AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning

Zhenbo Sun[†], Huanqi Cao[†], Yuanwei Wang[†], Guanyu Feng[†]
Shengqi Chen[†], Haojie Wang[†], Wenguang Chen[†‡]
[†]Tsinghua University, [‡]Peng Cheng Laboratory
{sunzb20,caohq18,wangyw20,fgy18,csq20}@mails.tsinghua.edu.cn,{wanghaojie,cwg}@tsinghua.edu.cn

Abstract

Large language models (LLMs) have demonstrated powerful capabilities, requiring huge memory with their increasing sizes and sequence lengths, thus demanding larger parallel systems. The broadly adopted pipeline parallelism introduces even heavier and unbalanced memory consumption. Recomputation is a widely employed technique to mitigate the problem but introduces extra computation overhead.

This paper proposes AdaPipe, which aims to find the optimized recomputation and pipeline stage partitioning strategy. AdaPipe employs adaptive recomputation to maximize memory utilization and reduce the computation cost of each pipeline stage. A flexible stage partitioning algorithm is also adopted to balance the computation between different stages. We evaluate AdaPipe by training two representative models, GPT-3 (175B) and Llama 2 (70B), achieving up to 1.32× and 1.22× speedup on clusters with NVIDIA GPUs and Ascend NPUs respectively.

ACM Reference Format:
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 *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651359

1 Introduction

Deep learning models have evolved to possess more than tens of billions of parameters recently in multiple scenarios, including natural language [3, 10] and computer vision [7]. As a data-driven and parameter-sensitive method, the performance of a deep learning model naturally develops as the size of its parameters and training data scales up [11]. Such large models cannot fit into the limited storage of one single computing device. Moreover, language models supporting long context [4, 26] receive much attention on various tasks, posing greater memory challenges to the training systems.

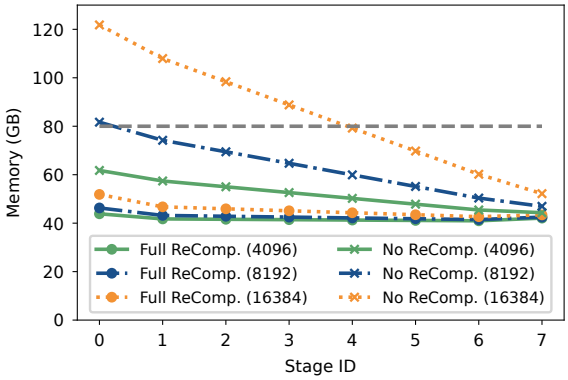


Figure 1. Simulated memory consumption for each stage during training GPT-3 with sequences of 4096, 8192, and 16384 tokens and different recomputation strategies. The dotted line is the hardware limit (80 GB) of the device. DP, TP, and PP are 1, 8, and 8, resp.

To support larger models and longer context, different parallel strategies have been proposed, including data parallelism (DP) [17, 21, 24], Tensor Parallelism (TP) [16, 20, 32, 33] and Pipeline Parallelism (PP) [12, 13]. The growing models that require supercomputer-scale hardware are leading the industry to adopt a mixture of these strategies. Pipeline parallelism is often used at the inter-node level during training. It incurs minimal communication and fits the interconnect between nodes which is normally slower than intra-node communication between accelerators.

Pipeline-parallel systems need to schedule the forward and backward passes of a large number of micro-batches to better fill the pipeline and reduce bubbles. However, computing the gradients requires intermediate results (i.e. activations) produced in the forward pass to be preserved until its corresponding backward pass is finished, bringing significant memory consumption. Even with proper scheduling, the micro-batches in one iteration inevitably result in huge and imbalanced memory usage across different stages. The state-of-the-art 1F1B scheduling mechanism [9] requires workers in stage i out of p stages to preserve the intermediates of at most $p - i$ micro-batches.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651359

To show the imbalance, we simulate the training process of the GPT-3 [3] model and present the results in Figure 1 as *no recomputation*. We also employ sequence parallelism [19] and Flash Attention [6] that can reduce the memory cost of activations and improve computation efficiency. As shown in Figure 1, the issue of memory consumption and imbalance becomes more severe with longer sequence length.

Given that pipeline parallelism distributes model parameters and ZeRO [28] distributes optimizer states, the predominant part of memory usage becomes the intermediate results. Recomputation [21] targets reducing memory usage by only saving part of the intermediate results in the forward pass. The rest will be recomputed during the backward pass. The most typical recomputation strategy for transformer models is to only preserve inputs of the decoder layers in the model, which we refer to as *full recomputation*. As Figure 1 shows, it relieves memory consumption, but at the cost of notably more computation.

Some techniques try to trade off between computation and memory consumption. Existing works [1, 2, 19, 39] either employ recomputation strategies on a coarse granularity or manually and uniformly select certain operators to be recomputed. These methods are not flexible enough to address the imbalanced memory consumption across different pipeline stages.

On the trade-off between computation cost and memory savings in recomputation, we uncover the chance to meet memory constraints at a minimal cost through recomputing partially and differently in each stage. The imbalanced memory consumption is thus turned into imbalanced computation, suggesting a further optimization that assigns a different number of layers to stages. Following this insight, we design our system, AdaPipe, with the following core ideas:

Adaptive Recomputation. AdaPipe supports finer granularity of recomputation strategies by splitting the decoder layer into finer computation units and allows different recomputation strategies in different stages. To automate the fine-grained recomputation in each stage, AdaPipe profiles the computation and memory cost of each operator and constructs a performance model considering the homogeneity of the transformer models to predict the memory consumption more precisely. We propose a dynamic programming (DP) algorithm that aims to find the optimal set of intermediate results to recompute for each pipeline stage.

Adaptive Partitioning. Adaptive recomputation turns imbalance of memory into imbalance of computation: Initial stages recompute more, while the following stages have more space to save the intermediates and perform less recomputation. Therefore, AdaPipe further adjusts the number of layers in pipeline stages, with initial stages responsible for fewer layers, and later stages more, thus balancing the computation cost. AdaPipe builds another level of DP algorithm upon the aforementioned recomputation DP results, to automatically and efficiently solve the optimizing problem of stage division.

The main contributions of AdaPipe are:

- We propose *Adaptive Recomputation*, which supports different recomputation strategies for different stages to fully utilize memory and reduce recomputation cost.
- We propose *Adaptive Partitioning* based on 1F1B pipeline scheduling mechanism to re-balance computation among stages with different recomputation strategies.
- We model the memory and time cost of different recomputation and stage partitioning strategies, and further design a two-level dynamic programming algorithm to search for the near-optimal plans combining Adaptive Recomputation and Partitioning.
- We implement AdaPipe on MindSpore and PyTorch, achieving up to 1.32× speedup on NVIDIA GPUs and 1.22× on Ascend NPUs.

The rest of this paper is organized as follows. Section 2 introduces the background of pipeline parallelism and recomputation technique. Section 3 presents the overview of AdaPipe. Adaptive recomputation and the searching algorithm are discussed in Section 4. Section 5 focuses on the cost model and the DP algorithm of adaptive partitioning. Section 6 describes the implementation of AdaPipe, and Section 7 evaluates its performance. Finally, Section 8 discusses related works, and Section 9 concludes this paper.

2 Background

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

Table 1. Notations Used in the Paper

Notation	Description
b	micro-batch size
n	number of micro-batches
L	number of layers
t	tensor parallel size
d	data parallel size (with ZeRO)
p	pipeline parallel size

2.1 Pipeline Parallelism

Pipeline parallelism involves partitioning the computation graph into sequential subgraphs and assigning subgraphs to different stages. In the forward pass, each sample will flow from the first stage to the last stage. Conversely, in the backward pass, the gradients of each sample will flow reversely. The computation of a single sample exhibits a sequential dependency across the stages, indicating that one sample cannot undergo simultaneous processing by two stages.

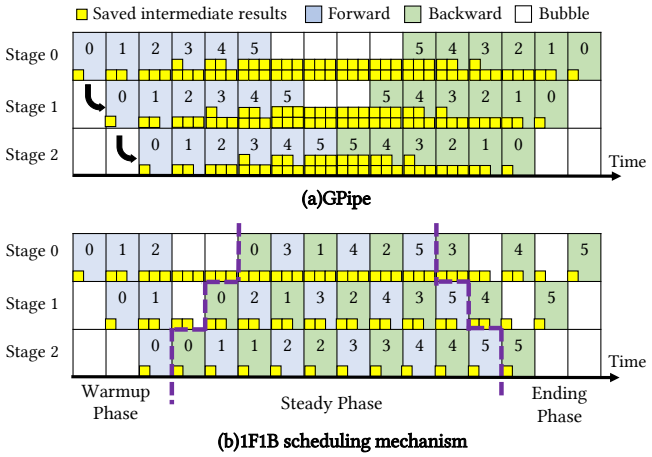


Figure 2. Scheduling mechanism of Pipeline Parallelism.

GPipe [13] employs a strategy of dividing samples into smaller micro-batches. The process begins by conducting the forward passes for all micro-batches sequentially, followed by the backward passes, as depicted in Figure 2 (a). This method allows GPipe to process different micro-batches at various stages concurrently, thereby reducing bubbles and enhancing the overall utilization of workers.

PipeDream [12] and DAPPLE [9] further propose the *one-forward-one-backward* (1F1B) scheduling mechanism, which effectively reduces memory consumption during training, as shown in Figure 2 (b). The scheduling process of 1F1B can be divided into three phases: warmup, steady, and ending phase. Given p stages and n micro-batches, workers in stage s will first perform the forward passes of $p-s$ micro-batches in the warmup phase, then process $n-p+s$ backward and forward passes alternately in the steady phase, and finally compute the backward passes of $p-s$ micro-batches in the ending phase.

The 1F1B mechanism has two main characteristics: (1) Imbalanced memory usage. Workers in the stage s are required to save the intermediate results of $p-s$ micro-batches during the steady phase. Although 1F1B reduces the memory cost from $O(n)$ (GPipe) to $O(p)$, it still results in imbalanced memory utilization across different stages. (2) Influence of the number of micro-batches on training efficiency. The number of bubbles of the 1F1B scheduling mechanism is $2p-2$, which only depends on the pipeline parallel size. When the pipeline parallel size is fixed, a larger number of micro-batches indicates a longer steady phase and higher efficiency. Since the running time of the steady phase is determined by the slowest stage, recent studies [12, 40] have proposed algorithms to partition the computation graph to minimize the maximum computation time across all stages.

Megatron-LM [25] proposes an interleaved 1F1B scheduling mechanism by splitting the computation graph into more subgraphs and assigning each stage with more than one subgraph. This method reduces the bubble ratio while bringing more communication overhead.

Chimera [22] and Hanayo [23] further introduce bidirectional pipelines that can reduce the bubble ratio. However, when the number of micro-batches exceeds the pipeline parallel size, their scheduling mechanisms tend to introduce more bubbles than the 1F1B scheduling mechanism. Additionally, these methods intensify memory pressure since model parameters are duplicated across pipeline stages.

2.2 Recomputation

The recomputation technique drops the intermediate results in the forward pass and recomputes them in the backward pass. While this approach effectively decreases the memory consumption of workers, it also leads to an increased computational burden, requiring the computation of the forward pass to be conducted again in the backward pass.

Prior study [5] reduces the memory usage of a model with L sequential layers to $O(\sqrt{L})$ by saving the activation every $O(\sqrt{L})$ layers. During the backward pass, it recomputes and saves the activations of $O(\sqrt{L})$ layers into a buffer. Once the gradients of these layers have been computed, the corresponding activations are released and the buffer is reused for the next $O(\sqrt{L})$ layers. Consequently, the size of the reserved buffer should also be proportional to $O(\sqrt{L})$.

Recent studies [1, 2] propose methods to find the best recomputation strategy for models in a chain structure with limited memory. However, since the architectures of the Attention and Feed-Forward layers in transformer models are complex, the above studies treat them as singular layers. The granularity of the recomputation strategy is thus too coarse, as both memory-intensive and computation-intensive operators exist within one layer. vPipe [39] can further divide Attention and Feed-Forward layers into finer layers by processing residual connections. Nevertheless, it is still not able to handle operators with multiple activation inputs beyond residual connections, such as the computation of the attention scores within Attention layers.

The memory consumption of activations within one decoder layer is huge, and this issue is further exacerbated when employing pipeline parallelism. When training large models with long sequences, storing all activations of decoder layers becomes impossible for workers in the first stage. Instead of saving all activations of one layer, recent work [19] proposes *selective recomputation*. It only recomputes the Softmax, Dropout, and Batch Matmal operators in each decoder layer because the activations of these ops occupy a large proportion of memory. Recently, Flash Attention [6] fuses the above operators into one operator and eliminates these memory-consuming activations naturally, which supersedes the selective recomputation strategy.

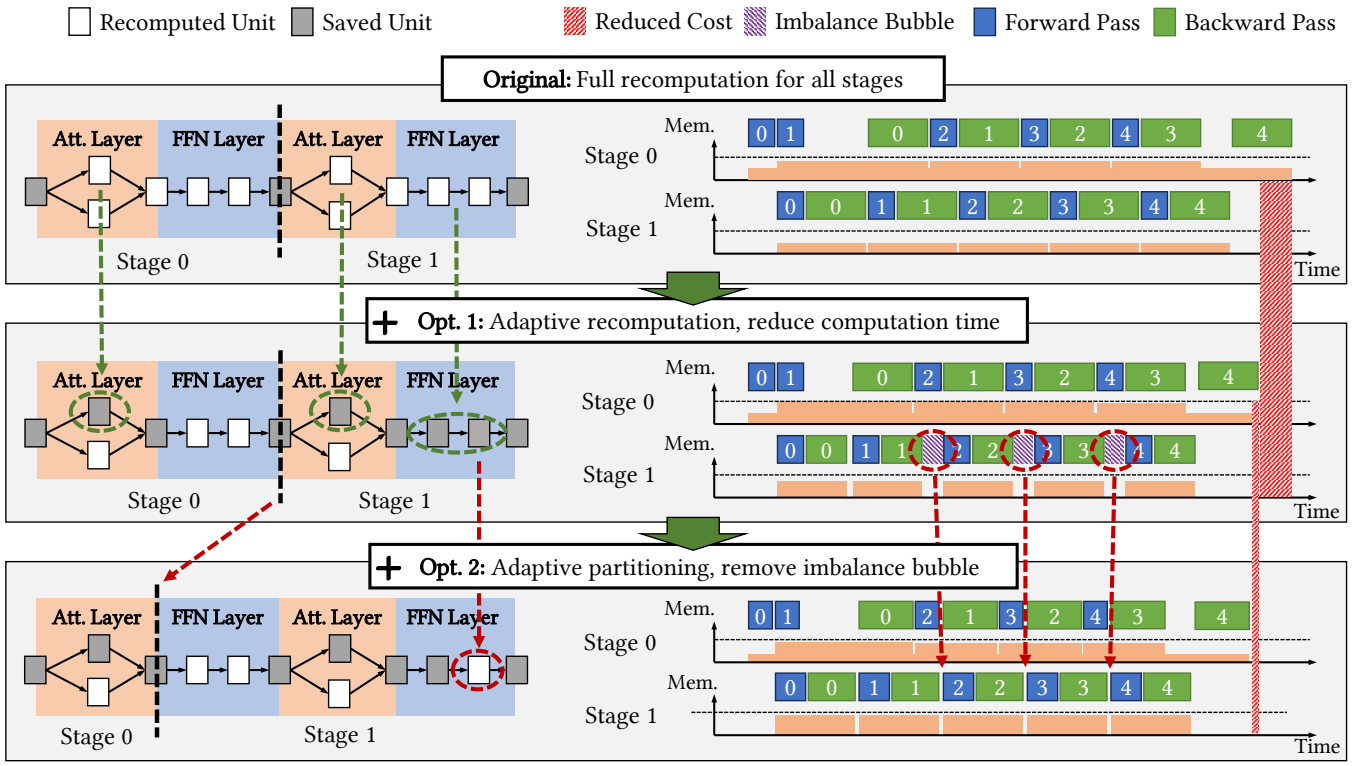


Figure 3. Overview of AdaPipe. The left part demonstrates the recomputation strategies of each unit and the partitioning strategies of the computation graph. The right shows the memory usage for intermediates and the timeline of stages.

3 Overview of AdaPipe

As mentioned in Section 2, imbalanced memory consumption exists across various stages within the pipeline parallelism framework. Specifically, with 1F1B scheduling mechanism, the last stage will have more free space for activations because they have fewer micro-batches to save, showing an opportunity to alleviate the need for recomputation.

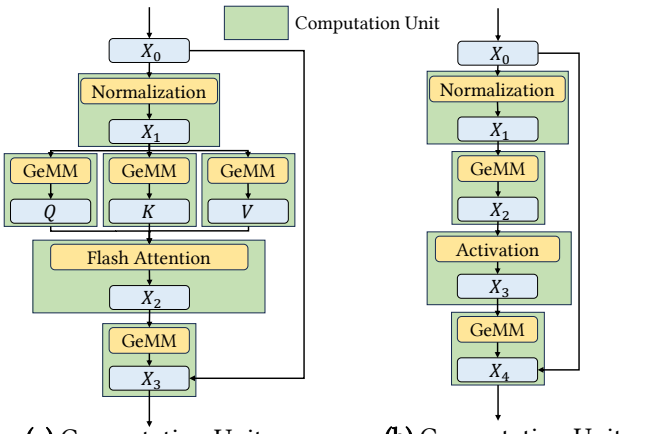
The top graph in Figure 3 shows a minimal case of 1F1B scheduling with only two stages. The layers are uniformly distributed among the stages, resulting in a well-balanced pipeline. In this configuration, all decoder layers are fully recomputed during the backward pass, ensuring minimal memory usage for intermediate results.

To demonstrate adaptive recomputation, we can exploit the remaining memory of workers in both stages as in the middle graph of Figure 3. The computation graph is split into various **computation units**, each comprising several operators that are either recomputed or saved together. For a recomputed unit, no internal intermediate tensors are preserved. On the contrary, a saved unit will get its internal and result tensors saved for backward just like without the recomputation technique. AdaPipe autonomously determines whether each unit is recomputed or saved for different stages. This results in a strategy where more intermediate results are saved in stage 0, and significantly more in stage 1, creating varied recomputation approaches for these stages. This optimized strategy enhances the performance of backward passes in stage 0 and 1, resulting in shorter warmup and ending phases. Yet the steady phase is not optimized much, because stage 0 needs to recompute more intermediates than stage 1, thus becoming the bottleneck.

In order to further enhance the efficiency of the steady phase, adaptive partitioning is employed, as illustrated in the bottom graph of Figure 3. Leveraging the sequential nature of the **layers** in transformer models, we can adjust the stage boundaries, introducing variability in the number or length of layers within each stage. Stage 0 can then transfer some layers to stage 1, thereby rebalancing and accelerating the steady phase of the pipeline. AdaPipe again automatically determines where to partition the computation graph according to the model structure and training configurations such as the number of micro-batches.

As we have mentioned in Section 2, the proportion of the steady phase of the 1F1B scheduling mechanism is determined by the number of micro-batches. When the number of micro-batches is small, adaptive recomputation contributes more to the optimization of AdaPipe, since it significantly improves the warmup and the ending phases. On the contrary, if more micro-batches are presented in one iteration, adaptive partitioning will show its effectiveness in the steady phase.

AdaPipe can optimize the strategies of recomputation and partitioning based on 3D parallelism, i.e. tensor, data, and pipeline parallelism. The tensor- and data-parallel sizes are the same across different stages. Given a 3D parallelism strategy, we first construct a performance model to analyze both the time and memory consumption, in which the time cost is optimized against the memory constraints. Then a two-level dynamic programming (DP) algorithm is proposed to optimize the problem. It is worth noting that the partitioning optimization cooperates with the previous recomputation optimization throughout the DP algorithm so that we don't fall into some local minimums. The performance model and algorithm details will be covered in the following Section 4 and Section 5. Both optimizations are automated, enabling AdaPipe to handle complicated large neural networks across different clusters and configurations.



(a) Computation Units for Attention Layer

(b) Computation Units for Feed-Forward Layer

Figure 4. Computation Units Division.

4 Adaptive Recomputation

Traditional frameworks typically employ the same recomputation strategy across diverse layers and stages. However, as highlighted in Section 2, there exists an inherent imbalance in memory usage among different stages. AdaPipe allows each stage to find the best recomputation strategy according to its memory pressure individually.

To search for the best recomputation strategy for different stages, we analyze the inner structure of the transformer models and split the layers into finer-grained **computation units**, each of which is a minimal group of operators to be recomputed or saved together. This gives us more flexible control over the trade-off between time and memory costs. Then we construct the cost model and design the algorithm to find the best recomputation strategy for different stages.

In this section, we first introduce the abstraction required for modeling the fine-grained recomputation, i.e., the computation units, and then dive into the cost model and optimization algorithm.

4.1 Computation Unit

We demonstrate the splitting of transformer layers into computation units, including the Attention layer and the Feed-Forward layer, in Figure 4. Instead of independently deciding whether to recompute each operator, we combine multiple operators when the intermediate results between them are not saved even in a non-recomputed backward pass: this happens to many operators in neural networks, including Transpose, Addition, etc. Other operators that require saving output tensors contribute to the boundaries between computation units.

Each tensor is bound to exactly one computation unit, which we refer to as its parent unit. For example, the parent unit of tensor Q includes the GEMM, addition, transpose, and division operators, in which only the GEMM operator is shown in Figure 4. If the tensor Q is decided to be saved, then these operators will not be required to recompute during the backward pass, unlike the full recomputation which will always recompute them all.

Some layers in the computation units, e.g., Flash Attention, may also save some tensors internally, together with their outputs. Such internally saved tensors are considered in our memory consumption modeling as well, which will be introduced later.

The division of the computation units in Figure 4 can be adapted for most transformer-like models, like GPT-3, Llama 2, and BERT [8].

4.2 Cost Model

As any methodology that involves a trade-off between memory and time consumption, modeling fine-grained recomputation naturally requires the consideration of both memory and time. The memory consumption will be used as the constraint, and the time cost will become the optimization target. Given a part of the whole network, we want to model the memory and time consumption as functions of the recomputation strategy, which is whether to save or to recompute each computation unit. Formally speaking, with a pipeline stage of computation units $\mathcal{U} = \{U_i\}$, we denote the recomputation strategy as a subset $\mathcal{R} = \{U \in \mathcal{U} \mid U \text{ is recomputed }\}$. We want $\text{Time}_{f|b}(\mathcal{R})$ (for both forward and backward) and $\text{Mem}(\mathcal{R})$ as our performance model.

For $\text{Time}_{f|b}(\mathcal{R})$, we first profile all computation units involved to obtain the forward and backward time of each computation unit, denoted as $\text{Time}_f(U)$ and $\text{Time}_b(U)$. Given a 3D parallelism strategy, this profiling process involves conducting a preliminary run of the training for 5 to 10 iterations and recording the timestamps before and after each computation unit. Then we can sum up to get

$$\text{Time}_f = \sum_{U \in \mathcal{U}} \text{Time}_f(U),$$

which doesn't change with regard to $\mathcal{R}$, and

$$\text{Time}_b(\mathcal{R}) = \sum_{U \in \mathcal{U}} \text{Time}_b(U) + \sum_{U \in \mathcal{R}} \text{Time}_f(U).$$

For $\text{Mem}(\mathcal{R})$, we consider the memory consumption in three parts. The first part is irrelevant to recomputation but only depends on the parallel strategy, which contains the parameters, optimizer states, and gradients. Assuming the computation graph consists of L_a Attention layers and L_f Feed-Forward layers, the number of the parameters of this stage will be $N = L_a P_a + L_f P_f$, where P_a and P_f is the parameter counts of the Attention and Feed-Forward layer correspondingly. Then the memory usage of the parameters is $2N/t$ bytes, given the 16-bit floating point precision. Similarly, we can compute the memory usage for the gradients as $2N/t$ bytes. Optimizer states are used by the optimizer to update the parameters, taking $k \times N/(td)$ bytes, where d represents the data parallel size and ZeRO stage 1 is employed. The value of k depends on the optimizer in use and its precision. For the FP32 Adam optimizer we are using, we have $k = 2 \times 4$ given the two FP32 states. Some frameworks will accumulate the gradients in FP32 precision or update the parameters in FP32 before converting them to half-precision for the next iteration. We also consider these factors when modeling the memory consumption of this part.

Then we discuss the memory required for buffering the recomputed results during the backward pass. Traditionally with full recomputation, we need to first recompute and save all intermediates within the last decoder layer before computing the gradients. After the gradients have been computed, we can drop these intermediates and perform the same actions on the preceding layer. Therefore the buffer is reused between layers and should be large enough to accommodate the intermediates within one single layer.

Heading to our fine-grained recomputation, the concrete size of the buffer will depend on the recomputation strategy. To simplify the model, we restrict the output tensors of each Attention layer and Feed-Forward layer to be saved by default, which means the last GEMM operators of these two layers won't be recomputed during the backward process. Therefore, our buffer size will never exceed the size of all intermediate results within one decoder layer. This restriction simplifies the memory model and is very efficient since the buffer size is influenced by many aspects, like the memory allocation algorithm and potentially reordered execution of the computation graph.

The last part is what we focus on, the saved intermediates. Each recomputation unit U will occupy a certain amount of memory, $\text{Mem}(U)$, for all its child tensors when configured as saved. This includes its output tensor and potentially some internal tensors, as mentioned above. In addition, we need to multiply the memory consumption of the intermediates by $p - s$, in which p is the total number of stages, and s is the number of the current stage, counting from first to last. Summarizing above, we have

$$\text{Mem}(\mathcal{R}) = \text{Const} + (p - s) \sum_{U \in \mathcal{U} \setminus \mathcal{R}} \text{Mem}(U).$$

The constant above represents the summation of all static memory consumptions. It is irrelevant to the recomputation strategy of this stage.

4.3 Solving the Optimization Problem

To find the optimal recomputation strategy $\mathcal{R}$, we need an upper bound of $\text{Mem}(\mathcal{R})$ to minimize $\text{Time}_b(\mathcal{R})$. Time_f is fixed so we can safely ignore it. As one may have noticed, the cost model constructed above naturally poses a knapsack problem, which can be efficiently solved through dynamic programming. We only need to subtract the Const memory consumption from the hardware memory limit to obtain the memory limit for intermediates. All memory limits below refer to the subtracted one.

Assuming there are N computation units in the stage s and the memory limit is M. $\tilde{T}_{s,i}(m)$ denotes the **maximum saved** recomputation cost with the first i units under the memory limit m for stage s, or formally speaking

$$\tilde{T}_{s,i}(m) = \max_{(p-s)\sum_{U_j \in \mathcal{U} \setminus \mathcal{R}, 0 \le j < i} \text{Mem}(U_j) < m} \sum_{U_j \in \mathcal{U} \setminus \mathcal{R}, 0 \le j < i} \text{Time}_f(U_j). \tag{1}$$

then we can have the following state-transition equation in Equation (2), where $i \in [1, N]$ and $m \in [1, M]$.

$$\begin{aligned} \tilde{T}_{s,i}(m) = \max(&\tilde{T}_{s,i-1}(m), \\ &\tilde{T}_{s,i-1}(m - (p - s)\,\text{Mem}(U_i)) \\ &+ \text{Time}_f(U_i)) \end{aligned} \tag{2}$$

With the above DP algorithm, we can compute the maximal saved recomputation cost as $\tilde{T}_{s,N}(M)$. Therefore, we can further compute the fixed forward time and minimal backward time of the computation graph G for stage s as

$$\begin{aligned} F_{G,s} &= \sum_U \text{Time}_f(U) \\ B_{G,s} &= \sum_U (\text{Time}_b(U) + \text{Time}_f(U)) - \tilde{T}_{s,N}(M). \end{aligned}$$

The results will be used in the adaptive partitioning algorithm detailed in Section 5.

5 Adaptive Partitioning

Extending the 1F1B scheduling mechanism, we further propose the adaptive partitioning algorithm aimed at adjusting the length of each stage, as detailed in Section 3. We treat the transformer models as a sequence of *layers*. These layers encompass the Embedding layer, the Attention layer, the Feed-Forward layer, and the Decoding Head layer. We then partition the sequence and assign each worker with one stage of a sub-sequence of layers. Such partitioning strategies will

not introduce extra communication between pipeline stages for homogeneous neural networks like transformers, since the tensors between layers are of the same shape.

The rest of this section will discuss how we model the cost of the 1F1B scheduling mechanism with regard to the partitioning strategy, and how we optimize the target to obtain the optimized partitioning strategy for the stages.

5.1 Cost Model

In Section 4, we have addressed the optimal time consumption of a given subgraph at stage s for both forward and backward pass as F_s and B_s. As is covered in Section 2, one iteration under the 1F1B scheduling mechanism consists of three phases: warmup, steady, and ending. We denote the maximum time of these phases from stage s to the last stage as W_s, S_s, and E_s.

As shown in Figure 2, for the warmup phase, W_s starts from the forward pass of the first micro-batch to the beginning of the first backward pass of the stage s. For instance, in Figure 3, W_1 only has one forward pass, while W_0 has two forward passes and one bubble. In this way, we can use Equation (3) to compute the final warmup time for all stages.

$$W_{s-1} = \max(W_s + B_s, (p-s)F_{s-1}) + F_{s-1} \tag{3}$$

Similarly, the ending phase starts from the end of the forward pass of the last micro-batch to the backward pass of the last micro-batch. We can use the same method to compute the time of the ending phase for all stages.

The steady phase S_s consists of $n-p+s$ forward and backward passes from stage s to the last stage. During the steady phase, each stage will communicate with the previous stage before the forward pass and communicate with the next stage before the backward pass. Therefore, the running time is dominated by the maximum summation of the forward time and backward time of these stages. We denote the maximum forward and backward time summation from stage s to the last stage as M_s. Then we can use the following equations to get the whole steady time of all stages:

$$\begin{aligned} M_{s-1} &= \max(M_s, F_{s-1} + B_{s-1}), \\ S_s &= (n-p+s)M_s. \end{aligned}$$

Then the total time of the 1F1B scheduling method is $W_0 + E_0 + S_0$. In this way, we build an accurate cost model for the 1F1B scheduling mechanism.

5.2 Solving the Optimization Problem

We further design another level of dynamic programming (DP) algorithm upon the previous algorithm in adaptive recomputation, to find the optimized partitioning strategy with the above cost model.

We use $f[s,i,j]$ and $b[s,i,j]$ to represent the minimal forward and backward time of the layer sequence from layer i to j for stage s. The arrays $f[s,i,j]$ and $b[s,i,j]$ can be obtained by performing the DP algorithm in Section 4 for different stages on different layer sequences. The output of our algorithm, $P[s,i]$, is the best partitioning strategy for the layer sequence from layer i to the last layer with stages from s to the last. Moreover, $P[s,i]$ is a state consisting of warmup time W, maximum forward and backward time M, ending time E, forward time F of the stage s, backward time B of stage s, and total time T. The algorithm is shown in Algorithm 1.

Algorithm 1: Algorithm for adaptive partitioning.

Input: $f[s,i,j]$: fwd. time of layers $i \sim j$ as stage s
Input: $b[s,i,j]$: bwd. time of layers $i \sim j$ as stage s
Output: $P[s,i]$: the best result from layer i to the last layer, containing W, E, M, F, B, T

```
for s = p − 2 to 0 do
  for i = L − p + s to 0 do
    for j = i to L − p + s do
      W ← f[s,i,j] + max(P[s+1,j+1].W
                         + P[s+1,j+1].B,
                         (p − s − 1) f[s,i,j])
      E ← b[s,i,j] + max(P[s+1,j+1].E
                         + P[s+1,j+1].F,
                         (p − s − 1) b[s,i,j])
      M ← max(P[s+1,j+1].M,
              f[s,i,j] + b[s,i,j])
      F ← f[s,i,j]
      B ← b[s,i,j]
      T ← W + E + (n − p + s)M
      if T < P[s,i].T then
        P[s,i] ← W, E, M, F, B, T
      end if
    end for
  end for
end for
```

With the above DP algorithm, AdaPipe can find the optimized partitioning strategy for the 1F1B scheduling mechanism automatically.

5.3 Complexity Analysis and Optimization

The complexity of Algorithm 1 is $O(pL^2)$. However, this algorithm necessitates the computation of $f[s,i,j]$ and $b[s,i,j]$, which are derived from the DP algorithm illustrated in Section 4. This requires executing the algorithm across all stages for every sub-sequence of layers from layer i to layer j, with $0 \le i \le j \le L$, resulting in a complexity of $O(pL^2)$.

The complexity of the DP algorithm in Section 4 is $O(mN)$, where m is the memory constraint and N is the number of computational units. Since the quantity of computational units increases linearly with the number of layers, the complexity can also be written as $O(mL)$. Therefore, the total complexity of the above searching process is $O(pL^2 + mpL^3)$.

Given that transformer models such as GPT-3 consist of homogeneous layers, some sub-sequences of layers can be isomorphic. For instance, the subgraph of layers 3–4 is isomorphic to that of layers 5–6, as they both have one Attention layer followed by a Feed-Forward layer. More generally, the subgraphs with the same number of layers and the same type of the initial layer (Attention layer or Feed-Forward layer) are isomorphic. By leveraging the isomorphism among subgraphs, we can reduce the execution number of the DP algorithm in Section 4 to $O(pL)$.

Furthermore, as sizes of activation tensors are typically times to some power of 2, we can find the greatest common divisor (GCD) of the memory costs of all units. Then we divide these memory costs and the limit m in Equation (2) by their GCD to reduce m and accelerate the DP algorithm.

With all aforementioned optimizations, the overall computational complexity of the searching process is $O(m'pL^2)$, where m' is m divided by GCD as discussed. For typical models like GPT-3 and Llama 2, the entire search process takes only seconds.

6 Implementation

AdaPipe consists of a search engine and an execution engine. The search engine first profiles the forward time and backward time of each computation unit. Then it performs the algorithms described above based on the training configurations and memory capacity. AdaPipe will find the optimized partitioning strategy with different recomputation strategies for each stage.

The execution engine supports hybrid parallelism strategies and different recomputation strategies for each computation unit across all stages. It will train large-scale models distributedly on the cluster with the obtained recomputation strategy and partitioning strategy.

We implement AdaPipe on top of two deep learning frameworks, MindSpore [14] and PyTorch [27]. MindSpore is an AI framework that will compile the computation graph before executing. It provides interfaces for users to partition the computation graph and decide whether operators should be recomputed, with which we implement the execution engine. For PyTorch, we implement the engine on Megatron-LM [33]. We rewrite the decoder layer to support the fine-grained recomputation strategies of AdaPipe.

7 Evaluation

7.1 Experimental setup

We evaluate AdaPipe on two clusters.

Cluster A. It has 8 nodes of NVIDIA DGX-A100 server, each equipped with 8 × A100 80GB Accelerators (GPUs), 64 CPU cores in 2 sockets, and 2 TB memory. The eight GPUs are connected with NVLink. Nodes are interconnected with 800 Gbps Infiniband HCAs.

Cluster B. It has 32 nodes of Huawei Atlas 800 server, each equipped with 8 × Ascend 910 32GB Accelerators (NPUs), 192 CPU cores in 4 sockets, and 2TB memory. In each node, 8 NPUs are installed on two NPU boards, and the 4 NPUs on each board are fully meshed by 30 GB/s links in all directions. Besides, each NPU directly connects to the CPUs through one PCIe 4.0 x16 link (32GB/s) and is equipped with one 100 Gbps NIC for inter-node networking.

Models and workloads. We evaluate AdaPipe on two representative models, GPT-3 (175B) [3] and Llama 2 (70B) [35]. The micro-batch size is set to 1 to save the memory of intermediate results. We change the sequence length and global batch size while keeping other parameters fixed in our experiments. Besides, when we double the sequence length, we will half the global batch size to keep the total number of tokens processed in one iteration unchanged.

The detailed configuration is listed in Table 2. On cluster A, evaluations of GPT-3 and Llama 2 are performed with 64 and 32 GPUs. On cluster B, we conduct both small-scale experiments using 256 and 128 NPUs, and large-scale experiments using 2048 and 1024 NPUs for GPT-3 and Llama 2, respectively. Additionally, all experiments conducted on Cluster A employ the PyTorch framework, whereas those on Cluster B use the MindSpore framework.

Table 2. Configurations Used in Evaluation.

Model	Cluster	#Dev	(Sequence Length, Global Batch Size)
GPT-3	A	64	(4096, 128), (8192, 64) (16384, 32)
	B	256	(4096, 256)
	B	2048	(4096, 2048)
Llama 2	A	32	(4096, 128), (8192, 64), (16384, 32)
	B	128	(4096, 256)
	B	1024	(4096, 1024)

Baseline. We select DAPPLE [9] and Chimera [22] as the baselines. These baselines are measured with full recomputation and no recomputation, denoted with the suffix *-Full* and *-Non*, respectively. Hanayo [23] is not measured as it cannot handle the situation where the number of micro-batches exceeds pipeline stages.

We evaluate the above baselines with Megatron-LM [25], the state-of-the-art framework for training large-scale models. It supports data parallelism, tensor parallelism, and pipeline parallelism. For data parallelism, it supports the ZeRO-1 technique, which partitions the optimizer state to reduce memory usage without increasing communication overhead.

For pipeline parallelism, it implements the synchronous gradient descent algorithm of the 1F1B scheduling mechanism, as proposed in DAPPLE [9]. Additionally, Megatron also integrates the Flash Attention technique.

We integrate the pipeline scheduling method of Chimera into Megatron since its original implementation doesn't support tensor parallelism. We also implement the forward doubling technique of Chimera to reduce the bubble when the number of micro-batches exceeds the number of stages.

We will evaluate DAPPLE, Chimera, and Chimera with forward doubling (denoted as ChimeraD) in our experiment. We also evaluate AdaPipe without the adaptive partitioning optimization, denoted as Even Partitioning. It will keep the same partitioning strategy as the baselines.

For cluster A, we will iterate all possible 3D parallelism strategies, and report the best performance among these strategies. The 3D parallelism requires the same data and tensor parallel size for each stage. Table 3 lists some examples of the 3D parallelism strategies. For Chimera and ChimeraD, we further iterate the number of replicated pipelines based on the 3D parallelism strategies and ensure the number is at least 2. We also require the tensor parallel size not larger than 8, as tensor parallelism across nodes will introduce massive communication in the network and significantly impact the overall performance.

For cluster B, we only experiment AdaPipe, Even Partitioning, and DAPPLE with certain parallel strategies from our experience, because the number of 3D parallelism strategies for 256 NPUs is large and the compilation for each parallel strategy takes around an hour using MindSpore.

We run each task for 20 iterations and measure the average iteration time of the last 2 iterations. For cluster A, we adjust the value of the initial loss scale to ensure there is no overflow and the optimizer state is allocated before the last 2 iterations.

7.2 End-to-End Performance

We profile the iteration time of AdaPipe, Even Partitioning, and baselines on clusters A and B. The speedup over DAPPLE with full and no recomputation is marked on each column in Figure 5, Figure 6 and Figure 7.

Llama 2 on cluster A Figure 6 shows that DAPPLE-Non achieves better performance than DAPPLE-Full by reducing the computation overhead. As DAPPLE-Non stores the activations of all micro-batches, it consumes more memory and exceeds the limit when the sequence length is 16384.

The basic scheduling units of Chimera only consist of p micro-batches, where p is the pipeline parallel size. When the number of micro-batches exceeds the number of pipeline stages, Chimera will concatenate two or more scheduling units. The forward passes of the second unit will occupy the bubbles at the end of the first unit. Given that the computation time of the backward pass is typically larger than that of the forward pass, this concatenation leads to bubbles between two consecutive scheduling units. Consequently, Chimera performs worse than DAPPLE as shown in Figure 5.

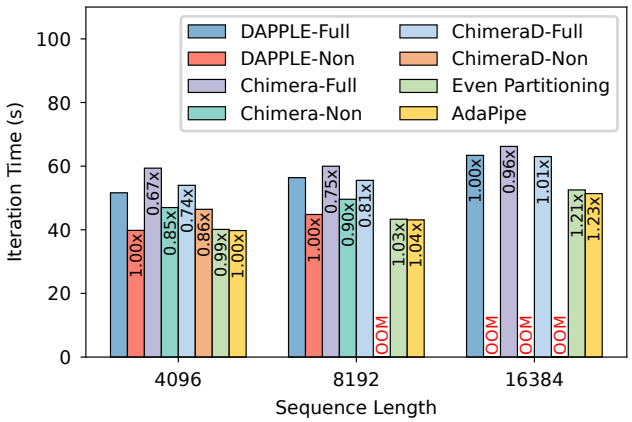


Figure 5. End-To-End Performance of Llama 2 (Cluster A).

ChimeraD alleviates the above problem by only doubling the micro-batch size of all forward passes, to equalize the computation of the forward and backward pass. Despite this adjustment, when full recomputation is employed, the computation time of the backward pass is still longer than that of the forward pass, and the bubbles between two consecutive scheduling units still exist. As a result, ChimeraD only outperforms DAPPLE when the number of micro-batches is small, as it indicates fewer scheduling units and bubbles within one iteration.

Furthermore, as ChimeraD needs to process two samples in each forward pass, the memory required to store intermediate results is consequently doubled when no recomputation is applied. As Figure 5, ChimeraD-Non exceeds the memory limit when the sequence length grows to 8192.

For cluster A, the memory capacity of GPUs is 80GB and is large enough to accommodate the intermediate results of DAPPLE-Non when the sequence length is 8192. In this situation, DAPPLE-Non is only feasible when the tensor parallel size is 8. However, AdaPipe and Even Partitioning achieve better performance when the tensor parallel size is 4. Detailed analysis of how different parallel strategies influence the performance is presented in Section 7.3.

When the sequence length grows to 16384, the size of intermediate results also becomes larger. On the one hand, DAPPLE-Non exceeds the memory limit under all parallel strategies. On the other hand, DAPPLE-Full cannot fully utilize the memory, leaving much memory unused. In this situation, AdaPipe and Even Partitioning both find the recomputation strategy that can fully utilize the memory and minimize the computation. Therefore, AdaPipe and Even Partitioning gains 1.23× and 1.21× speedup over the baselines.

GPT-3 on cluster A GPT-3 has 175 billion parameters and consumes more memory for intermediate results than Llama

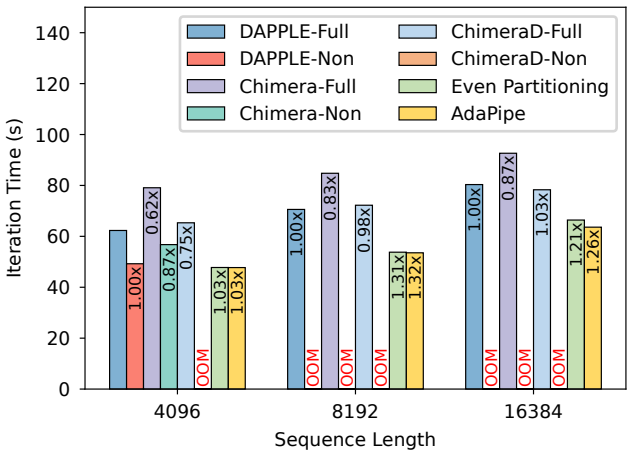


Figure 6. End-To-End Performance of GPT-3 (Cluster A).

2. When we extend the sequence length to 8192 and 16384, all baselines with no recomputation exceed the memory capacity under all parallel strategies. However, Even Partitioning and AdaPipe can still find better recomputation strategies according to the remaining memory size and achieve up to 1.32× and 1.31× speedup on cluster A.

As AdaPipe can repartition the computation graph based on the optimized computation cost to balance the computation cost, it outperforms Even Partitioning in most situations, especially when the sequence length is long and the memory is limited.

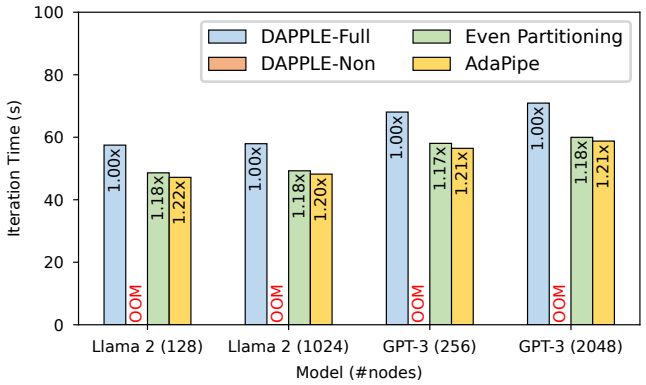


Figure 7. End-To-End Performance (Cluster B).

Cluster B The performance of DAPPLE, AdaPipe, and Even Partitioning on cluster B is shown in Figure 7. For GPT-3, both the tensor parallel size and pipeline parallel size are configured to 8. For Llama 2, the tensor parallel size is set to 4, while the pipeline parallel size is set to 8. In the experiment, we linearly scale the global batch size in proportion to the data parallel size on thousands of NPUs.

In contrast to the A100 GPUs, the memory capacity of the Ascend 910 is only 32GB, which significantly intensifies the memory constraints for storing intermediate results. For Llama 2 and GPT-3, DAPPLE-Non exceeds the memory limit when the sequence length is 4096, as shown in Figure 7. With the technique of adaptive recomputation, Even Partitioning and AdaPipe can fully exploit the available memory, minimizing the computation and enhancing the training efficiency. As a result, AdaPipe and Even Partitioning achieve up to 1.22× and 1.18× speedup over the baselines. Furthermore, AdaPipe and Even Partitioning both exhibit good weak scalability as shown in Figure 7.

7.3 Influence of Different Parallel Strategies

Table 3. Iteration Time of GPT-3 with Different Parallel Strategies on Cluster A (Sequence Length = 4096).

TP, PP, DP	DAPPLE-Full	DAPPLE-Non	Even Partitioning	AdaPipe
(1, 32, 2)	76.769	OOM	OOM	OOM
(2, 16, 2)	65.732	OOM	54.784	53.142
(2, 32, 1)	68.237	OOM	53.430	53.500
(4, 8, 2)	62.307	OOM	49.875	47.732
(4, 16, 1)	63.407	OOM	47.761	47.779
(8, 4, 2)	66.625	49.233	50.966	50.431
(8, 8, 1)	67.152	49.363	49.946	49.911

We list the iteration time of different parallel strategies for GPT-3 with sequence length as 4096 on cluster A in Table 3. Strategies that exceed the memory capacity for all methods are not listed. The iteration time of the best parallel strategies is marked red.

Parallel strategies influence the overall performance from different aspects. Smaller TP size reduces communication within one server and enhances the computation efficiency of operators as tensors have larger shapes.

However, when the number of devices is fixed, a smaller TP results in a larger DP or PP, which would adversely impact the overall efficiency. As shown in Figure 2, the bubble ratio of 1F1B scheduling mechanism can be represented as $\frac{p}{n}$, where p denotes the pipeline parallel size and n denotes the number of micro-batches. On the one hand, a larger PP would add more bubbles in the 1F1B scheduling mechanism as the number of micro-batches is unchanged. On the other hand, a larger DP means the number of micro-batches for a single data parallel group is decreased, therefore the bubble ratio is also increased.

If the benefits from less communication and higher computation efficiency outweigh the influence of a larger bubble ratio, then a smaller TP will gain better performance, like the strategy (4, 8, 2) and (8, 8, 1) in Table 3. Otherwise, a larger TP would be better, like the strategy (2, 16, 2) and (4, 8, 2) in Table 3.

We find that AdaPipe and Even Partitioning exceed the memory limitation when the strategy is (1, 32, 2). This issue stems from two primary factors. Firstly, AdaPipe restricts

the output tensors of both the Attention layer and the Feed-Forward layer to be saved, leading to more memory consumption than DAPPLE-Full even when all other computation units are recomputed. Secondly, the size of the output tensors can be very large when the tensor parallel size is 1.

7.4 Memory and Computation Analysis

We profile the memory usage and computation time of workers in each stage for the GPT-3 model on cluster A. Figure 8 and Figure 9 show the result with a sequence length of 16384 and parallelism strategy of (8, 8, 1).

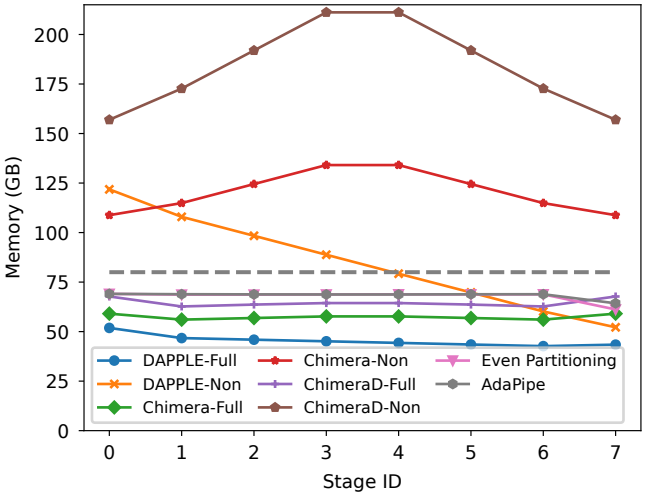


Figure 8. Peak Memory Usage of Each Stage. The memory capacity is represented by the grey dashed line.

We report the peak memory usage during the training process. Although DAPPLE, Chimera, and ChimeraD all exceed the memory limitation without recomputation, we can still estimate the peak consumption of these baselines.

For DAPPLE with full recomputation, it can be observed from Figure 8 that the first and the last stages consume more memory than the middle stages. The extra memory usage comes from the Embedding and Decoding Head layer at the front and the end of the model. Additionally, the peak memory of the middle stages decreases linearly with the stage number. From the profiling result, more than 30GB of memory is wasted by workers of all stages.

For DAPPLE without recomputation, workers of stage 0 require more than 80GB memory, which exceeds the capacity of the GPUs, rendering it unusable. Moreover, the peak memory of DAPPLE-Non is extremely imbalanced among different stages, where the memory consumption of the first stage is 2.33× that of the last stage.

For Chimera and ChimeraD, profiling is conducted with two replicated pipelines. As model parameters are replicated across two pipeline stages, Chimera-Full and ChimeraD-Full consume more memory than DAPPLE-Full.

For Chimera-Non and ChimeraD-Non, workers of the middle stages need to store the intermediate results of more micro-batches, consequently consuming more memory than workers at the initial and final stages, as depicted in Figure 8. Besides, ChimeraD-Non computes two samples at each forward pass and saves more amount of intermediate results than Chimera-Non.

AdaPipe and Even Partitioning support adaptive recomputation and allow fine-grained control of recomputation strategies over the computation units. Table 4 shows the number of saved computation units and layer numbers in each stage. The Embedding layer and Decoding Head layer are each counted as the extra layer in the first and last stages, respectively. The number of saved computation units increases with the stage ID because earlier stages need to save the intermediate results of more micro-batches. Figure 8 shows that each stage of AdaPipe and Even Partitioning occupies around 70GB of memory in a balanced manner. This is due to the conservative setting of the memory constraint at 70GB when executing the DP algorithm. The memory constraint can be elevated for better performance.

Table 4. Configuration of Recomputation and Stage Partitioning Produced by AdaPipe and Even Partitioning.

Method		Stages							
AdaPipe	Saved Units	39	48	50	55	68	70	85	124
	# of Layers	23	23	23	24	25	25	25	26
Even Part.	Saved Units	27	48	50	55	62	69	84	120
	# of Layers	25	24	24	24	24	24	24	25

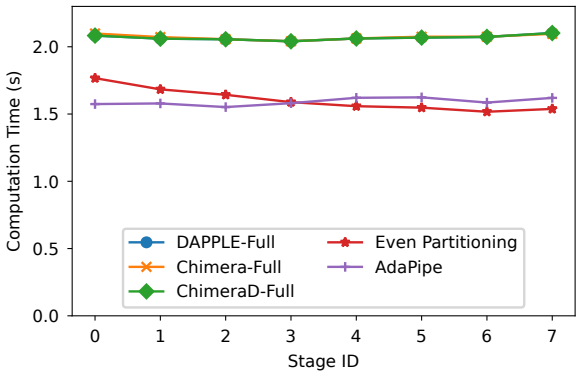


Figure 9. Computation Time of Each Stage.

Figure 9 shows the micro-step time of each stage, which is the summation of the forward and backward time of one

micro-batch. All stages have similar computation times for DAPPLE-Full, Chimera-Full, and ChimeraD-Full, for their computation workload of different stages is almost the same.

For Even Partitioning, the micro-step time decreases with a larger stage ID. As demonstrated before, workers in the front stages save fewer intermediates than workers in the subsequent stages. Therefore front stages conduct more recomputation work and have longer micro-step time. Figure 9 shows that the micro-time of the slowest stage is 1.17× that of the fastest stage, which influences the efficiency of the steady phases.

AdaPipe further partition the computation graph of the 1F1B schedule adaptively to balance the computation time of each worker. Table 4 shows that AdaPipe moves layers from earlier stages to later stages, therefore balancing the computation time of these stages, as shown in Figure 9. With this method, AdaPipe shortens the time of the steady phase and improves the overall performance.

7.5 Convergence Validation

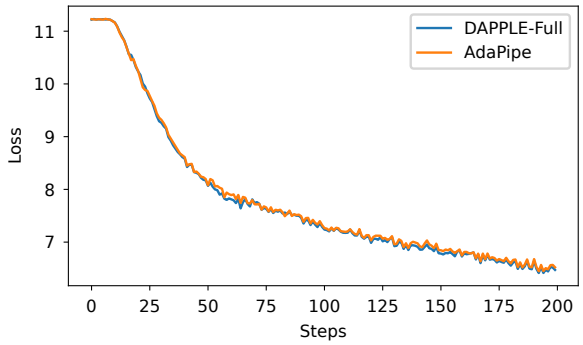


Figure 10. Loss Curve of Different Strategies.

We evaluate the convergence of AdaPipe and DAPPLE-Full on cluster B using the Llama 2 model. AdaPipe only reduces the repeated computation without changing the computation of each operator, therefore exhibiting similar convergence as the baseline, as depicted in Figure 10. The difference between the two loss curves comes from the different initialization of the parameters, as the graph partition strategy of AdaPipe is different from that of the baseline.

8 Related Work

Memory Optimization Techniques. Several studies have been conducted to address the memory problem when training large models. The offloading technique enables the training of large models with limited GPU memory by utilizing the host memory or disk storage. ZeRO-Offload [30] offloads the optimizer states from the GPU to the host memory. vDNN [31] employs host memory as the external buffer and properly schedules the prefetching and offloading operations to overlap the memory movement and computation. SuperNeurons [37] further combines the offloading and recomputation technique, which offloads activations of operators with heavy computation and recomputes operators with cheap computation. ZeRO-Infinity [29] leverages the storage of NVMe SSDs to train large models. However, the offloading technique incurs huge communication between the CPU and GPU. As the computational power of the GPUs grows significantly in recent years, especially with emerging high-performance Tensor Cores, it is more difficult to effectively overlap the communication with computation.

Considering the memory imbalance in the pipeline parallelism, BPipe [18] transfers the intermediate activations between GPUs across different stages to balance memory usage. However, this method incurs extra communication, and the tensor parallel size is limited as the first stage needs to be placed on the same node as the last stage. MPress [41] combines the recomputation and the offloading technique and allows different memory strategies in different stages, but their work only supports training within one single server.

Auto Parallelism. Different parallelism strategies would influence memory consumption and efficiency when training large models, as shown in Table 3. AdaPipe adopts 3D parallelism strategies. However, the tensor parallel and data parallel sizes can be different for each pipeline stage. Some existing works focus on the automated search for optimal parallelism strategies. Flexflow [15] and Tofu [38] design cost models and use DP algorithms to find the best intra-operator parallelism strategy. PipeDream [12] proposes a DP algorithm that splits the computation graph into balanced subgraphs. Unity [36] further combines algebraic transformation with parallelization. Piper [34] and Alpa [40] propose a hierarchical parallelism that incorporates data and tensor parallelism at the lower level while employing pipeline parallelism at the upper level. However, none of them consider the recomputation technique in their search space.

9 Conclusion

We propose AdaPipe, which uses adaptive recomputation and adaptive partitioning to find the optimized stage partitioning and recomputation strategies for pipeline parallelism during training large models. Evaluations on representative models show that AdaPipe accelerates the end-to-end training performance by up to 1.32× on NVIDIA GPUs and 1.22× on Ascend NPUs.

10 Acknowledgment

We would like to thank our shepherd and all anonymous reviewers for their insightful comments and feedback. We also thank Zhipu AI for sponsoring computation resources. This work is partially supported by the National Natural Science Foundation of China under grant U20B2044 and Peng Cheng Laboratory under grant PCL2022A05. Wenguang Chen is the corresponding author.

A Artifact

A.1 Artifact Description

Our artifact includes the source code of AdaPipe, instructions for preparing software dependencies, and scripts to reproduce the results on NVIDIA GPUs in Section 7. You can download it from Zenodo*.

- **Program:** AdaPipe and Megatron-LM
- **Compilers:** GCC 9.5.0, CUDA 12.1.0
- **Dataset:** Enwik8
- **Hardware:** 8 × A100 / A800 80GB GPUs and NVLink
- **Metrics:** Iteration time, maximum memory allocation
- **Disk space required:** ~100GB
- **Time needed to prepare:** ~6 hours
- **Time needed to run experiments:** 2 days

A.2 Installation

The detailed instructions for installing dependencies are explained in the `README.md` file of the `AdaPipe` directory.

Prepare Python environment. Install Anaconda to manage Python packages and create an environment with Python 3.10 for evaluation.

Prepare compilers and other packages. We use *spack* to install GCC 9.5.0, CUDA 12.1.0, and OpenMPI 5.0.2.

Install Python dependencies The Python package dependencies for AdaPipe are included in the `requirements.txt` of the `AdaPipe` directory.

A.3 Experiment Workflow

Before running the experiment, create a `hostfile` for OpenMPI, modify the path to the dataset and the IP address of the master server in the scripts for GPT-3 and Llama2, which are located in the `examples` directory.

We provide the script `global_test.sh` for conducting experiments of AdaPipe and Even Partitioning as shown in Figure 5 and Figure 6. It will invoke `gpt_experiment.sh` and `llama2_experiment.sh` with different training configurations like sequence length and global batch size. These scripts will iterate through all parallelism strategies and execute the profiling, searching, and measuring process of AdaPipe and Even Partitioning.

We also provide similar scripts for running baseline experiments in the `Megatron-LM` directory.

A.4 Evaluation and Expected Results

The results of the experiments can be found in the directories named `gpt_result` and `llama2_result`. These directories include results of different training configurations and parallelism strategies. The stdout of the training process and the logs for each worker are also recorded. The output files record the iteration time and loss at each step, while the log files capture the timestamps and memory information of each forward and backward pass.

We also provide the script `collect_result.py` to display a comprehensive summary of all experiments. The expected results for AdaPipe and baselines can be found in the file `expected_result.txt` in the `AdaPipe` and `Megatron-LM` directory.

*https://doi.org/10.5281/10.5281/zenodo.10877925

References

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

[2] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient combination of rematerialization and offloading for training dnns. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 23844–23857, 2021.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[4] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation, 2023.

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[7] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, Rodolphe Jenatton, Lucas Beyer, Michael Tschannen, Anurag Arnab, Xiao Wang, Carlos Riquelme, Matthias Minderer, Joan Puigcerver, Utku Evci, Manoj Kumar, Sjoerd van Steenkiste, Gamaleldin F. Elsayed, Aravindh Mahendran, Fisher Yu, Avital Oliver, Fantine Huot, Jasmijn Bastings, Mark Patrick Collier, Alexey A. Gritsenko, Vighnesh Birodkar, Cristina Vasconcelos, Yi Tay, Thomas Mensink, Alexander Kolesnikov, Filip Pavetic, Dustin Tran, Thomas Kipf, Mario Lucic, Xiaohua Zhai, Daniel Keysers, Jeremiah Harmsen, and Neil Houlsby. Scaling vision transformers to 22 billion parameters. *CoRR*, abs/2302.05442, 2023.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[9] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao,

Xiaoyong Liu, and Wei Lin. DAPPLE: a pipelined data parallel approach for training large models. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 431–445. ACM, 2021.

[10] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23:120:1–120:39, 2022.

[11] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, et al. Pre-trained models: Past, present and future. *AI Open*, 2021.

[12] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018.

[13] 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. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[14] Ltd. Huawei Technologies Co. Huawei mindspore ai development framework. In *Artificial Intelligence Technology*, pages 137–162. Springer, 2022.

[15] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2279–2288. PMLR, 2018.

[16] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[17] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 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)*, pages 463–479, 2020.

[18] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. Bpipe: Memory-balanced pipeline parallelism for training large language models. 2023.

[19] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022.

[20] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

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

[22] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[23] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 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*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 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*, pages 401–416, 2020.

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 58. ACM, 2021.

[26] OpenAI. Gpt-4 technical report, 2023.

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

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

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

[30] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 551–564. USENIX Association, 2021.

[31] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 18:1–18:13. IEEE Computer Society, 2016.

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

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[34] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 24829–24840. Curran Associates, Inc., 2021.

[35] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa,

Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[36] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022.

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

[38] Minjie Wang, Chien-Chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 26:1–26:17. ACM, 2019.

[39] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training. *IEEE Trans. Parallel Distributed Syst.*, 33(3):489–506, 2022.

[40] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[41] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. 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)*, pages 556–569, 2023.