AccTFM: An Effective Intra-Layer Model Parallelization Strategy for Training Large-Scale Transformer-Based Models

Zihao Zeng, Chubo Liu, *Member, IEEE*, Zhuo Tang, *Member, IEEE*, Kenli Li, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—Transformer-based deep neural networks have recently swept the field of natural language processing due to their outstanding performance, and are gradually spreading to more applications such as image/video processing. However, compared with general DNNs, training a sizeable transformer-based model is further time-consuming and memory-hungry. The existing distributed training strategies for general DNNs are not appropriate or can not efficiently handle transformer-based networks. In view of this, we propose an intra-layer model parallelization optimization strategy, AccTFM, which introduces a novel fine-grained pipeline execution and hybrid communication compression strategy to overcome the synchronization bottleneck. Specifically, on one hand, it first decouples the inter-layer computation and communication dependencies, and then searches for the optimal partitioning strategy to maximize the overlap of computation and communication. On the other hand, the hybrid communication compression module consists of token-level top-k sparsification and piecewise quantization methods aiming at minimizing communication traffic. Experimental results show that AccTFM accelerates transformer-based DNNs training by up to 2.08x compared to state-of-the-art distributed training techniques.

Index Terms—Communication hiding, deep learning, intra-layer model parallelization, quantization, Top-*k* sparsification

1 Introduction

Transformer-based deep neural network models (DNNs) [1] [2] have been widely exploited in the area of Natural Language Processing (NLP). They have shown unprecedented performance improvements over previous models based on Recurrent Neural Networks (RNN) or Long Short-Term Memory networks (LSTM) [3] [4] in various NLP tasks. Recently, transformer-based DNNs are spreading across wider fields such as image/video processing [5][6] and achieved better performance. For example, Vision Transformers (ViTs) [5] refreshed the state-of-the-art accuracy on ImageNet. However, the higher performance brought by transformer-based DNNs comes along with the explosion of model size. For example, with the state-of-the-art NLP models evolved from the base transformer model to GPT-3 [7], the number of parameters of the model increased significantly from 65 million to 175 billion, and it is far beyond single GPU memory (e.g., the latest NVIDIA V100 GPU with 32 GB memory). Furthermore, this phenomenon that improves application performance by developing larger-scale models continues, and the necessity to distributedly process transformer-based DNNs rises sharply in this era of faster services.

Existing distributed approaches for general DNNs mainly include Data Parallelization (DP) [8] [9] and Pipeline Parallelization (PP) [10]. In DP, each mini-batch input sample are distributed to multiple computing devices while each device holds a complete copy of the model weight parameters. Many studies have improved the performance of DP by mitigating the impacts of gradient synchronization, such as gradient scheduling [11], [12], [13], [14], [15] and compression [16], [17], [18], [19] techniques. Pipeline Parallelization, sometimes referred to as inter-layer model parallelization, vertically partitions the model into multiple consecutive groups of layers. A batch of training samples is further split into smaller micro-batches, which are then pipelined passes through each partition. Generally, a suitable model partitioning strategy is required to achieve load balancing between devices. To mitigate the pipeline bubble overhead, PipeDream [20] introduced a micro-batches scheduling strategy. In addition, each device stored multiple weight versions to ensure correct gradient computation.

However, the existing DP and PP training strategies can not tightly couple with the characteristics of numerous parameters of transformer-based DNNs, and thus are ineffective for training them. Recently, intra-layer Model Parallelization

- *Zihao Zeng, Chubo Liu, Zhuo Tang, and Kenli Li are with the College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Hunan 410082, China. E-mail: {zengzh, liuchubo, ztang, lkl}@hnu.edu.cn.*
- *Keqin Li is with the College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.*

Manuscript received 23 March 2022; revised 15 June 2022; accepted 23 June 2022. Date of publication 5 July 2022; date of current version 23 August 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB0300300, in part by the Programs of National Natural Science Foundation of China under Grant 62072165, and in part by the Fundamental Research Funds for the Central Universities.
(Corresponding authors: Chubo Liu and Kenli Li.)
Recommended for acceptance by D. Tiwari.
Digital Object Identifier no. 10.1109/TPDS.2022.3187815

TABLE 1
Distributed Training Solutions

Schemes	Parallelism	Parameter Update Mechanism	Main Technique(s)
SparCML [17], gTop-k[18]	DP	Synchronous	Gradient compression
MG-WFBP [11]	DP	Synchronous	Gradient merging
P3 [12], TicTac [13], PACE [15], ByteScheduler [14]	DP	Synchronous	Communication scheduling
Gpipe [10]	PP	Synchronous	Inter-layer pipeline execution
PipeDream [20]	DP & PP	Asynchronous	Worker partitioning & Computation scheduling
Xpipe [22]	PP	Asynchronous	Weight prediction
PipeMare [23]	PP	Asynchronous	Learning rate rescheduling & Discrepancy correction
Mesh-Tensorflow [24]	DP & Intra-layer MP	Synchronous	Tensorflow language extension
Megatron-LM [21]	Intra-layer MP	Synchronous	Intra-layer parameters partitioning strategy
AccTFM	Intra-layer MP	Synchronous	Fine-grained pipeline execution & Hybrid communication compression

(MP) [21] has been performed for accelerating transformer-based DNNs. It horizontally divides each layer of weight parameters. Different computing devices train their portion of parameters in parallel based on the same input sample. Unfortunately, similar to DP, the efficiency of the existing intra-layer MP approaches are also limited by the communication overhead between computing devices. Even worse, in low bandwidth networks such as 1 Gbps Ethernet commonly used in distributed development environments of small IT companies, the communication overhead will dominate the performance of intra-layer MP training. There is still a research gap in solving the communication barrier of intra-layer MP.

In this paper, we focus on speeding up the process of intra-layer MP training for transformer-based models. There are two major challenges for breaking the communication barrier of intra-layer MP. First, since intra-layer MP exerts more strong computation and communication dependencies between layers, it is challenging to get the most of both computing and network bandwidth resources. Specifically, the computation of the following layer can not start until the complete output of the preceding layer is obtained. During All-Reduce communication, aggregating output only from the part of workers will result in incorrect model predictions. Second, since the model convergency is sensitive to the communication content (i.e., intermediate output tensor and loss tensor of each layer) of intra-layer MP, it is challenging to achieve high communication compression ratios without accuracy loss. When employing intra-layer MP for transformer-based DNNs, taking the below 50% of elements of intermediate output tensor for synchronization will result in a notable accuracy loss. In this case, frequently-used top-k sparsification methods can not work due to expensive index transmission.

To this end, we propose AccTFM, an effective training acceleration approach for large-scale transformer-based DNNs, which exploits multi-level intra-layer MP optimization for distributed training in a cluster environment with limited network bandwidth. Specifically, the initial optimization is to decouple the computation and communication dependencies across the DNN layers, allowing for fine-grained pipeline execution between computation and communication operation. Then, a dynamic programming algorithm is developed to determine the optimal partitioning of each layer operation. Further, a hybrid communication compression strategy is explored and combined with fine-grained pipeline execution to enable the more effective intra-layer MP training. In summary, the specific contributions of this paper are the following:

- We propose AccTFM that leverages multi-level optimizaiton to provide high-performance distributed training for large-scale transformer-based DNNs. To the best of our knowledge, we are the first effort trying to break the communication barrier of intra-layer MP.
- We design fine-grained pipeline execution in intra-layer MP and propose a dynamic programming algorithm to determine the partitioning scheme of each layer, which aims to maximize the overlap of computation and communication.
- We develop a token-level top-k communication sparsification method, which eliminates the redundant communication of unimportant tokens according to their attention score.
- Based on the distribution characteristics of the values of the communication data chunk, we explore a piecewise quantization method, which can significantly reduce communication traffic in collaboration with the top-k sparsification method.
- Expensive experiments show that AccTFM can increase the training throughput of the transformer-based DNNs by 2.08x over the previous intra-layer MP methods without affecting the convergence of the model.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 introduces some preliminaries. Section 4 describes our proposed AccTFM in detail. Section 5 evaluates the performance of our method with extensive experiments, and finally, Section 6 concludes the paper.

2 Related Work

There are many efforts aimed at improving the training performance of deep neural network models. We only mention those most relevant to our work and list which scheme they can support in Table 1.

Data Parallelization. In data parallelization technology, each worker holds a complete copy of the model and only is responsible for processing a subset of the mini-batch training samples. The gradient communication process, which is implemented commonly by All-Reduce or Parameter Server (PS) architecture, is required to collect the local gradients computed by each worker.

Some studies explored gradient compression technologies to mitigate the communication overhead. Aji *et al.* [16] first proposed the gradient sparsification approach to accelerate the data parallelization training. It transmits only 1% of the most significant gradients during gradient synchronization and accumulates the dropped gradients to the next iteration. Considering inconsistent indices of sparse gradients from each worker, Renggli *et al.* [17] implemented a scalable communication library based on the All-Gather method. It switches the intermediate result vector to a dense format once its sparsity exceeds a threshold during gradient aggregation. Shi *et al.* [18] proposed a global top-k (gTop-k) gradient sparsification method to reduce communication complexity further. After aggregating the local gradient vector, gTop-k sparsifies the accumulating result once again. These works focus only on the communication operation in the data parallelization training process, which needs to find a trade-off between high compression rates and low accuracy losses.

The alternative approach is communication scheduling. Gradient computation and communication can be pipelined in most deep learning frameworks, e.g., Tensorflow, Pytorch, MXNet, S-Caffe, and Poseidon [25], [26], [27]. Considering that merging some small gradient communication operations into one can reduce communication startup time, MG-WFBP [11] proposed an optimization algorithm to determine which gradients should be merged. P3 [12] implemented overlapping of backward gradient communication and forward computation on MXNet PS architecture by using parameter slicing and priority-based parameter update. TicTac [13] was a similar idea and implemented on Tensorflow PS architecture. ByteScheduler [14] was a generic communication scheduler that supported different distributed architecture (i.e., All-Reduce and PS architecture) and DNN frameworks (i.e., Tensorflow, PyTorch, and MXNet). PACE [15] considered the DAG representation of DNN training and introduced preemptive communication scheduling combined with tensor fusion to achieve better overlapping of communication with computation. Despite these efforts to overcome the communication bottleneck of data parallelization, the storage wall is still not broken. Data parallelization solutions cannot operate when the model size surpasses the memory of a single worker.

Pipeline Parallelization. Pipeline parallelization is commonly referred to as inter-layer model parallelization. It divides the DNN model into sequential segments and assigns each one to a worker. A batch of input samples is also split into smaller micro-batches and continuously fed into the model in a pipeline manner.

Gpipe [10] is the first research on pipeline parallelization. It focuses on synchronous stochastic gradient descent training and utilizes re-materialization to reduce memory requirements. However, Gpipe exposes two performance drawbacks. First, there is a significant bubble overhead (i.e., some workers will be idle) due to backward propagation not starting until all micro-batches have been processed in the forward phase. Second, Gpipe is not equipped with load-balancing inter-layer partitioning solutions, and load-unbalancing between devices will undoubtedly damage training performance. Asynchronous pipeline parallelization technology has received recent interest to fill bubble overhead and improve training throughput. It needs to be carefully designed to control the impact on model convergence. Xpipe [22] and PipeMare [23] introduced weight prediction [28] and learning rate rescheduling algorithms, respectively, to mitigate the accuracy slide, but they did not focus on inter-layer partitioning of the model. PipeDream [20] filled idle workers by scheduling them to process the next batch of samples in advance and implemented load-balancing partitioning between workers. It maintains multiple weight versions to address the parameter inconsistency problem caused by multiple active batches in the pipeline. However, this way requires expensive memory space, limiting the capacity of the existing accelerator for training large-scale DNN models. Our AccTFM is based on intra-layer MP, which has good load-balanced and memory-friendly characteristics. Moreover, it is orthogonal to inter-layer pipeline parallelization and can be effectively combined to achieve higher performance gains [21].

Intra-layer Model Parallelization. The weight tensor and activation tensor (or loss tensor) are partitioned across multiple workers in intra-layer model parallelization technology. Each weighted layer requires a synchronization point to aggregate the results of all workers during the forward or backward propagation.

To avoid memory limits and improve training performance, AlexNet [29] introduced cross-GPU intra-layer MP that partitions convolutional filters between two GPUs. DistBelief [30] was early one of the DNN frameworks that support intra-layer MP. It implements synchronization between workers by using PS architecture. Dryden *et al.* [31] focused on CNN training and proposed a family of algorithms to explore intra-layer MP along the channel dimension. In the DNN computational graph, each tensor is either replicated or partitioned across individual workers. Based on this, Mesh-Tensorflow [24], a language extension of Tensorflow, is designed to achieve general distributed tensor computation that supports tensor division in arbitrary dimensions. Megatron-LM [21] implemented intra-layer MP training on PyTorch for transformer-based models. It specifies how model parameters are partitioned and only requires a few synchronization points during distributed training. Unfortunately, these efforts still suffer from severe communication bottlenecks and even worse in weak network connection environments. Taking a first step forward, our AccTFM breaks through the communication barrier of intra-layer MP with fine-grained pipeline execution and hybrid communication compression optimization.

3 Preliminaries

3.1 Transformer Architecture

The transformer-based DNNs are built by stacking multiple encoder and decoder modules. Specifically, each encoder or

decoder module contains two main components: (a) multi-head attention, and (b) a feed-forward network.

The multi-head attention relies on multiple self-attention, each of which operates on three tensors, queries (Q), keys (K), and values (V). It is computed by

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{1}$$

here, Q, K, and V are computed from the input tensors through three linear mapping layers respectively, and d_k is the dimension of keys. The different self-attention functions are independent of each other, and they represent different linear transforms and information extraction for the input tensors. These self-attention outputs are then concatenated and once again linearly transformed to produce the final values.

The second component of each encoder or decoder module is a feed-forward network. It contains two fully connected layers with a ReLU activation in the first layer that are written as

$$FFN(x) = ReLU(xW_1 + b_1)W_2 + b_2, \tag{2}$$

where W_1 and W_2 are the parameter matrices of two fully connected layers respectively.

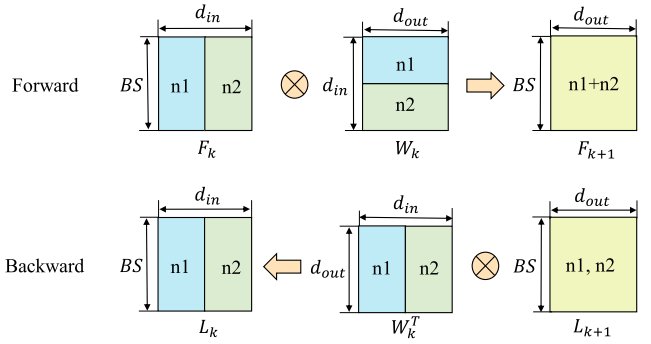
Generally, by using the more number of heads in multi-head attention and the larger inner-layer dimension in the feed-forward network, transformer-based DNNs can be extended to achieve superior accuracy in many NLP tasks.

3.2 Intra-Layer Model Parallelization

In intra-layer model parallelization, the parameter matrices of each layer are partitioned to multiple computing devices, and input tensors or output tensors of each layer are also partitioned accordingly [21][32], as shown in Fig. 1. When the parameter matrices are partitioned along the input dimension, the output tensors of this layer in each computing device are only a partial sum in the DNN forward execution phase. A global communication process is therefore required to obtain the complete output of the layer. When the parameter matrices are partitioned along the output dimension, the global communication process between computing devices takes place in the loss backpropagation phase of this layer. The data transmission unit between devices is the partial sum of the loss tensors of the layer.

On the whole, the parameter matrices of each layer have two partitioning ways, which will result in a total of 2^L intra-layer partitioning schemes for the whole DNN, where L is the number of layers of DNN. In fact, however, an ideal overall partitioning scheme is to divide the parameter matrices of each layer alternately along the input and output dimensions, which can effectively avoid complex inter-layer communication [33]. Generally, the necessary global communication process in each layer is implemented by All-Reduce primitive, both in the forward and backward phases of the DNN training process.

For distributed training of transformer-based DNNs, Megatron-LM is a recently proposed intra-layer MP approach. However, it is not optimized for expensive communication overhead between computing devices.



(a) Divide weight parameters of each layer along the input dimension.

(b) Divide weight parameters of each layer along the output dimension.

Fig. 1. Forward and backward propagation in intra-layer model parallelization.

3.3 Communication Model

The time cost of the global communication process between devices depends primarily on two variables: the number of devices, the message sizes. There are many optimized collective communication algorithms such as Ring-based All-Reduce and Recursive Doubling All-Reduce. They have their advantages and disadvantages when applied to different numbers of devices and message sizes. Without loss of generality, these collective communication algorithms can be modeled uniformly. We assume a bidirectional, direct peer-to-peer communication link between devices, and all devices can send and receive one message simultaneously. Thus, the time cost of All-Reduce operation can be modeled as

$$T_{comm}(m) = \alpha \times c_1 + \beta \times c_2 \times m, \tag{3}$$

where both α, the communication latency (or startup time), and β, the transmission time per byte, are relevant only to the hardware configuration. m represents the message size in bytes. In addition, c_1 and c_2 depend on the specific collective communication algorithm, which usually involves multiple communication iterations. Each communication iteration implies that all devices send or receive one message simultaneously. c_1 represents the number of iterations, and c_2 is the sum of message granularity for all communication iterations. The message granularity for each iteration is between 0 and 1, which is independent of message size.

Overall, the cost model of All-Reduce operations consists of startup time and data transfer time, which is a generally used Latency-Bandwidth communication model.

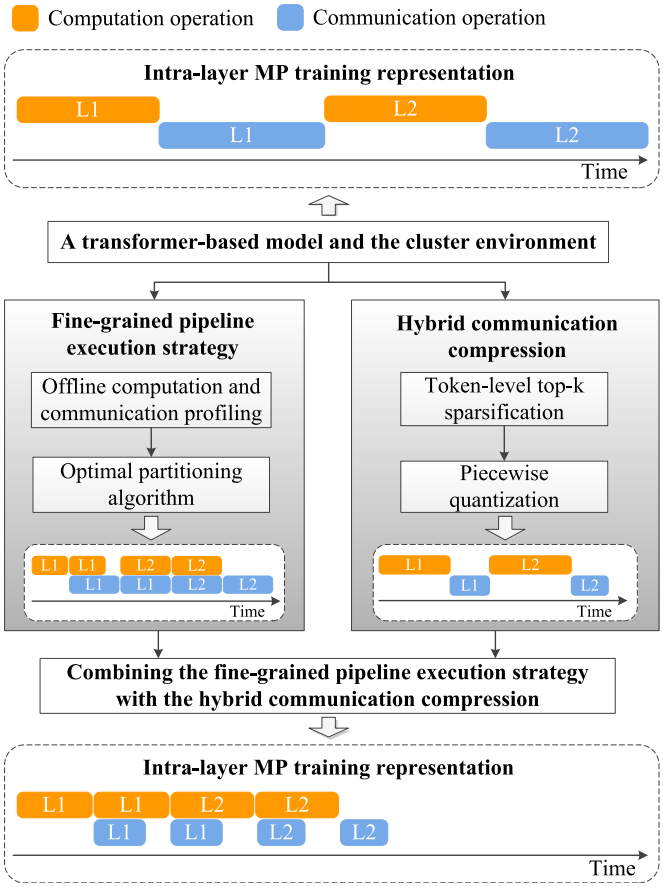


Fig. 2. AccTFM overview.

4 Methodology

4.1 Overview

For training acceleration of large-scale transformer-based DNNs, we propose AccTFM, which exploits multi-level intra-layer MP optimizations for distributed training in a cluster environment with limited network bandwidth. Fig. 2 shows the overview of the proposed AccTFM approach. It mainly involves fine-grained pipeline execution (overlapping computation and communication) and hybrid communication compression optimization for intra-layer MP.

To drive concurrent utilization of computing and network bandwidth resources, we first decouple the computation and communication dependencies between layers by splitting the operation of each layer along the sample dimension. It creates opportunities for computation and communication overlapping. Then, a dynamic programming partitioning algorithm was proposed to determine the partitioning scheme of each layer, which aims to maximize the overlap of computation and communication. It is a global optimization for all computation and communication tasks in intra-layer MP training.

For the local optimization of communication tasks, we develop a novel hybrid communication compression mechanism. Token-level top-k sparsification method removes redundant communication of unimportant tokens based on the attention matrice of each layer. Notably, the computational complexity of the token-level top-k sparsification method is negligible since it has a far smaller search space than the element-level top-k execution. At the same time, a piecewise quantization method is developed to compress the communication overhead further. It adaptively quantifies the elements in different ranges of values.

Algorithm 1. Determine Optimal Partition Size for Each Layer

Input:
1. Batch size, Bs,
2. The number of weighted layers in a DNN model, L,
3. The shape of input data for each layer, $S[1...L]$,
4. The communication latency, α, and the network bandwidth, b,
5. The number of optional partition schemes for each layer, $pn = 3$, and the corresponding partition size, $\boldsymbol{ps} = [1, 2, 4]$.

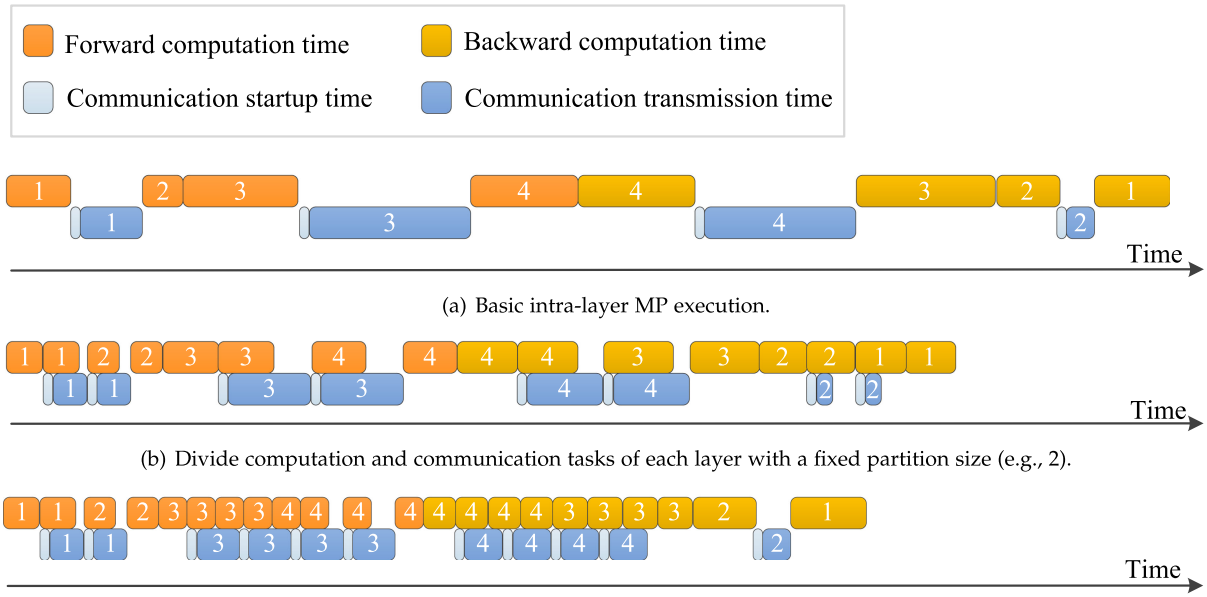
Output:
1.The optimal partition size for each layer, $\boldsymbol{optP}[1...L]$.

Evaluate the computation and communication overhead of each micro-batch (partition) in each layer, $\boldsymbol{T_{cp}}[1...L][1...pn], \boldsymbol{T_{cm}}[1...L][1...pn]$.
Initialize $\boldsymbol{par}[1...L][1...pn], \boldsymbol{optP}[1...L]$
Initialize $\boldsymbol{fE_{cp}}[1...pn], \boldsymbol{fE_{cm}}[1...pn][1...\max(\boldsymbol{ps})]$
Initialize $\boldsymbol{tE_{cp}}[1...pn], \boldsymbol{tE_{cm}}[1...pn][1...\max(\boldsymbol{ps})]$
for $l = 1$ **to** L **do**
  for $i = 1$ **to** pn **do**
    $\boldsymbol{tE_{cp}}[i], \boldsymbol{tE_{cm}}[i] \leftarrow \min_{1 \le j \le pn}\{\textsc{CalParEnd}(\boldsymbol{fE_{cp}}[j], \boldsymbol{fE_{cm}}[j], \boldsymbol{T_{cp}}[l][i], \boldsymbol{T_{cm}}[l][i], \boldsymbol{ps}[i])\}$
    $\boldsymbol{par}[l][i] \leftarrow \arg\min_{1 \le j \le pn}\{\textsc{CalParEnd}(\boldsymbol{fE_{cp}}[j], \boldsymbol{fE_{cm}}[j], \boldsymbol{T_{cp}}[l][i], \boldsymbol{T_{cm}}[l][i], \boldsymbol{ps}[i])\}$
  $\boldsymbol{fE_{cp}} \leftarrow \boldsymbol{tE_{cp}}$
  $\boldsymbol{fE_{cm}} \leftarrow \boldsymbol{tE_{cm}}$
$p \leftarrow \arg\min_{1 \le i \le pn}\{\boldsymbol{fE_{cm}}[i]\}$
$l \leftarrow L$
while $l > 0$ **do**
  $\boldsymbol{optP}[l] \leftarrow \boldsymbol{ps}[p]$
  $p \leftarrow \boldsymbol{par}[l][p]$
  $l \leftarrow l - 1$
return $\boldsymbol{optP}$

function CalParEnd ($preE_{cp}, \boldsymbol{preE_{cm}}, t_{cp}, t_{cm}, K$)
  Initialize $lastE_{cp}, lastE_{cm}, \boldsymbol{E_{cm}}[1...K]$
  $\boldsymbol{preE_{cm}} \leftarrow \textsc{SplitOrMerge}(\boldsymbol{preE_{cm}}, K)$
  $lastE_{cp} \leftarrow preE_{cp}$
  $lastE_{cm} \leftarrow \boldsymbol{preE_{cm}}[K]$
  for $i = 1$ **to** K **do**
    $lastE_{cp} \leftarrow \max(\boldsymbol{preE_{cm}}[i], lastE_{cp}) + t_{cp}$
    $lastE_{cm} \leftarrow \max(lastE_{cp}, lastE_{cm}) + t_{cm}$
    $\boldsymbol{E_{cm}}[i] \leftarrow lastE_{cm}$
  return $lastE_{cp}, \boldsymbol{E_{cm}}$

function SplitOrMerge $\boldsymbol{preE_{cm}}, K$
  Initialize $\boldsymbol{E_{cm}}[1...K]$
  Initialize $r \leftarrow K/len(\boldsymbol{preE_{cm}})$
  for $i = 1$ **to** K **do**
    $s \leftarrow \lceil i/r \rceil$
    $\boldsymbol{E_{cm}}[i] \leftarrow \boldsymbol{preE_{cm}}[s]$
  return $\boldsymbol{E_{cm}}$

Overall, fine-grained pipeline execution and hybrid communication compression component will work together to enable more efficient intra-layer MP training for large-scale transformer-based DNNs.



(a) Basic intra-layer MP execution.

(b) Divide computation and communication tasks of each layer with a fixed partition size (e.g., 2).

(c) Divide computation and communication tasks of each layer with various partition size.

Fig. 3. Three cases of intra-layer MP procedure with different execution approaches.

4.2 Fine-Grained Pipeline Execution in Intra-Layer MP

4.2.1 Problem Analysis

The primary intra-layer MP has strong dependencies between computation and communication. Generally, the computation of the next layer does not start until the complete output of the current layer is obtained. It means that the computing resources of each device are idle while All-Reduce communication operations are performed between devices. To address this problem, we need to decouple the computation and communication dependencies between layers. It is well-known that the operations of different samples are independent of each other, both in the forward and backward running phase of DNNs training. Hence, we first partition the operation of each layer along the batch size dimension, i.e., splitting a batch of training samples into several micro-batches. With the partitioning within a layer, the All-Reduce communication operation of one micro-batch can start asynchronously, and the computing resource is allocated to the next micro-batch. It can be viewed as a fine-grained pipeline parallelization for intra-layer operations, significantly different from the previous pipeline-based inter-layer MP. On the one hand, intra-layer pipelining only have two pipeline stages for all micro-batches, i.e., computation and communication stage in the layer. However, in the pipeline-based inter-layer MP, the number of pipeline stages equals the number of devices. Generally, the more pipeline stages are, the higher are the bubble overhead, i.e., the worse performance of pipeline parallelization. On the other hand, not only the computation and communication within the layer can overlap, but the pipelining execution in adjacent layers can overlap in our partitioning strategy. Hence, there is almost no bubble overhead when using intra-layer pipelining, and all devices simultaneously participate in computation and communication.

Furthermore, the partition size of each layer is an important factor influencing the performance of intra-layer MP training [14][25]. There is still room for improvement by carefully determining the partition size of each layer. Fig. 3 illustrates the execution comparison of intra-layer MP with different partitioning solutions. From a holistic perspective, due to the differences in computation density of each layer, using the same partition size for all layers of the DNN is not optimal. As shown in Fig. 3a, the 3rd layer of the example DNN is a computation-intensive layer. Specifying the partition size with 4 for the 3rd layer in Fig. 3c achieves more overlap of computation and communication than partition size with 2 in Fig. 3b, which leaves fewer resources idle. Therefore, the more computationally intensive layers should with larger partition sizes. Conversely, the less computationally intensive layers should with smaller partition sizes to fully populate the resource utilization of the computing device. From the perspective of each layer, small partition sizes miss some of the potential for hidden communication. Although larger partition sizes seem to overlap computation and communication considerably, they generate severe communication startup time and underutilization of computing resources. Moreover, the completion time of the current layer depends not only on partitioning by itself but also on the previous layer. For example, suppose the previous layer and the current layer take 4 and 2 partitions, respectively. In this case, the computation of the current layer cannot start until the communication operation of the first two partitions is completed in the previous layer, even if the computing resources are idle.

For simplicity, we only consider partition sizes with 1, 2, and 4 for each layer. In this case, there are still a total of 3^L partitioning schemes for a DNN with L layers, and brute force search is obviously not a practical solution.

4.2.2 Optimal Partitioning Strategy

Generally, a micro-batch finishes the computation task of a layer and then communicates with other computing devices to aggregate the complete output of that layer. The

computation task of the next layer will not perform until the communication task is completed. There exist inherent dependencies between computation and communication for each micro-batch in intra-layer MP. Moreover, the computation and communication tasks of different micro-batches, both intra-layer and inter-layer, can be executed concurrently. To maximize the overlap of computation and communication and improve intra-layer MP training performance, we explore a dynamic programming algorithm to determine the partitioning scheme of each layer. Considering intra-layer MP training in the N-node cluster environment with network bandwidth B, we can evaluate the All-Reduce communication cost of each micro-batch in each layer according to the communication model in Section 2. For the computation task of each micro-batch in each layer, we can also evaluate the actual time overhead by running on a specific computing device. In addition, we also assume that the execution time (computation or communication cost) of different micro-batches within the layer is the same. Typically, the evaluation for costs of computation and communication occurs before training.

Let's first consider the completion time of forward propagation when using intra-layer MP for distributed training. Algorithm 1 describes the proposed dynamic programming partitioning algorithm in detail. The algorithm first evaluates the computation and communication overhead of each micro-batch (partition) in each layer. Then (lines 5-10) the layer-wise minimum completion time is calculated. Here, the array $optP$ records the corresponding optimal partitioning scheme of the previous layer when minimizing the completion time for each layer based on the different partition sizes. Specifically, the "CalParEnd " function (lines 19-28) is responsible for deriving the computation and communication end time of each micro-batch in the next layer based on the dependencies of adjacent layers. Its required inputs are the computation end time of the last micro-batch and the communication end time of each micro-batch in the previous layer, i.e., $preE_{cp}$ and $preE_{cm}$, the computation and communication execution time of each micro-batch in the next layer, i.e., t_{cp} and t_{cm}, and the partition size of the next layer, i.e., K. Since the partition sizes of different layers may vary, it is necessary to split and merge micro-batches dynamically between layers. The "SplitOrMerge" function (lines 30-36) obtains the end time of the communication operations that the computation task of each micro-batch depends on in the next layer. Its required inputs are the communication end time of each micro-batch in the previous layer, i.e., $preE_{cm}$, and the partition size of the next layer, i.e., K. Finally, the optimal partitioning scheme for each layer is determined by backtracking (lines 11-17).

The development above can also be applied to determine the optimal partitioning scheme in backward propagation. Although each layer contains two computation operations in backward propagation, i.e., computing loss and gradient, it can be treated as a whole. When considering backward propagation alone, its optimal partitioning scheme can be determined by layer-wise dynamic programming partitioning algorithm in reverse order, which is essentially the same as Algorithm 1. Hence, the overall partitioning scheme is obtained by performing Algorithm 1 twice when using intra-layer MP for distributed training. Noticeably, the search complexity of the proposed partitioning algorithm is $O(L)$ for a DNN with L weighted layers, which is significantly smaller than the $O(3^L)$ complexity brought by brute force search.

4.3 Hybrid Communication Compression for Intra-Layer MP

Our communication compression module aims to clear meaningless consumption of network bandwidth resources during intra-layer MP training. To minimize communication costs, we jointly exploit token-level top-k sparsification and piecewise quantization methods while also considering the convergence and accuracy of intra-layer MP training.

4.3.1 *Token-Level Top-k Sparsification*

Top-k communication sparsification is designed to reduce the number of transmitted values. The element-level (fine-grained) top-k sparsification method plays a significant role in communication optimization of DP. When developing communication compression for intra-layer MP, the naive idea is to apply element-level top-k sparsification directly. However, the inherent drawback hinders its transplant to intra-layer MP. Rather than All-Reduce collective communication operation, element-level sparse communication in DP is usually implemented by the All-Gather operation due to differences in the index of top-k elements between computing devices. However, the communication traffic is increasing double and redouble with each iteration in the All-Gather operation. It is unfriendly for the intra-layer MP due to the poor sparsity of communication data. Thus, we should identify the global top-k index in advance so that sparse communication can still be implemented efficiently by the All-Reduce operation.

For transformer-based DNNs, there is a degree of token-level redundancy in processing NLP tasks. On the one hand, part of the redundancy comes from meaningless tokens such as articles, prepositions, and adverbs. On the other hand, the actual length of each sequence in a mini-batch of training samples is usually different, which results in meaningless computation and communication in batch processing. Therefore, rather than element-level, we reduce the communication overhead by token-level top-k (coarse-grained) sparsification during intra-layer MP training. The key idea of token-level top-k sparsification is that only significant tokens are transmitted in the All-Reduce operation, and meaningless tokens are pruned.

Specifically, we evaluated the importance of tokens based on the attention scores. For each layer of the transformer-based DNN, the attention score of each sequence is a three-dimensional tensor (i.e., the number of attention heads, the length of the input sequence, and the length of the encoded sequence or output sequence). During intra-layer MP training, each computing device stores only the part of attention heads. First, we aggregate the information of attention heads within each computing device to obtain the attention matrice. Then, the array An is generated by summing up each column of the attention matrice. Each element of the array An represents the attention score of the input token relative to the entire sequence instead of a single token. Since array An generated by each computing

device contains only local information, it is necessary to communicate array An by the All-Reduce operation to obtain the global attention score of all tokens. Considering a mini-batch of sequence, the communication traffic of this All-Reduce operation is batch size multiplied by sequence length, which is negligible compared to the output tensor of each layer. After that, all computing devices perform the top-k operation for the array An in parallel to select significant tokens and generate the mask. Finally, the All-Reduce operation is performed again to aggregate the sparse output tensor (i.e., the output values of selected tokens) in the forward propagation. Noticeably, the communication for the index of tokens is omitted because the token-level masks generated by all computing devices are consistent. For the communication of loss tensors in the backward propagation, each layer uses directly the same token-level mask as the forward propagation without recomputing the importance of tokens.

Assume that the dimensions of the intermediate output tensor of the transformer-based DNN are batch size Bs, sequence length l, hidden dimension h, then the search complexity of the token-level top-k sparsification method is $O(Bs \times l \times \log_2 k)$. It is far smaller than $O(Bs \times l \times h \times \log_2 k')$ brought by element-level top-k execution. Here, k represents the number of tokens selected, and k' is the number of elements to be transmitted.

4.3.2 Piecewise Quantization

Quantization is another communication compression method that can combine effectively with the top-k communication sparsification method. The communication traffic will be further reduced when using fewer bits to represent data transmitted. The generalized linear quantization is applied according to

$$step = \frac{\max(|\boldsymbol{R}|)}{2^{bit-1}-1}, \tag{4}$$

and

$$Qval = round\left(\frac{val}{step}\right) \times step. \tag{5}$$

It divides the numerical space of the output tensor or loss tensor $\boldsymbol{R}$ of each layer into multiple equidistant intervals, and each interval is mapped into a boundary value. Here, val is the original value of elements in the tensor $\boldsymbol{R}$, and $Qval$ is the interval's boundary value (quantized value) closest to val. Obviously, the larger the distance between $Qval$ and val, the more serious the accuracy loss for the model. The linear quantization can achieve the slightest mean square error with the original tensor $\boldsymbol{R}$ if its elements follow the uniform distribution approximately. However, the elements in the tensor, both output tensor in the forward propagation and loss tensor in the backward propagation, follow the Gaussian distribution roughly. The closer the interval is to 0, the more is its number of elements, and the less it is on the contrary.

Motivated by this, we exploit a piecewise quantization method, which aims to reduce MSE with the original tensor when the same compression rate is practised as the linear quantization. Instead of applying equidistant intervals, the fundamental idea of piecewise quantization is that the closer the interval is to 0, the smaller is the interval range so that most of the quantized value does not deviate much from the original value. Fig. 4 shows the piecewise quantization scheme. Here, the red dots on the top number axis are the original values, and the red dots on the bottom number axis represent the quantized values. The original values between the dashed lines are mapped to the same value. The segment between two short lines represents a cluster that contains multiple equidistant intervals. Different clusters hold the same size numerical space but with a different number of intervals. Let's first consider the positive interval. Specifically, for the output tensor or error tensor to be communicated in each layer, we divide its numerical space evenly into several clusters. Moreover, each cluster contains multiple equidistant intervals. Assuming the range of interval in the kth cluster is U_k, then it is $2U_k$ in the next cluster (i.e., $U_{k+1} = 2U_k$). For a given data bit width N and tensor $\boldsymbol{R}$, the piecewise quantization is applied according to

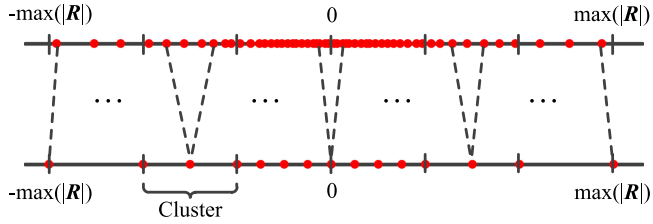


Fig. 4. Piecewise quantization.

$$Qval = C_k + round\left(\frac{val - C_k}{U_k}\right) \times U_k, \tag{6}$$

where

$$C_k = \frac{\max(|\boldsymbol{R}|)}{N} \times k, \tag{7}$$

and

$$U_k = U_0 \times 2^k, \tag{8}$$

and

$$k = floor\left(val \times \frac{N}{\max(|\boldsymbol{R}|)}\right). \tag{9}$$

The number of clusters is also set as N, $k \in \{0, \ldots, N-1\}$ is the index of the cluster that contains val, and C_k represents the left boundary of the kth cluster. The range of intervals U_0 can be derived from the number of clusters and the total number of intervals. Define p denote the number of intervals in the first cluster, and we have

$$p = \frac{\max(|\boldsymbol{R}|)}{N \times U_0}, \tag{10}$$

and

$$\sum_{i=0}^{N-1} \frac{p}{2^i} = 2^N - 1. \tag{11}$$

Thus we can obtain

$$U_0 = \frac{\max(|\boldsymbol{R}|)}{N \times 2^{N-1}}. \tag{12}$$

TABLE 2
Transformer-Based DNN Models for Training

Model	$d_f f$	$heads$	d_k, d_v	Parameters (Billions)	Batch size	Epochs
TF8	4096	8	128	0.24	128	200
TF16	8192	16	128	0.42	128	200
TF32	16384	32	128	0.77	128	200
TF64	32768	64	128	1.48	64	100

TABLE 3
The Accuracy (%) of AccTFM and Baseline

Methods	Models			
	TF8	TF16	TF32	TF64
Baseline	67.27	65.82	66.08	65.57
AccTFM	66.93	65.15	65.67	65.92

Overall, the data bit width that the numerical space of tensor $\boldsymbol{R}$ required is $N+1$ because the negative intervals are symmetric with positive.

5 Experiments

In this section, we evaluate the effectiveness of the AccTFM approach for the distributed training acceleration of transformer-based DNNs. As AccTFM adopts multi-level communication optimization, we present the overall performance improvement of AccTFM at first. Further, we conduct an ablation study to analyze the performance gains provided by fine-grained pipeline execution and hybrid communication compression for intra-layer MP, respectively.

5.1 Experimental Settings

Testbed. We conduct simulations for a 4-node cluster connected with 10-Gbps Ethernet, and each node is equipped with an Nvidia TITAN RTX GPU with 24 GB of memory. The nodes run Ubuntu 16.04 with Linux Kernel 4.4.0 and are installed with the Nvidia GPU driver at version 450.51 and CUDA-10.2. We use Python 3.8.5 and PyTorch 1.8.1 with cuDNN 7.6.5 as the deep learning toolkit. As to collective communication operations, we experiment with the recursive doubling All-Reduce algorithm because it consumes fewer communication iterations than other All-Reduce implementations, and consequently, less communication startup time.

Models and Hyper-Parameters. We focus on the Natural Language Processing tasks where transformer-based DNNs are most successfully applied. We choose four transformer-based DNN models from [24] and name TF8, TF16, TF32, and TF64, respectively. All models are trained on the WMT16 dataset, and the initial learning is set to 0.1. The details of other hyper-parameters are shown in Table 2. As the larger hidden layers and number of attention heads result in more powerful learning ability, fewer training iterations are required to reach convergence for the larger models.

Baseline. We compare AccTFM to Megatron-LM, which is the state-of-the-art intra-layer MP training method for transformer-based DNNs. For the multi-head attention blocks, Megatron-LM partitions the linear layer associated with queries (Q), keys (K), and values (V) in a column-parallel fashion, and the subsequent output linear layer is partitioned in a row-parallel. Similarly, the column-parallel followed by row-parallel fashion is applied for two linear layers in the feed-forward blocks. The column-parallel means weight parameters of the linear layer are divided along the output dimension, and the row-parallel is along the input dimension. For more efficient comparison, we use the same weight parameters partitioning way as Megatron-LM. In practice, although this layer-wise alternately partitioning way requires fewer synchronization points (communication operations), the AccTFM approach retains generality, which is not limited to any weight parameters partitioning way for intra-layer MP.

5.2 Performance Analysis

To evaluate the effectiveness of our proposed scheme AccTFM, we compare the training efficiency with Megatron-LM and the data parallelism solution, and data parallelism is implemented via PyTorch DDP [25]. Compared to baseline, the accuracy variation caused by AccTFM mainly comes from the hybrid communication compression module, which usually needs to consider the trade-off between compression ratio and accuracy loss. We take 50% sparsity (i.e., 50% of tokens will be pruned when performing an All-Reduce communication operation) for TF16 and TF32, while 60% for TF8 and TF64. The elements of transferred tokens are quantified to 4 bits for all models. This setting significantly reduces the communication traffic with negligible accuracy loss. We will analyze the impact of different compression ratios on accuracy and convergence in the following section. The fine-grained pipeline execution improves efficiency without accuracy loss because it maintains the strictly synchronous distributed training mode. Table 3 exhibits the training accuracy comparison between baseline and AccTFM based on the same model initialization and hyper-parameters.

We normalize the performance of AccTFM with respect to that obtained using Megatron-LM. As shown in Fig. 5, we achieve 2.08x, 1.82x, 1.49x, and 1.24x training performance improvements on TF8, TF16, TF32, and TF64 models, respectively. Since the baseline method serially executes all computation and communication operations and transfers

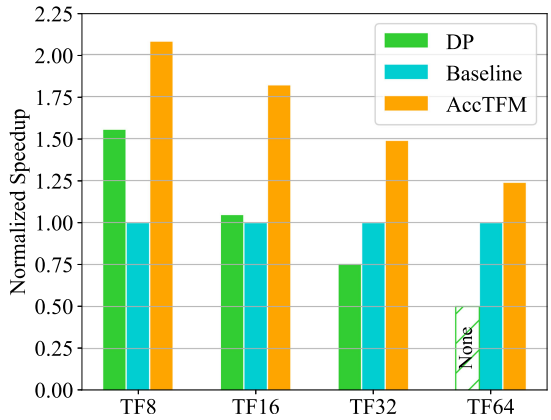


Fig. 5. The performance comparison of AccTFM with baseline and DP.

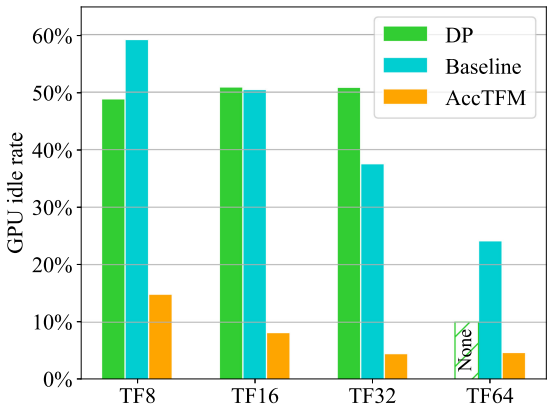


Fig. 6. The GPU idle rate when using baseline, DP, and AccTFM for distributed training.

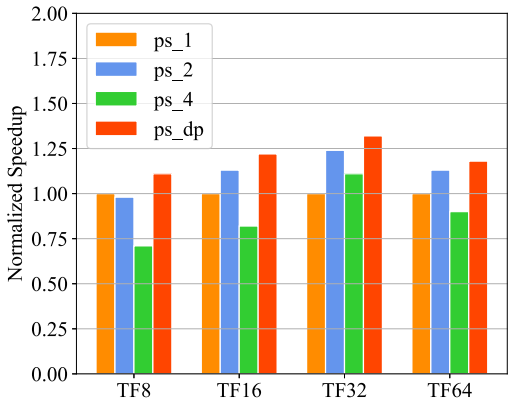


Fig. 7. The performance comparison between different partition sizes on transformer-based models.

the whole tensor during All-Reduce communication, its training performance is susceptible to communication costs. AccTFM overcomes the communication bottleneck by maximizing the overlap between computation and communication, combined with hybrid communication compression. Compared to data parallelism, AccTFM provides 1.34x, 1.74x, and 1.97x speedups on TF8, TF16, and TF32 models, respectively. However, data parallelism can not work for the TF64 model training due to the GPU memory constraint. By tracking the start and end timestamps of each operation in the model, we estimated the idle rate of each GPU, i.e., the waiting time for synchronization divided by the overall training duration. Without any communication optimization technique, i.e., using baseline, the time spent waiting for data synchronization during distributed training is as high as 59.15%, 50.46%, 37.55%, and 24.15% for TF8, TF16, TF32, and TF64 models, respectively, as shown in Fig. 6. The reason is that the training performance is dominated by communication overhead when using the baseline method, and GPUs are frequently left idle to wait for data synchronization. For data parallelism method, the statistics are 48.83%, 50.89%, and 50.85% on TF8, TF16, and TF32 models, respectively. Although Pytorch DDP optimizes the backward pass of training by bucketing gradients and overlapping computation with communication, there are still significant gradient synchronization overheads when processing transformer-based models. In contrast, this percentage is diminished to only 14.87%, 8.19%, 4.54%, and 4.73% when using AccTFM for distributed training. In AccTFM, when a micro-batch is performing an All-Reduce communication process, its neighboring micro-batch concurrently executes computation operation. Furthermore, hybrid communication compression makes communication cost accounts for a reduced fraction of the whole distributed training process. Therefore, AccTFM achieves a higher GPU utilization, i.e., a lower GPU idle rate.

5.3 Ablation Study

To analyze the performance of our AccTFM approach in-depth, we investigate the performance improvements provided by different components. The fine-grained pipeline execution improves intra-layer MP training performance by maximizing the overlap of layer-wise computation and communication time. The hybrid communication compression achieves training acceleration by reducing communication traffic directly.

5.3.1 *Performance Gains From Fine-Grained Pipeline Execution*

We normalize the performance of different partitioning strategies with respect to that obtained using baseline. As shown in Fig. 7, none of the partitioning schemes, which take partition sizes with 1, 2, and 4 for all layers uniformly, is optimal due to the layer-wise difference in computational density. We consider the extra waiting overhead of operation partitioning and merging for each layer and determine layer-wise partition size by dynamic programming, showing the best performance. Compared to baseline, we achieve 1.11x, 1.22x, 1.32x, and 1.18x training performance gains on TF8, TF16, TF32, and TF64 models, respectively. Unexpectedly, while specifying a partition size with 4 for each layer may seem to contribute more overlapping time, it performs sometimes even worse than without operation partitioning. In fact, partitioned too tiny increases overall communication startup time and underutilizes the computing resources of GPU when processing each micro-batch, especially for network layers that are non-computation-intensive or have low communication traffic.

To show the generality of our fine-grained pipeline strategy, we also conduct experiments on commonly used 2D and 3D CNN models, including AlexNet [29], VggNet [34],

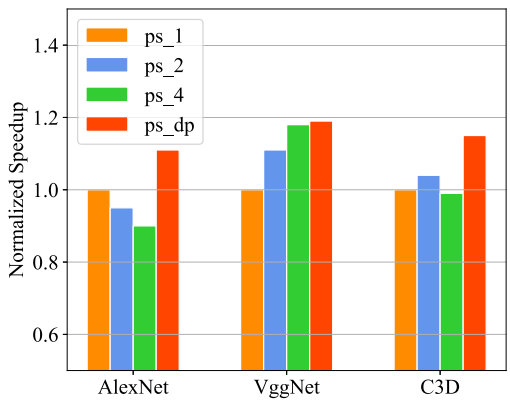
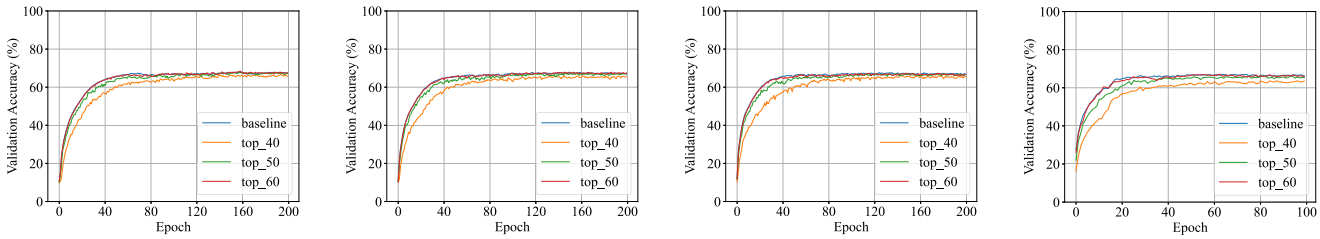
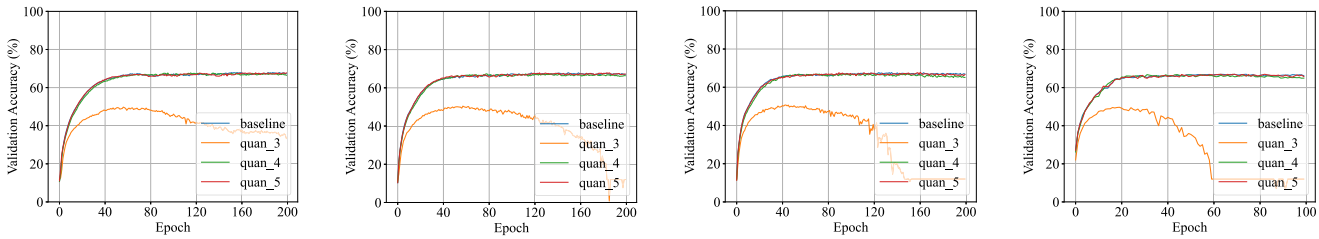


Fig. 8. The performance comparison between different partition sizes on CNN models.



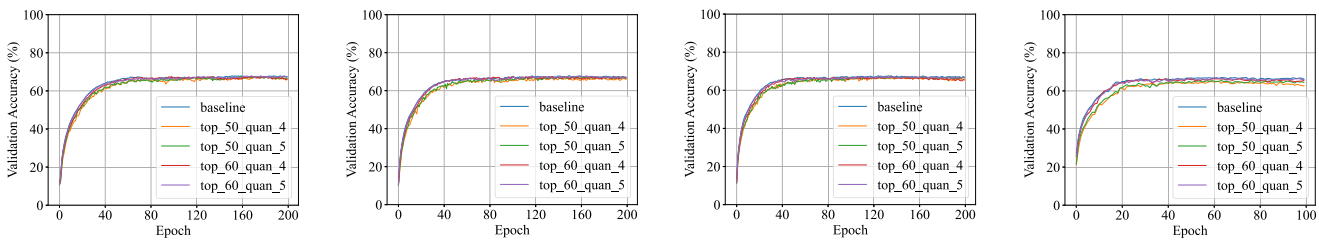
(a) TF8 (b) TF16 (c) TF32 (d) TF64

Fig. 9. The convergence of transformer-based models when using token-level top-k sparsification during All-Reduce communication.



(a) TF8 (b) TF16 (c) TF32 (d) TF64

Fig. 10. The convergence of transformer-based models when using piecewise quantization during All-Reduce communication.



(a) TF8 (b) TF16 (c) TF32 (d) TF64

Fig. 11. The convergence of transformer-based models when using a combination of sparsification and quantization during All-Reduce communication.

and C3D [35]. The ImageNet [36] dataset is used to train AlexNet and VggNet models, and the UCF101 [37] dataset is used for the training of C3D. The batch size is set to 64 for all CNN models. As shown in Fig. 8, the layer-wise partitioning strategy based on dynamic programming yields 1.15x mean performance increases on three CNN models. It also performs the best compared to partitioning schemes with the identical partition size for each layer. Furthermore, the performance specifying a partition size with 4 for all layers occasionally outperforms the partition size with 2 in CNN models, which indicates the execution efficiency of a particular partitioning scheme is connected to the model architecture. Larger partition sizes adapt to more computationally intensive layers and vice versa. Overall, our layer-wise partitioning strategy can cope successfully with the diversity of future DNN model architecture while still providing efficient execution.

5.3.2 *Performance Gains From Hybrid Communication Compression*

We first show the impact of different compression rates, including varying top-k sparsity and number of quantization bits, on accuracy and convergence. On the one hand, as shown in Fig. 9, specifying 50% and 60% sparsity during All-Reduce communication maintains essentially the same convergence speed as the baseline, while taking 40% sparsity impairs the validation accuracy of models slightly. On the other hand, although there is no drop of accuracy with 4-bit quantization, as shown in Fig. 10, further reducing the number of bits for transmitted elements (i.e., 3-bit quantization) has a significant impact on model convergence. Therefore, with a lower bound of 50% tokens sparsity and 4-bit quantization, we experiment with different sparsity and quantization bits combinations. As shown in Fig. 11, there has negligible accuracy loss when providing 16x communication compression rate (i.e., a combination of 50% tokens sparsity and 4-bit quantization) for TF16 and TF32 models, 13.3x compression rate (i.e., combine 60% tokens sparsity with 4-bit quantization) for TF8 and TF64 models. Fig. 12 indicates the training performance gains provided by this communication compression (CommC) configuration. Compared to baseline, it achieves 1.89x, 1.69x, 1.44x, and 1.21x speedup for the TF8, TF16, TF32, and TF64 models, respectively. Among these models, the TF8 model has the highest communication-to-computation ratio during distributed intra-layer MP training, followed by TF16, TF32, and TF64 models in descending order. Since our hybrid compression strategy focuses solely on the communication operation, according to Amdahl's law, a lower communication-to-

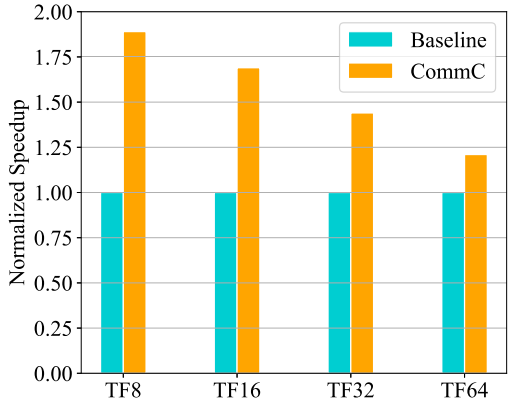


Fig. 12. The performance comparison of communication compression module with baseline.

computation ratio indicates less room for training performance improvement. Consequently, there are gradually reduced speedups for the TF16, TF32, and TF64 models compared to the TF8 model.

6 Conclusion and Future Work

In this research, we propose AccTFM, an efficient intra-layer MP optimization strategy for training large-scale transformer-based models. It integrates fine-grained pipeline execution and hybrid communication compression strategy to overcome the synchronous bottleneck of intra-layer MP. In fine-grained pipeline execution, computation and communication tasks of each layer are separated into smaller operations, and a dynamic programming partitioning algorithm is introduced to facilitate the maximum overlapping between computation and data transmission overhead. The hybrid communication compression module, which consists of token-level top-k sparsification and piecewise quantization methods, is designed to accelerate the synchronization process between workers. We choice four large-scale transformer-based DNNs to evaluate the effectiveness of AccTFM. Compared to existing intra-layer MP technology Megatron-LM, our approach can achieve 2.08x performance gains with negligible accuracy loss. Notably, we also conduct an ablation study to show the respective contributions of each component in AccTFM. The fine-grained pipeline execution with optimal partitioning can achieve speedup up to 1.32x when it serves independently. Under comparable convergence to the original intra-layer MP training, the hybrid communication compression scheme may reach a 16x compression ratio, resulting in a 1.89x training performance increase. Our AccTFM, we believe, is also a highly appealing training strategy for larger DNNs that are currently being developed.

In the future, we will improve AccTFM further on the large-scale distributed platform with heterogeneous devices to provide high-performance training services.

References

[1] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6000–6010.
[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[4] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1724–1734.
[5] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. Int. Conf. Learn. Representations*, 2021, pp. 1–21.
[6] A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić, and C. Schmid, "ViViT: A video vision transformer," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 6816–6826.
[7] T. B. Brown *et al.*, "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1877–1901.
[8] X. Lian, C. Zhang, H. Zhang, C. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5330–5340.
[9] J. Chen, K. Li, K. Bilal, X. K. zhouLi, and P. S. Yu, "A Bi-layered parallel training architecture for large-scale convolutional neural networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 965–976, May 2019.
[10] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.
[11] S. Shi, X. Chu, and B. Li, "MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 1903–1917, Aug. 2021.
[12] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *Proc. Mach. Learn. Syst.*, 2019, pp. 132–145.
[13] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," in *Proc. Mach. Learn. Syst.*, 2019, pp. 418–430.
[14] Y. Peng *et al.*, "A generic communication scheduler for distributed DNN training acceleration," in *Proc. ACM Symp. Operating Syst. Princ.*, 2019, pp. 16–29.
[15] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed DNN training," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 626–635.
[16] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2017, pp. 440–445.
[17] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "SparCML: High-performance sparse communication for machine learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 11:1–11:15.
[18] S. Shi *et al.*, "A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 2238–2247.
[19] H. Wang, S. Guo, Z. Qu, R. Li, and Z. Liu, "Error-compensated sparsification for communication-efficient decentralized training in edge environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 14–25, Jan. 2022.
[20] D. Narayanan *et al.*, "Pipedream: Generalized pipeline parallelism for DNN training," in *Proc. ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.
[21] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.
[22] L. Guan, W. Yin, D. Li, and X. Lu, "XPipe: Efficient pipeline model parallelism for multi-GPU DNN training," 2019, *arXiv:1911.04610*.
[23] B. Yang, J. Zhang, J. Li, C. Re, C. Aberger, and C. De Sa, "PipeMare: Asynchronous pipeline parallel DNN training," in *Proc. Mach. Learn. Syst.*, 2021, pp. 269–296.
[24] N. Shazeer *et al.*, "Mesh-TensorFlow: Deep learning for supercomputers," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 10435–10444.
[25] S. Li *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, 2020.
[26] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing MPI runtimes and caffe for scalable deep learning on modern GPU clusters," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 193–205.
[27] H. Zhang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 181–193.

[28] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Koster, "Pipelined backpropagation at scale: Training large models without batches," in *Proc. Mach. Learn. Syst.*, 2021, pp. 479–501.
[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.
[30] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1232–1240.
[31] N. Dryden, N. Maruyama, T. Moon, T. Benson, M. Snir, and B. V. Essen, "Channel and filter parallelism for large-scale CNN training," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 10:1–10:20.
[32] J. Chen, K. Li, Q. Deng, K. Li, and P. S. Yu, "Distributed deep learning model for intelligent video surveillance systems with edge computing," *IEEE Trans. Ind. Informat.*, to be published, doi: 10.1109/TII.2019.2909473.
[33] Z. Zeng, C. Liu, Z. Tang, W. Chang, and K. Li, "Training acceleration for deep neural networks: A hybrid parallelization strategy," in *Proc. 58th ACM/IEEE Des. Automat. Conf.*, 2021, pp. 1165–1170.
[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[35] D. Tran, L. D. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 4489–4497.
[36] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[37] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," 2012, *arXiv:1212.0402*.

Zihao Zeng received the bachelor's degree in computer science and technology from Jilin Agricultural University, in 2017, and the master's degree in computer technology from Hunan University, China, in 2019. He is currently working toward the PhD degree with Hunan University, China. His research interests include machine learning, parallel computing, and computer architecture. He has published three papers, including the 58th ACM/IEEE Design Automation Conference (DAC 2021), the 57th ACM/IEEE Design Automation Conference (DAC 2020), and the 21st IEEE International Conference on High Performance Computing and Communications (HPCC 2019).

Chubo Liu (Member, IEEE) received the BS and PhD degrees in computer science and technology from Hunan University, China, in 2011 and 2016, respectively. He is currently a professor of computer science and technology with Hunan University. His research interests are mainly in game theory, approximation and randomized algorithms, cloud and edge computing. He has published more than 20 papers in journals and conferences such as the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Industrial Informatics*, *IEEE Internet of Things Journal*, *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, *Theoretical Computer Science*, ICPADS, HPCC, and NPC. He won the Best Paper Award in IFIP NPC 2019 and the IEEE TCSC Early Career Researcher (ECR) Award, in 2019. He is a member of CCF.

Zhuo Tang (Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2008. He is currently a professor with the College of Computer Science and Electronic Engineering, Hunan University. He is also the chief engineer with the National Supercomputing Center, in Changsha. His majors are distributed computing system, cloud computing, and parallel processing for Big Data, including distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He has published almost 50 journal articles and book chapters. He is a member of ACM and CCF.

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar with the University of Illinois, Urbana-Champaign, from 2004 to 2005. He is currently a full professor of computer science and technology with Hunan University, the dean with the College of Information Sciences and Engineering of Hunan University, and the director in the National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, and grid and cloud computing. He has published more than 160 research papers in international conferences and journals such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Parallel and Distributed Computing*, ICPP, ICDCS, etc. He serves on the editorial board of the *IEEE Transactions on Computers*. He is an outstanding member of CCF.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York. He is also a national distinguished professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, Big Data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored more than 840 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds more than 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 5 most influential scientists in parallel and distributed computing based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the *ACM Computing Surveys* and the *CCF Transactions on High Performance Computing*. He has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Services Computing*, and *IEEE Transactions on Sustainable Computing*. He is a Member of Academia Europaea (The Academy of Europe).