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

Zhuohan Li [1] **Siyuan Zhuang** [1] **Shiyuan Guo** [1] **Danyang Zhuo** [2] **Hao Zhang** [1] **Dawn Song** [1] **Ion Stoica** [1]

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).

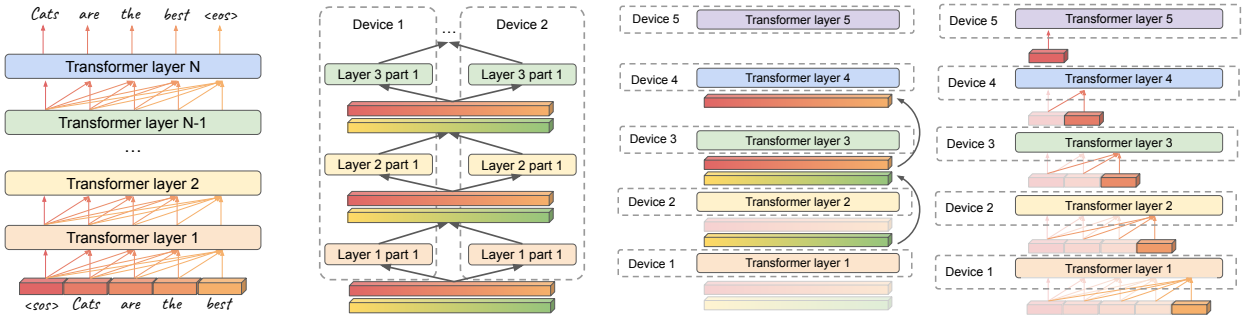
Our key observation in this paper is that Transformer-based language models have a key property: the computation of a given input token only depends on previous tokens, but not on future tokens. This lack of dependency on future tokens provides new opportunities for pipeline parallel training.[1] In 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

[1]UC Berkeley [2]Duke University. Correspondence to: Zhuohan Li <zhuohan@cs.berkeley.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

[1]In this paper, we focus on unidirectional autoregressive language models (e.g., GPT (Radford et al.; Brown et al., 2020)) but not bidirectional models like masked language models (e.g., BERT (Devlin et al., 2018)).



(a) Transformer-based LM (b) Operation partitioning (Megatron-LM) (c) Microbatch-based pipeline parallelism (GPipe) (d) Token-based pipeline parallelism (TeraPipe)

Figure 1. Different approaches of model parallel training of Transformer-based LMs. (a) shows a standard multi-layer Transformer LM. In each layer, each position only takes only its previous positions as input. (b) shows operation partitioning (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.

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 methods on an AWS cluster consisting of 48 p3.16xlarge instances.

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 spitted across multiple devices by partitioning A and B along its rows and columns, respectively.

$$XAB = X \cdot \begin{bmatrix} A_1 & A_2 \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = XA_1B_1 + XA_2B_2.$$

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.

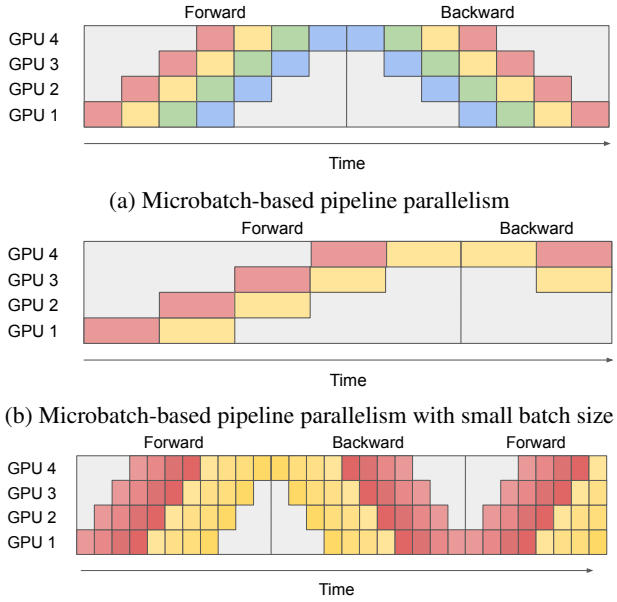
Many existing works (Jia et al., 2018; 2019; Wang et al., 2019; Shazeer et al., 2018) study how to optimize the

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 put 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 minibatch 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 microbach-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 (Sinharoy & 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).



(a) Microbatch-based pipeline parallelism

(b) Microbatch-based pipeline parallelism with small batch size

(c) TeraPipe

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 notate 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):

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

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 $(\langle sos \rangle, x_1, \ldots, x_{L-1})$ as input, where $\langle sos \rangle$ represents the start of a sentence, and outputs a probability distributions 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.): 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*:

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

$$\text{where } \alpha_{ts} = \text{softmax}\left(\frac{(W_Q h_t)^\top (W_K h_s)}{\sqrt{H}}\right); \quad (2)$$

$$\text{FFN}(h_t) = W_2 \sigma(W_1 h_t + b_1) + b_2. \quad (3)$$

$h_1, \ldots, h_L \in \mathbb{R}^H$ are hidden states correspond to each position of the input sequence, W and b are learnable parameters, and σ is the nonlinear activation function. An important note here: for each h_t, Eq. 2 takes only the hidden states before position t as inputs and Eq. 3 only takes h_t as input.

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 $\text{SelfAtt}(h_t)$ only depends on the hidden states of previous positions $(h_1, \ldots, h_{t-1})$, and the computation of a feed-forward layer $\text{FFN}(h_t)$ only depends on h_t 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 $(x_l, x_{l+1}, \ldots, x_r)$. 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 (Radford 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

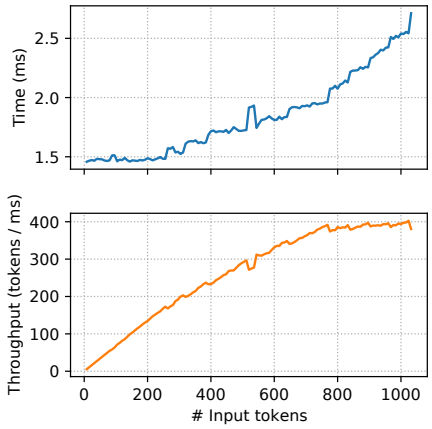


Figure 3. Forward propagation time and throughput for a single layer of GPT3-1B model with a single input sequence with different number of input tokens on a single NVIDIA V100 GPU, averaged by 30 independent runs. **Top:** Time per forward propagation. **Bottom:** Throughput measured by number of tokens per millisecond.

training throughput per layer for longer sequences due to the SIMD architecture and better locality, longer input slices lead to fewer pipeline stages within a sequence, which will increase the pipeline bubble, and thus reduce the pipeline efficiency and hurt the overall training speed.

Second, splitting inputs into multiple same-size chunks for pipelining, as normally done in existing work (Huang et al., 2019; Harlap et al., 2018), is not the ideal way for pipelining on the token dimension. For the self-attention layer, the computation of SelfAtt(h_1) only requires the hidden state h_1 from its previous layer, while the computation of 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 forward propagation time t_i for the

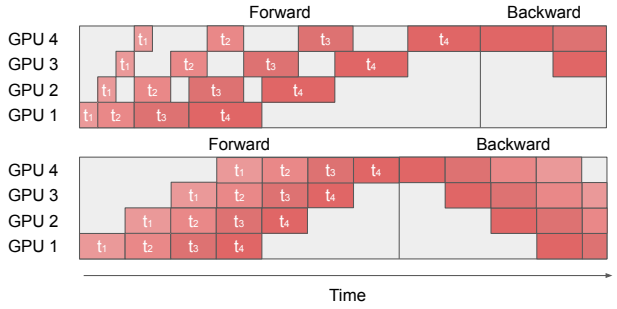


Figure 4. Execution timeline for inputs for uniform sequence split with non-uniform running time (top) and non-uniform sequence split with uniform running time (bottom). The total latency of a pipeline is determined by its slowest stage, and thus splits with non-uniform running time result in larger pipeline bubbles and inferior pipeline efficiency.

slice s_i on the cell c_k is determined by the length of the ith slice (l_i), the lengths of all the previous subsequences ($l_1, \ldots, l_{i-1}$), and the cluster specifications (e.g., GPU, bandwidth and latency of the underlying computer networks). We use t_{fwd} to denote the sum of the computation latency plus data transmission latency for a given l_i and the previous subsequences $l_1, \ldots, l_{i-1}$. We have:

$$t_i = t_{fwd}\left(l_i, \sum_{j=1}^{i-1} l_j\right). \tag{4}$$

Note the second term $\sum_{j=1}^{i-1} l_j$ is the total length of previous subsequences $s_1, \ldots, s_{i-1}$ to compute SelfAtt(s_t). As visualized in Figure 4, The optimal overall pipeline forward propagation latency is:

$$T^* = \min_{l_1,\ldots,l_M} \left\{ \sum_{i=1}^{M} t_i + (K-1) \cdot \max_{1\le j\le M}\{t_j\} \right\}. \tag{5}$$

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 $T = (t_1 + \ldots + t_4) + 3t_4$.

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 $t_{max} = \max_{1\le j\le M}\{t_j\}$ and minimize the first term for each different t_{max}. In other words, we reformulate T^* as:

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

$$S^*(L; t_{max}) = \min_{l_1+\cdots+l_M=L} \left\{ \sum_{i=1}^{M} t_i \mid t_i \le t_{max} \right\}. \tag{7}$$

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

Input: Forward propagation time function t_{fwd} and maximum per-slice time t_{max}.
Output: Minimal total forward propagation time $S^*(L; t_{max})$ and the corresponding slicing scheme $l_1, \ldots, l_M$.
// Dynamic programming for the total forward propagation time.
$S^*(0; t_{max}) \leftarrow 0$
for i **from** 1 **to** L **do**
  $S^*(i; t_{max}) \leftarrow \min_{1 \le k \le i} \{S^*(i-k; t_{max}) + t_{fwd}(k, i-k) \mid t_{fwd}(k, i-k) \le t_{max}\}$.
  $q_i \leftarrow \text{argmin}_{1 \le k \le i} \{r_{i-k} + t_{fwd}(k, i-k) \mid t_{fwd}(k, i-k) \le t_{max}\}$.
end for
// Derive the optimal slicing scheme.
$i \leftarrow L, l \leftarrow \{\}$
while $i > 0$ **do**
  $l.prepend(q_i)$
  $i \leftarrow i - q_i$
end while

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

$$S^*(i; t_{max}) = \min_{1 \le k \le i} \{S^*(i-k; t_{max}) + t_{fwd}(k, i-k) \mid t_{fwd}(k, i-k) \le t_{max}\}. \quad (8)$$

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

Complexity. With our DP algorithm, we can compute the best partition in $O(L^2)$ time for a fixed t_{max}. Note that in total there are at most $O(L^2)$ different choices ($t_{fwd}(i, j)$ for $i, j = 1, \ldots, L$) of t_{max}. We therefore can derive the optimal slicing scheme in $O(L^4)$ time.

Optimization. To further accelerate the above DP algorithm, we enumerate different t_{max} from small to large; when $K \cdot t_{max}$ is greater than the current best T, we stop the enumeration since larger t_{max} cannot provide a better slicing scheme. In addition, during enumeration of t_{max}, we only evaluate with t_{max} larger than the last t_{max} by at least ε. 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_{fwd}. To avoid the cost of evaluating $t_{fwd}(i, j)$ for all $O(L^2)$ combinations of i, j on real clusters, we use a simple performance model to estimate t_{fwd}. Specifically, we split $t_{fwd}(i, j)$ into two terms:

$$t_{fwd}(i, j) = t_{fwd}(i, 0) + t_{ctx}(i, j), \quad (9)$$

where $t_{fwd}(i, 0)$ is the forward propagation time without any extra context input and $t_{ctx}(i, j)$ 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 $t_{ctx}(i, j) = a_0 + a_1 i + a_2 j + a_3 ij$ 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_{bwd}, since the backward propagation computation in transformers is symmetric with its forward counterpart. One step further, we can replace all the t_{fwd} above with $t_{fwd} + t_{bwd}$ to derive the optimal slicing scheme that minimizes the total training time.

3.4. Combining with Other Parallel Training methods

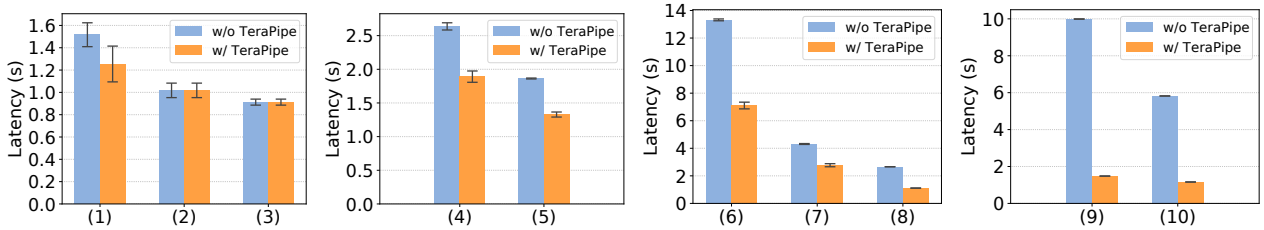
The new dimension to perform pipeline parallelism proposed by TeraPipe is orthogonal to all previous model parallel techniques, hence can be naturally combined with them. We explain next how TeraPipe can be combined with other parallelization methods and show, when combined, it significantly boosts parallelization 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 $(x^{(1)}, x^{(2)}, \ldots, x^{(B)})$, where each $x^{(i)}$ is an input sequence $(x_1^{(i)}, \ldots, x_L^{(i)})$ of length L, we partition the input batch into $(s^{(1)}, s^{(2)}, \ldots, s^{(D)})$, such that each $s_i^{(d)}$ includes $(x_l^{(a)}, x_{l+1}^{(a)}, \ldots, x_r^{(a)})$, $(x_l^{(a+1)}, x_{l+1}^{(a+1)}, \ldots, x_r^{(a+1)})$, $\ldots$, $(x_l^{(b)}, x_{l+1}^{(b)}, \ldots, x_r^{(b)})$, which is the subsequence from position l to r of input data a to b. During training, all slices $s_1^{(1)}, \ldots, s_M^{(1)}, s_1^{(2)}, \ldots, s_M^{(2)}, \ldots, s_1^{(D)}, \ldots, s_M^{(D)}$ 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 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 orthog-

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.

	Model	N	H	#Params	L	#GPUs	B	#Data	#Pipe	#Op
(1)	GPT3-1B	24	2048	1B	2048	192	128	8	24	1
(2)							72	2	12	8
(3)							72	1	24	8
(4)	GPT3-13B	40	5120	13B	2048	320	32	2	20	8
(5)							32	1	40	8
(6)	GPT3-44B	96	6144	44B	2048	384	8	4	96	1
(7)							8	2	24	8
(8)							8	1	48	8
(9)	GPT3-175B	96	12288	175B	2048	384	2	1	96	4
(10)							2	1	48	8



(a) GPT3-1B (b) GPT3-13B (c) GPT3-44B (d) GPT3-175B

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

onal 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.

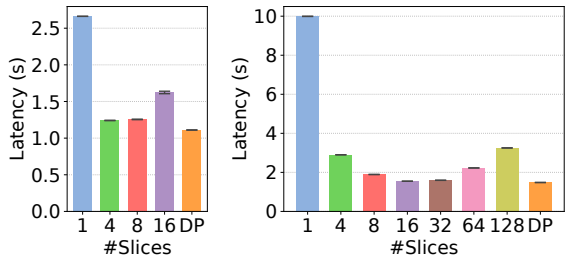
Combine with memory optimization. Same as previous pipeline parallel methods (Huang et al., 2019), TeraPipe stores the activations of a whole mini-batch in our implementation. TeraPipe can also be combined with various memory optimization techniques, e.g., gradient accumulation (Fan et al., 2020), rematerialization (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, operation partitioning, and pipeline parallelism setup combinations. The configuration details are shown in Table 1. For all configurations, we set the input sequence length $L = 2048$ following Brown et al. (2020). We evaluate the configurations on an AWS cluster with p3.16xlarge nodes (each with 8 NVIDIA V100 GPUs). For each model, we select a



(a) GPT3-44B (8) (b) GPT3-175B (9)

Figure 6. Training iteration latency of TeraPipe with uniform slicing scheme with different number of slices and the optimal slicing scheme find by the dynamic programming algorithm.

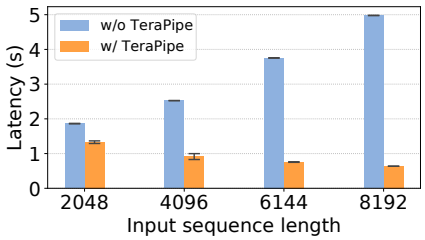


Figure 7. Training iteration latency of TeraPipe with different input sequence length for the GPT3-13B model.

cluster size based on its model size and number of layers so that each pipeline stage (each cell c_k) has the same number of layers. Since operation partitioning requires higher inter-connection speed compared to pipeline parallelism, we perform operation partitioning only inside a node, where all GPUs have high-speed inter-connection thanks to NVLink. For each configuration, we select the maximal batch size that can fit the memory of the GPUs.

We compare the per-iteration latency achieved by previous model parallel methods without TeraPipe and the latency achieved by TeraPipe for each configuration. Specifically, for the setup without TeraPipe, we measure the training latency with GPipe (Huang et al., 2019) as the pipeline parallel training method. For TeraPipe, we perform a joint dynamic programming on both batch and token dimension as shown in Section 3.4 and measure the training latency with the optimal slicing scheme found by the dynamic programming algorithm. All the latency results in the paper are averaged over 10 runs. The detailed numbers of the latency results and the solution find by the dynamic programming algorithm can be found in the supplementary material.

4.1. Main Results

We show the latency results for all configurations in Figure 5. TeraPipe accelerates the training for all models: For GPT3-1B, TeraPipe accelerates training for setting (1) by 1.21x. For setting (2) and (3), because of the large batch size, the optimal slicing scheme found by our dynamic programming algorithm only slices the batch dimension and thus TeraPipe does not provide speedup. For GPT3-13B, TeraPipe speeds up the training by 1.40x for both setting (4) and (5). For GPT3-44B, TeraPipe accelerates the training by 1.88x, 1.56x, and 2.40x for setting (6), (7), and (8), respectively. For GPT3-175B, TeraPipe accelerates the training by 6.75x and 5.02x for setting (9) and (10), respectively.

TeraPipe provides higher speedup for larger models: Larger models have a larger hidden state size H, and a larger portion of GPU memory is devoted to storing the model weights and hidden states. Therefore, the batch size B has to be decreased to fit the model into the GPU memory, as shown in the setup in Table 1. Smaller batch size B limits the previous microbatch-based pipeline parallel methods' ability to saturate the pipeline bubbles, while the token dimension used by TeraPipe still provides abundant opportunity to improve pipeline efficiency. In addition, larger models have more pipeline stages compared to smaller models, because larger models have more layers and each layer takes more memory than the smaller models. More pipeline stages require more input slices to saturate the pipeline.

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; Zaheer 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 ap-

plications 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

Appleyard, J., Kocisky, T., and Blunsom, P. Optimizing performance of recurrent neural networks on gpus. *arXiv preprint arXiv:1604.01946*, 2016.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

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

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

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al. Dapple: A pipelined data parallel approach for training large models. *arXiv preprint arXiv:2007.01045*, 2020.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pp. 103–112, 2019.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., and Gonzalez, J. E. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *arXiv preprint arXiv:1910.02653*, 2019.

Jia, Z., Lin, S., Ruizhongtai Qi, C., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. 02 2018.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

Kitaev, N., Kaiser, Ł., and Levskaya, A. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *ArXiv*, abs/1404.5997, 2014.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating*

Systems Design and Implementation ({OSDI} 14), pp. 583–598, 2014.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Manjikian, N. and Abdelrahman, T. S. Scheduling of wavefront parallelism on scalable shared-memory multiprocessors. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, volume 3, pp. 122–131. IEEE, 1996.

Petrowski, A., Dreyfus, G., and Girault, C. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6): 970–981, 1993. doi: 10.1109/72.286892.

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

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pp. 10414–10423, 2018.

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

Sinharoy, B. and Szymanski, B. Finding optimum wavefront of parallel computation. *Parallel Algorithms and Applications*, 2, 08 1994. doi: 10.1080/10637199408915404.

Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., and Metzler, D. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Wang, M., Huang, C.-c., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–17, 2019.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pp. 5753–5763, 2019.

Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., et al. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2020.

Zinkevich, M., Weimer, M., Li, L., and Smola, A. Parallelized stochastic gradient descent. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems*, volume 23, pp. 2595–2603. Curran Associates, Inc., 2010.