MPipeMoE: Memory Efficient MoE for Pre-trained Models with Adaptive Pipeline Parallelism

Zheng Zhang
School of Computer Science
WuHan University
zzhang3031@whu.edu.cn

Donglin Yang
Nvidia Corp.
dongliny@nvidia.com

Yaqi Xia
School of Computer Science
WuHan University
yaqixia@whu.edu.cn

Liang Ding
JD Explore Academy
JD.com Inc.
liangding.liam@gmail.com

Dacheng Tao
JD Explore Academy
JD.com Inc.
dacheng.tao@gmail.com

Xiaobo Zhou
University of Macau
waynexzhou@um.edu.mo

Dazhao Cheng
School of Computer Science
WuHan University
dcheng@whu.edu.cn

Abstract—Recently, Mixture-of-Experts (MoE) has become one of the most popular techniques to scale pre-trained models to extraordinarily large sizes. Dynamic activation of experts allows for conditional computation, increasing the number of parameters of neural networks, which is critical for absorbing the vast amounts of knowledge available in many deep learning areas. However, despite the existing system and algorithm optimizations, there are significant challenges to be tackled when it comes to the inefficiencies of communication and memory consumption.

In this paper, we present the design and implementation of MPipeMoE, a high-performance library that accelerates MoE training with adaptive and memory-efficient pipeline parallelism. Inspired by that the MoE training procedure can be divided into multiple independent sub-stages, we design adaptive pipeline parallelism with an online algorithm to configure the granularity of the pipelining. Further, we analyze the memory footprint breakdown of MoE training and identify that activations and temporary buffers are the primary contributors to the overall memory footprint. Toward memory efficiency, we propose memory reusing strategies to reduce memory requirements by eliminating memory redundancies, and develop an adaptive selection component to determine the optimal strategy that considers both hardware capacities and model characteristics at runtime. We implement MPipeMoE upon PyTorch and evaluate it with common MoE models in a physical cluster consisting of 8 NVIDIA DGX A100 servers. Compared with the state-of-art approach, MPipeMoE achieves up to 2.8× speedup and reduces memory footprint by up to 47% in training large models.

Index Terms—Mixture of Experts, Pipeline Parallelism, Distributed Training, Memory Efficiency

I. INTRODUCTION

Scaling up the model size of neural networks is one of the promising ways to improving model accuracy in a wide range of applications [1]–[6]. For example, in natural language processing (NLP), large pre-trained language models [7]–[10] have been shown effective in many domains such as language understanding [9], sequence generating [11], [12] and cross-lingual downstream transfer [13], [14]. Recently, Mixture-of-Experts (MoE) has been adopted to scale neural networks to an extreme size without introducing a proportional increase in computational cost [15]–[18]. The MoE architecture consists of many sub-models called *experts*. It employs a trainable gating network to intelligently forward the input token to specific experts. The sparse combination of experts makes it practical to save much computation capacity and improve model accuracy compared to dense models with the same computation resources. There are popular MoE-based models in recent years such as Google's Switch Transformer [17] and Meta's BASE Layer [19].

For training a MoE model, different experts are distributed across a large number of GPU servers. The training process requires All-to-All communication primitive operations to dispatch tokens to the desired experts and collect them after processing. This procedure is called expert parallelism [17], which is shown in Figure 1. In a distributed fashion, the main performance bottleneck comes from the communication phase. It is reported in literature [20] that a variant of MoE without All-to-All can achieve a relative improvement of communication cost for more than 90% in extreme cases. Besides, when scaling up model at extra-scale, the limited size of GPU DRAM has been a major challenge for researchers to explore deeper and wider neural networks.

Training a giant MoE model at the trillion scale requires tremendous hardware resources. For example, training a model consisting of 600 billion parameters in GShard [16] takes up to 96 hours on a cluster equipped with 2,048 TPUs. There are system and algorithm optimizations that tackle the intrinsic inefficiency of All-to-All synchronous communication in MoE [16], [20]–[22]. For example, the work [20] proposed a gating dropout algorithm to reduce the traffic of communication. Recently, FasterMoE [22] adopts pipeline parallelism to alleviate the overhead of communication with expert shadowing. It can achieve significant speedup upon the existing systems in training large MoE models. However, the granularity of pipelining is pre-defined and it is fixed throughout the training. In practice, the dynamic nature of communication demands for adaptive pipeline parallelism, because coarse-grained pipelining is sub-optimal in taking advantage of parallelism while very fine-grained pipelining incurs

significant overhead because of frequent kernel launches and GPU under-utilization. Furthermore, the existing approaches ignore memory efficiency in MoE training, which is however key to scaling up the model to extra-scale.

In this paper, we propose to address the inefficiency of communication and memory usage of MoE training in a holistic manner. First, to alleviate the overhead of communication, we analyze the system behaviors of communication and computation for the MoE architecture and design adaptive pipeline parallelism for MoE [23], which partitions a batch of tokens into several micro-batches and overlaps the execution of computation and communication. Different from Faster-MoE [22], we partition tokens in a more effective manner and propose an adaptive configuration algorithm to search for the optimal pipeline granularity.

Furthermore, we examine the memory footprint of MoE training, which mainly comes from three components: i) *model states of experts*, which include parameters, optimizer states and gradients; ii) *activations*, which need to be stored/stashed in the forward pass so that they can be used later in the backward pass; iii) *temporary buffers*, which store the gradients of activations during the backward pass that are discarded as soon as they are used. Among the three components, activations are the primary contributor to the memory footprint when the batch size is increased. As shown in Figure 1, expert parallelism [17] is designed to scale up the model size by distributing experts across devices evenly. Similarly to Zero Redundancy Optimizer [24], [25], it partitions parameters, optimizer states, and gradients of the model across devices, alleviating the memory footprint of model states in MoE. However, the memory footprints of activations and temporary buffers have the potential for further reduction.

We propose to reduce the memory footprint of activations and temporary buffers by sharing the same buffer for different partitions of tensors. Specifically, the memory of tensors T_{DI}, T_M, T_{DO} can be vastly reduced from m to $\frac{m}{n}$, in which m refers to the memory requirement and n is the number of partitions that determines the granularity of pipelining. But a new challenge is introduced, as activations are overwritten when different partitions request the same memory address. To deal with this problem, we resort to re-computation/communication [26] and CPU offloading [27], [28] for recovering activations in the backward pass. When the re-computation is enabled, the cost of computation can be overlapped with that of the communication, and vice versa. In addition, leveraging that modern GPUs support overlapping computations and data transfers over PCIe, we can offload data to the CPU in the forward pass and prefetch the data into GPUs accordingly. Specifically, T_{DI} can be obtained by either communication or CPU offloading while T_M can be obtained by either re-computation or CPU offloading. We establish a performance model to configure the ideal strategy at runtime.

In summary, we make the following contributions.

- We design adaptive pipeline parallelism for MoE by partitioning a batch of tokens into several micro-batches and overlapping the execution of computation and communi-
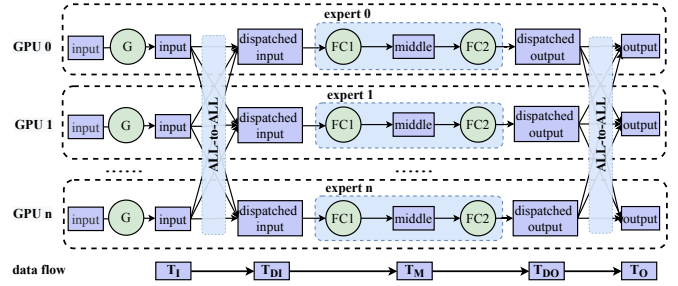

Fig. 1. The illustration of expert parallelism of MoE and its data flow. The green circles represent sub-modules of the MoE layer, and the purple rectangles represent activation tensors of MoE training. For simplicity, We take $T_I, T_{DI}, T_M, T_{DO}, T_O$ at the bottom of the figure as abbreviations of *input, dispatched input, middle, dispatched output, output* tensors, which are in purple color.

cation to improve the utilization of GPUs and network bandwidth. We present an online search algorithm to configure the optimal pipeline granularity.
- We analyze the memory footprint breakdown of MoE and find that activations and temporary buffers are the primary contributors to the memory footprint. With the pipeline parallelism, we propose to reduce the memory footprint of activations and temporary buffers by sharing the same memory buffer for different partitions.
- We tackle the problem that activations are overwritten when different partitions request the same memory space. We leverage re-computation/re-communication and CPU offloading for recovering activations in the backward pass based on performance modeling.
- We implement and integrate the proposed techniques into a library for MoE training, namely MPipeMoE. Experimental results show that MPipeMoE can achieve up to 47% memory footprint reduction and $2.8\times$ speedup over the state-of-the-art system FasterMoE.

The rest of this paper is organized as follows. Section II gives background and motivations for distributed training of MoE models. Sections III and IV describe the system design and implementation of MPipeMoE, respectively. Section V presents the experimental setup and evaluation results. Section VI reviews related works. Section VII concludes the paper.

II. BACKGROUND AND MOTIVATION

A. Mixture of Experts (MoE)

The transformer architecture was introduced to the NLP community due to its superior performance in sequence-to-sequence tasks, such as neural machine translation. A transformer model consists of a few blocks, each of which is formed by the self-attention, cross-attention, and Feed-Forward-Network (FFN) modules. Ever since, transformer-based models become the top performers in various NLP tasks, such as BERT [7], RoBERTa [8], and GPT-3 [10]. Scaling up the model size results in a significant increase in computational cost for both training and inference. For example, it takes 168 days to train a GPT-3 model with 178 billion parameters using 256 NVIDIA A100 GPUs [29].

MoE provides an efficient solution to reducing the cost of training extra-scale models, which incurs only sub-linear compute costs concerning the model size by sparsely activating a subset of the model parameters for given inputs. For example, the cost of training the Switch Transformer [17] with 1.6 trillion parameters are indeed less than the computation budget required to train a dense model with 10 billion parameters. The core component of these MoE models [15], [17], [25] is the MoE layer, which replaces the FFN sub-layer in the original dense transformers.

Expert Parallelism for MoE. To train a giant MoE model, expert parallelism [17] is widely applied to reduce the memory footprint by distributing different experts across devices. As shown in Figure 1, a gating network determines the destination device of each token, which is followed by All-to-All communication. After the dispatch All-to-All, each device executes the local expert, which is an FFN layer consisting of two linear layers and one activation function. Then, the second All-to-All communication is conducted to send the processed tokens back to the devices to which these tokens belong.

Inefficient Synchronous Communication. Each expert requires All-to-All communication to send/receive tokens to/from other devices. The communication phase becomes one of the most time-consuming factors in training MoE models [20], [22]. The All-to-All and expert process procedures are synchronous operations, which are blocked for waiting for the desired data.

B. Memory Footprint of MoE

1) Where did all the memory go: We first analyze the full spectrum of the memory footprint, including model states, activations, and temporary buffers.

Model States. Model states are one of the main contributors to memory consumption during training, which includes parameters, gradients, and optimizer states [24].

Activations. Activations are the intermediate tensors in forward computing, accounting for a significant amount of memory usage [26], especially for the large batch size.

Temporary Buffers. Temporary buffers are used to store intermediate results for a very short period, which are not required for future computation, i.e., the backward pass.

2) Formulation of Memory Footprint of MoE: To analyze the memory footprint of MoE, we demonstrate the detailed dataflow of the communication and expert computation, which is shown in Figure 1. Starting with the input tensor T_I, the All-to-All stage slices and dispatches the tensor across devices, which is referred to as T_{DI}. Every expert takes T_{DI} as the input and outputs tensors T_M and T_{DO} after two linear layers,
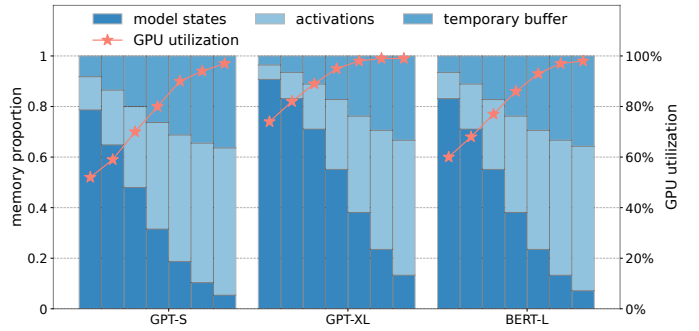

Fig. 2. Breakdown of memory footprint ratio within model states, activations, and temporary buffers. The experiments are conducted on three different MoE layers with various batch sizes of tokens ranging from 256 to 16k with exponential factor 2.

i.e., FFNs, in sequence. The activation function is omitted since in-place operations can be applied here. Finally, tensor T_O is obtained by the collective operations on slices of T_{DO}.

The memory footprint of model states, activation, and temporary buffers are denoted as $\mathcal{M}_{ms}$, $\mathcal{M}_{act}$, and $\mathcal{M}_{buf}$, respectively. We summarize other notations in Table I. The structure of an MoE layer consists of a gating network and an expert. As formulated in Equation (1), $E * M$ equals the number of parameters in the gating network and $2 * H * M$ equals that of an expert. Besides, Adam [30] is chosen as the default optimizer, requiring an additional memory footprint for momentum and variance. As a result, it takes 4 times the memory of parameters for storing model states, including parameters, gradients, momentum, and variance.

The memory footprint of activations is summarized in Equation 2, where the shape of tensors T_I, T_{Di}, T_{Do}, T_O is (B, M) and the shape of tensor T_M is (B, H). For simplicity, we do not consider small tensors such as the routing data of the gating network, because their sizes are one to two orders of magnitude smaller than other activation tensors.

In the backward pass, the GPU device is required to allocate temporary buffers to store the gradients of activations which will be discarded as soon as they are used. When operations are executed in sequence, only two adjacent tensors are required to be cached in the device. The formulation of memory footprint is presented in Equation 3, which is the peak requirement of temporary buffers.

$$\mathcal{M}_{ms} = 4 * (E * M + 2 * H * M) \tag{1}$$

$$\mathcal{M}_{act} = 4 * B * M + B * H \tag{2}$$

$$\mathcal{M}_{buf} = B * M + B * H \tag{3}$$

To visualize the memory consumption of the three discussed data types, we plot the ratio of memory footprint in different MoE settings, which are shown in Figure 2. It can be seen that activations and temporary buffers account for the major portions of the memory footprint with the increasing number of tokens. We also monitor the GPU utilization for the experiment. We observe that a small batch size leads to GPU under-utilization, especially for the MoE layer in GPT-S. As a
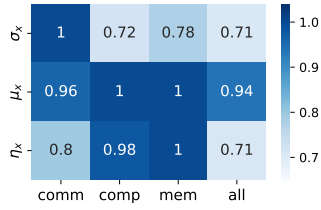
Fig. 3. The interference between different operations. The values in the grid represent the relative speed influenced by operations *GeMM computation, communication* and *memory copy*.

result, it is necessary to increase the batch size for higher GPU utilization. Based on the above observations, we motivate the need to reduce the memory footprint of activation tensors and temporary buffers to train the model with the large batch size.

C. Feasibility of Parallelism

The speed of the communication, computation, and memory copy is denoted as W_{comp}, W_{comm}, and W_{mem}, respectively. Ideally, three types of operations do not affect each other when they are being executed in parallel because they request individual hardware resources in principle. However, in a real environment, there exists resource competition when executing multiple operations in parallel CUDA streams. For example, the communication and memory copy race for memory bandwidth. Performance slowdown incurs if running multiple NVIDIA Collective Communication Library (NCCL) kernels concurrently with computation kernels on the same device. To quantify the degree of slowdown, we define the actual speed of communication, computation, and memory copy as $\mu_x W_{comp}$, $\sigma_x W_{comm}$, and $\eta_x W_{mem}$, in which μ_x, σ_x, and η_x represent their corresponding slowdown factors, respectively. The interference stream, i.e., x, can be any type of streams such as *comm*, *comp*, and *mem*. Specifically, *all* is regarded as the case that all three types of CUDA streams are executed in parallel. The values of μ, σ, and η indicate the feasibility of parallelism. For example, to take the advantage of overlapping between communication and computation, μ_{comm} and σ_{comp} are required to be greater than 0.5, otherwise the execution time of communication or computation would exceed the original end-to-end time, leading to deterioration of the end-to-end performance.

To better understand the interference between operations, we run a micro benchmark in our cluster and measure the actual speed of communication, computation, and memory copy in different situations. Results are demonstrated in Figure 3, from which we can learn that:

- Slowdown is introduced in communication if we execute computation with communication in parallel. However, it is feasible to overlap communication and computation as we can make sure that $\mu_{comm}, \sigma_{comp}$ are larger than 0.5.
- Computation is slightly influenced by other operations, which is negligible in terms of end-to-end performance. As a result, we set $\sigma = 1$ by default in this paper.

- There exists a performance slowdown when communication and memory copy streams are executed in parallel, which is because of bandwidth competition.

The observations and analysis above motivate us to design adaptive pipeline parallelism with memory efficiency.

III. SYSTEM DESIGN

A. Overview

We present the system design of MPipeMoE. First, we design adaptive pipeline parallelism and design an online pipeline granularity configuration algorithm to determine the optimal granularity for accelerating MoE training. Then, we propose the memory reusing component and build a performance model to select the optimal reusing strategy at runtime to reduce the memory footprint.

(a) Pipeline parallelism in GPipe.
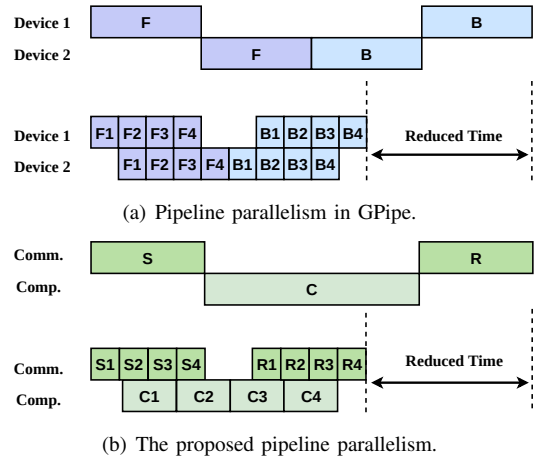
(b) The proposed pipeline parallelism.

Fig. 4. The illustration of GPipe and micro-batch pipeline parallelism in MPipeMoE. (a) F and B represent forward pass and backward, respectively. (b) S, C, and R represent the first All-to-All, computation of experts, and the second All-to-All. The serial number of every block represents the index of the micro-batch partition.

B. Micro-batch Pipelining

As stated previously, the All-to-All operation is the performance bottleneck to scaling out the training of MoE models. Pipeline parallelism, which is firstly introduced in GPipe [31], can reduce the overhead of communication by overlapping the computation and communication. As is shown in Figure 4(a), layers of the model are partitioned into multiple stages, which are mapped to separate devices for performing computation. To deal with the severe under-utilization caused by the sequential dependency of the neural network, GPipe divides the input mini-batch into smaller micro-batches, allowing different accelerators to work on different micro-batches simultaneously. Inspired by GPipe, the micro-batch parallelism can also be applied to the MoE layers to achieve end-to-end speedup. Note that pipeline is not a new idea, enabling adaptive pipelining for MoE requires online scheduling and the insight of computation separation because of the complex dependencies. The unique contribution of this paper lies in tackling these specific challenges in a holistic manner.
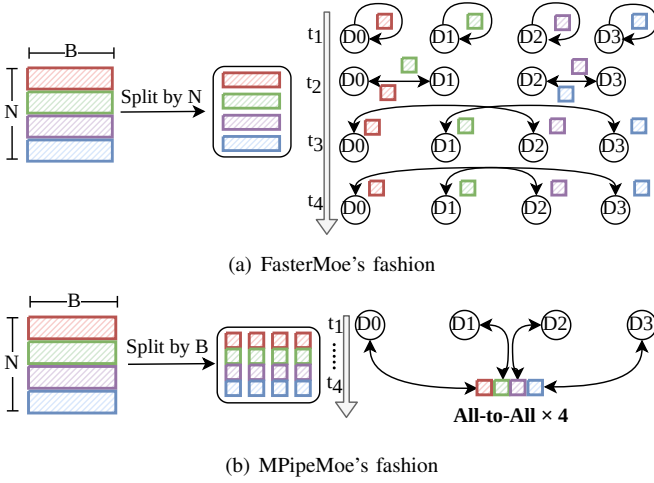
(a) FasterMoe's fashion

(b) MPipeMoe's fashion

Fig. 5. Comparison between FasterMoE and our methods.

Micro-batch pipelining for MoE. As shown at the top of Figure 4(b), only one mini-batch is active for computation or communication in the traditional expert parallelism. In this setup, computation and communication are 'idle' most time. With this in mind, we split a mini-batch of tokens into several micro batches and pipeline their execution one after the other as shown at the bottom of Figure 4(b). Upon completing the first All-to-All for a micro-batch, experts asynchronously execute calculation while simultaneously starting to receive another mini-batch. Then, the second All-to-All operation starts as soon as the calculation is finished. Furthermore, there is no dependency among operations of different partitions. Thus, we schedule S and R to be executed in the alternative as shown in Figure 7(a) for the better locality of memory accesses. The workflow "communication $\rightarrow$ computation $\rightarrow$ communication" is symmetric in the backward pass.

Comparison with FasterMoE in Pipeline Parallelism. FasterMoE [22] also adopts pipeline parallelism to improve the efficiency of MoE training. Different from FasterMoE, we apply a distinguishing method to split the batch data and propose a new optimization solution for communication. As shown in Figure 5, the shape of tensor T_I is (N, B), the first dimension is the number of devices while the second is the batch size of tokens. Each row of the tensor is assigned to the device, which is indicated in a different color in the figure. There exist two methods for splitting T_I into multiple partitions. The first method splits T_I along the first dimension. The All-to-All operation is partitioned into several point-to-point communications among workers for each partition as shown in Figure 5(a). The second method splits T_I along the second dimension as shown in Figure 5(b). The original All-to-All is split into a few fine-grained ones, each for one partition. FasterMoE adopts the former method, which has two disadvantages. First, the All-to-All communication is broken down into multiple point-to-point communications, making it infeasible to take advantage of optimizations offered by NCCL. Second, in the phase of communication, if the network

bandwidth is heterogeneous among workers, the synchronization procedure causes a waste of resources for those workers with higher bandwidth. As a result, MPipeMoE adopts the latter method for better performance.

Algorithm 1: Adaptive Pipeline Granularity Search

Input: the batch size of tokens B
Output: the number of partitions n

1 **global:** $\mathcal{S} = \{\}$;
2 **global:** $cache\_table = \{\}$;
3 **if** B in $cache\_table$ **then**
4 $\quad$ return $cache\_table[B]$;
5 **end**
6 $(\mathcal{R}_n, n) = find(\mathcal{S}, B)$;
7 **if** $n == -1$ **then**
8 $\quad$ $n = searchBestGran(B)$;
9 $\quad$ $(\mathcal{R}_n, n) = find(\mathcal{S}, B)$;
10 $\quad$ **if** $\mathcal{R}_n == \varnothing$ **then**
11 $\quad\quad$ $\mathcal{R}_n = range(B, B)$;
12 $\quad\quad$ $insert(\mathcal{S}, (\mathcal{R}_n, n))$;
13 $\quad$ **else**
14 $\quad\quad$ $\mathcal{R}_n = range(min(B, B_n^{lower}), max(B, B_n^{upper}))$;
15 $\quad$ **end**
16 **end**
17 $cache\_table[B] = n$;
18 return n;

C. Adaptive Pipelining Granularity Configuration

The effectiveness of pipeline parallelism is largely determined by the granularity of pipeline, which is determined by the number of partitions n. A coarse-grained granularity fails to take the benefit of pipeline because S, C, and R cannot be fully overlapped. On the other hand, a very fine-grained granularity could lead to GPU under-utilization. Therefore, it is necessary to configure for the optimal n at runtime to take full advantage of pipeline parallelism.

We consider the training process of MoE models, in which the batch size of tokens is split into n partitions. The micro-batch size equals B/n. It requires running dozens of iterations to search for the optimal configuration of n by calling the method $searchBestGran(B)$. Although the cost can be amortized by epochs, unfortunately, B is dynamic and span a wide range in MoE training [32]. Thus, it is time consuming to search for the optimal n for every value of B.

In order to reduce the searching space, we propose Algorithm 1 based on an intuitive hypothesis: n is monotonically increasing as B increases. As a result, the whole value domain of B can be a set of disjoint ranges $\{\mathcal{R}_n\}(\mathcal{R}_n = range(B_n^{lower}, B_n^{upper}))$, which is a one-to-one mapping to n. We denote the set of pairs $(n, \mathcal{R}_n)$ as $\mathcal{S}$. Given the batch size of tokens B, the optimal n can be looked up by finding a pair $(n, \mathcal{R}_n)$ that satisfies $B \in \mathcal{R}_n$ in $\mathcal{S}$, which is shown in line 6. If not found, $searchBestGran(B)$ is called to search

for the optimal configuration n by trials, i.e., lines 7-8. If n is not in $\mathcal{S}$, a new pair $(n, \mathcal{R}_n = (B, B))$ is inserted into $\mathcal{R}s$, i.e., lines 9-12. Otherwise, we merge B into range $\mathcal{R}_n$, as shown in lines 13-14. To eliminate the overhead of $find(B)$ method, we build a hash table to cache the best strategy for each specific B, which is illustrated in lines 3-5. We implement the set $\mathcal{R}s$ based on the binary-search-tree algorithm. The complexity of $find(B)$ and $insert(n, B)$ are both $O(log(n))$.
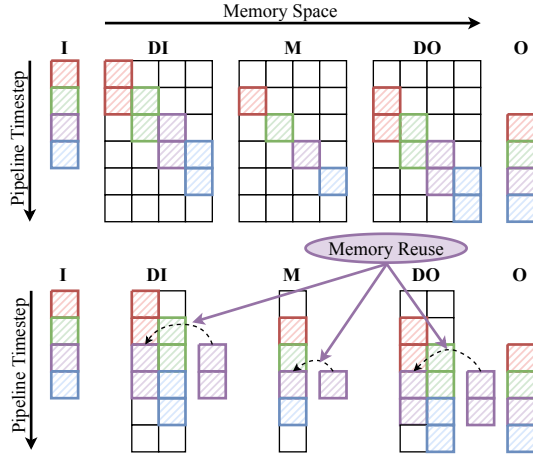
D. Memory Reusing

Fig. 6. The illustration of memory reusing. The top figure demonstrates "memory bubbles" in pipeline parallelism and the bottom one shows the compressed memory by memory reusing.

Tensors T_{DI}, T_M, and T_{DO} are split into n partitions in pipeline parallelism. Different partitions of tensors are activated at different times, resulting in "memory bubbles" as shown at the top of Figure 6. The same operation on different partitions is pipelined into a single stream and executed in sequence. We demonstrate that the input or output tensors of these operations can be shared among partitions to reduce memory redundancy. For example, the *i-th* partition of tensor T_M is activated for computation at time t and the *(i+1)-th* partition is activated at time $t + 1$. Thus we just can allocate one buffer memory to store partitions of T_M in turn. In this way, the required memory is reduced from m to $\frac{m}{n}$, where m is the original memory requirement. Similarly for T_{DI} and T_{DO}, it requires two buffers for communication and computation as shown at the bottom case of Figure 6.

The memory reusing method is applicable for temporary buffers. The peak memory requirement of temporary buffers equals that of activations in pipeline parallelism, thus we can obtain $\mathcal{M}_{buf}^{pipe}$ in Equation 4. With memory reusing, the corresponding reduced memory $\Delta\mathcal{M}_{buf}$ equals $\Delta\mathcal{M}_{act}$, which is presented in Equation 5. Finally, we can obtain the

memory saving ratio ϕ as formulated in Equation 6.

$$\mathcal{M}_{buf}^{pipe} = \mathcal{M}_{act}^{pipe} = 4 * B * M + B * H \quad (4)$$

$$\Delta\mathcal{M}_{buf} = \Delta\mathcal{M}_{act} = B * (2M * \frac{n-2}{n} + H * \frac{n-1}{n}) \quad (5)$$

$$\phi = \frac{\Delta\mathcal{M}_{act} + \Delta\mathcal{M}_{buf}}{\mathcal{M}_{ms} + \mathcal{M}_{act}^{pipe} + \mathcal{M}_{buf}^{pipe}} \quad (6)$$

After eliminating memory redundancy, tensors T_{DI}, T_M are overridden by other partitions. However, these tensors are required for computing the gradients in the backward pass. To restore tensors T_{DI}, T_M, we consider two methods as follows.
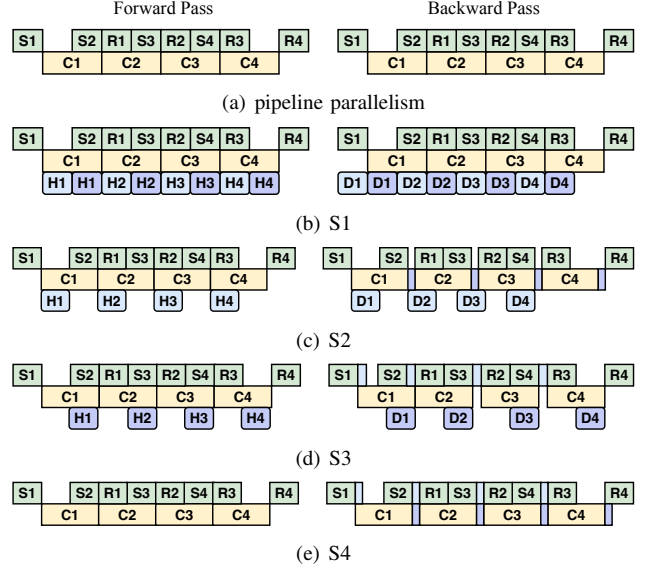

Fig. 7. The timeline of pipeline parallelism and memory reusing. H, D represent the host-to-device and device-to-host memory copies, respectively.

- Data offloading. Leveraging the fact that modern GPUs support overlapping computations and data transfers, we can swap data back to the CPU while computing. In the backward pass, data can be prefetched to the GPU memory in advance.
- Communication and Re-computation. Tensor T_{DI} can be transferred again from tensor T_I. And T_M can be re-computed from T_{DI}. Ideally, the additional cost of re-computation can be mitigated if communication is the bottleneck and vice versa.

As a result, we have four memory reusing strategies, i.e., $S1$, $S2$, $S3$, and $S4$ for MoE training, which are illustrated in Figure 7(b)-7(e). These strategies distinguish in adopting different methods to restore T_{DI} and T_M in the backward pass. Because there is no dependency among operations of different partitions, we schedule S and R in Figure 4(b) to be executed in an alternative manner for the better locality of memory accesses. Compared with the timeline of the pipeline without a memory reusing strategy as shown in Figure 7(a), $S1$, $S2$, and $S3$ require another CUDA stream to perform memory copy operations in parallel with computation and communication. Specifically, device-to-host and host-to-device memory copy

strategy	T_{DI}	T_M	μ	η	$\mathcal{Q}_{fw}, \mathcal{Q}_{bw}$
none	-	-	μ_{comp}	-	[2,2,0],[4,2,0]
S1	offload	offload	μ_{all}	η_{all}	[2,2,5],[4,2,5]
S2	comm.	offload	μ_{all}	η_{all}	[2,2,4],[4,3,4]
S3	offload	recompute	μ_{all}	η_{all}	[2,2,1],[5,2,1]
S4	comm.	recompute	μ_{comp}	-	[2,2,0],[5,3,0]

operations are involved in the forward pass and the backward pass, respectively. In $S2$ and $S4$, additional communication operations are introduced to restore T_{DI} in the backward pass. Additional computation operations are also required to restore T_M in $S3$ and $S4$.

E. Performance Model on Memory Reusing Strategies

In Section II-C, we validate the feasibility of pipeline parallelism and denote the speed of computation, communication, and memory copy as $\sigma_x W_{comp}$, $\mu_x W_{comm}$, and $\eta_x W_{mem}$, in which x refers to the interference stream. For simplicity, we define $v_0 = [v_{0,comp}, v_{0,comm}, v_{0,mem}]$ as the amount of different type of operations in Equations 7 to 9, where H, M are defined in Table I. Specifically, $v_{0,comp}$ and $v_{0,comm}$ are the amount of floating-point operations and All-to-All collective data volumes in MoE, respectively. $v_{0,mem}$ represents the amount of data volumes produced by moving tensor T_{DI} between the device and host. Because $H = 4 * M$ in most MoE models, copying tensor T_M requires four times more data than that of $v_{0,mem}$.

To quantify the workload on three streams, we define $\mathcal{Q} = [q_1, q_2, q_3]$ to represent the actual amount of relative operations. For instance, if not performing any memory reusing strategy, i.e., $\mathcal{Q}_{fw} = [2, 2, 0]$, there exists two GeMM operations and two All-to-All operations in the forward pass. And similarly, we can obtain $\mathcal{Q}_{fw}$ and $\mathcal{Q}_{bw}$ of four memory reusing strategies in Table II.

$$v_{0,comp} = b * H * M \qquad (7)$$
$$v_{0,comm} = b * M \qquad (8)$$
$$v_{0,mem} = b * M \qquad (9)$$

The execution time of a specific stream equals the total amount of operations divided by the processing speed. For instance, the computation time is $\frac{q_1 v_{0,comp}}{\sigma W_{comp}}$. Because different CUDA streams execute different tasks in parallel, the execution time of the end-to-end pipeline is determined by the slowest stream. We formulate the overall execution time $\mathcal{Q}$ in Equation 10, which is determined by $\mathcal{Q}$, μ, and η. Because the floating-point operations per second are stable as stated in Section II-C, α and β are nearly constant.

$$\mathcal{C} = max\left(\frac{q_1 v_{0,comp}}{\sigma W_{comp}}, \frac{q_2 v_{0,comm}}{\mu W_{comm}}, \frac{q_3 v_{0,mem}}{\mu W_{mem}}\right)$$
$$\approx \frac{1}{W_{comp}} max(q_1, q_2\alpha/\mu, q_3\beta/\eta) \qquad (10)$$
$$= \frac{1}{W_{comp}} max(\mathcal{Q} \cdot [1, 1/\mu, 1/\eta] \cdot [1, \alpha, \beta])$$

$$\text{in which} \quad \alpha = \frac{W_{comp}}{W_{comm}}, \qquad \beta = \frac{W_{comp}}{W_{mem}}$$

Table II summarizes the characteristics of four strategies, i.e., $\mu, \eta, \mathcal{Q}_{fw}$, and $\mathcal{Q}_{bw}$. We obtain the cost $\mathcal{C}$ for all strategies based on Equation 10, from which the one with the lowest cost is chosen as the optimal memory reusing strategy. Generally, strategies $S1$ and $S2$ introduce more memory copy operations, which tend to be I/O bound. In contrast, strategies $S3$ and $S4$ tend to be compute-bound.

IV. IMPLEMENTATION

MPipeMoE is an end-to-end MoE training library implemented on top of torch 1.9.0 with CUDA 11.1 [1]. A few key components and functionalities are implemented as follows.

A. Gating network and Experts

The gating network routes tokens to experts based on top-k algorithm. In this paper, we set k to 1. Increasing k is an equivalence of increasing B in the perspective of system performance. We implement a feed-forward network as the default expert, which is applicable for most transformer models.

B. Expert Parallelism

MPipeMoE distributes experts across different GPUs using the expert parallelism while running the remaining parts of the MoE model in data-parallel scheme. NCCL All-to-All collective operator is adopted to dispatch and collect tokens among GPUs.

C. Usability

It is easy for users to benefit from optimizations introduced by MPipeMoE using Python API, including adaptive pipeline parallelism and memory reusing, etc. As shown in the following code snippet, adding parameters such as *pipeline* and *memory_reuse* can bring the end-to-end speedup.

```python
import pmoe
moe_layer = pmoe.MoELayer(d_model=1024,
    d_hidden=4096, top_k=1,
    num_experts=64, pipeline=True,
    memory_reuse=True)
```

---

[1] https://github.com/whuzhangzheng/MPipeMoE

TABLE III
SPECIFICATIONS OF MoE LAYERS

Model Name	d_{model}	d_{hidden}	#experts
MoE-GPT3-S	768	3072	64
MoE-GPT3-XL	2048	8192	64
MoE-BERT-L	1024	4096	64

V. EVALUATION

A. Experimental Setup

1) Physical cluster: Experimental evaluation is performed on a physical cluster consisting of 8 NVIDIA DGX A100 servers. Each node is equipped with 8 NVIDIA A100 SXM 40GB GPUs and 200 Gbps HDR InfiniBand, backed by $96 \times$ 2nd-generation AMD EPYC CPU cores and 1.9 TiB memory. GPUs are connected by the 3-rd generation NVLink and NVSwitch within each machine. GPUs across different machines are connected through a 1,600 Gbps InfiniBand network with adaptive routing.

2) Models and configurations: The significant discrepancy of the MoE layer between different models lies in the size of experts, which is determined by M and H, and the number of tokens that is denoted as B. Here, we aim to validate the efficiency of the proposed methods on different expert sizes as well as batch sizes. We configure the different expert sizes of FFN from BERT [7] and GPT-3 [10] as shown in Table III, in which d_{model} refers to the dimension of token embedding and d_{hidden} refers to the hidden dimension of FFN layer of models, respectively. Note that this paper focuses on the MoE structure and related performance. We create a dummy dataset by generating random tokens as input to different models. For all experiments, we adopt Adam [30] as the optimizer. We evaluate the efficiency of MPipeMoE in terms of the average training time and the peak memory footprint.

B. Methodology

To demonstrate the performance gain and memory efficiency, we compare MPipeMoE against the state-of-art system *FasterMoE* [22], which implements dynamic shadowing and pipeline parallelism in MoE training. We also choose *FastMoE* as another competitor, which implements the primitive expert parallelism without pipeline parallelism.

We implement *MPipeMoE* and its variant *PipeMoE* to demonstrate the advantages of adaptive pipeline parallelism and memory efficiency. *PipeMoE* implements micro-batch size splitting, which also adopts multi CUDA streams to execute computation and communication in parallel. *MPipeMoE* is implemented on top of *PipeMoE*, which adopts adaptive memory reusing strategies to further reduce memory footprint.

C. Overall Speedup

Figure 8 shows the speedup of PipeMoE against FastMoE and FasterMoE in model training. Compared with FasterMoE, PipeMoE achieves an average speedup of $2.26\times$ on various models and batch sizes. Compared with FastMoE, PipeMoE
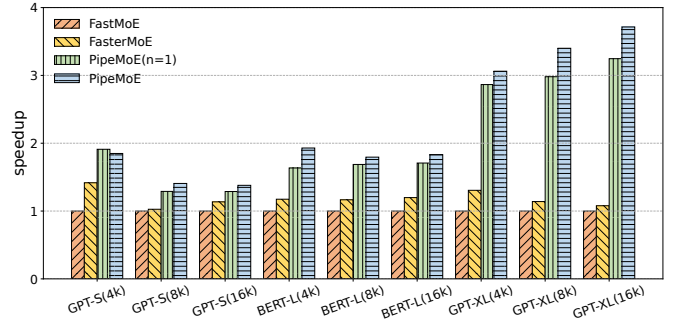

Fig. 8. The speedup of different methods in MoE training with the same model setting and batch size of tokens B. The format of x-axis is "model_name(B)".
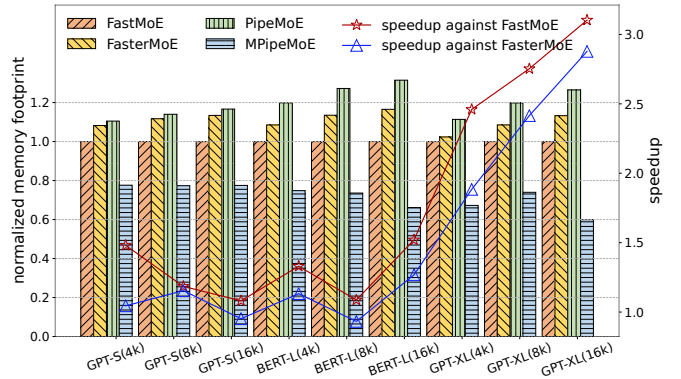

Fig. 9. The memory footprint reduction by MPipeMoE. The bars and the left y-axis show the ratio of memory footprint compared to FastMoE. The polyline and the right y-axis show the speedup of MPipeMoE compared to FastMoE and FasterMoE, respectively.

achieves up to $3.7\times$ speedup. FasterMoE outperforms Fast-MoE because of pipeline parallelism and overlapping of computation and communication. PipeMoE can improve the speedup up to $3.4\times$ against FasterMoE, largely because of the optimization of the pipeline granularity. PipeMoE also takes advantage of Tensor Core of GPUs to accelerate computation.

To validate the effectiveness of pipeline parallelism, we compare PipeMoE against PipeMoE(n=1). In PipeMoE(n=1), the communication and computation are executed in sequence. From the result, we can see that the implementation of pipeline brings benefits to various models with different batch sizes of tokens. The only exception is GPT-S with batch size $B = 4k$, which is not a computation-intensive workload. The result indicates that the pipeline cannot benefit the training process that is not compute-bound because the additional kernel launch overhead leads to lower GPU utilization.

D. Memory Footprint Reduction

Figure 9 presents the overall memory footprint of the approaches, where the left y-axis represents the memory footprint normalized to that of FastMoE. The result shows that MPipeMoE reduces the memory footprint by an average of 23% and up to 40% compared to FastMoE while still can achieve $3.1\times$ speedup in terms of training time. FasterMoE
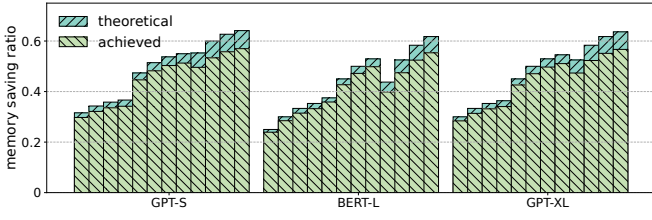
Fig. 10. The MPipeMoE achieved memory reduction ratios compared to their theoretical results on three model settings with the varying number of partitions n (2,4,8) and batch sizes (ranging from 4k to 32k).
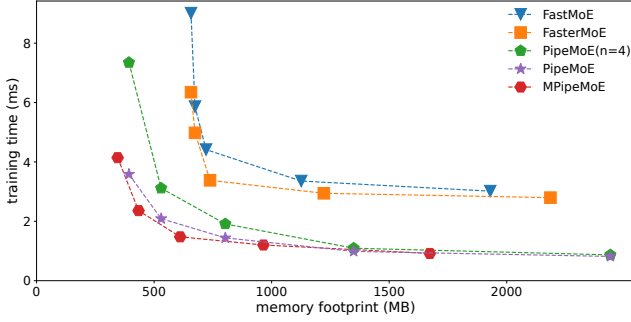

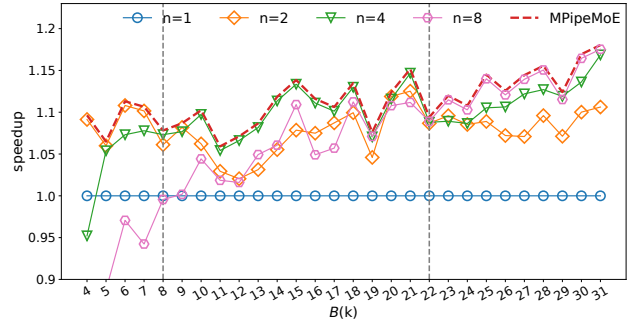Fig. 11. Overall performance breakdown of MPipeMoE on GPT-XL model.

Fig. 12. The effects of pipeline parallelism on various pipeline granularity. The dashed line represents the adaptive granularity selected by the configuration algorithm. The x-axis represents various B values.
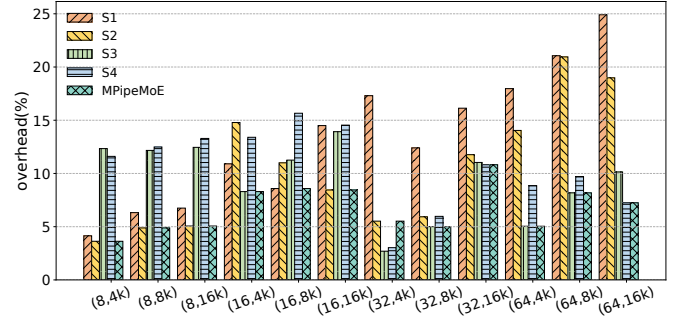

Fig. 13. The overhead of memory reusing strategies and the effectiveness of the strategy selection method in MPipeMoE. The ticks of the x-axis represent different numbers of GPUs N and the batch size of tokens B in format (N, B).

requires more memory than FastMoE because of the dynamic shadowing and smart scheduling. As a result, MPipeMoE achieves an average memory reduction of 27% and up to 47% compared with FasterMoE. Meanwhile, MpipeMoE achieves a speedup up to $2.8\times$ in terms of the training time.

In Section III-D, Equation 6 provides the theoretical bound of memory saving of MPipeMoE. To demonstrate the effectiveness of the analysis, we report the actually achieved memory saving ratio with the bound, which is depicted in Figure 10. We conduct experiments on three models. We configure the number of partitions n and the batch size of tokens B to different values to validate a wide range of cases. MPipeMoE achieves about 95% of the theoretical bound. Note that tensors with small sizes such as routing data produced by gating networks are not considered.

E. Performance Breakdown

To better understand the performance breakdown of MPipeMoE, we reveal the performance of the different methods in *memory-time* coordinates, in which the x-axis represents the memory footprint and the y-axis represents the training time. As shown in Figure 11, the one closer to the origin point illustrates better overall performance. MPipeMoE significantly outperforms FasterMoE and FastMoE. PipeMoE(n=4) reduces the training time because of a higher GPU throughput. PipeMoE outperforms PipeMoE(n=4) by configuring the optimal pipeline granularity at runtime. MPipeMoE achieves best memory efficiency by reusing memory partitions. The higher GPU utilization makes it possible to increase the batch size with the limited device memory space.

F. Effectiveness of Granularity Configurations

We illustrate the effectiveness of the adaptive pipeline granularity configuration of MPipeMoE, which is based on a hypothesis that n is monotonically increasing as B increases. We compare the performance due to different pipeline granularity with various batch sizes of tokens on model GPT-XL. Figure 12 shows that when the batch size is smaller than 8k, $n = 2$ is the best option. When the batch size is increased to 8k-22k, $n = 4$ ensures the best performance. $n = 8$ is the optimal configuration if the batch size is larger than 22k. MPipeMoE, which is denoted as a dashed line, performs the best in all situations. The results validate its effectiveness.

G. Overhead of Memory Reusing

In terms of speedup, MpipeMoE is indeed second to PipeMoE because MpipeMoE achieves memory efficiency at the same time, which however incurs non-trivial overhead. MpipeMoE features four memory reusing strategies, i.e., $S1$, $S2$, $S3$, and $S4$ defined in Table II, which resort to recomputation/communication and CPU offloading to restore activation tensors in the backward pass. For overhead analysis of the strategies, we conduct experiments with different numbers of GPUs N and various batch sizes of tokens B. Figure 13 presents the results, from which we can observe that:

- $S1$ and $S2$ perform better when N is small, e.g., 8, but worse with a larger N, e.g., 64. $S1$ and $S2$ introduce additional memory copy operations while $S2$ introduces additional communication operations. With the increasing number of workers, the cost of communication also increases, which results in the worse performance for $S2$ due to the competition on the memory bandwidth between memory copy and communication.
- Both $S3$ and $S4$ introduce additional computational costs, which perform worse if the workload is computation-bound, i.e., $N = 8$.
- $S4$ performs better than $S2$ if N equals 32 or 64, in which communication is the bottleneck because memory copy over PCIe in $S2$ slows down communication operations.
- There is not much performance variation with the varying batch sizes, indicating that the batch size is not sensitive to the configuration of strategy.

Based on these observations, we can conclude that there does not exist a single memory reusing strategy which can ensure the best performance under all situations. MPipeMoE builds a performance model based on Equation 10 to decide the optimal strategy considering both the hardware configurations and runtime characteristics.

VI. Related Work

Mixture-of-Experts (MoE). Several techniques have been proposed to improve the training efficiency of MoE models. Gating Dropout [20] allows tokens to ignore the gating network and keeps the input at the local machines, reducing the cross-machine communication. DeepSpeed MoE [21] proposes the hierarchical All-to-All and implements custom CUDA kernels to scale expert parallelism out to many devices as the latency increases linearly with the increase in devices. FasterMoE [22] designs a congestion-avoiding expert selection strategy that relieves network congestion to achieve lower training latency. Z-code multilingual Multitask MoE model [25] proposes the Zero Redundancy Optimizer to reduce memory footprint.

Data, Model, Pipeline, and Expert Parallelism. Parallelization is a key strategy for training large models at scale. For a model that fits in the device memory for training, data parallelism (DP) is used to scale training out to multiple devices. In DP, model parameters are replicated on each device. At each step, mini-batch data is divided evenly across all the data parallel processes, such that each process executes the forward pass and backward pass on a different subset of data samples, and uses averaged gradients across processes to update the model locally. To support training giant models, model parallelism (MP) [33], [34] and pipeline parallelism (PP) [33], Pipedream [35] splits the model among processes in either vertical or horizontal ways. Expert parallelism [17] is another form of model parallelism targeting expert parameters of MoE models. In expert parallelism, different experts are placed on different devices and executed in parallel. When experts reside on different GPU devices, explicit communication using the All-to-All primitive is required.

VII. Conclusion

MoE is a promising technology to improving model quality by scaling the neural network to an extra-scale. In this paper, we consider high performance and memory efficiency of MoE model training in a holistic manner. Toward this end, we design adaptive pipeline parallelism with online granularity configuration. Second, we analyze the memory footprint breakdown of MoE training and propose efficient memory reusing strategies to reduce memory requirements by eliminating memory redundancies. What is more, we develop an adaptive selection component to decide whether to offload or recompute the required tensors, which considers both the hardware capacities and model characteristics at runtime. We implement and integrate these features into MPipeMoE library and perform extensive evaluations. The results show that MPipeMoE achieves $2.8\times$ speedup and reduces memory footprint by up to 47% compared to FasterMoE.

References

[1] S. Arora, N. Cohen, and E. Hazan, "On the optimization of deep networks: Implicit acceleration by overparameterization," in *Proc. of ICML*, 2018.

[2] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. Van Der Maaten, "Exploring the limits of weakly supervised pretraining," in *Proc. of ECCV*, 2018, pp. 181–196.

[3] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning," in *Proc. of USENIX OSDI*, 2020, pp. 533–548.

[4] X. Jia, L. Jiang, A. Wang, W. Xiao, Z. Shi, J. Zhang, X. Li, L. Chen, Y. Li, Z. Zheng *et al.*, "Whale: Efficient giant model training over heterogeneous gpus," in *USENIX Annual Technical Conference*, 2022, pp. 673–688.

[5] S. Singh and A. Bhatele, "Axonn: An asynchronous, message-driven parallel framework for extreme-scale deep learning," in *Proc. of IEEE IPDPS*, 2022, pp. 606–616.

[6] S. Wang, O. J. Gonzalez, X. Zhou, T. Williams, B. D. Friedman, M. Havemann, and T. Woo, "An efficient and non-intrusive gpu scheduling framework for deep learning training systems," in *Proc. IEEE/ACM SC*, 2020.

[7] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, 2019, pp. 4171–4186.

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

[9] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[11] Z. Zhang, L. Ding, D. Cheng, X. Liu, M. Zhang, and D. Tao, "Bliss: Robust sequence-to-sequence learning via self-supervised input representation," *arXiv preprint arXiv:2204.07837*, 2022.

[12] Q. Zhong, L. Ding, J. Liu, B. Du, and D. Tao, "E2s2: Encoding-enhanced sequence-to-sequence pretraining for language understanding and generation," *arXiv preprint arXiv:2205.14912*, 2022.

[13] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, "Unsupervised cross-lingual representation learning at scale," in *Proc. of ACL*, 2020.

[14] Q. Zhong, L. Ding, Y. Zhan, Y. Qiao, Y. Wen, L. Shen, J. Liu, B. Yu, B. Du, Y. Chen *et al.*, "Toward efficient language model pretraining and downstream adaptation via self-evolution: A case study on superglue," *arXiv preprint arXiv:2212.01853*, 2022.

[15] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[16] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[17] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," 2021.

[18] S. He, L. Ding, D. Dong, B. Liu, F. Yu, and D. Tao, "Cherry hypothesis: Identifying the cherry on the cake for dynamic networks," *arXiv preprint arXiv:2211.05528*, 2022.

[19] M. Lewis, S. Bhosale, T. Dettmers, N. Goyal, and L. Zettlemoyer, "Base layers: Simplifying training of large, sparse models," in *Proc. of ICML*. PMLR, 2021, pp. 6265–6274.

[20] R. Liu, Y. J. Kim, A. Muzio, and H. Hassan, "Gating dropout: Communication-efficient regularization for sparsely activated transformers," in *Proc. of ICML*. PMLR, 2022, pp. 13 782–13 792.

[21] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation AI scale," in *Proc. of ICML*, vol. 162, 2022, pp. 18 332–18 346.

[22] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li, "Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models," in *Proc. of ACM PPoPP*, 2022, pp. 120–134.

[23] S. Wang, A. Pi, and X. Zhou, "Scalable distributed dl training: Batching communication and computation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[24] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proc. of IEEE/ACM SC*, 2020, pp. 1–16.

[25] Y. J. Kim, A. A. Awan, A. Muzio, A. F. C. Salinas, L. Lu, A. Hendy, S. Rajbhandari, Y. He, and H. H. Awadalla, "Scalable and efficient moe training for multitask multilingual models," *arXiv preprint arXiv:2109.10465*, 2021.

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

[27] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. of IEEE MICRO*, 2016, pp. 1–13.

[28] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *Proc. of ACM/IEEE ISCA*, 2020, pp. 926–939.

[29] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proc. IEEE/ACM SC*, 2021, pp. 1–15.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. of ICLR*, 2015.

[31] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[32] C. Hwang, W. Cui, Y. Xiong, Z. Yang, Z. Liu, H. Hu, Z. Wang, R. Salas, J. Jose, P. Ram *et al.*, "Tutel: Adaptive mixture-of-experts at scale," *arXiv preprint arXiv:2206.03382*, 2022.

[33] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," *Advances in neural information processing systems*, vol. 31, 2018.

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

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