# MEPipe: Democratizing LLM Training with Memory-Efficient Slice-Level Pipeline Scheduling on Cost-Effective Accelerators


Zhenbo Sun
Tsinghua University
Beijing, China
sunzb20@mails.tsinghua.edu.cn

Shengqi Chen
Tsinghua University
Beijing, China
csq20@mails.tsinghua.edu.cn

Yuanwei Wang
Tsinghua University
Beijing, China
wangyw20@mails.tsinghua.edu.cn

Jian Sha
Tsinghua University
Beijing, China
shaj24@mails.tsinghua.edu.cn

Guanyu Feng
Zhipu AI
Beijing, China
guanyu.feng@zhipuai.cn

Wenguang Chen
Tsinghua University
Beijing, China
cwg@tsinghua.edu.cn



## Abstract

The training of large language models (LLMs) typically needs costly GPUs, such as NVIDIA A100 or H100. They possess substantial high-bandwidth on-chip memory and rapid interconnects like NVLinks. The exorbitant expenses associated with LLM training pose not just an economic challenge but also a societal one, as it restricts the ability to train LLMs from scratch to a selected few organizations.

There is a significant interest in democratizing access to LLM training. This paper explores a potential solution by employing innovative parallel strategies on more affordable accelerators. Budget-friendly options like NVIDIA RTX 4090, while considerably less expensive and comparable in computational power to A100, are hindered by their limited memory capacity and reduced interconnect bandwidth, making the effective training of LLMs challenging.

Conventional parallel strategies often result in high communication costs or excessive memory usage. Our paper introduces MEPipe, a novel approach that includes a slice-level scheduling method for sequence pipeline parallelism. This method minimizes memory consumption without incurring additional communication overhead. Besides, MEPipe utilizes fine-grained weight gradient computation to reduce idle time and mitigate imbalanced computation among slices.

MEPipe has demonstrated up to 1.68× speedup (1.35× on average) on clusters equipped with 64 NVIDIA 4090 GPUs when training Llama models of varying sizes. 35% Model FLOPS Utilization (MFU) is achieved in training Llama 13B model, being 2.5× more cost-effective than A100 clusters.





*EuroSys '25, Rotterdam, Netherlands*





ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717469



***CCS Concepts:*** • **Computing methodologies** → *Neural networks*; **Parallel algorithms**.

***Keywords:*** distributed deep learning; large language model

**ACM Reference Format:**
Zhenbo Sun, Shengqi Chen, Yuanwei Wang, Jian Sha, Guanyu Feng, and Wenguang Chen. 2025. MEPipe: Democratizing LLM Training with Memory-Efficient Slice-Level Pipeline Scheduling on Cost-Effective Accelerators. In *Twentieth European Conference on Computer Systems (EuroSys'25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3717469


## 1 Introduction

Recently, large language models [3, 25, 34] have exhibited powerful abilities in various tasks. The enhancement of their ability is primarily attributed to the exponential growth in both data volume and model size [18]. However, the computation and storage requirements for training LLMs exceed the capacity of single accelerators, necessitating efficient parallel strategies for distributed training on large clusters.

Various parallel strategies have been proposed to improve the training efficiency of large models, including data parallelism (DP) [12], tensor parallelism (TP) [32], pipeline parallelism (PP) [7], and context parallelism (CP) [10, 21]. The characteristics of these parallel strategies will be discussed in Section 2. With the growth of model size and the number of accelerators, the aforementioned parallel strategies should be combined to achieve optimal training efficiency.

In general, accelerators used for training LLMs such as NVIDIA A100 GPUs are expensive because they should meet three essential requirements: powerful computation capacity, high-bandwidth interconnection, and substantial memory. In contrast, some inexpensive accelerators, such as the NVIDIA RTX 4090 GPUs, exhibit comparable computational power but are constrained with limited memory capacity and low interconnection bandwidth.

As of October 2024, the typical price for a server equipped with 8 NVIDIA A100 GPUs and NVLink interconnection is approximately 5 times compared to one with 8 NVIDIA RTX 4090 GPUs. The cost of training LLMs will be significantly reduced if we can train LLMs on RTX 4090 GPUs and achieve good performance. However, this approach presents huge challenges due to the limited memory capacity and interconnection bandwidth. These limitations call for better parallel strategies to minimize memory utilization and communication overhead simultaneously.

The memory requirements for training LLMs are substantial, including parameters, gradients, optimizer states, and activations. Among these components, activations consume a huge proportion of the total memory [16, 33]. To enable the training of LLMs on memory-limited accelerators like RTX 4090, activations should be partitioned by slicing models or individual samples.

Tensor parallelism partitions the parameters of each layer and context parallelism partitions single samples. However, both methods need communication of intermediate results for each layer in the forward and backward passes. This incurs substantial memory consumption and renders them unsuitable for accelerators with low-bandwidth interconnection like RTX 4090.

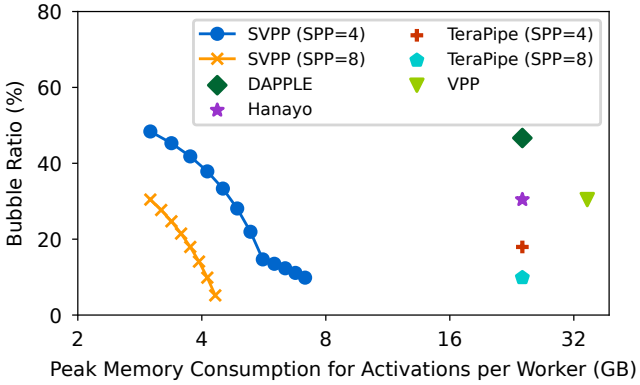


**Figure 1.** Bubble ratio and peak activation memory consumption of SOTA works on Llama 13B with context length as 4096, pipeline parallel size as 8, virtual pipeline size as 2, micro-batch size as 1, and number of micro-batches as 8.

Pipeline parallelism partitions the LLMs into sequential subgraphs and allocates each subgraph to different workers. To reduce the bubble ratio, traditional scheduling methods such as DAPPLE [7] require the first worker to compute several forward passes at the beginning of each iteration, leading to huge memory consumption for the activations.

Recent studies like Virtual Pipeline Parallelism (VPP) [24] and Hanayo [22] mainly focus on reducing the bubble ratio by partitioning the computation graph into finer *chunks* and assigning multiple chunks to individual workers. Although the activation memory of each forward pass is reduced, the first worker needs to compute more forward passes at the beginning of each iteration, failing to reduce the activation memory consumption.

Sequence pipeline parallelism (SPP), proposed by TeraPipe [20], reduces the bubble ratio by partitioning samples into finer slices. However, TeraPipe's scheduling method will process the forward passes of all samples before the first backward pass, leading to substantial memory consumption as workers need to preserve the intermediate activations of all samples. Figure 1 presents the bubble ratio and peak memory consumption of activations for the state-of-the-art approaches on Llama 13B model [34].

We propose *Sequence Virtual Pipeline Parallelism (SVPP)*, a slice-level scheduling method that optimizes the activation memory consumption by interleaving the forward and backward passes at the granularity of slices and advancing the first backward pass of each iteration. With SVPP, the maximum number of slices whose activations workers need to preserve is comparable to the maximum number of micro-batches they should save in other approaches. Given that the activations of each slice are substantially smaller than that of each micro-batch, the peak memory consumption for activations is thus reduced. Moreover, SVPP provides multiple variants of pipeline scheduling methods, each offering distinct memory consumption and bubble ratio.

As shown in Figure 1, when partitioning each sample into 4 and 8 slices, the reduction in peak memory consumption of activations exceeds 70% and 80%, respectively. This approach enables MEPipe to train large models on accelerators with limited memory capacity without incurring communication overhead.

To further improve training efficiency, we extend the technique of separating the computation of weight gradients and activation gradients, as proposed in zero bubble pipeline parallelism [27]. Since there are no dependencies between the computation of different weight gradients, we partition them into individual General Matrix Multiplications (GEMM) and schedule these GEMMs when workers are waiting for communication with other workers. This technique reduces the bubbles caused by imbalanced computation workload and bubbles at the ending phases of each iteration.

The main contributions of MEPipe are the following:

1. We propose SVPP, a slice-level pipeline scheduling method that consumes less memory and has fewer bubbles than other state-of-the-art works.
2. We put forward the technique of fine-grained weight gradient computation to mitigate the imbalanced computation among different slices and reduce the bubbles at the ending phases of each iteration.
3. We implement MEPipe on Megatron-LM, achieving up to 1.86× speedup over other approaches. MEPipe

achieves 116 TFLOPS peak performance and 35% Model FLOPS Utilization (MFU) for Llama 13B model on 64 RTX 4090 GPUs. MEPipe maintains the comparable iteration time to that observed on 32 A100 GPUs while being 2.5× more cost-effective.

The rest of this paper is organized as follows. Section 2 introduces the background of different parallel strategies. Section 3 presents the overview of MEPipe. The SVPP scheduling method and its analysis are presented in Section 4. Section 5 focuses on fine-grained weight gradient computation technique. Section 6 describes the implementation of MEPipe, and Section 7 evaluates the performance of MEPipe. Related works are introduced in Section 8. Finally, Section 9 discusses the challenges and insights, and Section 10 concludes this paper.

## 2 Background

We first define the notations used by the following sections in Table 1.

**Table 1.** Notations used in this paper.

| Notation | Description |
|---|---|
| $n$ | number of micro-batches |
| $d$ | data parallel size (with ZeRO) |
| $p$ | pipeline stage number |
| $s$ | sequence pipeline size |
| $v$ | virtual pipeline size |
| $A$ | activation memory consumption of one sample |

### 2.1 Pipeline Parallelism

Pipeline parallelism partitions the computation graph into sequential subgraphs and allocates each subgraph to different stages. Micro-batches are sequentially processed from the first stage to the last stage during forward propagation, and in the reverse order during the back propagation.

Due to computing dependency, a single micro-batch cannot be processed in two stages concurrently. Stages may become idle when waiting for the output from their preceding or succeeding stages. The period of idleness is called **bubbles**, and the proportion of time it occupies within one iteration is referred to as the **bubble ratio**. The bubble ratio serves as an important metric when evaluating the efficiency of different pipeline scheduling methods.

GPipe [9] divides individual batches into smaller micro-batches. It first executes the forward passes for all micro-batches and then followed by their backward passes. This approach reduces the bubble ratio by concurrently processing distinct micro-batches at different stages.

PipeDream [8] and DAPPLE [7] further reduce the activation memory consumption by proposing the 1F1B (*one-forward-one-backward*) scheduling method, as shown in Figure 2. Different colors represent different micro-batches. As activations are saved after the forward passes and can only be released after the backward passes, interleaving the forward and backward passes reduces the peak memory consumption for each worker. However, the first stage still needs to save activations for $p$ forward passes, leading to huge memory consumption.

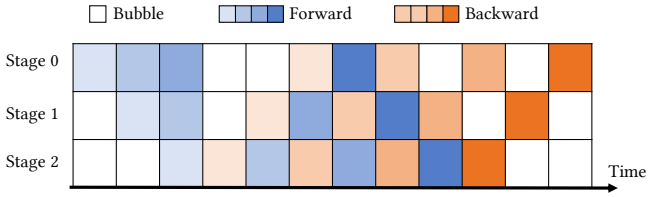


**Figure 2.** 1F1B pipeline scheduling in DAPPLE.

Chimera [19] and DeepSeek-V3 [5] design a novel bidirectional pipeline scheduling method to reduce the bubble ratio. DeepSeek-V3 further pairs the forward and backward passes to overlap the communication and computation for Mixture of Experts (MoE) models. The bidirectional scheduling method does not reduce the activations while consuming more memory as parameters are replicated across more than one stage.

Megatron-LM-v2 [24] introduces virtual pipeline parallelism (VPP), which partitions the computation graph into finer *chunks* and assigns multiple chunks to individual workers. During the forward propagation, one micro-batch will be processed from the first to the last stage multiple times and reversely during backward propagation. Hanayo [22] further presents an optimized wave-like pipeline scheduling method. These innovations reduce the bubble in the initial and ending phases of each iteration. Although the activations for individual forward passes are reduced, each stage needs to save activations of more forward passes. Therefore, the activation memory consumption remains large for these scheduling methods.

Zero bubble pipeline parallelism [27] optimizes the bubble ratio by separating the gradient computation of the activations and weights in the backward passes. This method delays the computation of weight gradients and advances the transfer of activation gradients to the preceding stage. The delayed gradient computation of weights fills the bubble at the ending phase of the latter stages.

TeraPipe [20] presents an innovative approach called sequence pipeline parallel (SPP), which partitions individual samples into smaller slices and schedules them similarly to GPipe, as shown in Figure 3. The numbers in the upper part

denote the index of each slice within a single sample. TeraPipe exploits the inherent characteristics of the decoder architecture, where tokens only depend on their preceding tokens. When processing the forward passes of the Attention Layers for each slice, workers need the key and value of all preceding slices to derive the final output.

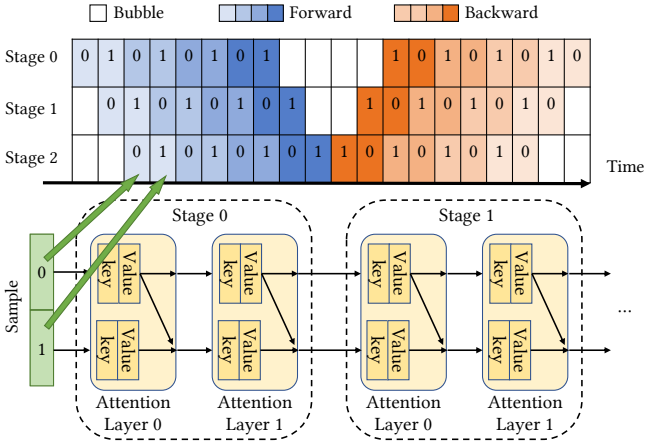


**Figure 3.** Pipeline scheduling of TeraPipe.

The fine-grained partitioning strategy reduces the bubble ratio and partitions the activations. However, TeraPipe's scheduling method is memory-consuming as workers need to preserve the activations of all samples before processing the first backward passes.

### 2.2 Other Types of Parallelism

In addition to pipeline parallelism, there are other important parallel strategies for training LLMs, including **data parallelism**, **tensor parallelism**, and **context parallelism**.

**Data parallelism.** Data parallelism (DP) [12, 18, 23] partitions input data across different workers while maintaining identical model parameters on each worker. In each iteration, workers need to synchronize the gradients after the backward passes. ZeRO [28] extends data parallelism by partitioning the optimizer states across different workers, to reduce memory consumption. ZeRO-2 and ZeRO-3 further partition the gradients and parameters. However, the integration of ZeRO-2 and ZeRO-3 with pipeline parallelism introduces substantial communication overhead, as each forward and backward pass requires the communication of parameters and gradients.

**Tensor parallelism.** Tensor parallelism (TP) [11, 17, 31] partitions the parameters of each layer and distributes these parts across different workers. During the forward and backward passes, the output activations of each layer should be synchronized across these workers, introducing significant communication overhead. Megatron-LM [32] reduces the communication volume by partitioning the weights of two contiguous linear layers in different dimensions. Due to the huge communication volume, tensor parallelism is limited to accelerators with high-bandwidth interconnections, like GPUs with NVLink technology.

**Context parallelism.** Context parallelism (CP) [10, 21] partitions single samples into smaller slices while maintaining identical parameters across different workers. This technique is initially proposed for training models with long context. When computing attention layers, workers need to communicate the key and value tensor with each other to derive the result, introducing communication overhead during the forward and backward passes. Although the communication volume of context parallelism is smaller than that of tensor parallelism, it still incurs substantial communication that would impact the overall performance. Since workers maintain identical parameters, the optimizer state partitioning technique in ZeRO can also be employed for context parallelism.

Table 2 shows the communication overhead and memory consumption of these parallel strategies.

**Table 2.** Comparison of different parallel strategies.

| | Comm. | Parameter Partition | Activation Partition | Optimizer Partition |
|---|---|---|---|---|
| TP | + + + + + | ✓ | ✓ | ✓ |
| CP (ZeRO) | + + ++ | ✘ | ✓ | ✓ |
| DP (ZeRO) | ++ | ✘ | ✘ | ✓ |
| PP | + | ✓ | ✘ | ✓ |
| SPP | + | ✓ | ✓ | ✓ |

## 3 Overview of MEPipe

With the growing scale of training datasets, the cost of training LLMs also increases rapidly. To democratize the training of LLMs to inexpensive accelerators with low-bandwidth interconnection, pipeline parallelism becomes necessary as it incurs minimal communication overhead.

As discussed in Section 2, existing pipeline scheduling methods require the first stage to preserve activations for several forward passes to reduce the bubble ratio, leading to substantial memory consumption.

To reduce the memory, MEPipe partitions each sample into finer slices as SPP and schedules at the granularity of slices. However, this slice-level scheduling introduces additional computational dependencies and workload imbalance across slices within a single sample. To address these issues, we propose two techniques: *SVPP scheduling method* and *fine-grained weight gradient computation*, which enable efficient slice-level scheduling for the first time.

The SVPP method schedules the forward and backward passes in slice granularity. Different variants of the scheduling methods are provided to suit different memory limits.

Fine-grained weight gradient computation further decomposes the weight gradient computation to the granularity

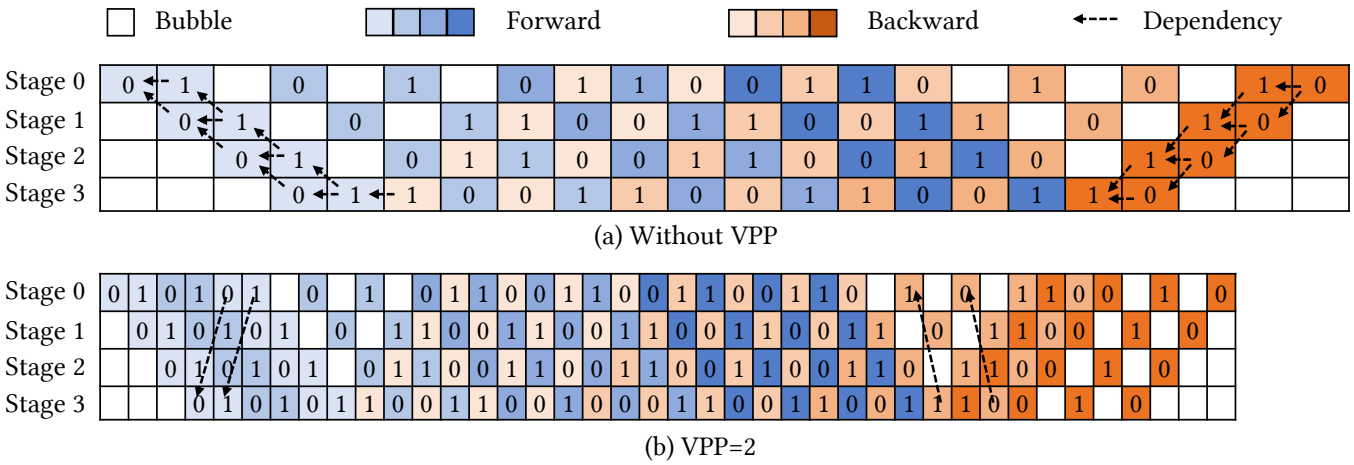


**Figure 4.** SVPP scheduling method with $p = 4$ and $s = 2$.

of individual General Matrix Multiplication (GEMM) operations. This method balances the computation of forward and backward passes, as different slices in sequence pipeline parallelism have different amounts of computation. Besides, this method advances the transmission of backward pass results, thereby enhancing the overall training efficiency.

## 4 Sequence Virtual Pipeline Parallelism

This section introduces the SVPP scheduling method and its generation process. Then we analyze the bubble ratio and memory consumption of SVPP and other state-of-the-art approaches. Finally, we discuss the selection of the variant of the scheduling method under different memory limitations.

### 4.1 SVPP Scheduling Method

As discussed in Section 2, sequence pipeline parallelism partitions individual samples into finer slices. The SVPP scheduling method interleaves forward and backward passes of single slices, thereby mitigating the peak memory utilization of activations. The design of scheduling methods should consider the dependencies between slices within one sample. The dependencies arise from the computation of the Attention Layers, whose results need the key and value components of all preceding slices, as shown in Figure 3. Taking the inherent dependencies into account, we design the scheduling method of SVPP in Figure 4.

Figure 4(a) and (b) illustrate the scheduling method with 4 pipeline stages and 4 samples, with each sample partitioned into 2 slices. Different colors denote different samples, while the numbers are indices of slices within a single sample. The arrows denote parts of dependencies between the forward and backward passes.

Figure 4(a) shows the scheduling method without the virtual pipeline parallelism. The forward pass of slice 1 has dependencies on both the forward pass of slice 0 in the same stage and the forward pass of slice 1 in the preceding stage. Moreover, the initiation of the backward pass of each sample requires the completion of all forward passes for slices 0 and 1 in that sample.

Figure 4(b) shows the scheduling method with the virtual pipeline size as 2. The computation graph is partitioned into 8 sequential chunks and each pipeline stage is allocated with 2 chunks. The processing of each slice needs two rounds of forward passes from stage 0 to stage 3. Therefore, the forward pass of the second chunk on stage 0 depends on the forward pass of the first chunk on stage 3, denoted with arrows in Figure 4(b). The bubbles in between can be filled with the forward passes of the next sample. Notably, the computation time for both forward and backward passes is reduced by half compared to the scheduling method in Figure 4(a), as each chunk contains half the number of layers. This strategy reduces the bubble ratio at the beginning and ending phases, enhancing overall training efficiency.

During each iteration, activations will be preserved after the forward pass and released after the corresponding backward pass. The peak memory consumption for activations is determined by the number of forward passes executed before the first backward pass. Due to the computation dependency across different stages, the first stage needs to execute more forward passes than subsequent stages at the beginning of each iteration, thus requiring more activation memory. Therefore, our analysis of memory consumption focuses on the first pipeline stage.

Let $A$ denote the memory volume of activations for a single sample. For Figure 4(a), the activation memory consumption for a single forward pass is $\frac{1}{8}A$, as the model is partitioned into 4 subgraphs and each sample is partitioned into 2 slices. The peak memory consumption of activations in Figure 4(a) is $\frac{5}{8}A$. However, as the computation graph is partitioned into 8 chunks in Figure 4(b), the activation memory

consumption for a single forward pass becomes $\frac{1}{16}A$. Consequently, the peak memory consumption of activations is $\frac{9}{16}A$ in Figure 4(b).

### 4.2 Variants of SVPP Scheduling Methods

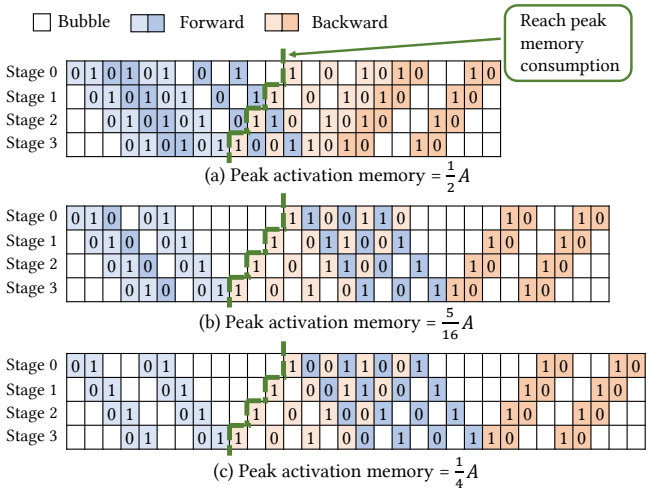


**Figure 5.** Variants of SVPP scheduling method with different peak memory consumption.

When the memory of accelerators is limited, SVPP can delay the forward passes after the first backward passes. Figure 5(a), (b), and (c) present different variants of the scheduling method when stage number is 4, virtual pipeline size is 2, and the number of micro-batches is 2.

This approach trades off memory consumption with the bubble ratio. For example, the scheduling method in Figure 5(c) reduces the memory consumption by 50% while increasing the bubble ratio by 50% compared to Figure 5(a). Theoretically, at least $v \times s$ forward passes must be executed before the first backward pass, as the first backward pass depends on the completion of all forward passes of the first sample. Each sample consists of $s$ slices, and each slice would be processed $v$ rounds across all stages, thus $v \times s$.

### 4.3 Generation of Scheduling Methods

The scheduling method is determined by the following five parameters: pipeline stage number ($p$), virtual pipeline size ($v$), sequence pipeline size ($s$), number of micro-batches ($n$), and the number of forward passes before the first backward pass ($f$). The generation process can be described as follows:

First, we arrange the forward and backward passes for the first micro-batch. Single bubbles will be inserted between two consecutive backward passes of different slices. These bubbles are left for the forward passes of the next micro-batches to maintain the peak memory utilization.

Subsequent micro-batches are processed sequentially. For the next $f - vs$ forward passes, we first place them into the bubbles between the forward passes of the first micro-batch when $v$ is larger than 1. The remaining forward passes will be placed at bubbles preceding the first backward pass (for stage 0) and bubbles between backward passes (for subsequent stages). Other forward passes are positioned at the bubble after one backward pass (for stage 0) sequentially. Backward passes are scheduled after the last forward passes of the corresponding micro-batch and the backward passes of preceding micro-batches. Similarly, single bubbles will be inserted in two consecutive backward passes.

After scheduling all micro-batches, the bubbles between two consecutive backward passes after the last forward pass can be removed to simplify the scheduling method, as shown in Figure 5(a).

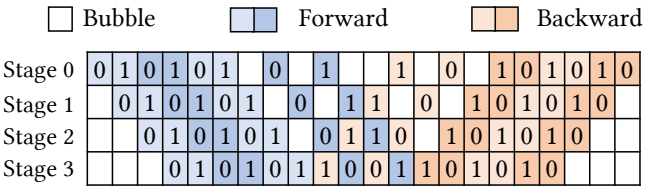


**Figure 6.** Optimized scheduling method of Figure 5(a). Peak activation memory is $\frac{1}{2}A$.

However, we can find the bubbles between the last few backward passes when the virtual pipeline size is larger than 1, as shown in Figure 5(a). This can be further optimized by rescheduling the backward passes, as depicted in Figure 6. Given that the backward passes of the last stage determine the scheduling of backward passes in preceding stages, optimization can be confined to the last stage.

To facilitate our analysis, we make the following definition regarding the relationships between backward passes. If the backward pass $x$ can only be computed after the backward pass $y$, we refer to $x$ as a child of $y$, and conversely, $y$ as the parent of $x$. We also use (Slice $i$, Chunk $j$) to represent the backward pass of the $i$-th slice on the $j$-th chunk.

The rescheduling method can be described as follows.

Firstly, we prioritize the backward passes based on the number of their children. Take Figure 4 for example, (Slice 1, Chunk 1) has 3 children: (Slice 0, Chunk 1), (Slice 1, Chunk 0), and (Slice 0, Chunk 0).

Secondly, we maintain a table to record the earliest possible initiation time for each backward pass. This table is dynamically updated according to the forward passes and the subsequent scheduling of their parent backward passes.

Finally, we reschedule the backward passes sequentially. For those positions before the last forward pass, we only substitute the backward passes and keep the same bubbles, to maintain the same peak memory utilization during the iteration. For positions after the last forward pass, we can schedule the remaining backward passes more flexibly as they won't influence the peak memory consumption.

When scheduling the backward passes, we will choose the backward pass with the highest priority among those whose dependent backward and forward passes have been completed, as indicated by the previously constructed table. After placing the backward pass, we can update the earliest initiation time of its children backward passes in the table.

### 4.4 Memory Consumption and Bubble Ratio

A comprehensive analysis of the theoretical activation memory consumption and bubble ratio of MEPipe and other state-of-the-art approaches is presented in Table 3.

As mentioned in Section 4.1, there are different variants of scheduling methods given the training configuration. We select the version that exhibits the lowest bubble ratio and maximal memory consumption for activations. For simplicity, we assume the computation graph is evenly partitioned, resulting in a balanced computation workload across all stages. Inter-stage communication is not considered here.

Zero bubble pipeline parallelism [27] is not discussed here either, as its method for partitioning the gradient computation of activations and weights is potentially applicable to all aforementioned approaches. For example, its scheduling methods, such as ZB-1P and ZBV, are extensions of the DAPPLE and Hanayo with the above method. This method reduces the bubbles at the beginning and ending phases during each iteration. However, in the scenario where the number of micro-batches is smaller than the pipeline stage number, this method will not be very useful as there are many bubbles in the middle phase in each iteration.

We also design a mechanism similar to zero bubble pipeline parallelism for MEPipe, which will be introduced in Section 5. For simplicity, this method will not be considered during the analysis.

Sequence pipeline parallelism has the problem of computation imbalance across different slices. The imbalance is primarily attributable to the computation of attention score, which constitutes less than 10% of the total computation when training a 7B model with a context length of 4096. The proportion is even smaller for models of greater scale. Consequently, the influence of computation imbalance is mild. Furthermore, the issue can be effectively mitigated with our technique that will be introduced in Section 5.

As illustrated in Table 3, the analysis is conducted under two distinct scenarios according to the relationship between the number of micro-batches and the pipeline stage number. Given that the training of LLMs is typically constrained by the global batch size (normally 2048 in Llama [34]), the condition $n \geq p$ represents the situation with a relatively modest number of accelerators. Conversely, $n < p$ represents the situation with a substantial number of accelerators (e.g., more than 2048 accelerators).

As shown in Table 3, in the scenario where $n \geq p$, MEPipe achieves a lower bubble ratio than other approaches. This enhanced efficiency is attributable to the simultaneous employment of virtual pipeline parallelism and sequence pipeline parallelism.

The activation memory consumption of MEPipe depends on the maximum number of forward passes before the first backward pass, as introduced in Section 4.1. In the scenario where $s \geq p$, there will be no bubble between the forward passes of the slices of different chunks within the first micro-batch. Consequently, the memory consumption can be expressed as $\frac{(vs+p-1)}{vsp}A$. Conversely, when $s < p$, there would be bubbles between the forward passes of different chunks within the first micro-batch, as shown in Figure 5(a). These bubbles will be filled with the forward passes of the next micro-batches. In this case, the peak memory consumption can be represented as $\frac{(v-1)p+s+p-1}{vsp}A$. Notably, the memory consumption of SVPP remains around $\frac{1}{\min(s,p)}A$, much smaller than other approaches.

In the scenario where $n < p$, the scheduling method of SVPP can still achieve a lower bubble ratio than other approaches. Furthermore, the memory consumption of SVPP in this scenario remains comparable to or less than that of other approaches.

### 4.5 Selection of the SVPP Scheduling Method Variants

As previously discussed, there are different variants of the SVPP scheduling method. When the memory capacity is limited, we need to choose the variant that satisfies the memory constraint and achieves the lowest bubble ratio.

We construct a memory model that can determine the variant of the SVPP scheduling method when given pipeline stage number $p$, data parallel size $d$, sequence pipeline size $s$, and virtual pipeline size $v$.

The memory model comprises three distinct components. The first is static memory, including parameters, gradients, and optimizer state. Let $m$ denote the number of parameters, the static memory consumption can be represented as $\frac{4m}{p} + \frac{8m}{dp}$ when we train the LLMs with half-precision and Adam [15] optimizer. Frameworks like Megatron-LM [32] may maintain copies of parameters and gradients in FP32, which would also be considered static.

The second component is the temporary memory for storing intermediate results. The size of temporary memory is determined by the model structure, as certain operators like the loss function need large temporary memory for intermediate results. This memory will be released after the computation and reused by other operators. To simplify, we regard the size of temporary memory as static during training.

The third component is the memory consumption of activations for forward passes. The memory consumption is also determined by the model structure and can be computed by summing the size of activations that should be preserved.

**Table 3.** Bubble ratio and activation memory of different SOTA scheduling methods. (-) indicates unsupported cases.

| Scheduling Method | $n \geq p$ (Small Cluster) | | $n < p$ (Large Cluster) | |
|---|---|---|---|---|
| | Bubble Ratio | Memory Consumption | Bubble Ratio | Memory Consumption |
| DAPPLE | $\frac{p-1}{p-1+n}$ | $A$ | $\frac{p-1}{p-1+n}$ | $\frac{n}{p}A$ |
| VPP | $\frac{p-1}{p-1+nv}$ | $\min((1+\frac{p-1}{pv}), \frac{n}{vp})A$ | - | - |
| Hanayo | $\frac{p-1}{p-1+nv}$ | $A$ | $\frac{vp+n-1-nv}{vp+n-1}$ | $\frac{n}{p}A$ |
| TeraPipe | $\frac{p-1}{ns+p-1}$ | $\frac{n}{p}A$ | $\frac{p-1}{ns+p-1}$ | $\frac{n}{p}A$ |
| SVPP | $\frac{p-1}{nsv+p-1}$ | $\frac{(v\max(p,s)+\min(p,s)-1)}{vsp}A$ | $\frac{p-1+(v-1)\max(p-sn,0)}{p-1+(v-1)\max(p-sn,0)+nvs}$ | $\min(\frac{v\max(p,s)+\min(p,s)-1}{vsp}, \frac{n}{p})A$ |
| SVPP ($s \to +\infty$) | 0 | $\frac{1}{p}A$ | 0 | $\frac{1}{p}A$ |

With the above three components, we can compute the maximum number of forward passes that can be executed before the first backward pass, thus determining the variant of the scheduling method.

## 5 Fine-grained Computation of Weight Gradients

Zero bubble pipeline parallelism decomposes the backward pass into the computation of activation gradients and weight gradients. We find that the computation of weight gradients can be further partitioned as there are no dependencies between the gradient computation of distinct weights. Consequently, we split the computation of weight gradients into the granularity of individual GEMMs and schedule these GEMMs dynamically.

Figure 7 shows the pipeline scheduling method when $p = 4$, $s = 2$, $v = 1$ and $n = 4$. We assume the forward time for slice 0 is 75% of that for slice 1. The imbalanced workload is primarily attributable to the structure of the Decoder Models [3]. As depicted in Figure 3, the computation of the Attention Layer for subsequent slices calls for the key and value components from preceding slices, resulting in a progressive increase in the computation workload for later slices.

For simplicity, we assume the backward time is the same as the forward time for each slice. We also assume the time for the weight gradient computation of each slice is the same as the forward time of slice 0, as weight gradient computation does not include the imbalanced computation of the attention score.

As shown in Figure 7, during the backward pass, GEMM operations of weight gradient computation are enqueued. These GEMMs are then dequeued and processed during the intervals when workers are waiting for the tensor from the preceding or subsequent stages.

In Figure 7(a), stage 0 executes the weight gradient computation immediately after its backward passes. If this computation is deferred, workers need to preserve the corresponding activations and activation gradients, leading to increased peak memory consumption.

However, subsequent stages consume less memory compared to preceding stages and can preserve more activations and activation gradients. This indicates that subsequent stages can postpone the weight gradient computation and advance the forward and backward passes of next slices. Consequently, bubbles in the ending phases can be used to compute weight gradients, yielding better training performance.

Moreover, if more memory is provided, we can advance the forward and backward passes in stage 0 to further reduce the bubbles, as shown in Figure 7(b). The memory consumption of forward and backward passes can be computed as discussed in Section 4.5. When computing the weight gradients, we can stop and process the next forward or backward pass as soon as there is enough memory. This approach enables the advancement of more forward and backward passes and improves efficiency.

Moreover, fine-grained weight gradient computation mitigates the imbalanced computation among different slices. The computation of attention score constitutes less than 10% of the total computation workload for 7B models with a context length of 4096. In such scenarios, scheduling single GEMMs dynamically is enough to fill the bubbles caused by imbalanced computation among different slices.

TeraPipe [20] solves this problem by partitioning each sample in a non-uniform way. They further design a dynamic programming algorithm to find the optimal partitioning strategy in their paper. However, operators like GEMM and FlashAttention [4] exhibit optimal performance when the input dimensions are the powers of 2, particularly on

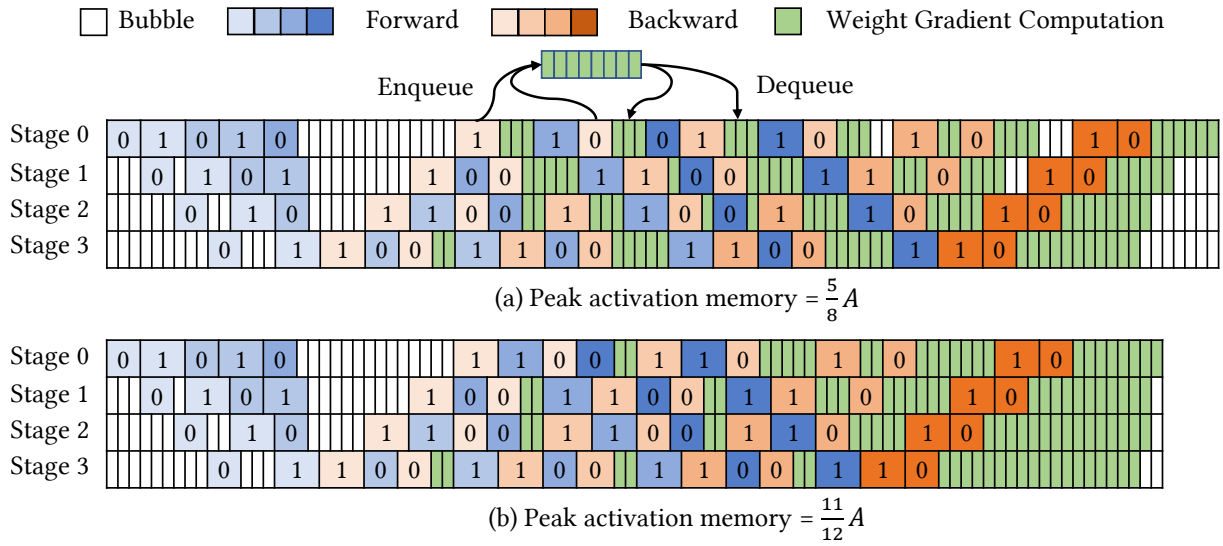


**Figure 7.** Fine-grained weight gradient computation.

modern accelerators like NVIDIA GPUs. The non-uniform partitioning approach potentially compromises the performance of these operators, thus impairing the overall training efficiency.

However, when training models with a context longer than 128,000 tokens, the computation of attention scores becomes significant, and the time of weight gradient computation is relatively small. In this scenario, the non-uniform partitioning strategy would be more efficient.

## 6 Implementation

We implement MEPipe on the Megatron-LM [24] framework using PyTorch [26]. MEPipe includes three main components: (1) a profiler that measures the computation time and memory consumption for each forward and backward pass; (2) an SVPP scheduler that determines the scheduling method with the parallel strategy and measured result; (3) an execution engine that conducts the scheduling method.

## 7 Evaluation

### 7.1 Experimental Setup

**Cluster.** We evaluate MEPipe on a cluster consisting of 8 servers. Each server is equipped with 8× NVIDIA RTX 4090 24GB GPUs, 64 CPU cores in 2 sockets, and 512 GB memory. The intra-node GPUs are connected with PCIe 4.0, and servers are interconnected with 100 Gbps Infiniband NICs.

**Models and Workloads.** We evaluate MEPipe on the representative model Llama 2 [34] with 7B, 13B, and 34B parameters. To mitigate the computation and memory imbalance caused by the embedding layer and the head layer at the front and the end of the model, we remove two transformer layers for the Llama models in different sizes. This approach is also adopted in training LLMs like Llama 3 [6].

**Configurations.** The global batch size imposes a critical constraint in training LLMs, as a larger global batch size can hinder the convergence of the loss curve. In Llama 2 [34], the global batch size is set to 1024 for models in different sizes. To emulate the training of LLMs on clusters containing 512, 1024, and 2048 accelerators, we keep a similar computation workload per accelerator by setting the global batch size as 128, 64, and 32, respectively. Furthermore, data parallelism is widely employed in large clusters, as pipeline parallelism is limited by the model structure, and tensor parallelism is restricted by the interconnection bandwidth. We set the minimal data parallel size to 2 to simulate realistic training on large clusters.

The detailed configuration is listed in Table 4.

**Table 4.** Evaluation configurations for Llama.

| Model | Hidden | Layer | #Dev | SeqLen | GBS |
|---|---|---|---|---|---|
| 7B | 4096 | 30 | 64 | 4096 | 128 |
| 13B | 5120 | 38 | | | 32 / 64 / 128 |
| 34B | 8192 | 46 | | | 128 |

**Baseline.** We choose DAPPLE [7], Virtual Pipeline Parallelism [38], and Zero Bubble Pipeline [27] as baselines.

For DAPPLE, we exhaustively search all possible combinations of the DP, PP, CP, and recomputation strategy to find the optimal combination of these strategies. Tensor parallelism is excluded here as it requires huge communication, and RTX 4090 GPUs are not equipped with high-bandwidth interconnect like NVLinks.

For virtual pipeline parallelism (VPP), we further search virtual pipeline size to find the optimal training strategy.

Zero bubble pipeline parallelism has several versions of scheduling methods, including ZB and ZBV, which are extensions of DAPPLE and Hanayo scheduling methods, respectively. They have different bubble ratios, communication overhead, and memory consumption. The recomputation technique is not compatible with zero bubble pipeline parallelism, as workers need to preserve activations and activation gradients for the postponed computation of weight gradients. Therefore, we only search the combination of the PP and CP size to find the optimal result.

The evaluation is conducted using 16-bit floating point precision. Each task is executed for 100 iterations to ensure the optimizer is allocated. We measure the average time of the last 10 iterations as the result.

### 7.2 End-to-End Performance

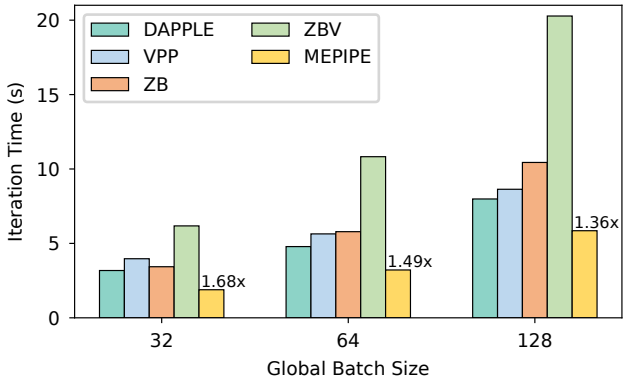


**Figure 8.** Iteration time of Llama 13B with different global batch sizes.

We measure the iteration time of the aforementioned approaches for the Llama 13B model with different global batch sizes and present the results in Figure 8. Table 5 shows the optimal training strategy for each scenario.

**Table 5.** Parallel configuration of all approaches on Llama 13B with different global batch sizes. The tuple elements are PP, CP/SPP, VP, and recomputation enabled, resp.

| System | 32 | 64 | 128 |
|---|---|---|---|
| DAPPLE | (8, 2, 1, ✘) | (8, 2, 1, ✘) | (8, 2, 1, ✘) |
| VPP | (4, 2, 2, ✓) | (4, 1, 2, ✓) | (4, 1, 2, ✓) |
| ZB | (8, 4, 1, ✘) | (8, 4, 1, ✘) | (8, 4, 1, ✘) |
| ZBV | (4, 8, 2, ✘) | (4, 8, 2, ✘) | OOM |
| MEPipe | (8, 4, 1, ✘) | (8, 4, 1, ✘) | (8, 4, 1, ✘) |

DAPPLE exhibits optimal performance with a context parallel size of 2 when the global batch size is 32. Although context parallelism introduces additional communication overhead, it increases the number of micro-batches for each data parallel group by partitioning individual samples. In scenarios with small global batch sizes, the performance improvement brought by bubble ratio reduction is more significant. When the global batch size grows to 64 and 128, the memory consumption of activations also increases, necessitating context parallelism to reduce it. Although the recomputation technique can also reduce the activation memory, it introduces additional computation overhead. A comprehensive analysis of the influence of different training configurations can be found in Section 7.3.

VPP schedules more forward passes at the beginning of each iteration, thereby consuming more activation memory. Moreover, as the computation graph should be partitioned evenly for all approaches and Llama 13B comprises 40 layers (including the embedding and head layer), the maximum number of stages for VPP only reaches 4. This leads to higher memory consumption for parameters and gradients (optimizer states are evenly distributed across all devices with the ZeRO technique). Consequently, VPP necessitates the recomputation technique to reduce memory of activations and shows worse performance than DAPPLE.

Theoretically, ZB exhibits comparable memory utilization to DAPPLE. However, we find that the PyTorch Allocator reserves more memory during the experiments for ZB, resulting in an out-of-memory error when the PP size is 8 and the CP size is 2. ZBV consumes more memory so the maximum number of stages only reaches 4, similar to VPP. Consequently, the feasible context parallel size for ZB and ZBV is 4 and 8, respectively, which leads to substantial communication overhead and worse performance.

Unlike the above approaches, MEPipe further partitions individual samples into slices, reducing the memory for activations without introducing additional communication or computation. As a result, MEPipe achieves 1.36× speedup over other approaches when the global batch size is 128.

The global batch sizes of 64 and 32 correspond to scaled training configurations with 1024 and 2048 accelerators, respectively. In this scenario, in addition to reducing memory, the SVPP scheduling method can achieve a lower bubble ratio by scheduling forward and backward passes in the granularity of slices. Therefore, MEPipe achieves higher speedups of 1.49× and 1.86× over other approaches. While larger sequence pipeline sizes yield smaller bubble ratios, they will impair the computation efficiency of operators like GEMM and FlashAttention. Section 7.3 analyzes the above trade-off in detail.

### 7.3 Influence of Different Parallel Strategies

Different parallel strategies influence the bubble ratio, memory consumption, communication overhead, and computation efficiency in distinct ways. In this section, we will analyze the impact of pipeline parallelism, context parallelism, sequence pipeline parallelism, and the recomputation technique on these factors.

**Table 6.** Influence of PP on DAPPLE for Llama 13B with global batch size as 64.

| (PP, DP, CP, Recomp.) | Bubble Ratio | Iteration Time |
|---|---|---|
| (2, 4, 8, ✘) | 11.1% | OOM |
| (4, 4, 4, ✘) | 15% | 6711.8 ms |
| (8, 4, 2, ✘) | 18% | 6226.3 ms |

**Pipeline parallelism (PP).** Pipeline parallelism divides the model into contiguous subgraphs. By increasing PP, the number of parameters assigned to each stage is reduced. This leads to a decrease in memory consumption and communication time for synchronizing the parameters and gradients. As shown in Table 6, workers exceed memory capacity when PP is set to 2, while increasing PP to 4 satisfies the memory limitation. It can be observed that increasing PP results in a higher bubble ratio. However, the incremental impact on the bubble ratio diminishes as PP increases. Therefore, a PP of 8 has better performance than a PP of 4.

**Table 7.** Influence of CP on DAPPLE for Llama 13B with global batch size as 32.

| (PP, DP, CP, Recomp.) | Bubble Ratio | Iteration Time |
|---|---|---|
| (8, 8, 1, ✘) | 63.6% | 3619.0 ms |
| (8, 4, 2, ✘) | 46.7% | 3199.7 ms |
| (8, 2, 4, ✘) | 30.4% | 3772.9 ms |

**Context parallelism (CP).** Context parallelism partitions each sample into finer slices and distributes them across different workers. CP reduces the memory consumption of activations by a factor of CP size. Moreover, the number of micro-batches for each data parallel group is increased, thereby reducing the bubble ratio, as shown in Table 7.

However, as shown in Figure 9, increasing CP impairs the performance of transformer layers, primarily due to two factors. First, CP introduces additional communication overhead, which becomes substantial as CP size increases. Second, CP results in the decreased computational performance of operators such as GEMM and FlashAttention. Furthermore, the Megatron-LM [24] framework partitions individual samples into $2 \times CP$ slices and assigns each worker two slices in the symmetric position within the sample, e.g., $(1, 4)$ for one worker and $(2, 3)$ for another. This approach balances the computation workload across different workers. However, the finer partition strategy leads to more severe degradation in computational performance.

As illustrated in Table 7, increasing CP from 1 to 2 reduces the iteration time, as the reduction in the bubble ratio outweighs the increased communication overhead and decreased operator performance. However, if we further increase CP to 4, the communication overhead and operator performance degradation become the dominating factors, leading to a longer iteration time.

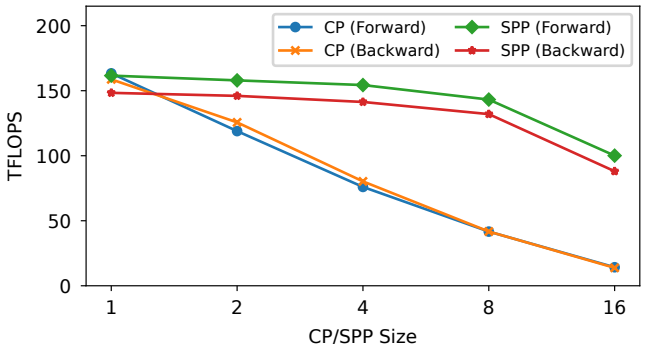


**Figure 9.** Measured performance of transformer layers with different CP/SPP sizes.

**Sequence Pipeline Parallelism (SPP).** Similar to CP, SPP reduces the bubble ratio and the memory consumption of activations. Moreover, SPP also suffers from the declined performance of operators like GEMM and FlashAttention. However, since SPP does not introduce communication overhead, the performance degradation is not as substantial as CP, as demonstrated in Figure 9. For instance, the performance of each transformer layer in Llama 13B only decreases by 12.6% when SPP increases from 1 to 8. The trade-off between the reduced bubble ratio and performance degradation of operators is similar to that in CP.

**Virtual Pipeline Parallelism (VPP) and ZBV.** Both VPP and ZBV partition the model into multiple chunks and allocate more than one chunk to each stage. While increasing the VP size reduces the bubble at the beginning and end phases in each iteration, this method introduces more inter-stage communication overhead. Furthermore, VPP may consume more memory for parameters and gradients. For example, the maximum number of chunks is limited to 8 for the Llama 13B model with 40 layers. Compared with traditional pipeline parallelism with 8 stages, VPP requires each worker to preserve more parameters, thereby consuming more memory.

**Recomputation Technique.** This technique drops most activations during forward passes and recomputes them in backward passes. This method reduces the activation memory consumption by 90%, at the cost of 33% more computation. To achieve equivalent memory reduction, the size of CP or SPP should exceed 8. The recomputation technique may be more efficient than increasing CP or SPP sizes if operator performance exhibits significant degradation in certain cases.

**Selection of the Optimal Parallel Strategy.** The goal is to find the optimal parallel strategy that satisfies the memory constraints and minimizes the computational overhead,

bubble ratio, and communication costs simultaneously. Memory consumption and bubble ratio can be computed theoretically from the parallel strategy. However, precise prediction of communication overhead and computational efficiency presents significant challenges. Therefore, we employ the grid search method to determine the optimal parallel strategy. This approach can be further optimized by constructing precise cost models in the future.

### 7.4 Influenence of Model Size

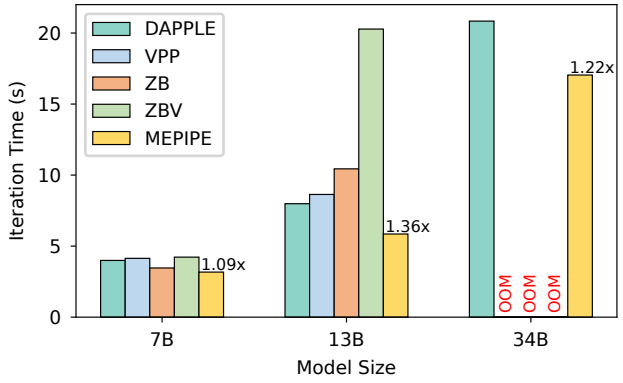


**Figure 10.** Iteration time of Llama in different sizes with global batch size as 128.

Figure 10 and Table 8 show the iteration time and the optimal training strategy of the above approaches on the Llama model when the global batch size is 128.

For Llama 7B, since the memory consumption is relatively small, ZB can train the model without context parallelism. In this situation, the speedup of MEPipe is primarily attributed to the smaller bubble ratio and the overlap of communication and computation.

**Table 8.** Parallel configuration of all approaches on Llama with different sizes. The cells are (PP, CP/SPP, VP, recomputation enabled).

| System | 7B | 13B | 34B |
|---|---|---|---|
| DAPPLE | (16, 1, 1, ✘) | (8, 2, 1, ✘) | (16, 2, 1, ✓) |
| VPP | ( 8, 2, 2, ✘) | (4, 1, 2, ✓) | - |
| ZB | (16, 1, 1, ✘) | (8, 4, 1, ✘) | - |
| ZBV-ZB | ( 8, 2, 2, ✘) | (4, 8, 2, ✘) | - |
| MEPipe | ( 8, 4, 1, ✘) | (8, 4, 1, ✘) | (16, 16, 1, ✘) |

For Llama 34B, the static memory consumption is very large. The mixed precision optimizer in Megatron-LM occupies around 6.375 GB for each worker, while the parameters and their gradients consume approximately $\frac{34\times4}{p}$ GB. For VPP and ZBV, as the maximum pipeline parallel size is 8, the static memory exceeds the capacity of the GPU.

DAPPLE and MEPipe can train the model with a pipeline parallel size of 16. However, the left memory for activations is around 5GB. DAPPLE employs the recomputation technique and context parallelism to reduce the memory for activations. By partitioning each sample into 16 slices, MEPipe can find the variant of the SVPP scheduling method that satisfies the memory limit. This schedule achieves a low bubble ratio and does not introduce extra computation or communication, thus achieving better performance than DAPPLE.

### 7.5 The Evaluation of Fine-grained Weight Gradient Computation

To evaluate the fine-grained weight gradient computation, we compare MEPipe with and without this technique on the Llama 13B model when the global batch size is 64 with the corresponding parallel strategy in Table 5. We profile the timeline for each pipeline stage and visualize the timeline in Figure 11 and 12.

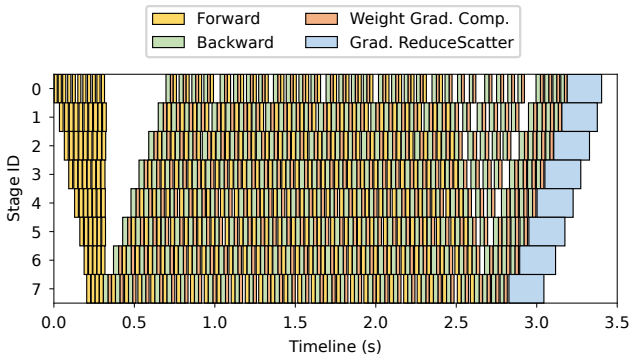


**Figure 11.** Timeline of different stages in MEPipe w/o fine-grained weight gradient computation.

Without fine-grained weight gradient computation, each worker will compute the weight gradient right after the corresponding backward passes. The computational workload is imbalanced across different workers and slices. Many bubbles can be found in the middle of the iteration in Figure 11.

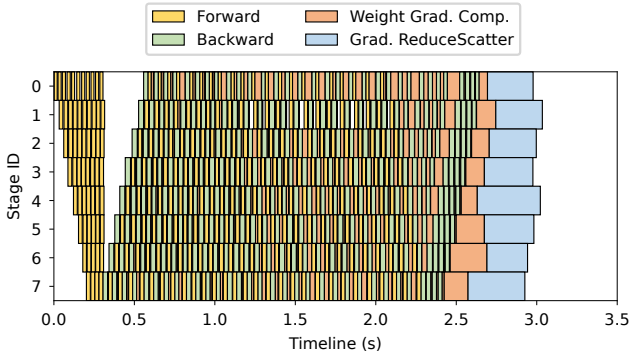


**Figure 12.** Timeline of different stages in MEPipe.

With the technique of fine-grained weight gradient computation, we can advance the forward and backward passes and fill the bubbles in the middle phase of each iteration with weight gradient computation. Additionally, the last few stages can deter more weight gradient computation until after the completion of all backward passes, effectively reducing the bubbles at the ending phase in each iteration. The fine-grained weight gradient computation brings a 9.4% performance improvement.

### 7.6 Analysis of FLOPS and Cost

**Table 9.** Comparison between A100 and RTX 4090.

| | A100 | RTX 4090 |
|---|---|---|
| Memory | 80 GB | 24 GB |
| FLOPS (FP16) | 312 TFLOPS | 330 TFLOPS |
| Bandwidth | 600 GB/s | 64 GB/s |
| Price (Server) | $150,000 | $30,000 |
| Llama 7B | 3216 ms | 3171 ms |
| | 220.4 TFLOPS | 111.7 TFLOPS |
| Llama 13B | 6131 ms | 5852 ms |
| | 221.4 TFLOPS | 116.0 TFLOPS |
| Llama 34B | 16167 ms | 17043 ms |
| | 213.9 TFLOPS | 101.5 TFLOPS |

We also measure the iteration time of Llama models on a cluster comprising 4 servers, each equipped with eight A100 80GB GPUs interconnected via NVLink. Servers are connected with 800 Gbps Infiniband NICs.

Table 9 compares memory capacity, computational power, bidirectional intra-node communication bandwidth, price of a single server with 8 GPUs, and the optimal iteration time for Llama models with a global batch size of 128 on the A100 cluster and RTX 4090 cluster.

As shown in Table 9, the iteration time of MEPipe on 64 RTX 4090 GPUs is comparable to the optimal iteration time on 32 A100 GPUs. In our experiments, we use FP32 as the accumulation type for GEMM kernels to maintain the same convergence, which influences the computational performance of RTX 4090 GPUs. As a result, a single RTX 4090 achieves approximately half the performance of a single A100. This can be further improved by rewriting the GEMM kernels in the future. Since one RTX 4090 server costs only one-fifth of one A100 server, MEPipe with the RTX 4090 cluster remains 2.5 times more cost-effective than the A100 cluster.

## 8 Related Work

**Recomputation Technique** The recomputation technique reduces memory by discarding intermediate activations during the forward passes and recomputing them in the backward passes. Previous studies [2, 18] propose methods to reduce the recomputation for models with sequential layers. For LLMs, Megatron-LM [24] employs the recomputation strategy that only preserves the input tensor of each decoder layer. A recent study [16] proposes selective recomputation, which strategically drops the memory-consuming intermediates within each layer. Recognizing that certain pipeline scheduling methods cause imbalanced memory consumption across different pipeline stages, AdaPipe [33] employs different recomputation strategies for each stage and redistributes the computation graph across stages to enhance training efficiency.

**Offloading Technique** The offloading technique mitigates the memory pressure by utilizing host memory or external storage. ZeRO-Offload [30] offloads the optimizer states and gradients to the host memory, and ZeRO-Infinity [29] further leverages NVMe SSDs. As offloading incurs communication overhead between GPUs and CPUs, these works carefully schedule the data movement to overlap them with computation.

SuperNeurons [35], vPipe [38], and MPresss [39] integrate the offloading technique with the recomputation technique. They propose algorithms to determine the recomputation and offloading strategies for various operators. Bpipe [14] transfers activations across different pipeline stages to mitigate the imbalanced memory consumption caused by certain pipeline scheduling methods.

## 9 Discussion

Although MEPipe reduces memory consumption and improves training efficiency, training LLMs on RTX 4090 GPUs still faces challenges.

First, training LLMs on a cluster of thousand RTX 4090 GPUs presents hardware reliability challenges. Recent studies in memory-based checkpointing [13, 36] reduce the fault recovery time to a few minutes. Given that the mean time between failures (MTBF) is approximately 12 hours for a thousand A100 GPUs [1], we estimate the cost of hardware failures is less than 5% for a thousand RTX 4090 GPUs.

Second, training LLMs in FP16 is prone to overflow and underflow issues, requiring techniques like sandwich layer normalization and embedding layer gradient shrink [37].

Third, the power consumption specifications for RTX 4090 and A100 GPUs are 450W and 400W, respectively. Since two RTX 4090 GPUs deliver computational performance comparable to a single A100 GPU, RTX 4090 clusters exhibit higher power consumption than A100 clusters with equivalent computational capabilities, resulting in greater operational costs. However, as discussed in Section 7.6, A100 clusters require a higher initial investment. Based on the industrial electricity rate of $0.1 per kilowatt-hour as of February 2025, it would take approximately 24 years for A100 clusters to achieve cost parity with RTX 4090 clusters.

Fourth, the grid search method used in our experiment incurs substantial overhead due to the large search space,

which expands with both model size and the number of accelerators. Consequently, this calls for automated parallelization frameworks that can construct cost models and identify the optimal strategy automatically.

Nevertheless, we believe that the SVPP scheduling method in MEPipe provides a new perspective for reducing the cost of training LLMs. The reduced memory and communication cost diminishes the traditional emphasis on memory capacity and interconnection bandwidth. The SVPP method broadens the design space for hardware manufacturers and potentially leads to more cost-effective solutions.

## 10 Conclusion

This paper presents MEPipe, a training system that incorporates a slice-level pipeline scheduling method to reduce memory consumption and bubble ratio. Moreover, MEPipe employs fine-grained weight gradient computation to balance the computation across different slices and reduce the bubbles at the ending phase of each iteration. Evaluations show that MEPipe achieves up to 1.68× speedup over other approaches on 64 NVIDIA RTX 4090 GPUs.

In addition to optimizing the training efficiency on existing low-end accelerators, our work aims to inspire more effective hardware-software co-design for future LLM accelerators.

## 11 Acknowledgment

We would like to thank Huanqi Cao, all anonymous reviewers, and our shepherd, Jia Rao, for their insightful comments and feedback. We also thank Zhipu AI for sponsoring computation resources. This work is supported by the National Natural Science Foundation of China under Grant Number U20B2044. Wenguang Chen is the corresponding author.

## References

[1] 2023. OPT-175B logbook. https://github.com/facebookresearch/metaseq/tree/main/projects/OPT/chronicles.

[2] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Alexis Joly, and Alena Shilova. 2019. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. *CoRR* abs/1911.13214 (2019). arXiv:1911.13214 http://arxiv.org/abs/1911.13214

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG]

[5] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). doi:10.48550/ARXIV.2412.19437 arXiv:2412.19437

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[7] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, et al. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 431–445. doi:10.1145/3437801.3441593

[8] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377 (2018). arXiv:1806.03377 http://arxiv.org/abs/1806.03377

[9] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 103–112. https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

[10] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *CoRR* abs/2309.14509 (2023). doi:10.48550/ARXIV.2309.14509 arXiv:2309.14509

[11] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks.. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 1–13. https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[13] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, et al. 2024. MegaScale: Scaling Large Language Model Training to More Than 10, 000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[14] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPIPE: memory-balanced pipeline parallelism for training large language models. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 682, 15 pages.

[15] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

[16] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. arXiv:2205.05198 [cs.LG]

[17] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[18] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.

[19] Shigang Li and Torsten Hoefler. 2021. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. doi:10.1145/3458817.3476145

[20] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. arXiv:2102.07988 [cs.LG]

[21] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2024. RingAttention with Blockwise Transformers for Near-Infinite Context. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=WsRHpHH4s0

[22] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 2023. Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 56, 13 pages. doi:10.1145/3581784.3607073

[23] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-performance heterogeneity-aware asynchronous decentralized training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 401–416.

[24] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 58. doi:10.1145/3458817.3476209

[25] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[27] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2024. Zero Bubble (Almost) Pipeline Parallelism. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=tuzTN0eIO5

[28] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 20. doi:10.1109/SC41405.2020.00024

[29] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59. doi:10.1145/3458817.3476205

[30] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[31] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084* (2018).

[32] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[33] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 86–100. doi:10.1145/3620666.3651359

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). doi:10.48550/ARXIV.2307.09288 arXiv:2307.09288

[35] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 41–53. doi:10.1145/3178487.3178491

[36] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 364–381. doi:10.1145/3600006.3613145

[37] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, et al. 2023. GLM-130B: An Open Bilingual Pre-trained Model. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=-Aw0rrrPUF

[38] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, et al. 2022. vPipe: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Trans. Parallel Distributed Syst.* 33, 3 (2022), 489–506. doi:10.1109/TPDS.2021.3094364

[39] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. 2023. MPress: Democratizing Billion-Scale Model Training on Multi-GPU Servers via Memory-Saving Inter-Operator Parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 556–569. doi:10.1109/HPCA56546.2023.10071077

# A Artifact Appendix

## A.1 Abstract

MEPipe is a memory-efficient training framework incorporating the slice-level pipeline scheduling method and fine-grained weight gradient computation for LLMs. The artifact comprises three components:

- The source code of MEPipe, DAPPLE, Virtual Pipeline Parallelism, and Zero Bubble Pipeline. We have enhanced the implementation of Zero Bubble Pipeline to support context parallelism.
- A subset of the OpenWebText dataset, processed with the Llama2 tokenizer.
- Scripts and instructions for validating the functionality and reproducing the experiment result in Section 7.2 and Section 7.4.

## A.2 Description & Requirements

**A.2.1 How to access.** The artifact can be retrieved from https://zenodo.org/records/14942680.

**A.2.2 Hardware dependencies.** MEPipe needs NVIDIA A100 / RTX 4090 GPUs or subsequent generation architectures. The functionality validation requires a single server equipped with 8 A100 40GB GPUs connected via PCIe. The reproduction of the experiments in the paper requires 8 servers interconnected with 100 Gbps Infiniband NICs, each having 8 NVIDIA RTX 4090 GPUs.

**A.2.3 Software dependencies.** The required software includes Python 3.10, CUDA 12.4, CUDNN 8.9.7, apex, TransformerEngine 1.12.0, and FlashAttention 3.5.8. Additional Python packages are documented in the `requirements.txt` in the MEPipe repository.

**A.2.4 Benchmarks.** We process the OpenWebText corpus with the Llama2 tokenizer for evaluation. Preprocessed dataset and the tokenizer can be found in the `dataset` directory within the artifact.

## A.3 Set-up

The detailed instructions for installing software dependencies and configuring the environment are documented in the `MEPipe.md` file within the MEPipe repository.

## A.4 Evaluation workflow

### A.4.1 Major Claims.

**C1:** MEPipe outperforms existing state-of-the-art works with different global batch sizes. This is proven by the experiment (E1) described in Section 7.2, results of which are illustrated in Figure 8. This can also be proved by the functionality experiment (E0).

**C2:** Sequence pipeline parallelism has superior performance compared to context parallelism when splitting each sample into the same number of slices. This is proven by the experiment (E2) in Section 7.3, results of which are illustrated in Figure 9.

### A.4.2 Experiments.

**Experiment (E0)** : [functionality] [4 compute-hours]: Validate the functionality of MEPipe.
[Preparation] Modify the `DATA_DIR` variable following the instructions in the `MEPipe.md` file.
[Execution] Run the test for MEPipe and other systems.

```
cd repo-name
sh e0_run.sh
sh e0_collect.sh
```

[Results] The expected results are in the `e0_expected.md` file within the respective repositories.

**Experiment (E1)** : [End-to-End] [24 compute-hours]: Compare the throughput of MEPipe with other systems on the Llama 13B model across different global batch sizes.
[Preparation] Modify the `DATA_DIR` variable following the instructions in the `MEPipe.md` file.
[Execution] Run the test for MEPipe and other systems.

```
cd repo-name
sh e1_run.sh
sh e1_collect.sh
```

[Results] The expected results are in the `e1_expected.md` file within the respective repositories. Then we can plot the Figure 8 with the following command:

```
python3 plot_e1.py
```

**Experiment (E2)** : [SPP/CP profiling] [1 compute-hours]: Profiles the performance of each transformer layer with different SPP/CP sizes.
[Preparation] Modify the `DATA_DIR` variable following the instructions in the `MEPipe.md` file.
[Execution] Run the test within the MEPipe and Megatron-LM repositories.

```
cd repo-name
sh e2_run.sh
sh e2_collect.sh
```

[Results] The expected results are in the `e2_expected.md` file within the MEPipe and Megatron-LM repository. Figure 9 can be plotted with the following command:

```
python3 plot_e2.py
```