Appendix Outline

This Appendix provides background and preliminaries, more details of four components, additional experimental details and results, and discussions. The organization is as follows:

Background and Preliminaries. Appendix A provides the introduction for Transformer models, freeze training, pipeline parallelism, data parallelism, and hybrid of pipeline parallelism and data parallelism. This section serves as the required knowledge to understand `PipeTransformer`.

More Details of `Freeze Algorithm`, `AutoPipe`, `AutoDP`, `AutoCache`. Appendix B explains more details of design motivation for freeze training algorithm and shows details of the deviation; Appendix C provides more analysis to understand the design choice of `AutoPipe`; Appendix D contains more details of `AutoDP`, including the dataset redistributing, and comparing another way to skip frozen parameters; Appendix E introduces additional details for `AutoCache`.

More Experimental Results and Details. In Appendix F, we provide hyper-parameters and more experimental results. Especially, we provide more details of speedup breakdown in F.2.

Discussion. In Appendix 6, we will discuss pretraining v.s. fine-tuning, designing better freeze algorithms, and the versatility of our approach.

A. Background and Preliminaries

A.1. Transformer Models: ViT and BERT

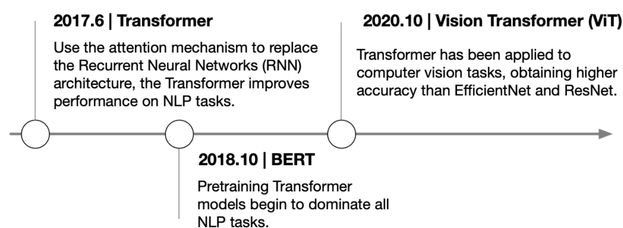

Figure 11. Evolution of Transformer Models.

Transformer. The Transformer model originates from the Natural Language Processing (NLP) community. It replaces the recurrent neural network (RNN) using a *self-attention* mechanism which relates different positions of a single sequence in order to compute a representation of the sequence. The transformer model has an encoder-decoder structure which is a classical structure for sequence modeling. The encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time. As shown in Figure 12, the Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder (left) and decoder (right). To better understand this architecture, we refer readers to the tutorial "The Annotated Transformer" [1].

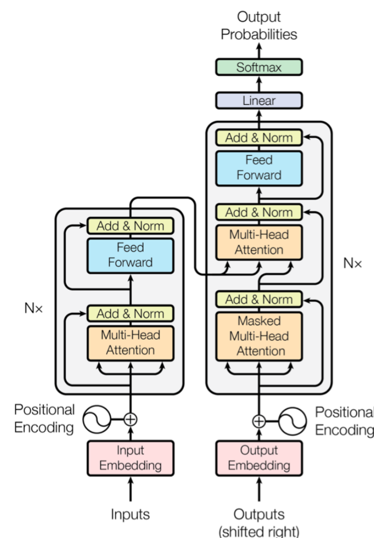

Figure 12. Transformer Model Architecture (Vaswani et al., 2017)

BERT (ViT). BERT (Devlin et al., 2018), which stands for Bidirectional Encoder Representations from Transformers, simply stacks multiple Transformer encoders (also called the Transformer layer, Figure 12, left). BERT $_{\text{BASE}}$ has 12 Transformer layers, and its total number of parameters is 110M. BERT $_{\text{LARGE}}$ has 24 Transformer layers, and its total number of parameters is 340M. BERT is pre-trained using unsupervised tasks (masked language model, and next sentence prediction) and then fine-tuned to various NLP tasks such as text classification and question answering.

Vision Transformer (ViT). ViT (Dosovitskiy et al., 2020) attains excellent results compared to state-of-the-art convolutional networks. Its architecture is shown in Figure 13. It splits an image into fixed-size patches, linearly embeds each of them, adds position embeddings, and feeds the resulting sequence of vectors to a Transformer encoder. Similar to BERT, the Transformer encode repeats multiple layers.

Model Architecture Comparison. Note that ViT and BERT's Transformer encoder places layer normalization

[1] `http://nlp.seas.harvard.edu/2018/04/03/attention.html`

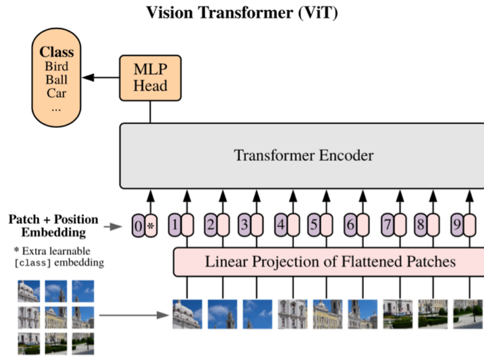

Figure 13. Vision Transformer (Dosovitskiy et al., 2020)

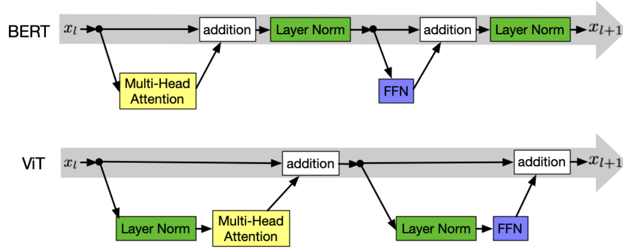

Figure 14. Comparison of Transform in BERT and ViT

in different locations. To understand the differences between these two architectures, please refer to the analysis in (Xiong et al., 2020). Due to this slight difference, our PipeTransformer source code implements the model partition of these two architectures separately.

A.2. Freeze Training.

The concept of freeze training is first proposed by (Raghu et al., 2017), which provides a posterior algorithm, named SVCCA (Singular Vector Canonical Correlation Analysis), to compare two representations. SVCCA can compare the representation at a layer at different points during training to its final representation and find that lower layers tend to converge faster than higher layers. This means that not all layers need to be trained through training. We can save computation and prevent overfitting by consecutively freezing layers. However, SVCCA has to take the entire dataset as its input, which does not fit an on-the-fly analysis. This drawback motivates us to design an adaptive on the fly freeze algorithm.

A.3. Pipeline Parallelism

In PipeTransformer, we reuse GPipe as the baseline. GPipe is a pipeline parallelism library that can divide different sub-sequences of layers to separate accelerators, which provides the flexibility of scaling a variety of different net-

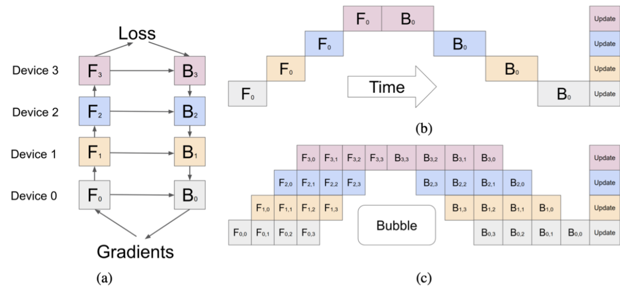

Figure 15. GPipe (Huang et al., 2018)

works to gigantic sizes efficiently. The key design in GPipe is that it splits the mini-batch into M micro-batches, which can train faster than naive model parallelism (as shown in Figure 15(b). However, as illustrated in Figure 15(c), micro-batches still cannot thoroughly avoid bubble overhead (some idle time per accelerator). GPipe empirically demonstrates that the bubble overhead is negligible when $M \geq 4 \times K$. Different from GPipe, PipeTransformer has an elastic pipelining parallelism in which K and pipeline number are dynamic during the training.

A.4. Data Parallelism

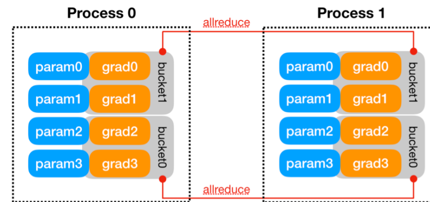

Figure 16. PyTorch DDP Bucket-based AllReduce

In PyTorch DDP (Li et al., 2020), to improve communication efficiency, gradients are organized into buckets, and AllReduce is operated on one bucket at a time. The mapping from parameter gradients to buckets is determined at the construction time, based on the bucket size limit and parameter sizes. Model parameters are allocated into buckets in (roughly) the reverse order of Model.parameters() from the given model. Reverse order is used because DDP expects gradients to be ready during the backward pass in approximately that order. Figure 16 shows an example. Note that, grad0 and grad1 are in bucket1, and the other two gradients are in bucket0. With this bucket design, DDP can overlap part of the communication time with the computation time of backward propagation.

A.5. Hybrid of Pipeline Parallelism and Data Parallelism

To understand the hybrid of pipeline parallelism and data parallelism, we illustrate the training process in Figure 17. This example is hybrid two-way data parallelism and two-stage pipeline parallelism: pipeline 0 has two partitions, using GPU 1 and 3; pipeline 1 also has two partitions, using GPU 0 and 2; two pipelines are synchronized by data parallelism. Each batch of training data is divided into micro-batches that can be processed in parallel by the pipeline partitions. Once a partition completes the forward pass for a micro-batch, the activation memory is communicated to the pipeline's next partition. Similarly, as the next partition completes its backward pass on a micro-batch, the gradient with respect to the activation is communicated backward through the pipeline. Each backward pass accumulates gradients locally. Subsequently, all data parallel groups perform `AllReduce` on gradients.

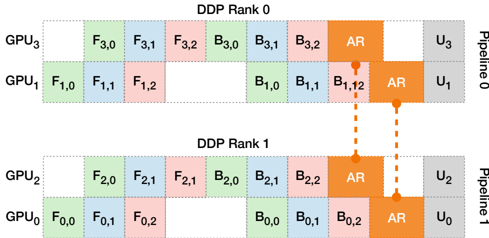

Figure 17. Illustration for Hybrid of Pipeline-parallel and Data-parallel

In this example, to simplify the figure, we assume that the bucket size is large enough to fit all gradients on a single device. That is to say, DDP uses one bucket per device, resulting in two `AllReduce` operations. Note that, since `AllReduce` can start as soon as gradients in corresponding buckets become ready. In this example, DDP launches `AllReduce` on GPU 1 and 3 immediately after $B_{3,1}$ and $B_{1,1}$, without waiting for the rest of backward computation. Lastly, the optimizer updates the model weights.

B. More Details of Freeze Algorithm

Explanation of Equation 1. In numerical optimization, the weight with the smallest gradient norm converges first. With this assumption, we use the gradient norm as the indicator to identify which layers can be frozen on the fly. To verify this idea, we save the gradient norm for all layers at different iterations (i.e., epoch). With this analysis, we found that in the later phase of training, the pattern of gradient norm in different layers matches the assumption, but in the early phase, the pattern is random. Sometimes, we can even see that the gradient norm of those layers close to the output is the smallest. Figure 18 shows such an example.

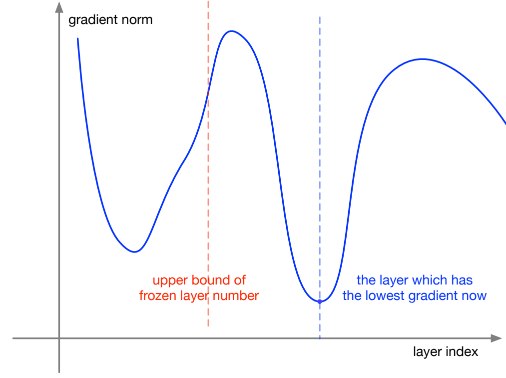

Figure 18. An example that the smallest gradient is not close to the input layer.

If we freeze all layers preceding the blue dash line layer, the freezing is too aggressive since some layers have not converged yet. This motivates us further amend this naive gradient norm indicator. To avoid the randomness of gradient norm at the early phase of training, we use a tunable bound to limit the maximum number of frozen layers. We do not freeze all layers preceding the layer with the smallest gradient norm for the case in the figure. Instead, we freeze layers preceding the bound (the red color dash line).

Deviation. The term $L_{\text{frozen}}^{(T-1)} + \alpha(L - L_{\text{frozen}}^{(T-1)})$ in Equation 1 can be written as:

$$L_{\text{frozen}}^{(T)} = (1-\alpha)^T [\frac{\alpha L}{1-\alpha} + \sum_{t=2}^{T} \frac{\alpha L}{(1-\alpha)^t}] \quad (3)$$

The deviation is as follows:

$$L_{\text{frozen}}^{(1)} = \alpha L \quad (4)$$

$$L_{\text{frozen}}^{(2)} = (L - L_{\text{frozen}}^{(1)})\alpha + L_{\text{frozen}}^{(1)} \quad (5)$$

$$L_{\text{frozen}}^{(T)} = (L - L_{\text{frozen}}^{(T-1)})\alpha + L_{\text{frozen}}^{(T-1)} \quad (6)$$

$$L_{\text{frozen}}^{(T)} = \alpha L + (1-\alpha) L_{\text{frozen}}^{(T-1)} \quad (7)$$

$$\frac{L_{\text{frozen}}^{(T)}}{(1-\alpha)^T} = \frac{\alpha L}{(1-\alpha)^T} + \frac{L_{\text{frozen}}^{(T-1)}}{(1-\alpha)^{(T-1)}} \quad (8)$$

$$\frac{L_{\text{frozen}}^{(T)}}{(1-\alpha)^T} = \frac{\alpha L}{(1-\alpha)} + \sum_{t=2}^{T} \frac{\alpha L}{(1-\alpha)^t} \quad (9)$$

$$(10)$$

C. More Details of `AutoPipe`

Balanced Partition: Trade-off between Communication and Computational Cost. Let us compute the communication cost in Figure 5. The intermediate tensor from partition $k-2$ needs two cross-GPU communications to arrive

to partition k. The parameter number of this intermediate tensor depends on the batch size and the Transformer model architecture. In BERT$_{\text{base}}$, the intermediate tensor width and height is the hidden feature size and sequence length, respectively (i.e., 1024, 512). If we use a batch size 300 in a pipeline, the total parameter number is $1024 \times 512 \times 300$. If we store it using `float32`, the memory cost is 0.63 GB. The GPU-to-GPU communication bandwidth is 15.754 GB (PCI 3.0, 16 lanes). Then one cross-GPU communication costs 40 ms. In practice, the time cost will be higher than this value. Therefore, two cross-GPU communications cost around 100 ms. To compare with the computation cost, we quantify the time cost for the forward propagation of a Transformer layer (12 million parameters), the time cost is around 35 ms, meaning that the communication cost for skip connection is far more than a specific layer's computation cost. Compared to a slightly unbalanced partition in parameter number wise, 100 ms is non-trivial. If we do not break the skip connection, the parameter number gap between different partitions is far less than 12 million (e.g., 4M or even less than 1 M). Therefore, this analysis explains partitioning without breaking the skip connection is a reasonable design choice. We also find that when the GPU device number in a machine is fixed (e.g., 8), the larger the model size is, the smaller the partition gap, which further indicates that our design's rationality.

Understanding Bubble in Pipeline. In the main text, Figure 6 depicts an example of running 4 micro-batches through a 4-device pipeline. Time flows from left to right, and each row denotes workload on one GPU device. `F` and `B` squares with the same color represent the forward and the backward pass time blocks of the same micro-batch. `U` represents the time block for updating parameters. Empty time blocks are bubbles. Assume that the load of the pipeline is evenly distributed amongst all devices. Consequently, all the time blocks during the forward pass are roughly in the same size, and similarly for backward time blocks. Note that the sizes of the forward time blocks can still differ from the backward ones. Based on these assumptions, we can estimate the per-iteration bubble size by simply counting the number of empty blocks during the forward and backward passes, respectively. In both the forward and backward pass, each device idles for $(K-1)$ time blocks. Therefore, the total bubble size is $(K-1)$ times per micro-batch forward and backward delay, which clearly decreases with fewer pipeline devices.

Relationship Between Number of Micro-batches per Mini-batch (M) and DDP. To understand the reason why M and `DDP` have mutual impacts, a thorough understanding of Section A.5 is needed first. In essence, `DDP` and pipelining has opposite requirement for M: `DDP` requires a relatively larger chunk of the bucket (smaller M) to overlap the communication (introduced in Section A.4), while pipelining requires a larger M to avoid bubble overhead (introduced in Section A.3). To further clarify, we must first remember that `DDP` must wait for the last micro-batch to finish its backward computation on a parameter before launching its gradient synchronization, then imagine two extreme cases. One case is that $M = 1$, meaning the communication can be fully overlapped with computation using buckets. However, setting $M = 1$ leads to a performance downgrade of pipelining (overhead of bubbles). Another extreme case is a very large M, then the communication time (labeled as green "AR" in Figure A.5) may be higher than the computation time for a micro-batch (note that the width of a block in Figure A.5 represents the wall clock time). With these two extreme cases, we can see that there must be an optimal value of M in a dynamical environment (K and parameter number of active layers) of `PipeTransformer`, indicating that it is sub-optimal to fix M during training. This explains the need for a dynamic M for elastic pipelining.

D. More details of `AutoDP`

D.1. Data Redistributing

In standard data parallel-based distributed training, PyTorch uses `DistributedSampler` to make sure each worker in DP only load a subset of the original dataset that is exclusive to each other. The example code is as follows:

```
self.train_sampler =
DistributedSampler(self.train_dataset,
num_replicas=num_replicas,
rank=local_rank)
```

Compared to this standard strategy, we made the following optimizations:

1. dynamic partition: the number of DP workers is increased when new pipelines have participated in DP. In order to guarantee that the data partition is evenly assigned after adding new pipes, the training dataset is repartitioned by rebuilding the `DistributedSampler` and setting new `num_replicas` and `rank` as arguments.

2. to reuse the computation of FP for frozen layers, we cached the hidden states in host memory and disk memory as well. Since the training requires to shuffle each epoch, the cache order of hidden features with respect to the order of original samples is different across different epochs. In order to identify which data point a hidden feature belongs to, we build a sample unique ID by returning `index` in the `get_item()` function of Dataset class. With this unique ID, we can find a sample's hidden feature with O(1) time complexity during training.

3. when data is shuffled in each epoch, a data sample trained in the previous epoch may be moved to another machine

for training in the current epoch. This makes the cache not reused across epochs. To address this issue, we fix a subset of entire samples in a machine and only do shuffle for this subset. This guarantees the shuffle during epochs is only executed inside a machine, thus the hidden feature's cache can be reused deterministically. To achieve this, rather than maintaining a global rank for DistributedSampler, we introduce `node_rank` and `local_rank`. `node_rank` is used to identify which subset of samples a machine needs to hold. `local_rank` is used by `DistributedSampler` to identify which part of the shuffle subset that a worker inside a machine should train. Note that this does not hurt the algorithmic convergence property. Shuffling for multiple subsets obtains more randomness than randomness obtained by a global shuffle, which further increases the robustness of training. The only difference is that some parallel processes in distributed training are fixed in part of the shuffled datasets. If a training task does not need to shuffle the dataset across epochs, the above-mentioned optimization will not be activated.

D.2. Skip Frozen Parameters in `AutoDP`

To reduce communication cost, another method is to use PyTorch DDP API [2]. However, this API is temporally designed for Facebook-internal usage, and we must carefully calculate and synchronize the information regarding which parameters should be skipped, making our system unstable and difficult to be debugged. Our design avoids this issue and simplifies the system design. Since `AutoPipe` stores $\mathcal{F}_{\text{frozen}}$ and $\mathcal{F}_{\text{pipe}}$ separately (introduced in Section 3.2.1), we can naturally skip the frozen parameters because `AutoDP` only needs to initialize the data parallel worker with $\mathcal{F}_{\text{pipe}}$.

E. More Details of `AutoCache`

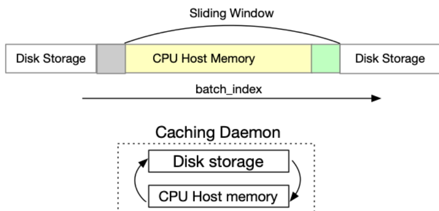

Figure 19. Hierarchical Caching

`AutoCache` supports hierarchical caching. Figure 19 shows our design. We maintain a sliding window to represent the maximum memory that the CPU host memory can hold, then move the window to prefetch the caching that the training requires and delete the caching that is consumed from the CPU host memory. In our implementation, we define the window size as the maximum batch number that the CPU host memory can hold. To avoid frequent memory exchange between disk storage and CPU host memory, we also define the block size that every time we prefetch (as the grey and green blocks are shown in the figure). In general, this hierarchical caching is useful when the training dataset is too large and exceeds the CPU host memory limit. However, we have to point out that this complex caching may not always be the optimal choice in the training system since the caching exchange itself may cost time. To this end, we suggest users of `PipeTransformer` using a relatively larger CPU host memory, which avoids activating the hierarchical caching and obtains faster training.

F. More Experimental Results and Details

F.1. Hyper-Parameters Used in Experiments

Table 3. Hyperparameters used in Experiments

Dataset	Model	Hyperparameters	Comments
SQuAD	BERT	batch size	64
		max sequence length	512
		learning rate	{1e-5, 2e-5, 3e-5, 4e-5, 5e-5}
		epochs	3
		gradient accumulation steps	1
ImageNet	ViT	batch size	400
		image size	224
		learning rate	{0.1, 0.3, 0.01, 0.03}
		weighs decay	0.3
		decay type	cosine
		warmup steps	2
		epochs	10
CIFAR-100	ViT	batch size	320
		image size	224
		learning rate	{0.1, 0.3, 0.01, 0.03}
		weighs decay	0.3
		decay type	cosine
		warmup steps	2
		epochs	10

In Table 3, we follow the same hyper-parameters used in the original ViT and BERT paper. Note that for ViT model, we use image size 224 for fine-tuning training.

F.2. More Details of Speedup Breakdown

Understanding the speed downgrade of `freeze only`. As shown in Figure 9, the `Freeze Only` strategy is about 5% slower than the `No Freeze` baseline. After the performance analysis, we found it is because `Freeze Only` changes memory usage pattern and introduced additional overhead in PyTorch's `CUDACachingAllocator` [3]. More specifically, to reduce the number of expensive CUDA memory allocation operations, PyTorch main-

[2]See the internal API defined by PyTorch DDP: https://github.com/pytorch/pytorch/blob/master/torch/nn/parallel/distributed.py, `_set_params_and_buffers_to_ignore_for_model()`.

[3]To understand the design of this API, please refer to Section 5.3 in the original PyTorch paper (Paszke et al., 2019). The source code is at https://github.com/pytorch/pytorch/blob/master/c10/cuda/CUDACachingAllocator.h

tains a `CUDACachingAllocator` that caches CUDA memory blocks to speed up future reuses. Without freezing, the memory usage pattern in every iteration stays consistent, and hence the cached memory blocks can be perfectly reused. After introducing layer freezing, although it helps to reduce memory footprint, on the other hand, it might also change the memory usage pattern, forcing `CUDACachingAllocator` to split blocks or launch new memory allocations, which slightly slows down the training. In essence, this underlying mechanism of `PyTorch` is not tailored for freeze training. Customizing it for freeze training requires additional engineering efforts.

F.3. Tuning α for ViT on ImageNet

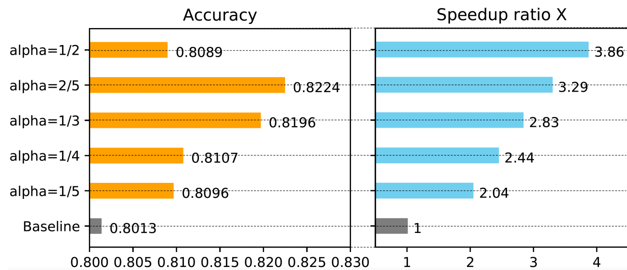

Figure 20. Tuning α for ViT on ImageNet

F.4. The Method That Can Accurately Measure the Communication Cost

Since PyTorch DDP overlaps communication with computation, the time difference between a local training iteration and a distributed training iteration does not faithfully represent the communication delay. Moreover, as DDP also organizes parameters into buckets and launches an `AllReduce` for each bucket, recording the start and finish time of overall communications is also insufficient. To correctly measure DDP communication delay, we combined the DDP communication hook with `CUDAFuture` callback. We developed a communication hook function that records a start `CUDA` event immediately before launching `AllReduce`. Then, in the `CUDAFuture` returned by the `AllReduce` function, we install a callback that records a finish `CUDA` event immediately after the non-blocking `CUDAFuture` completes. The difference between these two `CUDA` events represents the `AllReduce` communication delay of one bucket. We collected the events for all buckets and removed time gaps between buckets if there were any. The remaining duration in that time range accurately represents the overall DDP communication delay.

Table 4. Overheads of pipe transformation (seconds)

Pipeline Transformation	Overall Time Cost	Dissect		
		C	P	D
initialization (length = 8)	18.2	16.6	0.7	0.9
length is compressed from 8 to 4	10.2	8.3	1.3	0.6
length is compressed from 4 to 2	5.5	3.8	2.1	0.7
length is compressed from 2 to 1	9.5	2.3	6.1	1.0

*C - creating CUDA context; P - Pipeline Warmup; D - DDP.

F.5. Overheads of Pipe Transformation

We have verified the time cost of pipeline transformation. The result in Table 4 shows that the overall cost of pipeline transformation is very small (less than 1 minute), compared to the overall training time. Therefore, we do not consider further optimization.