AdaPtis: Reducing Pipeline Bubbles with Adaptive Pipeline Parallelism on Heterogeneous Models

Jihu Guo*
Fudan University & Shanghai AI Laboratory
Shanghai, China
guojihu@pjlab.org.cn

Tenghui Ma*
Fudan University & Shanghai AI Laboratory
Shanghai, China
matenghui@pjlab.org.cn

Wei Gao†
Hong Kong University of Science and Technology
Hong Kong, China
csgaowei@ust.hk

Peng Sun
Shanghai AI Laboratory
Shanghai, China
sunpeng@pjlab.org.cn

Jiaxing Li
Shanghai AI Laboratory
Shanghai, China
lijiaxing@pjlab.org.cn

Xun Chen
SenseTime
Shenzhen, China
chenxun@sensetime.com

Yuyang Jin
Tsinghua University
Beijing, China
jinyuyang@tsinghua.edu.cn

Dahua Lin
Chinese University of Hong Kong & Sensetime Research
Hong Kong, China
dhlin@ie.cuhk.edu.hk

Abstract

Pipeline parallelism is widely used to train large language models (LLMs). However, increasing heterogeneity in model architectures exacerbates pipeline bubbles, thereby reducing training efficiency. Existing approaches overlook the co-optimization of model partition, model placement, and workload scheduling, resulting in limited efficiency improvement or even performance degradation. To respond, we propose AdaPtis, an LLM training system that supports adaptive pipeline parallelism. First, we develop a pipeline performance model to accurately estimate training throughput. Second, AdaPtis jointly optimizes model partition, model placement, and workload scheduling policies guided by this performance model. Third, we design a unified pipeline executor that efficiently supports the execution of diverse pipeline strategies. Extensive experiments show that AdaPtis achieves an average speedup of 1.42× (up to 2.14×) over Megatron-LM I-1F1B across various LLM architectures and scales.

1 Introduction

Large Language Models (LLMs) have demonstrated impressive performance across a broad range of domains [3, 7, 21, 30, 51, 58]. However, this rapid advancement comes with a steep increase in training cost [7, 16, 57], highlighting that even a modest reduction in training time (e.g., 5%) can yield substantial cost savings [7, 32, 48, 63].

Pipeline Parallelism (PP) [14] is critical in accelerating LLM training [7, 30, 36]. A typical pipeline involves three phases. ***1) Model Partition***: the model is split into a sequence of stages, each comprising a group of consecutive layers. ***2) Model Placement***: these stages are mapped to devices such as GPUs, and the inter-device data dependencies, where the computation of each stage depends on the result from the preceding stage [14], are accordingly established. ***3) Workload Scheduling***: the forward and backward passes of each micro-batch are scheduled across the assigned stages following a predefined policy, while respecting the data dependencies. Due to the inherent inter-device data dependency, pipeline bubbles (i.e., device idle time) are inevitable [14].

To reduce the bubble ratio, many PP methods [8, 13, 15, 19, 23, 28, 30, 33, 36, 40, 50, 54, 62, 63] have been proposed to optimize the *static pipeline*, which remains model partition, model placement, and workload scheduling fixed even when training configurations (e.g. micro-batch size, number of model layer, number of stages) change, such as S-1F1B [47] and GPipe [14]. These methods can be regarded as *partially adaptive pipelines*, since each optimizes an individual phase of the pipeline while keeping the others static. For example, I-1F1B [36] adapts model placement by dividing stages into smaller *virtual stages*; ZB [40] tunes workload scheduling by reordering the computation of micro-batches; and Mist [63] adjusts model partition by changing the number of layers per stage. In all cases, only one dimension of the pipeline is adapted, while the remaining dimensions remain fixed in the style of S-1F1B [47]. Such approaches are effective primarily on relatively homogeneous model architectures (e.g., LLaMA-2 [53], GPT-3 [3]), where computational and memory demands are well balanced across stages.

However, modern models are increasingly heterogeneous [1, 2, 27, 30, 35, 52], and this heterogeneity significantly increases pipeline bubbles. For example, Gemma [52] has a substantially larger vocabulary than LLaMA-2 [53] (256K vs.

*Equal contribution
†Corresponding author

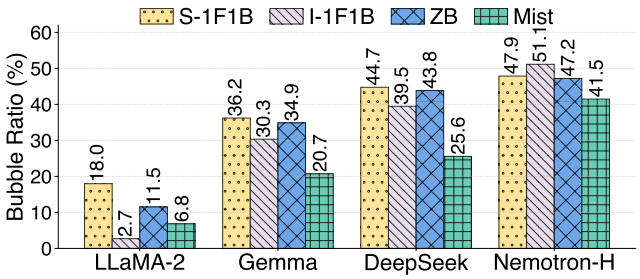

Figure 1. Bubble ratios of pipeline parallelism methods on different models. The model and training configurations are set to $L = 32, P = 4, T = 2, G = 16, nmb = 16$ on 8 GPUs.

32K). Beyond vocabulary size, DeepSeek [30] integrates both Feed-Forward Network (FFN) and Mixture-of-Experts (MoE) layers [20], while Nemotron-H [2] combines Self-Attention (SA) [55] with Mamba [10], leading to even greater architectural heterogeneity. As shown in Figure 1, existing PP methods including I-1F1B [36], ZB [40], and Mist [63] suffer from higher bubble ratios on Gemma [52], DeepSeek [30], and Nemotron-H [2] compared to LLaMA-2 [53].

We also observe from Figure 1 that, compared with the widely used S-1F1B [47], prior partially adaptive PP methods often deliver only marginal improvements and sometimes even degrade efficiency on heterogeneous models. A key reason is that they optimize only one aspect of pipeline parallelism while overlooking the co-optimization of model partition, model placement, and workload scheduling. For example, I-1F1B [36] adapts model placement alone but leaves model partition and workload scheduling untuned. This partially adaptive pipeline shows a greater bubble ratio than the static pipeline S-1F1B [47], as illustrated on Nemotron-H [2] in Figure 1. These results suggest that efficient training of heterogeneous models requires an adaptive pipeline parallelism that co-optimizes model partition, model placement, and workload scheduling, rather than tuning them individually (as analyzed in § 3.2).

Nevertheless, co-optimizing model partition, placement, and workload scheduling in adaptive pipeline parallelism on heterogeneous models presents challenges: *1)* co-optimization is highly complex, since the optimization of each part influences the others; *2)* co-optimizing these three phases results in an exponentially large search space; and *3)* the pipeline executor should be able to efficiently handle the complicated communication and computation in diverse pipelines.

To address these challenges, we propose `AdaPtis`, an LLM training system that enables adaptive pipeline parallelism that co-optimizes the model partition, model placement, and workload scheduling. First, we build a Pipeline Performance Model (§4.2) that provides accurate estimates of computation, communication, and memory costs (including detailed bubble time and overlap time) for arbitrary combinations of model partition, model placement, and workload scheduling policies. Second, Pipeline Generator (§4.3) jointly optimizes the model partition, model placement, and workload scheduling policies with the performance estimation of Pipeline Performance Model. Pipeline Generator efficiently explores the large search space, starting with predefined settings for model partition, placement, and workload scheduling. It tunes the performance bottleneck phase per iteration to progressively optimize pipeline performance, thereby accelerating pipeline generation. Third, to support adaptive execution, `AdaPtis` implements a unified Pipeline Executor (§4.4) that carefully orchestrates computation and communication instructions based on the model partition, model placement, and workload scheduling policies generated by Pipeline Generator, providing efficient pipeline execution.

We implement `AdaPtis` atop PyTorch [38] and conduct extensive experiments across different scales of Gemma [52], DeepSeek [29], and Nemotron-H [2]. The results show that `AdaPtis` consistently outperforms prior PP policies, including S-1F1B [47], I-1F1B [36], ZB [40], and Mist [63], achieving average speedups of 1.34× (up to 1.54×), 1.42× (up to 2.14×), 1.34× (up to 1.51×), and 1.20× (up to 1.27×), respectively.

Table 1. Symbols used in this paper.

Sym.	Description	Sym.	Description
L	Number of layers	D	Data parallelism size
S	Number of stages	T	Tensor parallelism size
H	FFN hidden size	P	Pipeline parallelism size
V	Vocabulary size	E	Expert parallelism size
G	Global batch size	nmb	Number of micro-batches
F	Forward computation	B	Input gradient computation
		W	Parameter gradient computation

2 Background

2.1 Distributed Training Parallelisms

Table 1 lists relevant notations used in this paper.

Data Parallelism (DP) [24, 46] splits the input training data into smaller mini-batches. Each device maintains a copy of parameters and takes its assigned mini-batches as input. When applying sharding techniques [43, 61], the redundant memory overhead can be reduced.

Tensor Parallelism (TP) [47] partitions a layer into smaller layer shards across multiple GPUs. TP alleviates the GPU memory pressure but compensates for heavy collective communication (e.g., `all-reduce`). Hence, TP is usually adopted within a single node to avoid cross-node communication [36].

Pipeline Parallelism (PP) [14] introduces point to point (P2P) communication to efficiently support cross node communication. As shown in Figure 2, PP consists of three phases, including model partition, model placement, and workload scheduling. First, the model is divided into sequential pipeline stages, each comprising a set of consecutive layers. Second, these stages are assigned to P groups of devices. Intermediate results are transmitted via P2P communication between devices hosting adjacent stages. Third, the input mini-batch is further subdivided into micro-batches, which are processed

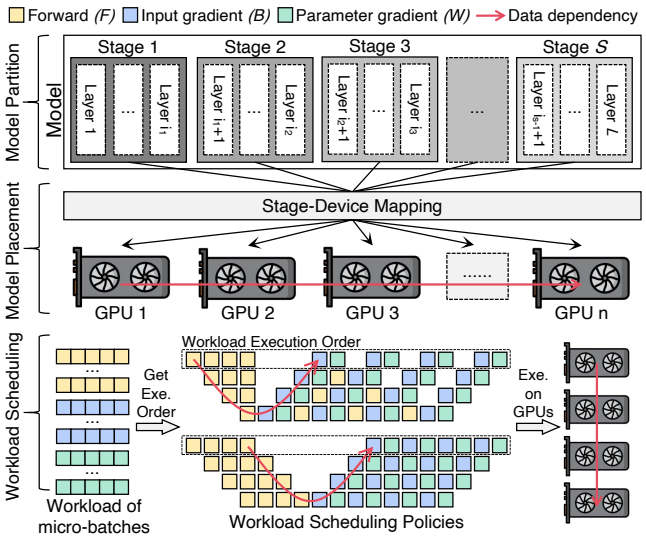

Figure 2. Illustrations of Model Partition, Model Placement, and Workload Scheduling in Pipeline Parallelism.

in a pipelined manner across all stages to improve device utilization. We next introduce these three phases of PP.

2.2 Model Partition

At the beginning of pipeline parallelism, the model is partitioned into multiple stages. A common strategy is to evenly allocate transformer layers across stages, with the input layer assigned to the first stage and the output layer to the final stage [8, 13, 15, 30, 33, 36, 40, 42, 47]. With the growing heterogeneity of model architectures, this model partition scheme inevitably leads to an increasing size of bubbles. To reduce bubbles, recent approaches adjust the number of layers per stage [50, 54, 62, 63]. These methods formulate the model partition task as an Integer Linear Programming (ILP) problem, which is solved through dynamic programming [50, 54] or ILP solvers [6, 12]. While these approaches mitigate imbalance to some extent, methods such as Mist [63] still exhibit substantial bubbles when training heterogeneous models like Nemotron-H [2], as illustrated in Figure 1.

2.3 Model Placement

Model placement refers to mapping the pipeline stages onto physical devices, which determines the inter-device data dependencies. A widely adopted strategy is to sequentially map the stages to devices [8, 13, 15, 40, 50, 54, 62]. This approach assumes that the number of stages equals the pipeline parallelism size ($S = P$), with each stage assigned to a unique device. To reduce bubbles at the beginning and end of each training step, I-1F1B [36] introduces the *virtual pipeline stages*, which split pipeline stages into smaller ones. Hanayo [33] builds upon this idea but applies a wave-like stage-to-device mapping policy. Other approaches, such as Chimera [23] and DualPipe [30], duplicate model parameters to form multiple pipelines, allowing for the concurrent execution of multiple micro-batches across model replicas. However, the same benefits can be achieved through *virtual pipeline stages* without the memory redundancy introduced by parameter duplication [33]. All these methods rely on manually defined model placement policies (e.g., manually setting the number of *virtual pipeline stages* in I-1F1B [36], pipelines in Chimera [23], or waves in Hanayo [33]), which limits their adaptivity.

2.4 Workload Scheduling

Based on the inter-device data dependencies defined by model placement, the workloads, consisting of the forward pass (F) and the backward pass of each micro-batch, are scheduled according to a predefined execution order. When the backward splitting strategy [30, 37, 40] is applied, the backward pass is further divided into input gradient computation (B) and parameter gradient computation (W).

The execution order can be manually designed, as in GPipe [14] and S-1F1B [47], or automatically generated, as in ZB [40] and Tessel [28]. Manually designed orders are fixed; therefore, they can not adapt to configuration changes. Automatic scheduling methods model the problem as a Job Shop Scheduling Problem (JSSP), which is NP hard, and employ ILP Solvers to obtain workload scheduling policies. However, the large search space makes finding the optimal workload scheduling infeasible in practice. For example, Tessel [28] restricts the search space to a small set of candidate patterns, which limits flexibility and performance. ZB [40] only reschedules W to fill bubbles, leaving other stages underutilized. DynaPipe [15] focuses on scheduling micro batch execution to mitigate data heterogeneity, but it lacks mechanisms to address model architectural heterogeneity.

3 Motivation

3.1 Increasing Heterogeneity in Model Architectures

Nowadays, the evolution of LLMs shows a clear trend toward increasing architectural heterogeneity [1, 2, 11, 20, 30, 35, 58]. Instead of relying solely on standard transformer blocks [55], modern LLMs integrate diverse attention and FFN mechanisms. For example, DeepSeek [11, 29, 30] adopts dense FFNs in the first k layers (with k manually defined) and sparse MoE layers in the later layers, and replaces SA [55] with MLA [30]. Nemotron-LLaMA [1] introduces variable FFN sizes within transformer blocks, while Minimax-M1 [35] alternates between Lightening Attention [41] and SA. Nemotron-H [2] combines Mamba [10] and SA to form a heterogeneous attention design, and Jamba [27] goes further by integrating SA, Mamba, FFN, and MoE layers into a single architecture. Beyond the hidden layers, heterogeneity also emerges in the output layer, as vocabulary sizes expand rapidly in models such as Gemma [52], LLaMA-3 [7], and Qwen [58].

This growing heterogeneity in model architectures results in larger pipeline bubbles compared to previous models, such as LLaMA-2 [53], as shown in Figure 1. This result highlights

Table 2. Taxonomy of existing pipeline parallelism approaches. ● : support, ○ : not support.

	Model Partition Tuning	Workload Scheduling Tuning	Model Placement Tuning	Co-optimization
S-1F1B [47]	○	○	○	○
Alpa [62]	●	○	○	○
Metis [54]	●	○	○	○
AdaPipe [50]	●	○	○	○
Mist [63]	●	○	○	○
ZB [40]	○	●	○	○
DynaPipe [15]	○	●	○	○
Tessel [28]	○	●	○	○
Mario [32]	○	●	○	○
I-1F1B [36]	○	○	●	○
Chimera [23]	○	○	●	○
Hanayo [33]	○	○	●	○
AdaPtis (Ours)	●	●	●	●

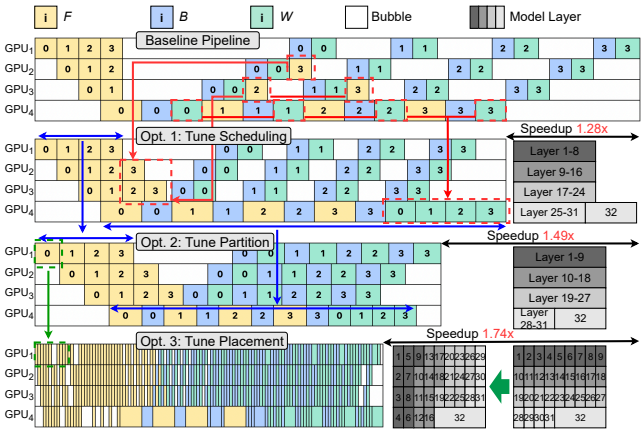

Figure 3. [Motivation]. Illustration of co-optimizing workload scheduling, model partition, and model placement for accelerating training on a heterogeneous model with a large vocabulary size (e.g. Gemma [52]) with $L = 32, P = 4, nmb = 4$.

the necessity of adaptive pipeline parallelism to mitigate bubbles and improve training efficiency.

3.2 Demand for Adaptive Pipeline Parallelism

Reducing pipeline bubbles in heterogeneous models calls for adaptive pipeline parallelism with co-optimization of model partition, placement, and workload scheduling. However, prior approaches typically optimize only a single aspect of the pipeline, and thus lack the co-optimization of all three dimensions, as summarized in Table 2.

Figure 3 presents a case study of co-optimizing model partition, model placement, and workload scheduling on a Gemma-like model [52]. We use the mainstream S-1F1B pipeline [47] as the baseline. First, we optimize workload scheduling under the baseline model partition and model placement by advancing F and B, then delaying W within the memory constraint, which yields a 1.28× speedup compared to the baseline. Second, since GPU_4 exhibits longer execution time than other GPUs, we redistribute the layers

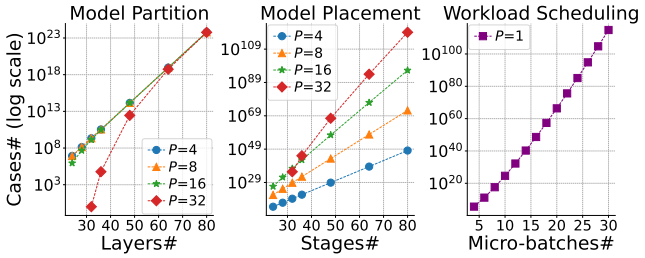

Figure 4. [Motivation]. The vast search space calls for an efficient pipeline generation method.

to tune the model partition and mitigate the computational imbalance among devices, achieving a 1.49× speedup over the baseline. Finally, we refine model placement to make each computation finer-grained and apply workload scheduling optimization on the new model partition and model placement. These optimizations substantially reduce pipeline bubbles, resulting in a 1.74× overall speedup compared to the baseline. This case study demonstrates that adaptive pipeline parallelism, achieved through the co-optimization of model partition, placement, and workload scheduling, significantly contributes to reducing bubbles and improving training efficiency on heterogeneous models.

3.3 Challenges in Pipeline Generation and Execution

Generation: The large search space poses a fundamental challenge to the efficiency of co-optimizing model partition, model placement, and workload scheduling. As shown in Figure 4, the number of cases for model partition, model placement, and workload scheduling policies grows exponentially with the number of layers, stages, and micro-batches, respectively. This explosive growth in the search space makes the DP- [50, 54, 62] or ILP-based [28, 32, 40, 62] methods incur impractically long search time (Section §5.6). To make the search tractable, some approaches [28, 40] manually restrict the search space. However, such restrictions risk excluding high-performance pipeline configurations.

Execution: To enable adaptive pipeline parallelism, a unified executor is essential. Prior approaches typically rely on execution engines tailored to specific workload scheduling policies [8, 36, 47, 50, 63]. Although some methods support adaptive workload scheduling [15, 28, 32, 40], co-optimizing the pipeline between model partition, model placement, and workload scheduling substantially increases the complexity of communication dependencies. Consequently, these partially adaptive methods are unable to adapt to diverse model partition and model placement policies. Therefore, an effective executor should be flexible enough to support diverse model partition, model placement, and workload scheduling policies, while remaining efficient in orchestrating fine-grained computation and communication.

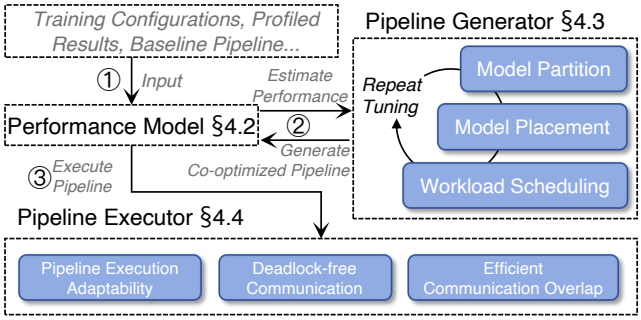

Figure 5. AdaPtis system design overview.

Table 3. Symbols used in pipeline performance model.

Symbol	Description
C_s	Computation cost of stage s.
M_s	Memory cost of stage s.
C_d	Computation cost of device d.
M_d	Memory cost of device d.
M_d^{capacity}	Memory capacity of device d.
T_d	Runtime on device d.
A_d	Activation memory on device d.
G_d	Gradient memory on device d.
Layers(s)	Set of layers assigned to stage s.
Stages(d)	Set of stages assigned to device d.
BubbleTime(d)	Total pipeline bubble time on device d.
OverlapTime(d)	Time overlap on device d.

4 AdaPtis Design

4.1 Overview

Figure 5 shows the workflow of AdaPtis: ① Pipeline Performance Model evaluates the performance of a given pipeline under specified input parameters. ② Pipeline Generator leverages these estimates to iteratively tune model partition, model placement, and workload scheduling, and then generates the co-optimized pipeline. ③ Pipeline Executor executes the co-optimized pipeline while applying communication optimizations to further improve efficiency.

4.2 Pipeline Performance Model

To support the co-optimization of model partition, model placement, and workload scheduling, we construct a fine-grained Pipeline Performance Model. It takes training settings, profiled data, and pipeline configurations as input and outputs the execution time of each device (e.g., T_d) and its memory cost (e.g., M_d). These outputs contain detailed execution metrics such as BubbleTime(d), OverlapTime(d), A_d, and G_d, which are used to estimate pipeline performance and guide subsequent tuning (§4.3). The notations used in this section are summarized in Table 3, and Algorithm 1 presents the performance modelling procedure in three **Steps**.

Step 1: Given a model partition, each stage is assigned a different number and type of layers, each associated with profiled computation and memory costs. The total computation and memory cost of a stage is determined by accumulating the cost of executing a specific computation type (e.g., F, B, or W) across all layers assigned to that stage.

Step 2: Given a model placement, the computation and memory cost of each device is obtained by summing the costs of all stages assigned to it. In addition, communication time and the overlapping time between communication and computation are considered, both of which are determined by the workload scheduling results.

Step 3: Given the model partition, model placement, and workload scheduling results, Pipeline Performance Model simulates the execution behavior of each device and identifies when and where device idle time (BubbleTime(d)) occurs, and detects opportunities where communication can be overlapped with computation (OverlapTime(d)). Based on the simulation results, Pipeline Performance Model provides feedback for Pipeline Generator (§4.3) to refine model partition, model placement, and workload scheduling, thereby reducing pipeline bubbles and increasing overlap time.

Algorithm 1 Pipeline Performance Model

Input: *Pipeline conf., Training conf., Profiled data.*
Output: List of T_d, M_d

/* **Step 1: Layer-level cost aggregation** */
for each stage s **do** ▷ *Model Partition*
  $C_s \leftarrow \sum_{l \in \text{Layers}(s)} \text{ProfiledCompCost}(l)$
  $M_s \leftarrow \sum_{l \in \text{Layers}(s)} \text{ProfiledMemCost}(l)$
end for

/* **Step 2: Stage-level cost aggregation** */
for each device d **do** ▷ *Model Placement*
  $C_d \leftarrow \sum_{s \in \text{Stages}(d)} C_s + \text{ProfiledCommCost}(s)$
  $M_d \leftarrow \sum_{s \in \text{Stages}(d)} M_s$
end for

/* **Step 3: Runtime and memory estimation** */
for each device d **do** ▷ *Workload Scheduling*
  $T_d \leftarrow C_d + \text{BubbleTime}(d) - \text{OverlapTime}(d)$
  $A_d \leftarrow \text{TotalActMem}(d)$
  $G_d \leftarrow \text{TotalGradMem}(d)$
  $M_d \leftarrow M_d + A_d + G_d$
end for

4.3 Pipeline Generator

Pipeline Generator is designed to optimize the pipeline by leveraging the performance estimation of Pipeline Performance Model and then generating pipelines with co-optimized model partition, model placement, and workload scheduling.

Optimization Objective. The pipeline execution time is determined by the slowest device, which has the maximum T_d among all devices. In addition, the memory usage M_d on each

device must not exceed its memory capacity M_d^{capacity}. Then the pipeline optimization objective is defined as follows:

$$\min \quad \max_d T_d \tag{1}$$

$$\text{s.t.} \quad M_d^{\text{capacity}} \geq M_d \quad \forall d \tag{2}$$

Efficient Exploration of Pipelines. As illustrated in Figure 6, Pipeline Generator reduces the search space by selecting representative baseline pipelines and using them as starting points for optimization. For model partition, we adopt the policies from S-1F1B [47] and Mist [63]. For model placement, the baselines include S-1F1B [47], I-1F1B [36], and Hanayo [33]. For workload scheduling, we consider S-1F1B [47] and ZB [40]. The performance of these baselines is estimated by the Pipeline Performance Model, which enables Pipeline Generator to prune low-performing candidates, thereby accelerating the exploration of pipelines.

From a chosen baseline, Pipeline Generator iteratively tunes the model partition, model placement, and workload scheduling using performance estimations from the Pipeline Performance Model. In each iteration, Pipeline Generator identifies the bottleneck phase of the pipeline (e.g., the phase contributing the largest BubbleTime(d)) and tunes it accordingly. This phase-by-phase tuning method avoids the combinatorial explosion of jointly searching model partition, model placement, and workload scheduling policies, while directly addressing pipeline performance bottlenecks. Since model partition or model placement changes affect execution time and inter-device dependencies, workload scheduling is adjusted in tandem. If a tuning step degrades pipeline performance, it is rolled back, and alternative adjustments are attempted. The tuning process repeats until no further improvement can be achieved, at which point Pipeline Generator outputs the co-optimized pipeline.

Model Partition Tuning. As shown in **Step 1** of Algorithm 1, the model partition determines the computation and memory cost of each stage. Previous works [54, 62, 63] focus on balancing C_d across devices without considering other factors that affect execution time. In contrast, our method incorporates the impact of model placement and workload scheduling, since the device execution time T_d depends not only on C_d but also on BubbleTime(d) and OverlapTime(d). Specifically, as illustrated in Figure 6(a), for each model partition policy, we perform workload scheduling, analyze BubbleTime(d) on devices, and then tune the model partition by transferring layers from the stage with the lowest bubble ratio to the stage with the highest bubble ratio. This process continues until the difference in BubbleTime(d) across devices is smaller than the maximum C_s.

Model Placement Tuning. Pipeline Generator also tunes model placement to reduce bubbles. As shown in Figure 6(b), first, Pipeline Generator adjusts the stage-device mapping by permuting the assignment of layers of each stage across

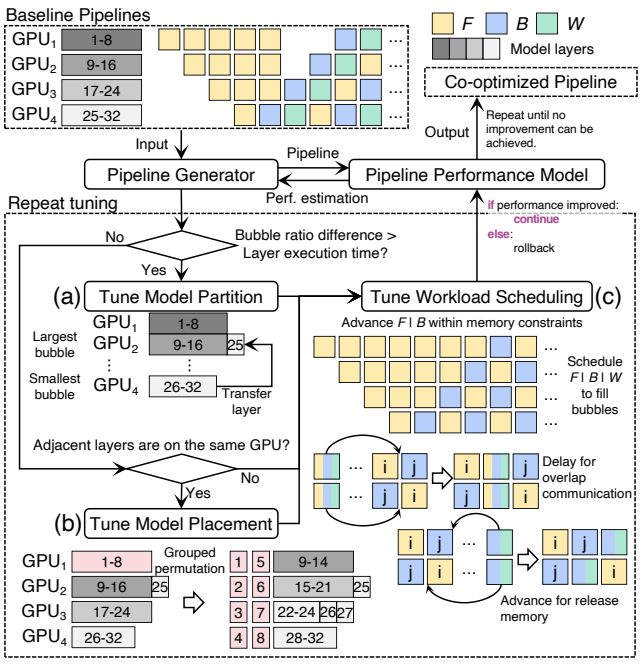

Figure 6. Illustrations of repeat tuning (a) model partition, (b) model placement, and (c) workload scheduling.

devices, placing adjacent layers on different devices. This permutation increases the effective number of pipeline stages, resulting in more fine-grained computation tasks. Second, Pipeline Generator performs workload scheduling on the permuted model placement while keeping the model partition fixed. If BubbleTime(d) is reduced after permutation, the new placement is accepted as the updated baseline, and the process is repeated. Otherwise, the permutation is rolled back. To accelerate model placement tuning, Pipeline Generator employs grouped permutations, which simultaneously permute all layers on a stage, rather than one layer at a time.

Workload Scheduling Tuning. Figure 6(c) shows three policies in tuning workload scheduling. First, given the model partition and placement, Pipeline Generator advances the F first, followed by the B, and finally the W within the memory constraints to fill bubbles. This workload scheduling strategy is based on how data dependencies are handled. Specifically, both the F pass and B involve inter-device data dependencies. For instance, the F on stage i depends on the result from stage $i-1$, and the B pass on stage i depends on the result from stage $i+1$. In contrast, W has no inter-device data dependencies since it relies only on B on the same device; thus, W is suited for filling pipeline bubbles.

Second, Pipeline Generator adopts an overlap-aware scheduling policy to increase OverlapTime(d). The key idea is to avoid scheduling dependent computation tasks consecutively and instead delay certain computations to enable communication overlap. For example, consider two devices that need to exchange intermediate results of F_i and B_j. If the subsequent computations that depend on these results are

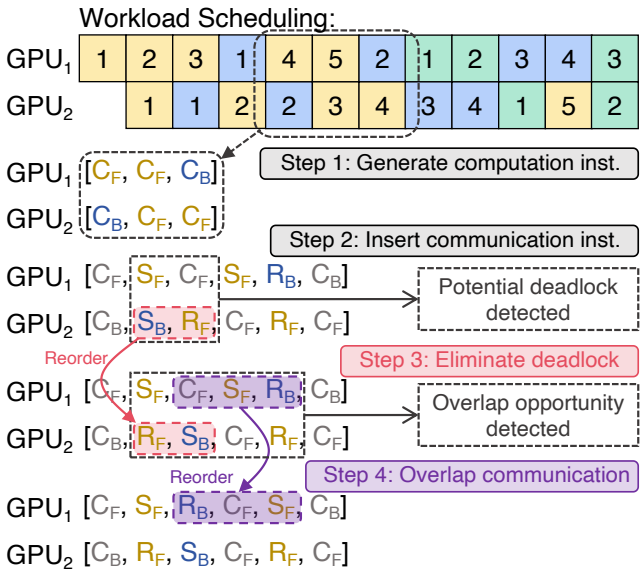

Figure 7. Illustration of pipeline execution instruction generation and communication optimizations.

scheduled immediately, communication and computation are forced to proceed sequentially. In contrast, by first executing other independent computations and delaying the dependent ones, communication can be overlapped with computation. This overlap-aware policy not only increases OverlapTime(d) but also reduces BubbleTime(d).

Third, to meet the Constraint 2, Pipeline Generator advances some B and W to advance the memory release time and avoid Out-of-Memory (OOM) issues. During workload scheduling, Pipeline Generator records the workload scheduling results and identifies potential OOM time. It then advances the execution of the latest B and W to ahead of this time to free up memory, continuing this process until all potential OOM errors are resolved.

4.4 Pipeline Executor

The design goals of Pipeline Executor are supporting pipeline execution adaptability and efficiency.

Table 4. Pipeline Execution Instructions.

Instructions	Descriptions
`compute_F|B|W` ($C_{F\|B\|W}$)	Computation of F, B, W.
`send_F|B_start` ($S_{F\|B}$)	Build P2P comm. on `sender`.
`receive_F|B_start` ($R_{F\|B}$)	Build P2P comm. on `receiver`.
`wait_F|B_receive` ($W_{F\|B}$)	Asynchronous comm. on `receiver`.

Pipeline Execution Adaptability. To execute the pipeline under various workload scheduling policies, Pipeline Executor abstracts computation and communication into a set of instructions (summarized in Table 4) and orchestrates them to realize specific pipeline execution. As for model partition and model placement, Pipeline Executor employs a flexible mechanism that enables flexibility in both stage division and device mapping. The model can be partitioned into stages with varying numbers of layers, and the total number of stages can be tuned according to model partition results. The stage-to-device mapping is also adaptable, allowing different numbers of stages to be assigned per device.

As illustrated in **Step 1** of Figure 7, Pipeline Executor generates computation instruction lists based on the workload scheduling results. In this step, the original execution order of F, B, and W in the workload scheduling policy is preserved on each device to ensure data dependencies are maintained.

In **Step 2**, communication instructions ($S_{F|B}$, $R_{F|B}$, and $W_{F|B}$) are inserted into the instruction lists. Specifically, for each computation instruction, Pipeline Executor first checks whether it requires intermediate results from other devices. If so, a $R_{F|B}$ instruction is inserted before the computation instruction. A corresponding $W_{F|B}$ instruction is inserted between $R_{F|B}$ and $C_{F|B}$ to ensure that the required data has been received before computation begins (omitted in Figure 7 for clarity). Next, Pipeline Executor checks whether the computation instruction generates intermediate results that need to be transferred to other devices. If so, a $S_{F|B}$ instruction is arranged right after the computation instruction to begin intermediate data transmission immediately.

Deadlock-free Communication. Naively inserting $S_{F|B}$ and $R_{F|B}$ instructions before and after computation instructions may lead to deadlock. Since $S_{F|B}$ and $R_{F|B}$ must be executed synchronously on both the data sender and receiver sides, an improper execution order can cause a communication deadlock. For example, as illustrated in **Step 2** of Figure 7, GPU_1 executes S_F after C_F, while GPU_2 performs S_B after C_B. This execution order leads to a cross-dependency situation in which both GPU_1 and GPU_2 are waiting for the corresponding $R_{F|B}$, resulting in both sides being blocked.

To address this issue, Pipeline Executor traverses all communication instructions on each device and checks whether each $S_{F|B}$ on the sender and its corresponding $R_{F|B}$ on the receiver are ordered correctly. If a pair of communication instructions is identified as potentially leading to deadlock, Pipeline Executor reorders them to ensure deadlock-free execution, as demonstrated in **Step 3** of Figure 7.

Efficient Communication Overlap. To realize efficient communication overlap, Pipeline Executor carefully reorders communication instructions to align with the pipeline execution behavior defined by the workload scheduling. Due to inter-device data dependencies, computation and communication operations are often interleaved, naturally enabling opportunities for overlap. However, naively inserting communication instructions before the corresponding computation instructions can miss such opportunities.

For example, as shown in **Step 3** of Figure 7, on GPU_1, the R_B instruction is placed directly before C_B. In this case, the communication cannot be overlapped with computation because C_B depends on the data received by R_B.

To improve overlap, Pipeline Executor traverses the instruction list to identify an earlier insertion point for R_B. As illustrated in **Step 4** of Figure 7, Pipeline Executor finds an ideal position before C_F. Since there is no data dependency between R_B and C_F, this reordering enables GPU_1 to perform C_F while asynchronously executing R_B, thus increasing the overlap between communication and computation.

5 Evaluation

5.1 Experimental Settings

Implementation. We implement AdaPtis on top of Pytorch [38] with approximately 14,000 lines of Python code.
Testbed. We conduct experiments on a cluster equipped with 128 NVIDIA H800 GPUs. Each node connects 8 GPUs via NVLink, while inter-node communication is handled through InfiniBand. The training performance is tested on Gemma [52], DeepSeek [30], and Nemotron-H [2]. The detailed model parameter configurations are listed in Table 5.

Table 5. Model parameter configurations.

Model	Size	L	V	H	FFN Type	Attn. Type
Gemma [52]	Small	32	256K	1536	FFN	SA
	Medium	64	512K	1536	FFN	SA
	Large	128	1024K	1536	FFN	SA
DeepSeek [30]	Small	16	128K	2048	FFN+MoE	MLA
	Medium	32	256K	2048	FFN+MoE	MLA
	Large	64	512K	2048	FFN+MoE	MLA
Nemotron-H [2]	Small	28	128K	1024	FFN	SA+Mamba
	Medium	56	256K	1024	FFN	SA+Mamba
	Large	112	512K	1024	FFN	SA+Mamba

Baselines. To ensure fair and reproducible comparisons, we integrate the open-source implementations of selected baselines into our framework for evaluation against AdaPtis. **1) S-1F1B** [47], a widely adopted PP strategy in mainstream training frameworks such as Megatron-LM [47] and DeepSpeed [42]. **2) I-1F1B** [36] optimizes the model placement of S-1F1B by leveraging *virtual pipeline stages* to reduce pipeline bubbles. **3) ZB** [40], a pipeline approach that incorporates adaptive workload scheduling upon S-1F1B to fill pipeline bubbles. **4) Mist** [63], a state-of-the-art automatic training framework that supports adaptive model partition. We do not include Tessel [28] as a baseline because its code is not publicly available. AdaPipe [50] and Mario [32] are also excluded, as their recomputation optimization techniques are orthogonal to both AdaPtis and the baselines considered. It is worth noting that recomputation [5] can also be incorporated into AdaPtis, which we leave for future work.
Training Configurations. We evaluate with $P = 4, 8, 16$ and conduct a grid search over D, T, and E to determine the optimal parallelism settings. Training throughput is measured in Tokens per Second (TS). We further assess the performance of each method under varying input sequence lengths, numbers of micro-batches (*nmb*), and GPU counts.

5.2 E2E Performance and Analysis

As shown in Figure 8, AdaPtis consistently achieves the superior training throughput across diverse models and scales, highlighting its strength in handling model heterogeneity. Compared with S-1F1B, I-1F1B, ZB, and Mist, AdaPtis delivers average speedups of up to 1.34×, 1.42×, 1.34×, and 1.20× across Gemma, DeepSeek, and Nemotron-H models, under both 2K and 4K sequence length settings. These gains translate to throughput improvements ranging from 11% to over 59%, demonstrating both the robustness and generality of AdaPtis across heterogeneous models.

In contrast, existing PP methods fall short in efficiently training heterogeneous models. ZB shows only marginal improvements (only 1.02× over S-1F1B), while I-1F1B even degrades throughput by up to 22% on Nemotron-H (Large). These results demonstrate that optimizing a single phase of the pipeline is insufficient to address the increasing bubbles caused by model heterogeneity. By co-optimizing the model partition, model placement, and workload scheduling policies, AdaPtis achieves higher training throughput than these baselines, indicating the efficiency of co-optimization in reducing bubbles in heterogeneous models.

5.3 Throughput with Different Sequence Lengths

Figure 9 shows that AdaPtis sustains high training efficiency across sequence lengths ranging from 1K to 32K. Across all tested sequence lengths, AdaPtis consistently outperforms the baselines and achieves speedups up to 1.54×, 2.14×, 1.51×, and 1.27×, over S-1F1B, I-1F1B, ZB, and Mist, respectively. Compared with these baselines, AdaPtis obtains higher improvements as the sequence length increases from 1K to 32K. When compared with Mist, AdaPtis achieves relatively smaller but still steady improvements, with average speedups of 1.15× across all sequence lengths. These results highlight the efficiency of AdaPtis in handling training workloads across different sequence lengths.

5.4 Ablation Study of Pipeline Co-optimization

Figure 10 shows that co-optimization of adaptive workload scheduling, model partition, and model placement yields substantial improvements in training efficiency, whereas optimizing only one of them provides limited gains or may even degrade performance. Specifically, when co-optimizing model placement, workload scheduling, and model partition on the baseline method, performance improves by 1.32×, 1.37×, and 1.33× on Gemma, DeepSeek, and Nemotron-H, respectively. In contrast, tuning only a single phase leads to marginal benefits or performance degradation (e.g., applying model placement alone on Nemotron-H results in a 16% slowdown, consistent with the I-1F1B results in Figure 8).

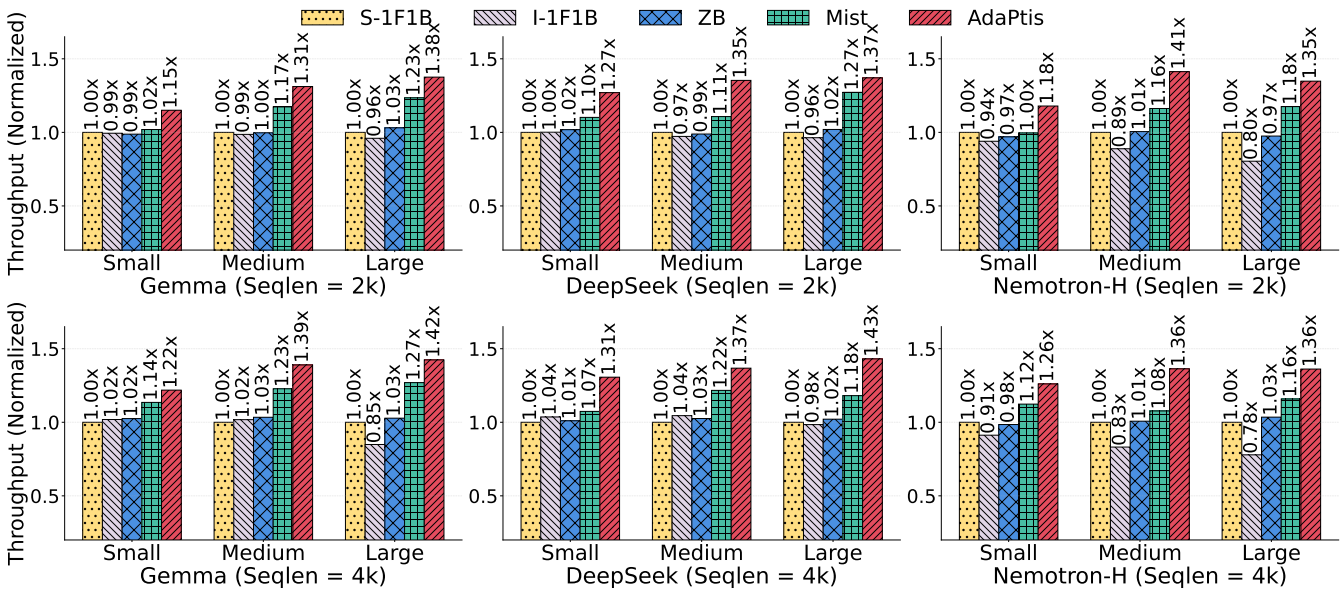

Figure 8. End-to-end training throughput of different PP methods on various model types and model sizes with input sequence length = (2K, 4K). The numbers above the bars indicate the normalized speedup over S-1F1B [47].

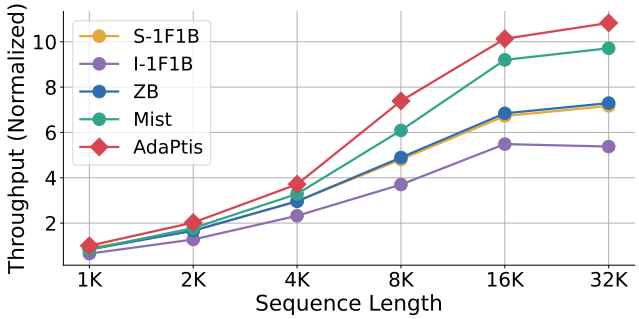

Figure 9. Throughput of AdaPtis, Mist, ZB, I-1F1B, and S-1F1B on Nemotron-H (Large) with $P = 8, T = 4, G = 64, nmb = 64$ across various input sequence lengths.

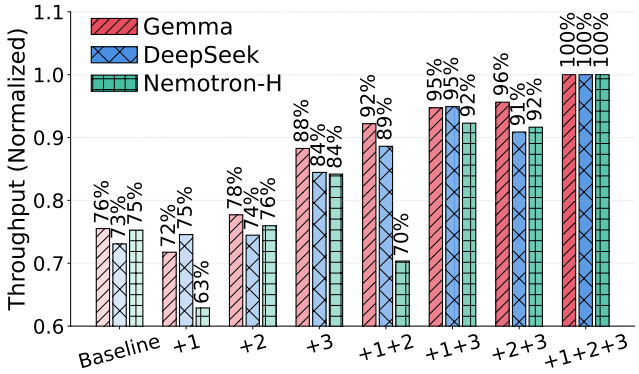

Figure 10. Ablation study of pipeline co-optimization with ① adaptive model placement, ② adaptive workload scheduling, and ③ adaptive model partition across models.

Moreover, Figure 11(a), (c), and (e) show the real traces of S-1F1B, Mist, and AdaPtis. It is evident that by co-optimizing

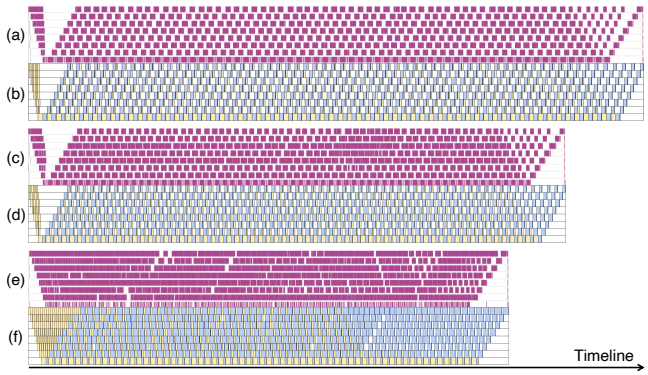

Figure 11. Real ((a), (c), (e)) and Pipeline Generator simulated ((b), (d), (f)) traces of S-1F1B, Mist, and AdaPtis, respectively, on Nemotron-H (Large) with $P = 8, T = 4, G = 64, nmb = 64, SeqLen = 4K$. In real traces, purple rectangles represent GPU kernels, and white rectangles represent bubbles. In simulated traces, yellow/blue rectangles represent GPU kernels, while white rectangles indicate bubbles.

the model partition, model placement, and workload scheduling, AdaPtis reduces pipeline bubbles (i.e., the blank area in the figure) and exhibits improved training efficiency.

5.5 Performance Model Fidelity

As shown in Figure 12, our Pipeline Performance Model achieves high modeling accuracy, with an average throughput prediction error of 2.12% across different methods for Nemotron-H models at $Seqlen = 4K$. The predicted throughput, which is normalized to S-1F1B since the modeling result is relative rather than absolute, closely matches the profiled real throughput results. It is worth noting that the maximum errors are 4.42% for AdaPtis, 4.55% for I-1F1B, 2.12% for ZB,

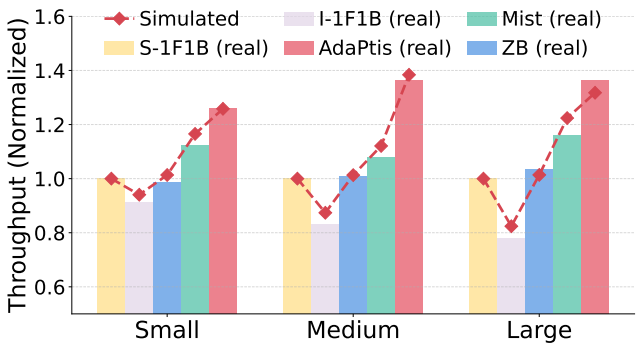

Figure 12. Pipeline performance model fidelity experimental results on Nemotron-H models with *Seqlen* = 4*K*.

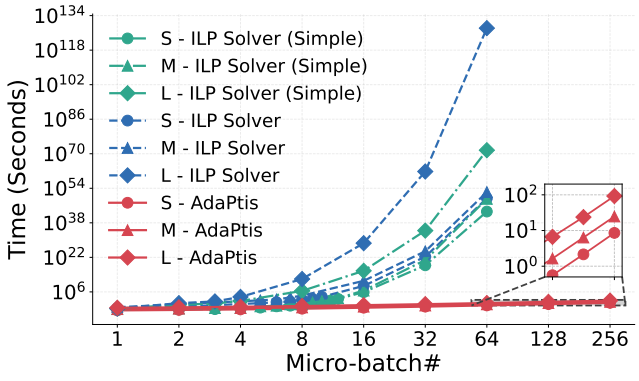

Figure 13. Pipeline generation time of ILP solver method [28, 39, 40] and AdaPtis across different model size (e.g. Small, Medium, and Large) and parallelism configurations.

and 6.57% for Mist. Furthermore, as shown in Figure 11, the simulated pipeline traces from Pipeline Generator closely match the real traces. These results collectively demonstrate the high accuracy of our Pipeline Performance Model.

5.6 Pipeline Generation Time

As shown in Figure 13, AdaPtis demonstrates superior efficiency and scalability in pipeline generation compared with the ILP solver-based approaches [28, 39, 40]. `ILP Solver (Simple)` applies only adaptive workload scheduling, while `ILP Solver` further incorporates adaptive workload scheduling, model partition, and model placement. As the problem size increases, ILP-based methods incur rapidly growing overhead. For small cases, we measure the actual solving time (less than 10^5 seconds), and for larger cases, we approximate it using `scipy.optimize.curve_fit`, since directly solving them is impractical. In contrast, AdaPtis requires significantly less time, often completing even substantial pipeline generation problems (e.g., large models, P, and 256 micro-batches) within 100 seconds. Overall, AdaPtis achieves remarkable efficiency and scalability in pipeline generation.

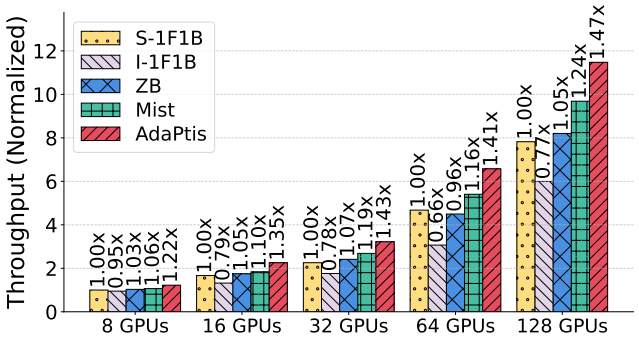

Figure 14. Strong scaling experimental results of training Nemontron-H (Large) with *Seqlen* = 4K on 128 GPUs.

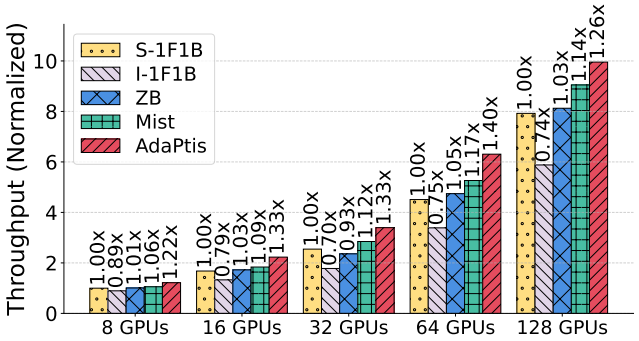

Figure 15. Weak scaling experimental results of training Nemontron-H (Large) with *Seqlen* = 4K on 128 GPUs.

5.7 Scalability Experimental Results

Strong Scaling. As shown in Figure 14, AdaPtis demonstrates superior scalability in our strong scaling experiments. We fix the sequence length, model size of the Nemotron-H (Large), pipeline parallelism size, and the number and size of micro-batches, while increasing the number of GPUs from 8 to 128. Across all configurations, AdaPtis consistently delivers the highest throughput. Furthermore, when scaling from 8 to 128 GPUs, AdaPtis achieves a scaling efficiency of 534%, outperforming the second-best Mist, which attains 514%. These results demonstrate that AdaPtis can efficiently accelerate the training tasks with more GPUs.

Weak Scaling. As demonstrated in Figure 15, AdaPtis maintains its computational efficiency in our weak scaling experiments. We fix the sequence length, model size of the Nemotron-H (Large), pipeline parallelism size, and the number and size of micro-batches, while increasing the number of GPUs from 8 to 128 and the global batch size from 32 to 512. Experimental results show that AdaPtis outperforms the baselines in throughput across all configurations. Moreover, when scaling from 8 to 128 GPUs, AdaPtis achieves a speedup of 519%, demonstrating its scalability in training larger global batch sizes with larger clusters.

6 Related Works

Token-level pipeline. TeraPipe [26] introduces token-level pipeline parallelism by splitting input sequences into finer-grained shards, which can be integrated into existing PP methods. Subsequent works further combine the existing PP methods with the token-level pipeline parallelism to mitigate pipeline bubbles [49] and alleviate memory pressure [4, 25].
Data heterogeneity. Training dataset has different lengths of sequences, making micro-batch execution time vary [15, 56, 60]. Some works apply micro-batch reordering [15, 56, 60], adaptive parallelism settings [56], or balanced data distribution [9] to alleviate bubbles caused by data heterogeneity.
Memory optimizations. To reduce memory overhead, *recomputation* [5] is proposed to discard the activations and recompute them before using. *ZeRO*[43–45] uses GPU memory, CPU memory, or even SSD to alleviate memory pressure. These techniques are orthogonal to PP and can be combined to reduce memory usage [32, 50]. Other works focus on addressing the memory imbalance in pipelines [17, 25, 59] by adjusting the memory allocation among stages.
Automatic parallelization. Many works [18, 22, 31, 34] focus on automatically finding optimal parallelism combinations to improve training performance. To reduce bubbles, some works [54, 62, 63] adjust the number of layers of each stage. However, the co-optimizing of model partition, model placement, and workload scheduling is not considered.

7 Conclusion

We propose `AdaPtis`, an LLM training system with adaptive pipeline parallelism. To alleviate the increasing pipeline bubbles caused by model heterogeneity, `AdaPtis` generates pipelines with co-optimized model partition, model placement, and workload scheduling policies, and executes these pipelines with communication optimizations. Extensive experimental results demonstrate that compared with the existing PP methods, `AdaPtis` achieves superior training efficiency on various types and scales of heterogeneous LLMs.

References

[1] Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, et al. 2025. Llama-nemotron: Efficient reasoning models. *arXiv preprint arXiv:2505.00949* (2025).

[2] Aaron Blakeman, Aarti Basant, Abhinav Khattar, Adithya Renduchintala, Akhiad Bercovich, Aleksander Ficek, Alexis Bjorlin, Ali Taghibakhshi, Amala Sanjay Deshmukh, Ameya Sunil Mahabaleshwarkar, et al. 2025. Nemotron-h: A family of accurate and efficient hybrid mamba-transformer models. *arXiv preprint arXiv:2504.03624* (2025).

[3] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[4] Qiaoling Chen, Shenggui Li, Wei Gao, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2025. SPPO:Efficient Long-sequence LLM Training via Adaptive Sequence Pipeline Parallel Offloading. arXiv:2503.10377 [cs.DC] https://arxiv.org/abs/2503.10377

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[7] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[8] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[9] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. 2025. ByteScale: Communication-Efficient Scaling of LLM Training with a 2048K Context Length on 16384 GPUs. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 963–978.

[10] Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2023).

[11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[12] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[13] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[15] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2024. DynaPipe: Optimizing multi-task training through dynamic pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 542–559.

[16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[17] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPIPE: memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*. PMLR, 16639–16653.

[18] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. 2023. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1466–1478.

[19] Joel Lamy-Poirier. 2023. Breadth-first pipeline parallelism. *Proceedings of Machine Learning and Systems* 5 (2023), 48–67.

[20] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[21] Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, et al. 2025. Minimax-01: Scaling foundation models with lightning attention. *arXiv*

preprint arXiv:2501.08313 (2025).

[22] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. 2022. Amp: Automatically finding model parallel strategies with heterogeneity awareness. *Advances in Neural Information Processing Systems* 35 (2022), 6630–6639.

[23] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[25] Zhouyang Li, Yuliang Liu, Wei Zhang, Tailing Yuan, Bin Chen, Chengru Song, and Di Zhang. 2025. SlimPipe: Memory-Thrifty and Efficient Pipeline Parallelism for Long-Context LLM Training. *arXiv preprint arXiv:2504.14519* (2025).

[26] 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. In *International Conference on Machine Learning*. PMLR, 6543–6552.

[27] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avshalom Manevich, Nir Ratner, Noam Rozen, Erez Schwartz, Mor Zusman, and Yoav Shoham. 2024. Jamba: A Hybrid Transformer-Mamba Language Model. *arXiv preprint* (2024). https://arxiv.org/html/2403.19887v2 Equal contribution.

[28] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. 2024. Tessel: Boosting distributed execution of large dnn models via flexible schedule search. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 803–816.

[29] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[30] 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. *arXiv preprint arXiv:2412.19437* (2024).

[31] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. 2024. Aceso: Efficient Parallel DNN Training through Iterative Bottleneck Alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 163–181.

[32] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. 2025. Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 197–211.

[33] 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*. 1–13.

[34] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878* (2022).

[35] MiniMax, :, Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, Chengjun Xiao, Chengyu Du, Chi Zhang, Chu Qiao, Chunhao Zhang, Chunhui Du, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dong Li, Enwei Jiao, Haigang Zhou, Haimo Zhang, Han Ding, Haohai Sun, Haoyu Feng, Huaiguang Cai, Haichao Zhu, Jian Sun, Jiaqi Zhuang, Jiaren Cai, Jiayuan Song, Jin Zhu, Jingyang Li, Jinhao Tian, Jinli Liu, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kaiyi Feng, Ke Yang, Kecheng Xiao, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Li, Lin Zheng, Linge Du, Lingyu Yang, Lunbin Zeng, Minghui Yu, Mingliang Tao, Mingyuan Chi, Mozhi Zhang, Mujie Lin, Nan Hu, Nongyu Di, Peng Gao, Pengfei Li, Pengyu Zhao, Qibing Ren, Qidi Xu, Qile Li, Qin Wang, Rong Tian, Ruitao Leng, Shaoxiang Chen, Shaoyu Chen, Shengmin Shi, Shitong Weng, Shuchang Guan, Shuqi Yu, Sichen Li, Songquan Zhu, Tengfei Li, Tianchi Cai, Tianrun Liang, Weiyu Cheng, Weize Kong, Wenkai Li, Xiancai Chen, Xiangjun Song, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xinzhu Hou, Xuan Lu, Xun Zou, Xuyang Shen, Yan Gong, Yan Ma, Yang Wang, Yiqi Shi, Yiran Zhong, Yonghong Duan, Yongxiang Fu, Yongyi Hu, Yu Gao, Yuanxiang Fan, Yufeng Yang, Yuhao Li, Yulin Hu, Yunan Huang, Yunji Li, Yunzhi Xu, Yuxin Mao, Yuxuan Shi, Yuze Wenren, Zehan Li, Zelin Li, Zhanxu Tian, Zhengmao Zhu, Zhenhua Fan, Zhenzhen Wu, Zhichao Xu, Zhihang Yu, Zhiheng Lyu, Zhuo Jiang, Zibo Gao, Zijia Wu, Zijian Song, and Zijun Sun. 2025. MiniMax-M1: Scaling Test-Time Compute Efficiently with Lightning Attention. arXiv:2506.13585 [cs.CL] https://arxiv.org/abs/2506.13585

[36] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[37] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. 2022. Out-of-order backprop: An effective scheduling technique for deep learning. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 435–452.

[38] 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).

[39] Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. 2024. Pipeline Parallelism with Controllable Memory. *arXiv preprint arXiv:2405.15362* (2024).

[40] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2024. Zero Bubble (Almost) Pipeline Parallelism. In *The Twelfth International Conference on Learning Representations*.

[41] Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. 2024. Various lengths, constant speed: Efficient language modeling with lightning attention. *arXiv preprint arXiv:2405.17381* (2024).

[42] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*. PMLR, 18332–18346.

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[44] 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 *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[45] 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 21)*. 551–564.

[46] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[47] 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. *arXiv preprint arXiv:1909.08053* (2019).

[48] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 13693–13696.

[49] Ao Sun, Weilin Zhao, Xu Han, Cheng Yang, Xinrong Zhang, Zhiyuan Liu, Chuan Shi, and Maosong Sun. 2024. Seq1f1b: Efficient sequence-level pipeline parallelism for large language model training. *arXiv preprint arXiv:2406.03488* (2024).

[50] 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*. 86–100.

[51] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[52] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786* (2025).

[53] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[54] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast Automatic Distributed Training on Heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 563–578.

[55] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[56] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. 2025. Flexsp: Accelerating large language model training via flexible sequence parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 421–436.

[57] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[58] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[59] Man Tsung Yeung, Penghui Qi, Min Lin, and Xinyi Wan. 2024. Balancing Pipeline Parallelism with Vocabulary Parallelism. *arXiv preprint arXiv:2411.05288* (2024).

[60] Zili Zhang, Yinmin Zhong, Ranchen Ming, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, and Xin Jin. 2024. Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models. *arXiv preprint arXiv:2408.04275* (2024).

[61] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[62] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[63] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. 2025. Mist: Efficient Distributed Training of Large Language Models via Memory-Parallelism Co-Optimization. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1298–1316.