TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models

Zhuohan Li
Siyuan Zhuang
Shiyuan Guo
Danyang Zhuo
Hao Zhang
Dawn Song
Ion Stoica

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-258
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-258.html

December 16, 2021
TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models

by Zhuohan Li

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Ion Stoica
Research Advisor

12/15/2021

(Date)

Professor Dawn Song
Second Reader

12/14/2021

(Date)
TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models

by Zhuohan Li

Abstract

Model parallelism has become a necessity for training modern large-scale deep language models. In this work, we identify a new and orthogonal dimension from existing model parallel approaches: it is possible to perform pipeline parallelism within a single training sequence for Transformer-based language models thanks to its autoregressive property. This enables a more fine-grained pipeline compared with previous work. With this key idea, we design TeraPipe, a high-performance token-level pipeline parallel algorithm for synchronous model-parallel training of Transformer-based language models. We develop a novel dynamic programming-based algorithm to calculate the optimal pipelining execution scheme given a specific model and cluster configuration. We show that TeraPipe can speed up the training by 5.0x for the largest GPT-3 model with 175 billion parameters on an AWS cluster with 48 p3.16xlarge instances compared with state-of-the-art model-parallel methods.
Abstract

Model parallelism has become a necessity for training modern large-scale deep language models. In this work, we identify a new and orthogonal dimension from existing model parallel approaches: it is possible to perform pipeline parallelism within a single training sequence for Transformer-based language models thanks to its autoregressive property. This enables a more fine-grained pipeline compared with previous work. With this key idea, we design TeraPipe, a high-performance token-level pipeline parallel algorithm for synchronous model-parallel training of Transformer-based language models. We develop a novel dynamic programming-based algorithm to calculate the optimal pipelining execution scheme given a specific model and cluster configuration. We show that TeraPipe can speed up the training by 5.0x for the largest GPT-3 model with 175 billion parameters on an AWS cluster with 48 p3.16xlarge instances compared with state-of-the-art model-parallel methods. The code for reproduction can be found at https://github.com/zhuohan123/terapipe.

1. Introduction

Transformer-based language models (LMs) have revolutionized the area of natural language processing (NLP) by achieving state-of-the-art results for many NLP tasks, including text classification, question answering, and text generation (Brown et al., 2020; Radford et al.). The accuracy of a Transformer-based LM grows substantially with its model size, attributing to the fact that they can be unsupervisedly trained on almost unlimited text data. Today, a large LM, such as GPT-3 (Brown et al., 2020), can have more than 175B parameters, which amounts to 350 GB, assuming 16-bit floating-point numbers. This significantly exceeds the memory capacity of existing hardware accelerators, such as GPUs and TPUs, which makes model-parallel training a necessity, i.e., partitioning the model on multiple devices during the training process.

Because of the demands for efficient LM training, many researchers and industry practitioners have proposed different ways for model parallel training. One approach is to partition the weight matrices and dispatch smaller matrix operations to parallel devices (Figure 1b; Shoeybi et al., 2019; Shazeer et al., 2018). Another approach is to split a batch of training data into many microbatches and then evenly pipeline the layer computations across different microbatches and devices (Figure 1c; Huang et al., 2019). Unfortunately, these approaches either introduce excessive communication overheads between compute devices, or lead to reduced efficiency due to pipeline “bubbles” (i.e. device idle time, see Section 2 and 3.2 for details).

Our key observation in this paper is that Transformer-based language models have a key property: the computation of a given input token only depends on previous tokens, but not on future tokens. This lack of dependency on future tokens provides new opportunities for pipeline parallel training. In particular, it allows us to create a fine-grained pipeline within a single training sequence for Transformer-based LMs, by parallelizing the computation of the current token on the current layer with the computation of the previous token on the next layer of the model. For example, in Figure 1d, we can pipeline the execution across all 5 devices within a single input sequence. Similar to other synchronous model parallel training methods, e.g., Gpipe (Huang et al., 2019), Megatron-LM (Shoeybi et al., 2019), we do not change the underlying optimization algorithm, so the resulting model has exactly the same accuracy.

However, leveraging the token dimension for efficient model parallel training raises several challenges. First, if the partitioning along the token dimension is too fine-grained, it leads to under-utilization on devices that require large blocks...
of data for efficient processing (e.g., GPU). Second, since each token position in the sequence depends on all previous tokens, different positions in a transformer layer exhibit uneven computation loads. This means that uniformly partitioning along the token dimension might cause uneven load across devices, and degenerate the training efficiency.

To this end, we design and implement TeraPipe, a high-performance synchronous model parallel training approach for large-scale Transformer-based language models, which exploits the token dimension to pipeline the computation across devices. TeraPipe uses a small number of simple workloads to derive a performance model and then uses a novel dynamic programming algorithm to compute the optimal partitioning of the token dimension for the pipeline. TeraPipe is orthogonal to previous model-parallel training methods, so it can be used together with these methods to further improve the training performance. Our evaluation shows that for the largest GPT-3 model with 175 billion parameters, TeraPipe achieves a 5.0x speedup improvement over the state-of-the-art synchronous model parallel training approach (Shazeer et al., 2018) study how to optimize the performance synchronous model parallel training approach (Shoeybi et al., 2019). An allreduce operation is required to synchronize the results of each layer. (c) shows microbatch-based pipeline parallelism (Huang et al., 2019), which allows different microbatches (red and green bars) to be executed on different layers of the DNN in parallel. (d) show TeraPipe (our work), which pipelines along the token dimension.

Our paper makes the following contributions:

- We propose a new dimension, token dimension, for pipeline-parallel training of Transformer-based LMs.
- We develop a dynamic programming algorithm to compute a partition along the token dimension to maximize pipeline parallelism.
- We implement TeraPipe and show that we can increase the synchronous training throughput of the largest GPT-3 model (with 175 billion parameters) by 5.0x over the previous state-of-the-art model-parallel methods.

2. Related Work

Data parallelism scales ML training by partitioning training data onto distributed devices (Zinkevich et al., 2010; Krizhevsky, 2014; Goyal et al., 2017; Rajbhandari et al., 2019). Each device holds a model replica, works on an independent data partition, and synchronizes the updates via allreduce (Krizhevsky, 2014) or a parameter server (Li et al., 2014). Data parallelism alone is not enough to train large-scale DNNs due to two main reasons: (1) every device has to have enough memory to store the model and the gradients generated during the training process; (2) communication can be a performance bottleneck to synchronize model parameters.

Model parallelism allows for training models larger than the memory capacity of a single device, by partitioning the model (e.g., layers) into disjoint parts and executing each on a dedicated device. Existing model parallel training approaches can be roughly categorized as: operation partitioning and pipeline parallelism.

Operation partitioning. One way to split the model is to partition and parallelize computational operations across multiple devices. For example, the computation of matrix multiplications (matmul) $XAB$ can be split across multiple devices by partitioning $A$ and $B$ along its rows and columns, respectively.

$$XAB = X \cdot [A_1 \quad A_2] \cdot [B_1 \quad B_2] = XA_1B_1 + XA_2B_2.$$  

This means we can have one device calculate $XA_1B_1$ and another device calculate $XA_2B_2$ in parallel. After that, cross-device communication is needed to compute the sum of these two parts.

Many existing works (Jia et al., 2018; 2019; Wang et al., 2019; Shazeer et al., 2018) study how to optimize the
Partitioning schemes for different operations to maximize throughput and minimize communication overheads, among which, Megatron-LM (Figure 1b; Shoeybi et al., 2019) designs partitioning schemes specifically for large-scale Transformers. However, due to the excessive communication required to collect partial results after each layer, it is not efficient when the bandwidth between devices is limited (Shoeybi et al., 2019). Flexflow (Jia et al., 2018) proposes a framework to find the optimal operation partitioning, but it cannot model the new dimension proposed in our work.

**Pipeline parallelism** partitions a DNN into layers and puts different layers onto different devices (Figure 1c; Petrowski et al., 1993). Each device computes the input on a given layer and sends the result to the next device. Pipeline parallelism significantly reduces communication between devices, because only devices holding neighboring layers need to communicate and they only need to communicate the activations on a particular layer.

Previous pipeline parallel training methods are based on microbatch pipelining, e.g., GPipe (Huang et al., 2019). This means the computation for a given microbatch in a minibatch on a layer can run in parallel with the next microbatch in the same minibatch on the previous layer. However, microbatch-based pipeline parallelism still cannot achieve high efficiency due to its pipeline bubbles. This is because the start of the forward propagation on a microbatch requires the backward propagation of the previous minibatch to complete (Figure 2a). This problem becomes more severe when model sizes increase (see Section 3.2). Harlap et al. (2018) propose using an asynchronous training algorithm to mitigate the effect of pipeline bubbles in microbatch-based pipeline parallel training, but asynchronous training introduces uncertainty in model accuracy and is thus not widely adopted for training DNNs.

**Wavefront parallelism** is a variant of pipeline parallelism, broadly applied in shared-memory multiprocessors (Singhroy & Szymanski, 1994; Manjikian & Abdelrahman, 1996). In deep learning, it has been used to accelerate the computation of multi-layer RNNs on a single GPU (Appleyard et al., 2016), where different input positions of different layers can execute in parallel in a wavefront fashion to maximize the utilization of the GPU. However, wavefront parallelism cannot accelerate the execution of Transformers because there is no dependency between different input positions within a single Transformer layer to begin with. In addition, wavefront parallelism uses fine-grained per-word pipelining due to the temporal data dependency in RNNs, while too fine-grained pipelining in TeraPipe would lead to inferior pipeline efficiency (see Section 3.2 and 3.3).

![Diagram](image-url) **Figure 2.** Execution timeline for different pipelining methods. Grey blocks indicate GPUs idle time (a.k.a. pipeline bubbles). (a) Microbatch-based pipeline parallelism (e.g. GPipe). Each color corresponds to a microbatch. (b) Microbatch-based pipeline parallelism with longer sequence (hence smaller minibatch size due to fixed GPU memory). Pipeline bubbles significantly increase. (c) TeraPipe. Pipeline bubbles are substantially reduced because of the improved pipelining granularity.

### 3. Method

In this section, we briefly introduce language modeling and Transformers. Based on their structures, we identify new opportunities for performing pipelining along the input sequence (which we will note as the token dimension in the rest of the paper). With that, we derive the optimal slicing scheme over the token dimension to maximize pipeline efficiency using a dynamic programming algorithm. Finally, we show how to combine our new method with existing parallel training techniques.

#### 3.1. Language Modeling and Transformers

The task of language modeling is usually framed as unsupervised distribution estimation of a text corpus $\mathcal{X}$, where each example $x \sim \mathcal{X}$ is a variable length sequence of tokens $(x_1, x_2, \ldots, x_L)$. Since language has a natural sequential ordering, it is common to factorize the joint probability over the tokens as the product of conditional probabilities (a.k.a. autoregressive decomposition; Bengio et al., 2003):

$$
    P(x) = \prod_{t=1}^{L} P(x_t | x_1, \ldots, x_{t-1}). 
$$

Transformer (Vaswani et al., 2017) is the state-of-the-art architecture for modeling these conditional probabilities. As
visualized in Figure 1a, a Transformer-based LM \( F \) takes the sequence \( \langle \text{sos} \rangle, x_1, \ldots, x_L \) as input, where \( \langle \text{sos} \rangle \) represents the start of a sentence, and outputs a probability distribution \( p_t \) at each position \( t \) that models the conditional probability \( P(x_t | x_1, \ldots, x_{t-1}) \) as in Eq. 1. In practice, \( F \) is stacked with many Transformer layers \( F = f_N \circ f_{N-1} \circ \cdots \circ f_1 \) (Vaswani et al., 2017; Radford et al.; Yang et al.): \( f_1 \) takes the embedding of the original sequence as input, while \( f_i \) \((i > 1)\) takes the output of \( f_{i-1} \) as input. The main components of a Transformer layer \( f \) contain a self-attention layer and a position-wise feed-forward network layer:

\[
\text{SelfAtt}(h_i; h_1, \ldots, h_{i-1}) = \sum_{s=1}^{t} \alpha_{ts} \cdot (W_V h_s),
\]

where \( \alpha_{ts} = \text{softmax} \left( \frac{(W_Q h_t)^\top (W_K h_s)}{\sqrt{H}} \right) \); \( \text{FFN}(h_i) = W_2 \sigma(W_1 h_i + b_1) + b_2. \)

The operation and data dependency in Transformers make it more amenable to parallelization on GPUs/TPUs compared to RNNs (Vaswani et al., 2017). Therefore, Transformers have been scaled to enormous datasets and achieved state-of-the-art performance on a wide range of NLP tasks (Vaswani et al., 2017; Devlin et al., 2018; Radford et al.; Yang et al., 2019; Brown et al., 2020; Liu et al., 2019). Recently, people show that the accuracy of LMs can consistently improve with increasing model sizes (Radford et al.; Yang et al., 2019). While the growing model size greatly exceeds the memory capacity of a single GPU (Brown et al., 2020), model parallelism becomes a necessity for training large-scale LMs (Shoeybi et al., 2019).

3.2. Pipeline Parallelism Within a Sequence

In this subsection, we expose the limitations of existing pipelining parallelism approaches, and develop the proposed new pipelining method for Transformer-based LMs.

Typically, to perform pipeline parallelism, a Transformer model \( F \) is partitioned into multiple cells \( c_1, \ldots, c_K \). Each cell \( c_k \) consists of a set of consecutive Transformer layers \( f_j \circ \cdots \circ f_{i+1} \circ f_i \) so that \( F = c_K \circ \cdots \circ c_2 \circ c_1 \). Each \( c_k \) is placed and executed on the \( k \)-th device (e.g. GPU). The output of cell \( c_k \) is sent to cell \( c_{k+1} \) during forward propagation, and the backward states computed on cell \( c_{k+1} \) is sent to cell \( c_k \) during backward propagation. Since each layer \( f \) exhibits the same structure, the entire LM can be uniformly partitioned: each cell possesses the same number of layers hence the same amount of computation workload, to reach optimal pipeline efficiency (see Figure 2).

However, previous pipelining methods (Huang et al., 2019; Harlap et al., 2018) do not perform well on large Transformer-based LMs due to the growing model size. Consider a minibatch of size \( B \). The input to a Transformer layer \( f \) is a 3-dimensional tensor \( (h^{(1)}, h^{(2)}, \ldots, h^{(B)}) \) of size \((B, L, H)\), where \( L \) is the sequence length and \( H \) is the hidden state size. To improve accuracy, large LMs are often configured to have a large \( L \) to capture longer-term dependency in language sequences (Tay et al., 2020; Zaheer et al., 2020). To fit the model into a GPU, the minibatch size \( B \) has to decrease accordingly. The pipeline bubbles become larger (Figure 2b) because fewer input sequences can be processed in parallel.

In this work, we make a key observation: for Transformer-based LMs, with appropriate scheduling, the token dimension \( L \) can be pipelined for parallel training; and this pipelining dimension is complementary to other model parallelism approaches. Precisely, for an input hidden state sequence \((h_1, h_2, \ldots, h_L)\), the computation of a self-attention layer \( \text{SelfAtt}(h_i) \) only depends on the hidden states of previous positions \((h_1, \ldots, h_{i-1})\), and the computation of a feed-forward layer \( \text{FFN}(h_i) \) only depends on \( h_i \) itself. These offer a new opportunity for pipelining: the computation of layer \( f_i \) at step \( t \) can commence once the hidden states of previous steps \((< t)\) at \( f_{i-1} \) are ready, which, also, can be parallelized with the computation of latter steps at \( f_{i-1} \), illustrated in Figure 1d. This property enables us to perform pipeline parallelism within a single input sequence. Specifically, we can split an input sequence \( x_1, \ldots, x_L \) into \( s_1, \ldots, s_M \), where each subsequence \( s_i \) consists of tokens \((x_j, x_{j+1}, \ldots, x_{x_r})\). The computation of \( c_1, \ldots, c_K \) over \( s_1, \ldots, s_M \) can be pipelined, for example: when \( c_k \) computes over \( s_i \), \( c_k+1 \) can process \( s_{i-1} \) and \( c_{k-1} \) can process \( s_{i+1} \) in parallel.

Considering that nowadays LMs operate on sequences with thousands of tokens (Brown et al.; Brown et al., 2020) (e.g. 2048 for GPT-3), the token dimension opens substantial space to improve the pipelining efficiency. However, applying it in practice is still challenging, especially on GPUs, for the following reasons.

First, finer-grained pipelining (i.e. picking a small \(|s_i|\)) is prone to underutilizing the computational power of GPUs, and thus lowering the training throughput. As shown on the top part of Figure 3, for a single layer of the GPT3-1B model (see Table 1 for specs), the forward propagation time for an input sequence with a single token is the same as an input sequence with 256 tokens. In this case, the GPU is not being fully utilized for input sequence lengths less than 256. This means a large subsequence length is needed to achieve high throughput for a single layer (see the bottom part of Figure 3). On the other hand, although GPUs have better
We propose a dynamic programming (DP) algorithm to parallelize propagation latency, where the efficiency of computation.

Let’s first consider the latency of forward propagation. As shown in Section 3.2, all cells \( c_k \) have exact same amount from its previous layer, while the computation of \( \text{SelfAtt}(h_1) \) only requires the hidden state \( h_1 \) from its previous layer, while the computation of \( \text{SelfAtt}(h_L) \) takes all \( h_1, \ldots, h_L \) as inputs, as shown in Figure 1a. Therefore, the computation load on a later token position in a sequence is heavier than that of previous tokens. Since the total latency of a pipeline is determined by its slowest stage (Figure 4), an optimal slicing scheme should have a long slice in the beginning and a shorter slice in the end. We next develop methods to select the optimal slicing scheme over the token dimension.

### 3.3. Selecting Optimal Slicing Scheme

We propose a dynamic programming (DP) algorithm to partition the input sequence to achieve the optimal pipeline efficiency. Specifically, given a partitioned Transformer-based LM \( F = c_K \circ \cdots \circ c_1 \) and a training input sequence of length \( L \), the goal of the algorithm is to find the slicing scheme \( l_1, \ldots, l_M \) to minimize the total forward and backward propagation latency, where \( l_i = |s_i| \) is the length each sub-sequence slice \( s_i \) (\( l_1 + \cdots + l_M = L \)).

Let’s first consider the latency of forward propagation. As shown in Section 3.2, all cells \( c_k \) have exact same amount of computation.

The overall latency consists of two terms: The first term here is the total forward propagation time on a device (i.e. on a cell \( c_k \)); The second term is the overhead brought by the pipeline execution, which is determined by the slowest component in the whole pipeline multiplied by the number of pipeline stages \( K \) minus 1. For example, on the top of Figure 4, the total execution time will be \( T = (l_1 + \cdots + t_4) + 3t_4 \).

Our goal is to find the optimal slicing scheme \( l_1, \ldots, l_M \) that achieves the optimal latency \( T^* \). We choose to first enumerate the second term \( t_{\max} = \max_{1 \leq j \leq M} \{ t_j \} \) and minimize the first term for each different \( t_{\max} \). In other...
TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models

Algorithm 1 Selecting optimal slicing scheme given $t_{\text{max}}$.

**Input:** Forward propagation time function $t_{\text{fwd}}$ and maximum per-slice time $t_{\text{max}}$.

**Output:** Minimal total forward propagation time $S^\ast(L; t_{\text{max}})$ and the corresponding slicing scheme $l_1, \ldots, l_M$.

// Dynamic programming for the total forward propagation time.

$S^\ast(0; t_{\text{max}}) \leftarrow 0$

for $i$ from 1 to $L$ do

$S^\ast(i; t_{\text{max}}) \leftarrow \min_{1 \leq k \leq i} \{ S^\ast(i - k; t_{\text{max}}) + t_{\text{fwd}}(k, i - k) \mid t_{\text{fwd}}(k, i - k) \leq t_{\text{max}} \}$

$q_i \leftarrow \arg\min_{1 \leq k \leq i} \{ r_i - k + t_{\text{fwd}}(k, i - k) \mid t_{\text{fwd}}(k, i - k) \leq t_{\text{max}} \}$

end for

// Derive the optimal slicing scheme.

$i \leftarrow L$

while $i > 0$ do

$t_{\text{fwd}}(q_i)$

$i \leftarrow i - q_i$

end while

words, we reformulate $T^\ast$ as:

$$T^\ast = \min_{t_{\text{max}}} \left\{ S^\ast(L; t_{\text{max}}) + (K - 1) \cdot t_{\text{max}} \right\} ,$$  \hspace{1cm} (6)

$$S^\ast(L; t_{\text{max}}) = \min_{t_1 + \ldots + t_L = L} \left\{ \sum_{i=1}^M t_i \mid t_i \leq t_{\text{max}} \right\} .$$  \hspace{1cm} (7)

Note that $S^\ast(\cdot; t_{\text{max}})$ has the following optimal substructure:

$$S^\ast(i; t_{\text{max}}) = \frac{\min_{1 \leq k \leq i} \{ S^\ast(i - k; t_{\text{max}}) + t_{\text{fwd}}(k, i - k) \mid t_{\text{fwd}}(k, i - k) \leq t_{\text{max}} \}}{}.$$  \hspace{1cm} (8)

Therefore, we can get the slicing scheme $l_1, \ldots, l_M$ that achieves the total forward propagation time $S^\ast(L; t_{\text{max}})$ with Algorithm 1. By enumerating all different $t_{\text{max}}$, we can get the optimal slicing scheme that reaches the optimal overall pipeline latency $T^\ast$.

**Complexity.** With our DP algorithm, we can compute the best partition in $O(L^2)$ time for a fixed $t_{\text{max}}$. Note that in total there are at most $O(L^2)$ different choices ($t_{\text{fwd}}(i, j)$ for $i, j = 1, \ldots, L$) of $t_{\text{max}}$. We therefore can derive the optimal slicing scheme in $O(L^2)$ time.

**Optimization.** To further accelerate the above DP algorithm, we enumerate different $t_{\text{max}}$ from small to large; when $K \cdot t_{\text{max}}$ is greater than the current best $T$, we stop the enumeration since larger $t_{\text{max}}$ cannot provide a better slicing scheme. In addition, during enumeration of $t_{\text{max}}$, we only evaluate with $t_{\text{max}}$ larger than the last $t_{\text{max}}$ by at least $\varepsilon$. In this case, the gap between the solution found by the DP algorithm and the global optima is at most $K \cdot \varepsilon$. We choose $\varepsilon = 0.1$ ms in our evaluation and observe that the solution given by Algorithm 1 and the real optimal solution ($\varepsilon = 0$) are always the same in all our evaluated settings. With these two optimizations, the dynamic programming can finish within a minute in our evaluations.

**Estimating $t_{\text{fwd}}$.** To avoid the cost of evaluating $t_{\text{fwd}}(i, j)$ for all $O(L^2)$ combinations of $(i, j)$ on real clusters, we use a simple performance model to estimate $t_{\text{fwd}}$. Specifically, we split $t_{\text{fwd}}(i, j)$ into two terms:

$$t_{\text{fwd}}(i, j) = t_{\text{fwd}}(i, 0) + t_{\text{ctx}}(i, j),$$  \hspace{1cm} (9)

where $t_{\text{fwd}}(i, 0)$ is the forward propagation time without any extra context input and $t_{\text{ctx}}(i, j)$ is the latency overhead brought by the extra context input. We measure the first term with all $L$ choices of $i$ and we fit a simple linear model $t_{\text{ctx}}(i, j) = a_0 + a_1 i + a_2 j + a_3 ij$ for the second term with a subset of all $(i, j)$ combinations. In our experiments, the linear model can achieve $a < 2\%$ relative prediction error compared to the actual overhead.

The development above can be applied to backward propagation time $t_{\text{bwd}}$, since the backward propagation computation in transformers is symmetric with its forward counterpart. One step further, we can replace all the $t_{\text{fwd}}$ above with $t_{\text{fwd}} + t_{\text{bwd}}$ to derive the optimal slicing scheme that minimizes the total training time.

### 3.4. Combining with Other Parallel Training methods

The new dimension to perform pipeline parallelism proposed by TeraPipe is orthogonal to all previous model parallel techniques, hence can be naturally combined with them. We explain next how TeraPipe can be combined with other parallelization methods and show, when combined, it significantly boosts parallelization performance in Section 4.

**Combine with microbatch-based pipeline parallelism.** To combine with microbatch-based pipeline parallelism (Huang et al., 2019), we slice the batch dimension and the token dimension jointly to form the pipeline. Specifically, consider a training input batch $(x_1(1), x_2(1), \ldots, x_B(1))$, where each $x_i(1)$ is an input sequence $(x^{(i)}_1, \ldots, x^{(i)}_L)$ of length $L$, we partition the input batch into $(s_1^{(a)}, s_2^{(a)}, \ldots, s_L^{(a)})$, such that each $s_i^{(a)}$ includes $(x^{(a)}_1, x^{(a)}_{i+1}, \ldots, x^{(a)}_r)$, $(x^{(a+1)}_1, x^{(a+1)}_{i+1}, \ldots, x^{(a+1)}_r)$, $\ldots$, $(x^{(b)}_1, x^{(b)}_{i+1}, \ldots, x^{(b)}_r)$, which is the subsequence from position $l$ to $r$ of input data $a$ to $b$. During training, all slices $(s_1^{(1)}, s_2^{(1)}, \ldots, s_L^{(D)})$, $(s_1^{(2)}, s_2^{(2)}, \ldots, s_L^{(D)})$, $\ldots$, $(s_1^{(D)}, s_2^{(D)}, \ldots, s_L^{(D)})$ can execute on cells $c_1, \ldots, c_K$ in a pipelined fashion. To jointly optimize the sequence slicing and batch splitting, the DP algorithm in Section 3.3 can be extended to include the batch dimension: we can first run the whole DP algorithm in Section 3.3 for all different batch sizes $b$ from 1 to $B$. For each
Table 1. Model settings and parallel training setups used in the evaluation. N: Number of Transformer layers. H: Hidden state size. #Params: Number of total parameters. L: Input sequence length. #GPUs: Total number of GPUs. B: Batch size. #Data: Number of data parallel shards. #Pipe: Number of pipeline stages. #Op: Number of GPUs used for operational partitioning by each Transformer layer.

<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th>H</th>
<th>#Params</th>
<th>L</th>
<th>#GPUs</th>
<th>B</th>
<th>#Data</th>
<th>#Pipe</th>
<th>#Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) GPT3-1B</td>
<td>24</td>
<td>2048</td>
<td>1B</td>
<td>2048</td>
<td>192</td>
<td>72</td>
<td>2</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>(2)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4) GPT3-13B</td>
<td>72</td>
<td></td>
<td>13B</td>
<td>2048</td>
<td>320</td>
<td>32</td>
<td>2</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>(5)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>(6)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>(7)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>(8)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>(9) GPT3-175B</td>
<td>96</td>
<td>12288</td>
<td>175B</td>
<td>2048</td>
<td>384</td>
<td>8</td>
<td>1</td>
<td>96</td>
<td>4</td>
</tr>
<tr>
<td>(10)</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Training iteration latency for all configurations with and without TeraPipe. Details for each configuration are listed in Table 1.

$b$, we derive the optimal $T_b$ and the corresponding slicing scheme $s_b$. With all $T_b$ and $s_b$, we only need to determine the size of each slice in the batch dimension $b_1, \ldots, b_D$ such that $b_1 + \cdots + b_D = B$ and $T_{b_1} + \cdots + T_{b_D}$ is minimized. This reduces to a 1D knapsack problem and can be solved using off-the-shelf solvers.

**Combine with operation partitioning.** TeraPipe is orthogonal from operation partitioning in the sense that: operation partitioning is intra-operation parallelism that parallelizes the execution of a single operation, whereas TeraPipe pipelines the execution of different operations. To combine with operation partitioning, we distribute each pipeline parallel cell $c_K$ to a set of target devices and then perform operation partitioning across target devices.

**Combine with data parallelism.** Similarly, because data parallelism maintains multiple identical copies of the model, we can perform model parallelism for each data parallel model replica and synchronize the gradient updates between the replicas after each forward and backward propagation.

**Combine with memory optimization.** Same as previous pipeline parallel methods (Huang et al., 2019), TeraPipe stores the activations of a whole mini-batch in our implementation. TeraPipe can also be combined with various memory optimization techniques, e.g., gradient accumulation (Fan et al., 2020), rematerialization (Chen et al., 2016; Jain et al., 2019), or memory swapping (Ren et al., 2021). See supplementary material for more discussions on combining TeraPipe with gradient accumulation.

4. Evaluation

TeraPipe is a synchronous model parallel training method that performs exactly the same underlying optimization algorithm as training the model on a single device. The optimization performance of TeraPipe (i.e., training loss versus training iterations) is hence the same compared to training on a single device. Therefore, in this paper, we focus on the per-iteration latency (i.e., wall-clock time used per training iteration) as our evaluation metric.

We evaluate TeraPipe following the setup in Brown et al. (2020). Specifically, we test 3 settings in Brown et al. (2020): GPT3-1B, GPT3-13B, and GPT3-175B, which have 1 billion, 13 billion, and 175 billion parameters in total, respectively. Note that GPT3-175B is the largest setting in Brown et al. (2020). In addition, we also test on a GPT3-44B model with half the hidden state size $H$ of the GPT3-175B model, which includes 44 billion parameters in total.

For each model, we select multiple data parallelism, oper-
4.2. Dynamic Programming

In this subsection, we provide an ablation study on the effectiveness of the dynamic programming algorithm proposed in Section 3.3. We compare the training latency with the slicing scheme found by the dynamic programming algorithm, to a simple heuristic that slices the input sequence uniformly. Specifically, we evaluate GPT3-44B with setting (8) and GPT3-175B with setting (9). For the uniform slicing baseline, we slice the whole input on the batch dimension and range the number of slices on the token dimension from 1 to 16 and 1 to 128 for two settings, respectively, and evaluate the iteration latency for each uniform slicing scheme.

The result is shown in Figure 6. As in Section 3.2, too fine-grained pipeline (e.g. #slices=128 in Figure 6b) performs badly because of the underutilization of the GPUs. Also, too coarse-grained pipeline (e.g. #slices=4 in Figure 6b) has large pipeline bubbles, which leads to high iteration latency. In addition, because of the non-uniform running time brought by the Transformer structure, the slicing scheme derived by the dynamic programming program achieves better performance compared to the best uniform sliced pipeline: the optimal solutions found by dynamic programming are 1.12x and 1.04x faster compared to the best uniform slicing scheme for GPT3-44B and GPT3-175B model, respectively.
4.3. Longer Sequence Length

A growing set of works start to focus on increasing the input sequence length of the Transformers (Tay et al., 2020; Zafar et al., 2020; Kitaev et al., 2020). Long sequence length enables Transformers to reason about long-term dependencies and thus extends its applicability to more complex applications such as modeling documents. However, longer sequences increases the memory usage of a single input sequence, and decreases the maximum batch size allowed, which limits the pipeline efficiency of previous microbatch-based pipeline parallelism methods.

In this subsection, we vary the sequence length from 2048 to 8192 for the GPT3-13B model (setting (5)) and evaluate the training iteration latency. Because of the growth in memory usage, the batch sizes for sequence length 4096, 6144, 8196 are reduced to 8, 4, 2, respectively. We show the results in Figure 7. TeraPipe achieves 2.76x, 4.97x, 7.83x speedup for sequence length 4096, 6144, and 8196, respectively. As the sequence length grows, the gap between the performance with and without TeraPipe significantly increases, as expected. Meanwhile, longer sequence length provides more space on the token dimension and thus TeraPipe can perform even better – TeraPipe enables efficient training of future-emerging LMs with growing sequence lengths.

5. Conclusion

We present TeraPipe, a high-performance token-level pipeline parallel algorithm for training large-scale Transformer-based language model. We develop a novel dynamic programming-based algorithm to calculate the optimal pipelining execution scheme, given a specific LM and a cluster configuration. TeraPipe is orthogonal to other model parallel training methods and can be complemented by them. Our evaluations show that TeraPipe accelerates the synchronous training of the largest GPT-3 models with 175 billion parameters by 5.0x on an AWS cluster with 48 p3.16xlarge instances compared to previous methods.

Acknowledgement

We thank our anonymous reviewers for their insightful feedback. We also thank Lianmin Zheng and many others at the UC Berkeley RISELab for their helpful discussion and comments. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References


Appendix

A. Combine TeraPipe with Gradient Accumulation

TeraPipe and gradient accumulation (GA) are orthogonal and TeraPipe can further speed up over GA. To see this, we visualize a 3-stage pipeline training with an input batch of 6 training sequences below, similar to Figure 2 in the main paper.

In (a), we show the case where each GPU is capable of storing the intermediate activations of at most 3 input sequences. With scheduling algorithms like DAPPLE (Fan et al., 2020), GA indeed increases the pipeline efficiency. However in (b), when each GPU can only support 2 input sequences (due to large model size), the forward pass of input sequence 3 cannot start on GPU 1 until sequence 1 finishes the backward pass and release the memory of its intermediate activations. The memory constraint limits the pipeline efficiency: only two GPUs can work at a time, and GA cannot solve the issue. In (c), we follow the setting in (b) but enable TeraPipe to split a training sequence into two. TeraPipe improves the pipeline efficiency compared to (b) thanks to more fine-grained pipelining: the three can work at the same time.

In our experiments, we have 48 pipeline stages but a single GPU is only capable to hold 2 input sequences due to its memory capacity. Even with newer GPUs (e.g. 80GB A100, 5x memory compared to V100s in the paper), their memory capacity is still not enough to fulfill the pipeline with 48 input sequences. Therefore, even with GA, TeraPipe is still expected to significantly improve the training efficiency.

B. Implementation

We implement TeraPipe with PyTorch (Paszke et al., 2019) and NCCL (NCCL). We use Megatron-LM (Shoeybi et al., 2019) as the library for operation partitioning and implement microbatch-based pipeline parallelism and data parallelism by ourselves. The core of TeraPipe is implemented using 1714 lines of Python. We include the code in the supplementary material and the code will be open-sourced.

C. Experiment Results

Here, we include the detailed numbers (mean and standard deviation of the latency) and the slicing schemes found by the DP algorithms for all experiments in the main paper. Specifically, we list the details of Figure 5, 6, and 7 in Table 2, 3, and 4.
### Table 2. Detailed numbers and slicing schemes in main experiments (Figure 5 in the main paper).

<table>
<thead>
<tr>
<th>Model</th>
<th>Setting</th>
<th>Algorithm</th>
<th>Slicing Scheme</th>
<th>Latency (s)</th>
<th>TFlops (per GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT3-1B</td>
<td>5, (1)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 16</td>
<td>1.517 ± 0.107</td>
<td>0.8841</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 16</td>
<td>1.254 ± 0.160</td>
<td>1.0695</td>
</tr>
<tr>
<td>GPT3-1B</td>
<td>5, (2)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 36</td>
<td>1.018 ± 0.065</td>
<td>2.9643</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 36</td>
<td>1.018 ± 0.065</td>
<td>2.9643</td>
</tr>
<tr>
<td>GPT3-1B</td>
<td>5, (3)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 72</td>
<td>0.913 ± 0.027</td>
<td>6.6105</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 72</td>
<td>0.913 ± 0.027</td>
<td>6.6105</td>
</tr>
<tr>
<td>GPT3-1B</td>
<td>5, (4)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 16</td>
<td>2.637 ± 0.055</td>
<td>3.0305</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 16</td>
<td>1.891 ± 0.084</td>
<td>4.2261</td>
</tr>
<tr>
<td>GPT3-1B</td>
<td>5, (5)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 32</td>
<td>1.863 ± 0.007</td>
<td>8.5792</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [704, 668, 656]]] * 32</td>
<td>1.328 ± 0.037</td>
<td>12.0354</td>
</tr>
<tr>
<td>GPT3-44B</td>
<td>5, (6)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 8</td>
<td>13.319 ± 0.067</td>
<td>0.2148</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 8</td>
<td>7.103 ± 0.243</td>
<td>0.4028</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 4</td>
<td>4.311 ± 0.032</td>
<td>1.3274</td>
</tr>
<tr>
<td>GPT3-175B</td>
<td>5, (9)</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 2</td>
<td>9.990 ± 0.005</td>
<td>1.1300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 2</td>
<td>1.481 ± 0.002</td>
<td>7.6225</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 2</td>
<td>5.822 ± 0.003</td>
<td>1.9390</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [128]]] * 2</td>
<td>1.160 ± 0.001</td>
<td>9.7318</td>
</tr>
</tbody>
</table>

### Table 3. Detailed numbers and slicing schemes in ablation studies on the effectiveness of the dynamic programming algorithm (Figure 6 in the main paper).

<table>
<thead>
<tr>
<th>Model</th>
<th>Setting</th>
<th>Algorithm</th>
<th>Slicing Scheme</th>
<th>Latency (s)</th>
<th>TFlops (per GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT3-44B</td>
<td>6, (a)</td>
<td>#Slices=1</td>
<td>[[1, [2048]]] * 8</td>
<td>2.662 ± 0.001</td>
<td>4.2995</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=4</td>
<td>[[1, [2048]]] * 8</td>
<td>1.241 ± 0.003</td>
<td>9.2226</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=8</td>
<td>[[1, [2048]]] * 8</td>
<td>1.255 ± 0.004</td>
<td>9.1197</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=16</td>
<td>[[1, [2048]]] * 8</td>
<td>2.241 ± 0.003</td>
<td>9.2226</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DP</td>
<td>[[1, [2048]]] * 8</td>
<td>2.241 ± 0.003</td>
<td>9.2226</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=1</td>
<td>[[1, [2048]]] * 2</td>
<td>9.990 ± 0.005</td>
<td>1.1300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=4</td>
<td>[[1, [2048]]] * 2</td>
<td>2.962 ± 0.003</td>
<td>3.8900</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=8</td>
<td>[[1, [2048]]] * 2</td>
<td>1.892 ± 0.002</td>
<td>5.9667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Slices=16</td>
<td>[[1, [2048]]] * 2</td>
<td>1.547 ± 0.01</td>
<td>7.2973</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DP</td>
<td>[[1, [2048]]] * 2</td>
<td>1.547 ± 0.01</td>
<td>7.2973</td>
</tr>
</tbody>
</table>

### Table 4. Detailed numbers and slicing schemes in experiments with longer sequence lengths (Figure 7 in the main paper).

<table>
<thead>
<tr>
<th>Model</th>
<th>Input Sequence Length</th>
<th>Algorithm</th>
<th>Slicing Scheme</th>
<th>Latency (s)</th>
<th>TFlops (per GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT3-13B</td>
<td>2048</td>
<td>w/o TeraPipe</td>
<td>[[1, [2048]]] * 32</td>
<td>1.863 ± 0.007</td>
<td>8.5792</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [2048]]] * 32</td>
<td>1.863 ± 0.007</td>
<td>8.5792</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>w/o TeraPipe</td>
<td>[[1, [4096]]] * 8</td>
<td>2.526 ± 0.001</td>
<td>1.3819</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [4096]]] * 8</td>
<td>2.526 ± 0.001</td>
<td>1.3819</td>
</tr>
<tr>
<td></td>
<td>6144</td>
<td>w/o TeraPipe</td>
<td>[[1, [6144]]] * 4</td>
<td>3.754 ± 0.006</td>
<td>0.5322</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [6144]]] * 4</td>
<td>3.754 ± 0.006</td>
<td>0.5322</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>w/o TeraPipe</td>
<td>[[1, [8192]]] * 2</td>
<td>4.978 ± 0.004</td>
<td>0.2007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [8192]]] * 2</td>
<td>4.978 ± 0.004</td>
<td>0.2007</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>w/o TeraPipe</td>
<td>[[1, [1024]]] * 2</td>
<td>6.6105 ± 0.002</td>
<td>0.1570</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w/ TeraPipe</td>
<td>[[1, [1024]]] * 2</td>
<td>6.6105 ± 0.002</td>
<td>0.1570</td>
</tr>
</tbody>
</table>