Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism

Weijian Liu
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
liuweijian22s@ict.ac.cn

Mingzhen Li[*]
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
limingzhen@ict.ac.cn

Guangming Tan
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
tgm@ict.ac.cn

Weile Jia[*]
SKLP, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
jiaweile@ict.ac.cn

Abstract

Large language models have to be trained in parallel due to their large number of parameters and significant memory footprint. Among various parallelism techniques, pipeline parallelism is widely adopted in inter-node scenarios with minimal communication overhead. However, state-of-the-art pipeline schemes lead to extra and imbalanced memory footprints, leaving room for further improvement. In this paper, we propose *Mario*, a pipeline optimizer that automatically tessellates activation checkpointing to existing pipeline schemes, enabling training larger models (or longer sequences) with less and balanced memory footprint across GPUs and improved GPU utilization. First, the activation recomputation can be effectively overlapped in the bubbles by moving it earlier in the execution process, thereby improving overall efficiency. With eliminated memory footprint through checkpointing, *Mario* allows for preposing more forward computation into the pipeline bubbles, making more room for further overlapping with greater flexibility, and thus exploiting the bubbles. Then we design a lightweight pipeline simulator to model execution behavior w/o|w/ *Mario*. Finally, we introduce an automatic pipeline scheduler specifically for *Mario*, capable of searching for near optimal combination of checkpointing and pipeline configurations within minutes. Experimental results on GPT3 and LLaMA2 models show that *Mario* can speed up existing state-of-the-art pipeline schemes (w/o|w/ checkpointing) including 1F1B, Chimera, and Interleave by 1.16×|1.57× on average. This work paves a new direction for effective low-cost pipeline training.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**.

Keywords: large-scale training, pipeline parallelism, activation checkpointing

1 Introduction

Large language models (LLMs), which are built upon the transformer architecture [42], have become foundational models in the field, playing a crucial role in a wide range of applications. However, training these LLMs comes with significant resource demands, particularly in memory capacity and computation power. Larger and deeper models with an increased number of parameters are designed to achieve higher quality and sample efficiency [19], and the longer sequence length allows for capturing more contextual information but introduces quadratic complexity in memory and computation, further intensifying resource requirement [6].

As one essential parallelization scheme, pipeline parallelism partitions the model layers into multiple stages, where each device is responsible for one or more stages. This approach processes micro-batches in a pipeline manner to enhance the device utilization. It can be integrated with data, tensor (including sequence) parallelism [22, 25, 31, 35] to jointly train large-scale models such as GPT-4 [1], which has up to 1.76 trillion parameters.

Recently, a series of pipeline parallelism schemes have been proposed to reduce the pipeline bubbles by scheduling the instructions (forward and backward computations of micro-batches). GPipe [13] simply processes forward computations of all micro-batches, followed by all backward computations. The 1F1B schedule [7, 29] restricts that each

[*]Corresponding authors

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710878

stage can only process one micro-batch in the steady phase. Chimera [23] embeds bidirectional 1F1B pipelines simultaneously to overlap their bubbles mutually. ZB-H1 [34] (which splits the backward to fill bubbles) and Hanayo [28] (which proposes the wave-like pipeline based on Chimera) both enhance throughput by delaying the release of activation memory. However, they trade off decreased memory efficiency for reduced bubbles.

The activations (i.e., intermediate tensors) generated in the forward computation must be retained in memory to calculate the gradients during the backward computation, which significantly contributes to the overall memory footprint [22, 33, 36]. All pipeline schemes suffer from imbalanced memory footprint across devices. This imbalance arises because multiple micro-batches are processed simultaneously, requiring storing several replicas of activations corresponding to these unfinished micro-batches. For example, in the 1F1B scheme with 16 GPUs, the activation of the first device can be 16 times larger than that on the last device. To reduce the memory footprint of activations, checkpointing [4, 21, 26], swapping [12, 37], and compressing [14, 46] have been proposed. Among them, checkpointing is widely adopted due to its lower recomputation overhead [6, 21, 22, 26] compared to swapping and compressing.

Integrating activation checkpointing to pipeline parallelism can directly eliminate the activation memory to tackle imbalanced memory footprint across devices. However, this integration introduces additional bubbles—primarily from recomputation, in the critical path of pipeline, significantly deteriorating the training efficiency. Previous work [13] has reported about 23% overhead compared to the pipeline without checkpointing. Other approaches [6, 22, 39] select certain operators for checkpointing. However, they still overlook the performance opportunity of tessellating checkpointing into pipeline bubbles and congest the recomputation in the critical path. For example, they typically insert recomputation directly before corresponding backward computation, increasing the backward computation time by 50%. To achieve better performance through tessellation, we propose two strategies: *1)* tweaking the pipeline to hide the recomputation within existing bubbles, and *2)* reshaping the bubbles by pre-executing more forward computations, as the memory footprint is independent of unfinished micro-batches.

To overcome above issues, we propose *Mario*, a pipeline optimizer that tessellating the activation checkpointing optimizations to existing pipeline schemes. *Mario* eliminates the imbalanced memory footprint of activations through checkpointing, and reshapes pipeline bubbles to hide the recomputation instructions, thereby minimizing checkpointing overhead to near zero in most cases. Additionally, *Mario* can further leverage the memory space freed up by these optimizations to train LLMs. For example, *Mario* enables a larger micro-batch size to improve device utilization or longer sequence length (seqlen) to better handle extended contexts. Unlike pipeline schemes that reduce bubbles at the expense of increasing memory footprint, *Mario* simultaneously reduces memory footprint and overlaps pipeline bubbles. Note that *Mario* is still orthogonal to existing pipeline schemes (e.g., 1F1B [29], Chimera [23], Interleave [31], Hanayo [28]) as ***Mario aims to strengthen existing pipeline parallelism schemes with near zero-cost activation checkpointing***.

Mario optimizes the pipeline schemes by manipulating instruction lists, which are determined when the scheme is given, in an ahead-of-time (AOT) manner. *Mario* takes the configurations of the pipeline scheme and the model as input (§4), then applies four optimization passes by identifying and substituting corresponding pipeline patterns (§5.1), and then leverages the simulator-based performance model (§5.2) to search for optimal parameter configurations (§5.3). The outputted instruction lists can be directly executed by *Mario* to train LLMs. Specifically, the key contributions are as follows:

- We design a pipeline optimizer that tessellates activation checkpointing into pipelines, effectively reducing and balancing memory footprint across devices. It hides recomputation overhead within the naturally existing and specifically reshaped pipeline bubbles.
- We propose an accurate simulator-based performance model that provides instant performance feedback, combining dynamic programming and predefined dependencies to estimate execution behaviors precisely.
- We exploit the enlarged pipeline scheduling space to improve device utilization, leveraging the memory saved through checkpointing. Additionally we introduce an automatic scheduler to determine better parameter configurations.
- We deploy *Mario* in Megatron-DeepSpeed[*] to optimize the training of transformer-based models, such as GPT3 and LLaMA2. Evaluation results show that *Mario* can speed up state-of-the-art pipeline schemes.

2 Background

2.1 Pipeline Parallelism

To train large models and process large datasets, several parallelization schemes are adopted to reduce training time. In *data parallelism (DP)*, each device holds a complete replica of model weights for mini-batch training, while straightforward, is not memory-efficient due to duplication of weights across devices. ZeRO [35, 36] improves *DP* by partitioning the parameters, gradients, and optimizer states across all devices and only stores one replica. In *tensor parallelism (TP)* (including *sequence parallelism (SP)*), each device computes a portion of model operators using split model weights. However, *TP* suffers from high communication overhead to gather/scatter the split tensors, especially in cross-node training. In *pipeline parallelism (PP)*, each device holds one pipeline stage (several layers) and processes micro-batches (part of a

[*]https://github.com/microsoft/Megatron-DeepSpeed

Table 1. Memory footprint across pipeline schemes.

Scheme	Weights Mem.	Activation Mem.	Activation Mem.(*Mario*)
GPipe [13]	M_w	$N \times M_\theta$	M_θ
1F1B [7]	M_w	$[M_\theta, D \times M_\theta]$	M_θ
Interleave [31]	M_w	$[D+1, 3D-2] \times M_\theta/2$	$M_\theta/2$
Chimera [23]	$2 \times M_w$	$[(D/2+1) \times M_\theta, D \times M_\theta]$	M_θ
Hanayo [28]	M_w	$[(D+1)/2 \times M_\theta, D \times M_\theta]$	$M_\theta/2$

Table 2. Symbols.

D	# of devices
S	# of pipeline stages
N	# of micro-batches in one iteration
M_w	Memory footprint of weights
M_θ	Memory footprint of activations
T_{F_i}	Exec. time of forward in *i-th* stage
T_{B_i}	Exec. time of backward in *i-th* stage

Table 3. Instructions.

$(C)FW_m^p$	(Ckpt) Forward computation
BW_m^p	Backward computation
SA_m^p	Send activation
RA_m^p	Receive activation
SG_m^p	Send gradient
RG_m^p	Receive gradient
RC_m^p	Recomputation
AR^p	Allreduce for *DP*
OS^p	Optimizer step

mini-batch) in a pipeline manner. Activations (intermediate tensors) are communicated between neighboring stages in a peer-to-peer (*p2p*) manner. In practice, *PP* is orthogonal to *DP* and *TP*, and can be combined with them for better resource utilization [3, 16, 49].

In this paper, we focus on synchronous *PP* approaches. These methods trigger gradient synchronization and pipeline flushes to ensure all micro-batches in a training iteration use the same weights, which, however, introduces pipeline bubbles [23]. Instead, asynchronous approaches suffer from *1)* mismatched weight versions [45]) or *2)* weight staleness [29, 30], both of which deteriorate the convergence [2, 5].

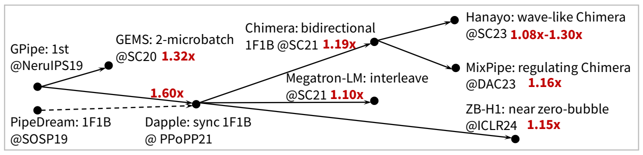


Figure 1. Development of pipeline parallelism schemes.

The typical *PP* schemes and their improvement on training throughput are summarized in Figure 1. Note that training GPT4 model costs over $100 million, even a 10% throughput improvement can save millions of dollars. Existing *PP* works strive to reduce the pipeline bubbles and improve computation efficiency but usually trade off the memory efficiency. We summarize the weight memory and activation memory of the aforementioned pipeline schemes in Table 1, with the frequently used symbols provided in Table 2. As for weight memory, Chimera maintains two replicas of model weights (i.e., $2M_w$ for *up* and *down* pipelines), and others only maintain one replica. The peak activation memory is determined by the number of on-the-fly micro-batches (those whose activations are stored from micro-batch issuance until the backward computation is complete). Thus the initial pipeline stages typically consume more memory than the final stages. Note that the peak activation memory can be linearly correlated to the index of devices participating in a pipeline (i.e., $\beta D M_\theta$). As shown in the last column of Table 1, with *Mario*, the peak activation memory can be reduce to M_θ or $M_\theta/2$, making it independent from D with near zero cost (see §5).

2.2 Activation Checkpointing and Recomputation

Activation checkpointing trades off additional computations for a lower memory footprint [4, 11, 21, 26]. It allows dropping the activation tensors in the forward computation and recomputing them by replaying the forward computation (i.e., recomputation) in the backward computation. In general, due to the lower overhead of recomputation than data transmission between GPU and CPU, activation checkpointing is widely adopted in popular deep learning frameworks.

Several works have applied checkpointing to train large models, but they only treat checkpointing as a remedy to rescue the memory footprint introduced by pipeline schemes. GPipe [13] injects multiple micro batches into the pipeline concurrently, leading to multiple versions of activation tensors. It uses checkpointing to reduce activation tensors and also considers scheduling recomputation earlier to overlap with the bubbles. Megatron-LM also uses checkpointing both naively [31] and selectively [22] to reduce the peak memory. AdaPipe [39] selects certain operators for checkpointing and then re-balances pipeline stages. PipeDream-2bw [30] saves two versions of model weights, and thus turns to checkpointing to compensate for the extra memory footprint.

However, the potential of combining checkpointing with pipeline parallelism remains under-explored. By eliminating activation memory, pipeline execution can be adjusted in more flexible ways, yet prior works have overlooked the acceleration opportunities in this context.

3 Motivation

3.1 Demand for Efficient Checkpointing in Pipeline

To eliminate the linear correlation of activation memory footprint and the number of devices, current pipeline schemes rely on activation checkpointing at the expense of recomputation overhead, which GPipe [13] reports at 23% of a single training iteration. Theoretically, the recomputation time of a block is equivalent to the forward time, resulting in about 33% overhead (assuming the backward time is twice as long as the forward time, denoted as t). Figure 2 shows the 4-stage 1F1B execution (each device holds one stage) of a GPT3-125M model, comparing the baseline without checkpointing to the steps with checkpointing (step 1-4). In ***Step 1***, when checkpointing is applied and recomputation for stage

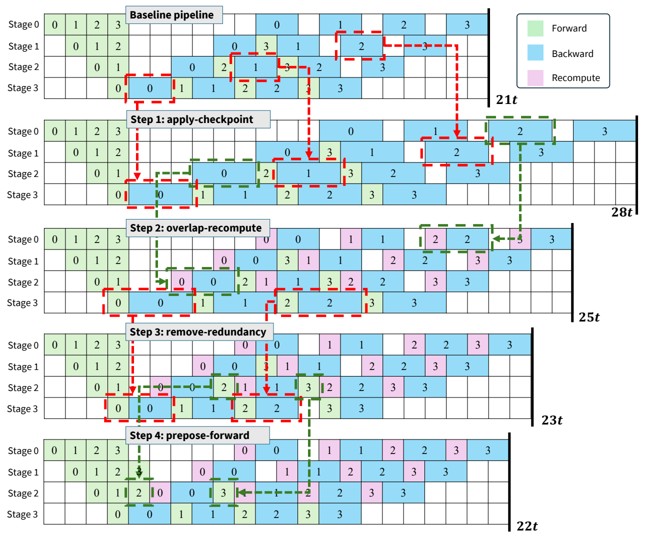


Figure 2. A 4-stage 1F1B pipeline can achieve near zero-cost activation checkpointing. X-axis indicates timeline and y-axis indicates pipeline stages. The red/green boxes highlight the optimizations in different steps.

i is placed just before the backward (of stage i), the total computation time increases from $21t$ (baseline) to $28t$.

However, the activation checkpointing can be near zero-cost if recomputation is carefully tessellated into the pipeline. We illustrate this approach in the following steps: ***Step 2:*** Start the recomputation earlier to fill the recomputation into pipeline bubbles, reducing the total time to $25t$; ***Step 3:*** Remove the redundant recomputation, particularly those adjacent to their corresponding backward computation, further reducing the time to $23t$; ***Step 4:*** Employ fine-grained co-scheduling, such as preposing extra micro-batches to reshape the pipeline bubbles, reducing the time to $22t$, near to baseline of $21t$. We further reproduce above steps using Megatron-DeepSpeed on a node with 4 A100-40G GPUs. This demonstrates that near zero-cost activation checkpointing is achievable. Consequently, we can *1)* reduce the memory footprint, *2)* alleviate the imbalanced activation memory among devices, and *3)* enlarge the feasible schedule space (with checkpointing) of pipeline parallelism.

3.2 Complicated Performance Modeling of Pipeline

Building an accurate performance model for pipeline schedules, especially when coupled with activation checkpointing, is highly complex. The strawman approach about counting the forward grids and backward grids is oversimplified and fails to capture real-world scenarios. This is due to several factors: *1)* Stage splitting is often imbalanced because the pipeline contains not only identical `transformer` layers, but also various other layers (e.g., `embedding` and `layer_norm` in first/last stage. *2)* The execution time of forward and backward computations are not necessarily in a simple integer ratio, such as the commonly assumed 1:2. For instance, according to Ref.[22], the ratio of a transformer layer is about 1:1.6). Using biased performance data based on those assumptions can lead to a sub-optimal pipeline modeling.

3) Current pipeline schemes are tightly integrated with customized frameworks (e.g., DeepSpeed schedules instructions based on a *grid_id* formula, Chimera uses queues, and Hanayo employs *action list* that is not open-source). This makes it challenging to evaluate the checkpointing strategies across multiple schemes without executing real runs.

However, real runs to evaluate these pipeline schedules are both expensive and time-consuming. For example, even the initialization time for the Megatron-LM framework on 2,048 GPUs can cost 1047 seconds [18]. Therefore, a reliable and portable performance modeling tool (e.g., a pipeline simulator) is essential. Such a tool can estimate the efficiency of pipeline schemes in a lightweight and fair manner, facilitating further parameter tuning without the need for extensive real-world testing.

4 Overview

Goal. *Mario* aims to optimize the pipeline schedule by integrating the activation checkpointing, thereby reducing and balancing the memory footprint across devices at near zero cost. Therefore, *Mario* enables using larger micro-batch size to improve computing efficiency and larger sequence length to improve model quality. Note that *Mario* focuses on *PP*, and only optimizes the synchronous pipeline schedule, making it orthogonal to *DP/TP/SP* schemes. The optimized pipeline schedule by *Mario* can be seamlessly integrated into other parallelism strategies (see experiments in §6.5 and §6.7).

User Interface. To use *Mario*, a user needs to specify *Mario* configuration and the model configuration, as shown in Listing 1. *Mario* configuration includes the pipeline scheme, global batch size, number of devices and memory per device. Currently, *Mario* supports typical pipeline schemes, including "V" shape (1F1B), "X" shape (Chimera), and "W" shape (Interleave). Note that users can also select the "Auto" option to automatically determine the optimal pipeline scheme.

Once specified, the pipeline scheme is expanded into an instruction list of a training iteration for each device, which functions similar to the intermediate representation (IR) of the pipeline. As shown in Figure 3(b), the instruction list mainly consists of the basic forward (FW_m^p) and backward (BW_m^p) instructions. Each instruction has two key attributes, micro-batch id (subscript m) and partition id (superscript p). Refer to §5.2 for details. The model configuration is instantiated into a trainable model, which is then partitioned into multiple pipeline stages. These stages are subsequently mapped to the available devices.

Design Overview. As shown in Figure 3, *Mario* has three main components: a graph tuner, a simulator-based performance model, and a schedule tuner. Given an input, the

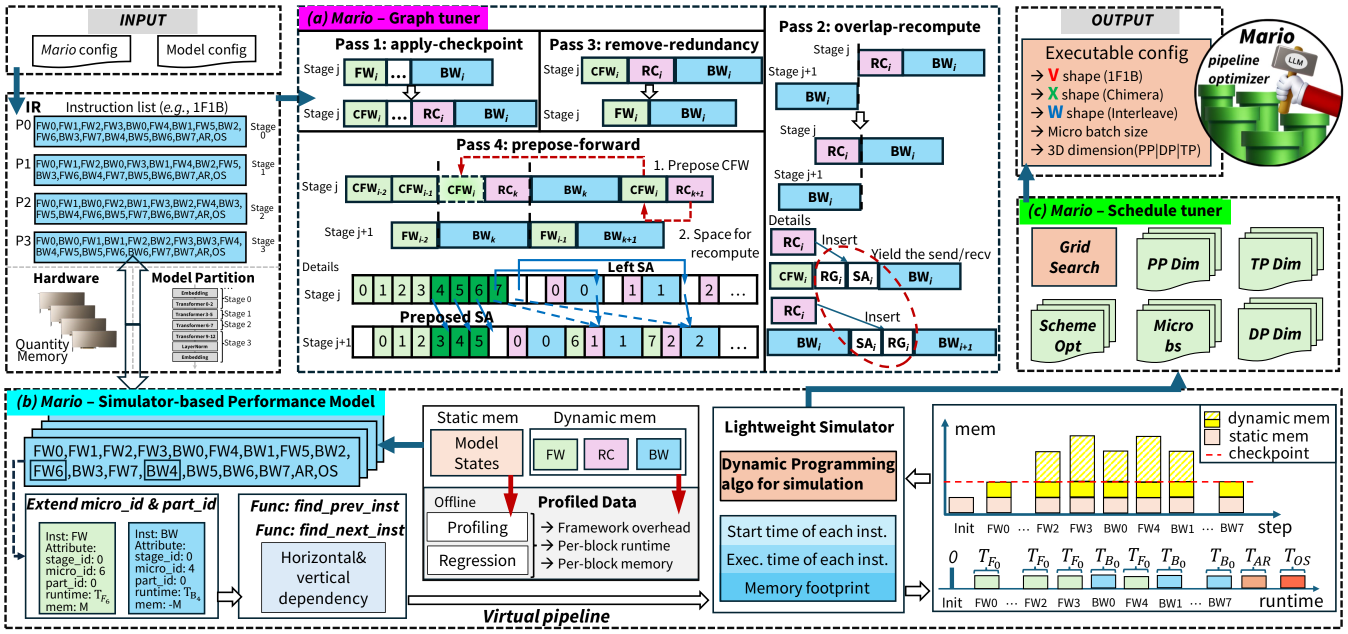

INPUT
Mario config
Model config
IR Instruction list (e.g., 1F1B)
P0 FW0,FW1,FW2,FW3,BW0,FW4,BW1,FW5,BW2, FW6,BW3,FW7,BW4,BW5,BW6,BW7,AR,OS
P1 FW0,FW1,FW2,BW0,FW3,BW1,FW4,BW2,FW5, BW3,FW6,BW4,FW7,BW5,BW6,BW7,AR,OS
P2 FW0,FW1,BW0,FW2,BW1,FW3,BW2,FW4,BW3, FW5,BW4,FW6,BW5,FW7,BW6,BW7,AR,OS
P3 FW0,BW0,FW1,BW1,FW2,BW2,FW3,BW3,FW4, BW4,FW5,BW5,FW6,BW6,FW7,BW7,AR,OS
Hardware
Quantity
Memory
Model Partition
(a) Mario – Graph tuner
Pass 1: apply-checkpoint
Pass 3: remove-redundancy
Pass 2: overlap-recompute
Pass 4: prepose-forward
1. Prepose CFW
2. Space for recompute
Left SA
Preposed SA
Details
Insert
Yield the send/recv
OUTPUT
Executable config
→ V shape (1F1B)
→ X shape (Chimera)
→ W shape (Interleave)
→ Micro batch size
→ 3D dimension(PP|DP|TP)
Mario pipeline optimizer
(c) Mario – Schedule tuner
Grid Search
PP Dim
TP Dim
Scheme Opt
Micro bs
DP Dim
(b) Mario – Simulator-based Performance Model
FW0,FW1,FW2,FW3,BW0,FW4,BW1,FW5,BW2, FW6,BW3,FW7,BW4,BW5,BW6,BW7,AR,OS
Extend micro_id & part_id
Inst: FW Attribute: stage_id: 0 micro_id: 6 part_id: 0 runtime: T_{F_6} mem: M
Inst: BW Attribute: stage_id: 0 micro_id: 4 part_id: 0 runtime: T_{B_4} mem: -M
Func: find_prev_inst
Func: find_next_inst
Horizontal& vertical dependency
Static mem
Dynamic mem
Model States
FW RC BW
Offline
Profiled Data
Profiling
Regression
→ Framework overhead
→ Per-block runtime
→ Per-block memory
Virtual pipeline
Lightweight Simulator
Dynamic Programming algo for simulation
Start time of each inst.
Exec. time of each inst.
Memory footprint
mem
dynamic mem
static mem
checkpoint
step
runtime

Figure 3. Overview of *Mario*: (a) graph tuner, (b) simulator-based performance model, and (c) schedule tuner.

graph tuner tessellates activation checkpointing and recomputation into the pipeline, identifying patterns to overlap the recomputation within the pipeline bubbles and reshaping these bubbles for further overlapping. To estimate the efficiency of the tuned pipeline, the lightweight **pipeline simulator** predicts per-block latency and memory usage. It models the *virtual pipeline* of all possible pipeline schemes, accounting for the horizontal/vertical dependencies through a dynamic programming algorithm. Note that the simulator can also depict the overhead of distributed training frameworks (e.g., DeepSpeed). The **schedule tuner** further refines the schedules generated by the graph tuner through an iterative approach. It utilizes the lightweight simulator for rapid throughput estimation of compatible schedules, thereby avoiding costly execution on GPU clusters.

Listing 1. Programming interface of *Mario*.

```
import mario
mario_conf = { 'pipeline_scheme': 'Auto|V|X|W|...',
               'global_batch_size': 128,
               'num_device':        32,
               'memory_per_device':'40G'          }
model_conf = { 'type':              'GPT3|LLaMA2|...',
               'hidden_size':     4096,         ...}
schedule = mario.optimize(mario_conf, model_conf,
     global_bs)
mario.run(schedule)
```

5 Design

5.1 Graph Tuner

We build the computational graph of pipeline execution based on the initial pipeline schedule (i.e., the instruction list). In fact, the instruction list is similar to the intermediate representation (IR) of the pipeline, as each instruction represents computation/communication type and its dependencies, rather than directly executing them on real hardware. An instruction's dependency indicates that instruction I must be executed after the completion of a instruction set S. These dependencies can be categorized into horizontal and vertical ones (see §5.2). Our focus is on the forward (FW) and backward (BW) instructions. Based on these, we insert additional auxiliary instructions (e.g., LM, SG, RG, SA, RA) into the instruction list to complete the pipeline execution procedure. The frequently used instructions are summarized in Table 3. When plotting the computational graph in figures, for simplicity, we assume *1)* the latency across stages are balanced, and *2)* the backward latency is twice that of forward. Note that our simulator uses the real latency for high fidelity.

Mario applies graph-level optimization passes to modify and transform the computational graph, ensuring that all dependencies of the computational graph are maintained after each pass. We now discuss the graph-level passes, as shown in Figure 3(a). For simplicity, we demonstrate them using the instruction list of the 1F1B scheme, where each device is mapped to one pipeline stage, allowing us to omit the superscript m. These passes is general to other pipeline schemes and have already been applied to Chimera and Interleave (see experiments in §6). These optimization passes are applied iteratively, meaning each pass can be applied multiple times to refine the computational graph.

Pass 1: apply-checkpoint – *Apply activation checkpointing to all paired forward and backward instructions.* The forward instruction (FW_i) is replaced by the checkpointed forward (CFW_i). And a recomputation (RC_i) instruction is inserted before the corresponding backward (BW_i) instruction to ensure that the activation tensors required by the backward instruction are restored. Specifically, the distance between RC_i and BW_i should be minimized because the activation generated by RC_i is kept in memory till it is consumed by BW_i. A shorter distance results in a lower memory footprint for a longer period, allowing other expensive instructions to be scheduled. Therefore, only one replica of activation is kept in memory for each stage, balancing the memory footprint across devices.

Pass 2: overlap-recompute – *Overlap the recomputation instructions in pipeline bubbles by preposing recomputation and backward instructions together.* The recomputation instruction RC_i relies only on the forward instruction FW_i of the same device, so it can be placed in a space between FW_i and BW_i, which usually contains pipeline bubbles. The backward instruction BW_i relies on both RC_i from the same device (denoted as device j) and BW_i from the next device $j+1$. Therefore, RC_i of device j can be executed concurrently with BW_i of device $j+1$, effectively hiding it within the pipeline bubbles. To maintain pipeline efficiency, RC_i should be placed before RG_i (which receives the gradient of BW_i from device $j+1$). If RC_i is incorrectly placed after RG_i, it must wait for RG_i to finish, which, in turn, is dependent on with SG_i from device $j+1$, causing RC_i on device j to wait for BW_i on device $j+1$ and losing the opportunity for concurrent execution.

Pass 3: remove-redundancy – *Remove redundant activation checkpointing.* When CFW_i and BW_i are adjacent, the corresponding activation is dropped and then restored instantly once the BW_i starts. It leads to higher memory footprint similar to the scenario without checkpointing.

Pass 4: prepose-forward – *Prepose the checkpointed forward instructions to the earliest pipeline bubbles.* When there are pipeline bubbles before the checkpointed forward instruction (e.g., CFW_i), it can be preposed into the bubbles, leaving its original position idle. This allows *pass 2* to move the recomputation (e.g., RC_{k+1}) into this idle place, improving the opportunity for future RC_{k+m} ($m > 1$) instructions to be overlapped. In classical pipelines without checkpointing, forward instructions can also be preposed, but at the cost of increased memory footprint due to more on-the-fly micro-batches, which makes this approach infeasible. Specifically, the SA_i instruction corresponding to CFW_i should also be considered to avoid communication deadlock, which can be handled in two scenarios. *1)* If CFW_i in the next device is also preposed (e.g., $CFW_{4|5}$), SA_i can be preposed along with CFW_i. *2)* If not (e.g., $CFW_{6|7}$), SA_i should remain in the original place. The output of CFW_i is temporarily stored in a buffer, and SA_i reads the buffer before sending it to RA_i on the next device. This is necessary because *Mario* supports a blocking p2p communication approach, and SA_i and RA_i must be paired to avoid deadlock.

5.2 Simulator-based Performance Model

To comprehend the computational graph of the pipeline, we build a simulator-based performance model. Unlike common analytical performance models [23, 49], we formulate our estimation as a dynamic programming algorithm, eliminating the need to manually identify the critical path or make approximations about a perfect pipeline. The simulation latency is approximately 700ms for training GPT3-13B (64 micro-batches, Chimera scheme) with 32 GPUs, significantly accelerating the schedule tuner (§5.3).

Virtual Pipeline. We introduce *virtual pipeline* to unify the representation of various pipeline schemes. Although these schemes differ significantly in instruction execution order (e.g., a device may accommodate multiple stages, and logically neighboring stages might not reside in neighboring devices), they all adhere to the fundamental principle that *FW* instructions are executed across all stages, followed by *BW* instructions in a reverse order for each micro-batch. The *virtual pipeline* is designed to encapsulate this principle.

To identify different stages held by a single device, we extend pipeline instructions with an additional attribute, the partition id (denoted as *part_id*, superscript p). Together with the micro-batch id (denoted as *micro_id*, subscript m) attribute, the virtual pipeline can abstractly represent almost all pipeline schemes. For instance, the bidirectional pipelines in Chimera (i.e., an *up* one and a *down* one) are assigned *part_id* 0 and 1, respectively, while the model chunks of Interleave are identified by *part_id* corresponding to chunk_id.

As shown in Algorithm 1, the dependencies among instructions in the virtual pipeline (of a given *micro_id*) are streamlined into two functions: `find_prev_inst` (which locates the instruction in previous stage) and `find_next_inst` (which locates the instruction in next stage). Since *Mario* maintains an instruction list for each device, the target instruction can be indexed by device_id, micro_id, part_id, and inst_type. For simplicity, consider the `find_prev_inst` function for *FW* instructions. The logical direction, or *step*, is opposite for *FW* and *BW* in the virtual pipeline (line 2). In the 1F1B pipeline, the device for the target instruction is indexed by advancing a step in the logical direction (line 5). For Chimera, which features bidirectional pipelines, the up/down pipeline ($part\_id = 0|1$) follows the logical/opposite direction, respectively (line 7). In the Interleave pipeline, where each device handles multiple stages in a cyclic manner, the device of target instruction is indexed using modular arithmetic in a logical direction, with part_id adjustments if the target instruction crosses stages (line 9-10). Additionally, we provide a flexible interface for users (line 12) to extend `find_prev_inst` and `find_prev_inst` functions to support other emerging pipeline schemes.

Algorithm 1 Find dependencies in *virtual pipeline*.

Input: device_id d, micro_id m, part_id p, instruction_type t of current instruction
Output: ids of the target instruction

```
1: function FIND_[PREV|NEXT]_INST($d$, $m$, $p$, $t$)
2:     $step = -1| + 1$ if find_prev_inst|find_next_inst      ▷ logical direction
3:     $d_o = d$, $m_o = m$, $p_o = p$, $t_o = t$              ▷ initialize outputs
4:     if scheme == "1F1B" then
5:         $d_o = d + step$
6:     else if scheme == "Chimera" then
7:         $d_o = d + step \mid d - step$ if $part\_id = 0|1$
8:     else if scheme == "Interleave" then
9:         $d_o = (d + step + \text{num\_device}) \% \text{num\_device}$
10:        if $d_o \neq d + step$ then $p_o = p - 1$
11:    else if scheme == ... then
12:        ...                                                  ▷ support other pipeline schemes
       return $d_o$, $m_o$, $p_o$, $t_o$
```

Lightweight Profiling. To support the simulator, we collect the computation time, memory footprint, and communication time through offline profiling. The profiling is designed to be lightweight, collecting only ten training iterations (e.g., profiling LLaMA2-13B only takes 142 seconds), as we find this to be sufficient based on our tests. We follow these guidelines: *1)* Treat the transformer block as the basic unit for profiling to exploit the repetitive structures of LLMs. *2)* Select the *(D-1)*-th device in the 1F1B scheme for profiling, as it typically contains several transformer blocks and has more available memory. *3)* Categorize the memory footprint into static part and dynamic components. *4)* Focus on the overall patterns of the warm-up, stable, and cool-down phases (defined in Ref.[27]) rather than individual send/recv instructions. With the profiled data, we build the performance estimators. We apply linear regression ($y = a \times n + b$) to predict execution time and static/dynamic memory based on the number of transformer blocks (n). And the bias b represents the framework overhead. Predicting *p2p* communication time follows a similar approach, as the warm-up and cool-down phases are fixed, so we use n to denote the number of micro-batches and apply linear regression.

Dynamic Programming Formulation. To derive the timeline for pipeline instructions, we develop a dynamic programming algorithm based on the estimators described above. The key is preserving the dependencies among instructions and gradually inferring the earliest start time of each instruction. Pipeline execution is organized by the inter-instruction dependencies, allowing an instruction to execute instantly once its dependencies are satisfied. The start time of instruction I is determined by the latest finish time of its preceding instructions: vertical dependencies (represented in virtual pipeline via `find_prev_inst` method) across devices and horizontal dependencies (containing dependencies to recomputation, embedded in ordered instruction list) within a device.

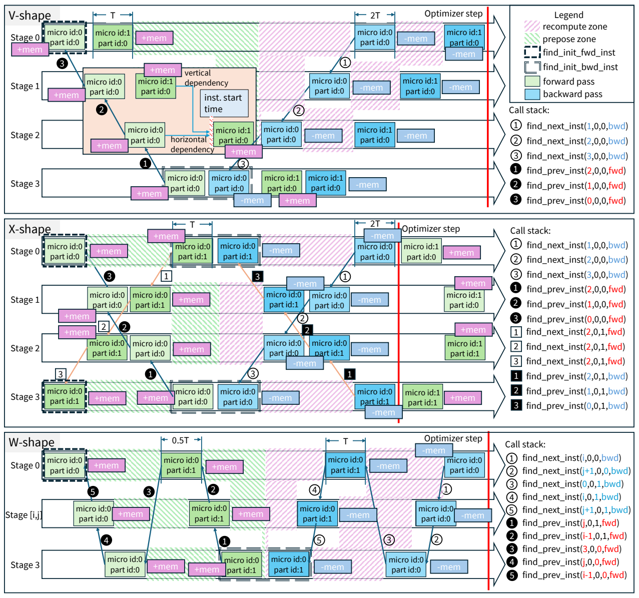
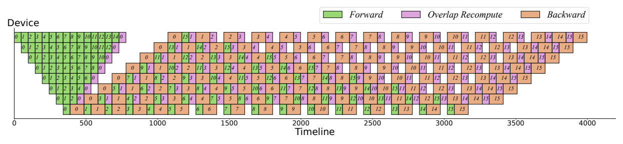

V-shape
X-shape
W-shape
Stage 0
Stage 1
Stage 2
Stage 3
Stage [i,j]
T
2T
0.5T
Optimizer step
micro id:0 part id:0
micro id:1 part id:0
micro id:0 part id:1
micro id:1 part id:1
+mem
-mem
vertical dependency
horizontal dependency
inst. start time
Legend
recompute zone
prepose zone
find_init_fwd_inst
find_init_bwd_inst
forward pass
backward pass
Call stack:
① find_next_inst(1,0,0,bwd)
② find_next_inst(2,0,0,bwd)
③ find_next_inst(3,0,0,bwd)
❶ find_prev_inst(2,0,0,fwd)
❷ find_prev_inst(1,0,0,fwd)
❸ find_prev_inst(0,0,0,fwd)
Call stack:
① find_next_inst(1,0,0,bwd)
② find_next_inst(2,0,0,bwd)
③ find_next_inst(3,0,0,bwd)
❶ find_prev_inst(2,0,0,fwd)
❷ find_prev_inst(1,0,0,fwd)
❸ find_prev_inst(0,0,0,fwd)
1 find_next_inst(2,0,1,fwd)
2 find_next_inst(2,0,1,fwd)
3 find_next_inst(2,0,1,fwd)
1 find_prev_inst(2,0,1,bwd)
2 find_prev_inst(1,0,1,bwd)
3 find_prev_inst(0,0,1,bwd)
Call stack:
① find_next_inst(i,0,0,bwd)
② find_next_inst(j+1,0,0,bwd)
③ find_next_inst(0,0,1,bwd)
④ find_next_inst(i,0,1,bwd)
⑤ find_next_inst(j+1,0,1,bwd)
❶ find_prev_inst(j,0,1,fwd)
❷ find_prev_inst(i-1,0,1,fwd)
❸ find_prev_inst(3,0,0,fwd)
❹ find_prev_inst(j,0,0,fwd)
❺ find_prev_inst(i-1,0,0,fwd)

Figure 4. Simulation process on `V|X|W`-shape pipeline.

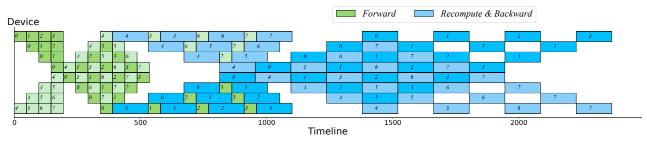
The detailed timeline simulation process is shown in Figure 4. It generally contains five main steps: *1)* Assign latency to each instruction using the pre-trained estimators. *2)* Initialize the *FW* instructions with no predecessor and mark them as "already initialized" (*find_init_fwd_inst*). *3)* Update the start time of the *FW* instructions. *4)* Identify the *BW* instructions that have only their corresponding *FW* as the predecessor (*find_init_fwd_inst*) and mark them. *5)* Update the start time of the *BW* instructions. Thus, the overall computation time is derived from the finish time of the last *BW* instructions across all devices.

The memory simulation is performed at device level. We accumulate the estimated static memory and track the peak dynamic memory (by accumulating the dynamic memory of all *FW* instructions). If RC_i follows FW_i, the dynamic memory of FW_i is accumulated to calculate the peak memory and then subtracted to maintain the current memory.

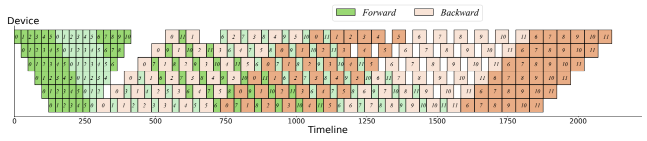
Visualization. *Mario* supports the visualization of the computational graphs. This feature allows users to intuitively observe pipeline execution states, the distribution of bubbles, and other critical aspects, rather than relying on solely on execution time and throughput. Such visualization facilitates fine-grained adjustments to checkpointed pipelines, enhancing the overall optimization process. Besides, Mario supports more pipelines (not limited to `V|X|W`-shape pipeline), through the virtual pipeline abstraction and heuristics, which is applicable to explore new pipeline structures. Some visualization examples are shown in Figure 5.



(a) 1F1B pipeline with *Mario* (all passes)



(b) Chimera pipeline with checkpointing (pass 1)



(c) Interleave pipeline without checkpointing

Figure 5. Pipeline visualization through *Mario* simulator.

Together with TP and DP. We treat *TP* as an internal block within pipeline stages. The changes in memory, additional communication time, and latency brought by *TP* directly affect the instructions and will be captured in profiling. As for *DP*, only the allreduce time in the cool-down phase of pipeline is modeled, with no need to duplicate pipeline simulations along the *DP* dimension.

5.3 Schedule Tuner

The schedule tuner optimizes parameter tuning based on *Mario* configuration and model configuration. It involves two key components: the graph tuner, which applies efficient activation checkpointing to the pipeline schemes, and the simulator-based performance model, which estimates the performance of pipeline schedules. Since *Mario* significantly reduces the memory footprint by freeing most activation tensors, we can increase the micro-batch size to better utilize the available memory. Although *Mario* primarily optimizes the pipeline parallelism (*PP*), it is also compatible with hybrid scenarios involving *PP*, *TP*, and *DP* parallelism. Besides, *Mario* can also reduce the devices used in the *PP* dimension, and leverage the idle devices in the *DP* dimension.

As shown in Equation 1, we formulate the parameter tuning as an optimization problem. The parameters to be tuned include: *1) a*: Whether to enable the activation checkpointing *Mario*, *2) b*: Which pipeline scheme should be selected, *3) pp*: The dimension of pipeline parallelism, *4) dp*: The dimension of data parallelism, *5) mbs*: The micro-batch size, and *6) dmem*: The available device memory. We denote the instruction-grained latency and memory estimator derived from profiling as E, and the simulator-based performance model as F. The primary objective is to maximize the training throughput estimated by F and E with the above parameters.

To support the schedule tuner, we extend the simulator-based performance model F. When the estimated peak memory exceeds *mem*, F returns zero as a penalty. We also extend F to support the *dp* parameter, which multiplies an efficiency coefficient to model the scalability of data parallelism. Additionally, we keep *TP* dimension constant, with its latency and memory impact already modeled by F and E.

For solving the optimization problem, we employ a grid search rather than more complicated heuristics or neural networks. The search space is manageable, and each iteration is lightweight, thanks to the efficiency and accuracy of the simulator, which can usually preserve the partial order (see Figure 10). Note that without our simulator, each search iteration on real devices would take several minutes, which is time-consuming.

$$\begin{aligned} \max \quad & F(b, E(a, b, pp, mbs), dmem, dp) \\ s.t. \quad & a \in \{True, False\} \\ & b \in \{1F1B, Chimera, Interleave\} \\ & 4 \le pp \le D \\ & dp \times pp = D \\ & mbs \in \{1, 2, 4, 8, \ldots\} \\ & dmem \text{ available device memory specified by users} \\ & D \text{ \# of available devices} \end{aligned} \tag{1}$$

6 Evaluation

In this section, we apply *Mario* to three state-of-the-art (SOTA) synchronous pipeline schemes: 1F1B [7], Chimera [23], and Interleave [31]. And we evaluate *Mario* on training two GPT3 models (with 1.6B and 13B parameters) and two LLaMA2 models (with 3B and 13B parameters). The model configurations are summarized in Table 4.

All experiments are conducted on a cluster with 16 nodes, each equipped with two 64-core ARMv8 CPUs (Kunpeng 920) and 4 NVIDIA A100-40G GPUs (driver 510.85.02, CUDA 11.8, and cuDNN 8.4). *Mario* is implemented on the Megatron-DeepSpeed (v0.2.0) framework with DeepSpeed (v0.11.2). We use the 1F1B implementation provided by Megatron-DeepSpeed. For Chimera and Interleave, we directly pick their open-source pipeline schedules[†]. To ensure a fair comparison, we execute their schedules with *Mario* by transforming them into the instruction lists used by *Mario*.

We evaluate four configurations: *1)* `base` ("baseline") represents the original pipeline scheme without any extra optimization. *2)* `ckpt` ("checkpoint") represents the pipeline scheme with activation checkpointing enabled. *3)* `ovlp` ("overlap") represents the `ckpt` optimized by the four passes of *Mario*. *4)* `lmbs` ("larger micro-batch size"). We further increase the micro-batch size on `ovlp` configuration, while keeping the global batch size unchanged (resulting in fewer schedulable micro-batches), because larger global batch size

[†]The pipeline schedules of Chimera [23] and Interleave [31] are directly picked from https://github.com/Shigangli/Chimera/blob/main/chimera_pipeline_rank.py & https://github.com/NVIDIA/Megatron-LM/blob/main/megatron/core/pipeline_parallel/schedules.py.

could lead to convergence problem [41, 43]. For simplicity, we denote the pipeline schemes by their visualization shape, V (1F1B), X (Chimera), and W (Interleave). Besides, we adopt the *training throughput* as the performance metric, because it is the ultimate of other intermediate metrics (e.g., bubble ratio).

Table 4. LLMs in experiments.

Model	Hidden Size	Layers	Attention Heads	seqlen
GPT3-1.6B	1024	128	16	1024
GPT3-13B	3000	128	40	1024
LLaMA2-3B	2048	64	16	1024
LLaMA2-13B	4096	64	32	1024

6.1 Performance on GPT3-1.6B and LLaMA2-3B

We evaluate *Mario* using a pipeline with 8 A100-40G GPUs against the baselines `V-base`, `X-base` and `W-base`. We use GPT3-1.6B and LLaMA2-3B to avoid out-of-memory (OOM) exceptions caused by imbalanced activation memory in the baselines, so that we can compare different configurations. The global batch size is set to 128 and the micro-batch size is set the same as Table 5 (i.e., the Micro BS column).

As shown in Figure 6, `lmbs` achieves the highest throughput, followed by `base`, `ovlp` and `ckpt`. Specifically, compared to `base`, `lmbs` achieves 1.16× (V), 1.00× (V), 1.23× (X), 1.17× (X), 1.52× (W), and 1.40× (W) speedup, with an average speedup of 1.25×. Through activation checkpointing, the peak memory footprint can be reduced to 1/2 to 1/3 of `base`. With this reduced memory usage, we can double the micro-batch size to utilize available memory and improve computational efficiency. Besides, the reduced number of micro-batches lead to less *p2p* communication across GPUs, indicating that *Mario* can outperform the baselines even without checkpointing.

Compared to `base`, `ovlp` shows a 29.22% slowdown on average. Although this may seem inconsistent with the "near zero-cost" claim, this slowdown is due to the extra overhead that prevents *BW* from fully overlapping with *RC* and *FW* of the previous device. However, as the ratio of transformer computation increases(e.g., by increasing model size or micro-batch size), this overhead can be migrated, allowing *Mario* to achieve near zero-cost in those scenario (see the gray cells in Table 5). The performance improvement from `ovlp` to `ckpt` is due to that *Mario* applys pass 2&3&4 to `ckpt`.

Additionally, Interleave (W) is designed to consume more memory to reduce bubbles. Theoretically (c.f. Table 1), with D devices in the pipeline, the peak activation memory of is $(3D-2) \times M_\theta/2$ with Interleave (W), larger than $D \times M_\theta$ with 1F1B (V). In evaluation, we have configured the micro-batch size to allow 1F1B (V) fill up the GPU memory, using the same micro-batch size for Interleave (W) would result in out-of-memory exception. To maintain a consistent global batch size among V|X|W, the micro-batch size for

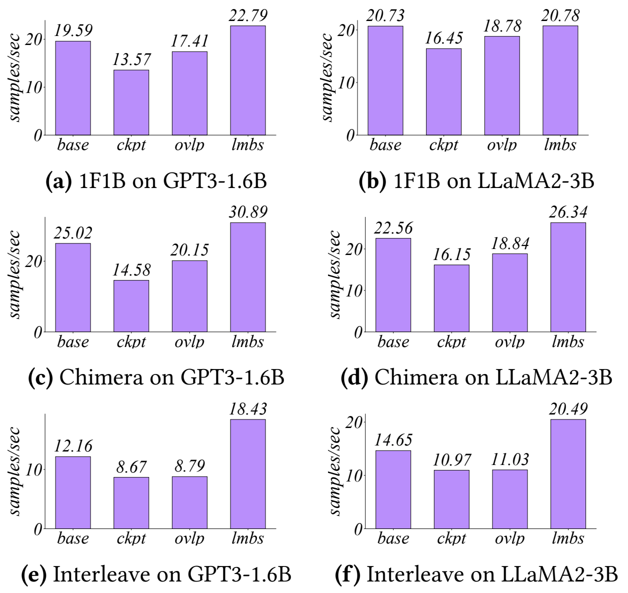


Figure 6. Performance on GPT3-1.6B and LLaMA2-3B with 8 A100-40G GPUs.

`W-base|ckpt|ovlp|lmbs` is set to half of `V|X`. As a result, the training throughput for W is lower than others.

Note that he purpose of our evaluation is to demonstrate the speedup after applying *Mario* to different pipeline schemes, and we have no intention to benchmark different pipeline schemes.

6.2 Performance on GPT3-13B and LLaMA2-13B

We evaluate *Mario* with a pipeline containing 32 A100-40G GPUs . When the models are large enough and have to be trained on 32 GPUs, the main reason of OOM is the imbalanced activation memory footprint across devices, making the GPU memory underutilized. Compared to `ckpt`, `ovlp` and `lmbs` achieves 1.13× and 1.36× speedup, respectively. By using `ckpt`, `ovlp`, and `lmbs`, the activation memory can be eliminated, thereby balancing the memory footprint across devices. This allows for adopting larger micro-batch sizes to further improve the device utilization. Consequently, `lmbs` shows superior performance compared to `ckpt` and `ovlp`.

Notably, on LLaMA2-13B, `V-ovlp` achieves 94.7% throughput of `V-base`, indicating *Mario* incurs only ~5% overhead compared to `V-base` baseline. In `V-ovlp`, *Mario* employs activation checkpointing without changing micro-batch size, global batch size or other hyper-parameters. Therefore, ***the recomputation overhead is almost entirely hidden, reinforcing the "near zero-cost" claim in our paper title***.

6.3 Peak Memory Footprint

We collect statistics on peak memory footprint when training GPT3 (1.6B and 13B parameters) and LLaMA2 (3B and

Table 5. Performance on GPT3-13B and LLaMA2-13B with 32 A100-40G GPUs. The underlined values are estimated by *Mario* simulator, which is originally OOM.

	Config	Global BS	Micro BS	Memory (Min,Max GB)	Throughput (samples/sec)
GPT3-13B	V-base	128	2	[10.35, 122.41]	20.42
	V-ckpt	128	2	[9.85, 14.10]	14.13
	V-ovlp	128	2	[9.85, 14.10]	17.68
	V-lmbs	128	4	[12.29, 16.61]	18.37
	X-base	128	2	[71.90, 124.27]	23.19
	X-ckpt	128	2	[19.53, 27.61]	19.13
	X-ovlp	128	2	[19.53, 27.61]	22.24
	X-lmbs	128	4	[19.78, 28.59]	23.40
	W-base	128	1	[65.270, 178.97]	22.23
	W-ckpt	128	1	[9.77, 13.62]	13.81
	W-ovlp	128	1	[9.77, 13.62]	14.10
	W-lmbs	128	2	[9.87, 14.11]	21.48
LLaMA2-13B	V-base	128	2	[8.17, 35.85]	25.11
	V-ckpt	128	2	[7.90, 11.37]	21.80
	V-ovlp	128	2	[7.90, 11.37]	24.78
	V-lmbs	128	4	[8.06, 12.06]	26.55
	X-base	128	2	[30.06, 44.34]	28.76
	X-ckpt	128	2	[15.67, 22.58]	24.46
	X-ovlp	128	2	[15.67, 22.58]	27.90
	X-lmbs	128	4	[15.86, 23.27]	29.48
	W-base	128	1	[13.60, 29.74]	17.83
	W-ckpt	128	1	[7.85, 11.89]	16.21
	W-ovlp	128	1	[7.85, 11.89]	17.16
	W-lmbs	128	2	[7.94, 11.92]	26.95

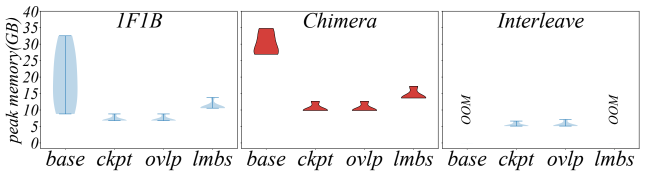
13B parameters) models on 8 and 32 A100-40G GPUs, respectively. As shown in Figure 7, activation checkpointing significantly reduces and balances the memory usage across devices. Specifically, compared to `ckpt`, the `ovlp` does not introduce any additional memory footprint. The `lmbs`, which increases the micro-batch size, uses some extra memory to achieve higher SM utilization and memory utilization. Despite this, even with `lmbs`, the memory footprint across different devices is still more balanced than `base`.

As a result, state-of-the-art pipeline schemes (e.g., `V`, `X`, `W`) often exhibit high but imbalanced peak memory usage, leading to frequent OOM exception. This highlights the need for efficient memory optimizations, which *Mario* provides through a systematic design.

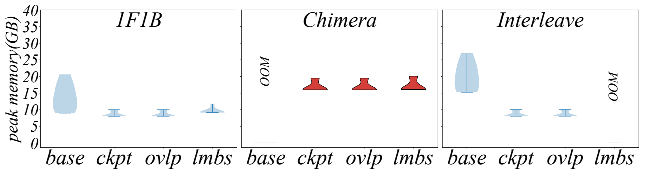
6.4 Model Parameters Scaling

We conduct model parameter scaling with a pipeline containing 16 A100-40G GPUs, focusing on the GPT3 model. The sequence length is set to 1024, the number of transformer layers to 64, and the number of attention heads to 32, with a global batch size of 64. We scale the number of parameter by adjusting the hidden layer size, starting from 512 and increasing it by 256 each time until OOM occurs.

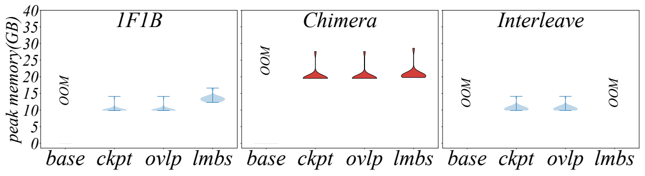
As shown in Figure 8, *Mario* has significantly increased the feasible parameter size. Specifically, `V-base` can handle 3B parameters, but after *Mario* optimizations, `V-ovlp` and `V-lmbs` can handle 16B parameters, resulting in a 5.3× improvement. Similarly, `X-ovlp` and `X-lmbs` can now handle 7B parameters, which is 2.3× larger than `X-base`. `W-ovlp` and `W-lmbs` can handle ~20× the parameter size of `W-base`. The high throughput of Chimera (`X-lmbs`) is due to its bidirectional design, but maintaining two replicas of the model parameters limits its scalability to larger models.



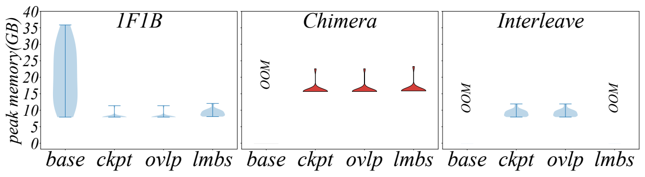
(a) GPT3-1.6B



(b) LLaMA2-3B



(c) GPT3-13B



(d) LLaMA2-13B

Figure 7. Peak memory footprint across devices.

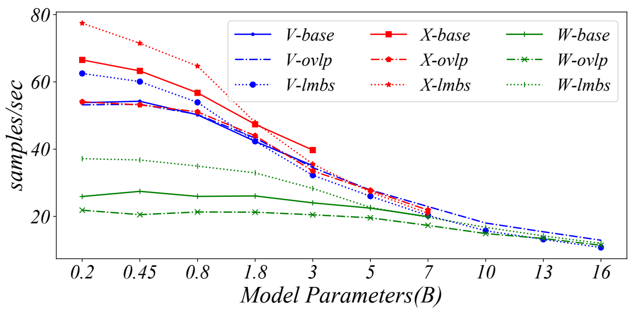


Figure 8. Model parameters scaling on GPT3 models with 16 A100-40G GPUs.

6.5 Sequence Length Scaling

We conduct sequence length scaling using a pipeline with 16 A100-40G GPUs and the GPT3-1.6B model. The micro-batch size is set to 1, and the global batch size is twice the number of stages. We increase the sequence length from 1024 by 64 at a time untill an OOM occurs. We test three configurations: *a) PP:8 TP:1*, *b) PP:8 TP:2*, and *c) PP:8 TP:2* optimized by *Mario*, where the numbers represent *PP/TP* dimension.

As shown in Figure 9, *PP:8 TP:2* optimized by *Mario* supports a larger sequence length than others, with an average increase of 1.49× over *PP:8 TP:2* and 2.80× over *PP:8 TP:1*. This improvement is due to *Mario* eliminating a significant amount of activation memory, allowing the saved memory to accommodate longer sequence length.

6.6 Simulator Accuracy

To evaluate the accuracy of our simulator-based performance model, we conduct experiments using GPT3-1.6B on 8 A100-40G GPUs. As shown in Figure 10, the estimated peak memory closely matches the actual measurements, with a mean absolute percentage error (MAPE) of only 5.1%. The simulator also reveals that the framework (including Megatron, DeepSpeed, PyTorch, CUDA context, etc.) consumes about 2GB GPU memory. While *Mario* tends to slightly overestimate the throughput due to some un-modeled behaviors, and the MAPE is 9.4%. Importantly, the partial order in estimated throughput aligns well with real runs, making it sufficient for further parameter tuning.

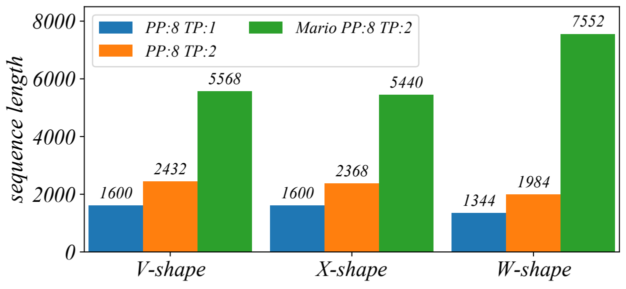
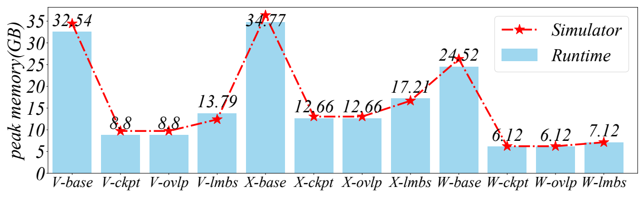


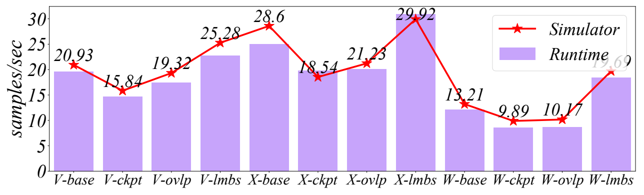
Figure 9. Sequence length scaling on GPT3-1.6B with 16 A100-40G GPUs.

6.7 Cluster Experiment

We conduct experiments using the GPT3-13B model on a cluster with 64 A100-40G GPUs. In contrast to previous experiments, we introduce data parallelism here. The throughput curve of parameter tuning is shown in Figure 11. The labels (in the format of *x-y-z*) represents the configurations, where *x*, *y*, and *z* represents the pipeline scheme, *PP* dimension, and micro-batch size, respectively. *TP* dimension is set to 1 to allow for a wide range of *PP* scaling, and *DP* dimension is set to $64/PP/TP$ to ensure utilizing all GPUs.



(a) On peak memory.



(b) On training throughput.

Figure 10. Accuracy of *Mario* simulator.

The best configurations of 1F1B, Chimera, and Interleave are *V-64-16*, *X-64-16*, and *W-64-32* with *Mario* enabled. It is obvious that choosing a proper *PP* dimension is crucial. If *PP* is increased while keeping other parameters unchanged, throughput drops sharply (e.g., *V-16-1* → *V-32-1 to V-64-1*). However, as the micro-batch size increasing, throughput rises significantly (e.g., *V-64-1* → *V-8-2*). Generally, a larger *PP* paired with larger micro-batch size yields higher throughput, which validates why (`lmbs`) can enhance training throughput. With *Mario*, the *PP* dimension can reach 64 on 64 GPUs. However, as shown in Table 5, without *Mario*, using 32 GPUs will trigger OOM, especially with larger *PP* dimension, making feasible configurations far from optimal.

The optimal configuration identified through parameter tuning aligns the manually tuned configuration. The total tuning time for this experiment is 210 seconds, whereas manually configuring the parameters and making adjustments based on log feedback takes approximately 10 minutes once. Besides, we have tested the tuning on 1024-GPU scenario and it only takes 1060 ms per iteration with 240 configurations.

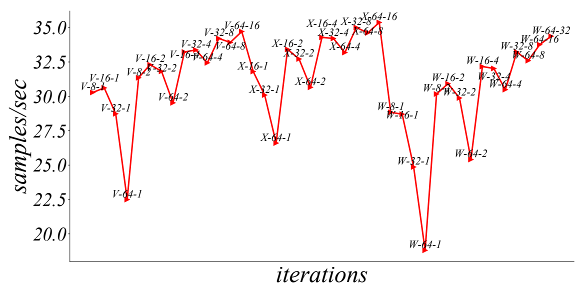


Figure 11. Throughput curve along tuning iterations.

7 Discussion

7.1 Pipeline Stage Partition

Through stage partition, either computation balance or memory balance can be achieved, but not both. The original pipelines show *descending* activation memory footprint along GPU indexes. To balance the memory, one can partition the stages with *ascending* workloads (Transformer layers). However, it inevitably increases the computation imbalance, which is even worse for the pipeline.

In fact, even partitioning is commonly adopted by the state-of-the-art pipeline schemes (e.g., Chimera [23], Hanayo [28]), mainstream parallel frameworks (i.e., Megatron-LM), and other memory optimization techniques (e.g., BPipe [20]). There are indeed papers leveraging imbalanced stages, *however, their partitioning is only slightly uneven*, which is not enough to balance activation memory. For example, AdaPipe [39] partitions GPT3-175B with 23∼26 layers per stage, and the average speedup compared to even partitioning is only <4%.

When implementing *Mario* before, we have also performed some demonstrating experiments on GPT3-1.6B (128 Transformer layers) with 8 GPUs. We adopt a strategy of *varying k layers uniformly across stages*, where $k \in -1, -2, 0, +1, +2$. Only $k = -1$ shows performance improvement (around 3%) when without checkpointing. However, in that scenario, the stage closer to the last stage would have less space to hide the recomputation, leading to 2∼3% performance degradation.

7.2 Memory Fragmentation

Memory fragmentation is common for *dynamic fine-grained* checkpointing (e.g., DTR [21], MegTaichi [11], AdaPipe [39]). However, *Mario* adopts the *static* (drop and then recompute all activation tensors without runtime estimation) *coarse-grained* (the whole pipeline stage) checkpointing, and thus has much less fragmentation. Besides, fragmentation-related optimizations can be directly applied to *Mario*.

7.3 Apply *Mario* to Larger Systems

Due to hardware limitations, we could not access more GPUs, and thus failed to further scale *Mario* to 100∼1000 GPUs. However, note that our evaluations on 8-GPU and 32-GPU pipelines can already represent large-scale scenarios. Specifically, according to the best practices of LLM training (c.f. Megatron-LM), the *PP* dimension typically ranges from 4 to 16. For example, when using 6,144 H100 GPUs to train a 462B model, the *PP* dimension is only 16=6144(# of GPUs)÷48(DP-dimension)÷8(TP-dimension). Therefore, our 32-GPU pipeline is already large enough.

Besides, larger systems should involve ZeRO-DP/Offload, TP, and other emerging techniques. We will discover their "bubbles" (i.e., idle device time) and better hide the recomputation of *Mario* under these newly discovered bubbles.

8 Related works

Tackling Memory Wall in LLM Training. An critical issue in training LLM is the memory wall [9, 28, 36]. To address this, parallel training frameworks such as Megatron-LM [22, 31], DeepSpeed [35, 36], and Colossal-AI [24] distribute memory footprint across devices by partitioning tensors along different dimensions. ZeRO [35, 36] and FSDP [48] partitions the optimizer states. ZeRO-Offload [38], PatrickStar [8], and Sentinel [37] utilize heterogeneous memory for offloading. Some works [4, 15, 21, 39] can trade computation for memory through activation checkpointing.

Performance Models in LLM Training. Reconciling parallel schemes in large search space impacts the efficiency of LLM training. Therefore, performance models are needed to guide the space searching. Chimera [23], MixPipe [47], ZB-H1 [34], and AdaPipe [39] have built performance models for their pipelines. DynaPipe [17] build cost models for dynamic micro-batching planer. Tessel [27] uses repetend-centered model to guild pipeline design. AutoDDL [3], Crius [44] models hybrid parallelization scenarios. FasterMoe [10] models MOE training with roofline-like models.

Leveraging Pipeline Bubbles. Pipeline bubbles can overlap tasks including checkpointing. PipeFisher [32] overlaps the K-FAC optimizer. Bamboo [40] overlaps redundant computation for fault tolarance. BPipe [20] overlaps activation transmission for balanced memory. And *Mario* can further adopt the split backward parts of ZB-H1 [34] to overlap remaining bubbles, which is our future work.

9 Conclusion

With *Mario*, we successfully demonstrate the feasibility of overlapping activation checkpointing with the bubbles in pipeline parallelism schemes, achieving near zero-cost memory optimization. By leveraging the eliminated memory space, *Mario* enables the increase of micro-batch size without altering the global batch size to enhance training throughput and supports longer sequence length for improved model quality. In the future, we hope that *Mario* will inspire greater attention to the potential of near zero-cost activation checkpointing across all parallelism dimensions in LLM training.

Acknowledgments

This work is supported by the following funding: National Science Foundation of China (No. 92270206, T2125013, 62372435 62032023, 61972377, T2293702), CAS Project for Young Scientists in Basic Research (No. YSBR-005), Beijing Natural Science Foundation (No. 4254087), China National Postdoctoral Program for Innovative Talents (No. BX20240383). Mingzhen Li and Weile Jia are the corresponding authors.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt,

Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4, Article 65 (aug 2019), 43 pages. https://doi.org/10.1145/3320060

[3] Jinfan Chen, Shigang Li, Ran Guo, Jinhui Yuan, and Torsten Hoefler. 2024. AutoDDL: Automatic Distributed Deep Learning With Near-Optimal Bandwidth Cost. *IEEE Transactions on Parallel and Distributed Systems* 35, 8 (2024), 1331–1344. https://doi.org/10.1109/TPDS.2024.3397800

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

[5] Daning Cheng, Shigang Li, Hanping Zhang, Fen Xia, and Yunquan Zhang. 2021. Why Dataset Properties Bound the Scalability of Parallel Machine Learning Training Algorithms. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1702–1712. https://doi.org/10.1109/TPDS.2020.3048836

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[7] 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. 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* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 431–445. https://doi.org/10.1145/3437801.3441593

[8] Jiarui Fang, Yang Yu, Zilin Zhu, Shenggui Li, Yang You, and Jie Zhou. 2021. PatrickStar: Parallel Training of Pre-trained Models via a Chunk-based Memory Management. *arXiv preprint arXiv:2108.05818* (2021).

[9] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall . *IEEE Micro* 44, 03 (May 2024), 33–39. https://doi.org/10.1109/MM.2024.3373763

[10] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 120–134. https://doi.org/10.1145/3503221.3508418

[11] Zhongzhe Hu, Junmin Xiao, Zheye Deng, Mingyi Li, Kewei Zhang, Xiaoyang Zhang, Ke Meng, Ninghui Sun, and Guangming Tan. 2022. MegTaiChi: dynamic tensor-based memory management optimization for DNN training. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 25, 13 pages. https://doi.org/10.1145/3524059.3532394

[12] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. https://doi.org/10.1145/3373376.3378530

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[14] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Los Angeles, CA, USA, 776–789. https://doi.org/10.1109/ISCA.2018.00070

[15] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 497–511. https://proceedings.mlsys.org/paper_files/paper/2020/file/0b816ae8f06f8dd3543dc3d9ef196cab-Paper.pdf

[16] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 673–688. https://www.usenix.org/conference/atc22/presentation/jia-xianyan

[17] 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* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 542–559. https://doi.org/10.1145/3627703.3629585

[18] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[19] 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).

[20] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPipe: Memory-Balanced Pipeline Parallelism for Training Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, Red Hook, NY, USA, 16639–16653. https://proceedings.mlr.press/v202/kim23l.html

[21] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations*.

[22] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023).

[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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. https://doi.org/10.1145/3458817.3476145

[24] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proceedings of the 52nd International Conference on Parallel Processing* (Salt Lake City, UT, USA) *(ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 766–775. https://doi.org/10.1145/3605573.3605613

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[26] Jianjin Liao, Mingzhen Li, Hailong Yang, Qingxiao Sun, Biao Sun, Jiwei Hao, Tianyu Feng, Fengwei Yu, Shengdong Chen, Ye Tao, Zicheng Zhang, Zhongzhi Luan, and Depei Qian. 2023. Exploiting Input Tensor Dynamics in Activation Checkpointing for Efficient Training on GPU. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, St. Petersburg, FL, USA, 156–166. https://doi.org/10.1109/IPDPS54959.2023.00025

[27] Z. Lin, Y. Miao, G. Xu, C. Li, O. Saarikivi, S. Maleki, and F. 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 Computer Society, Los Alamitos, CA, USA, 803–816. https://doi.org/10.1109/HPCA57654.2024.00067

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

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 7937–7947. https://proceedings.mlr.press/v139/narayanan21a.html

[31] 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. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[32] Kazuki Osawa, Shigang Li, and Torsten Hoefler. 2023. PipeFisher: Efficient training of large language models using pipelining and Fisher information matrices. *Proceedings of Machine Learning and Systems* 5 (2023), 708–727.

[33] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. https://doi.org/10.1145/3373376.3378505

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

[35] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[36] 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[37] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 598–611. https://doi.org/10.1109/HPCA51647.2021.00057

[38] 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)*. USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[39] 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 86–100. https://doi.org/10.1145/3620666.3651359

[40] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 497–513. https://www.usenix.org/conference/nsdi23/presentation/thorpe

[41] 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)*. USENIX Association, Santa Clara, CA, 563–578. https://www.usenix.org/conference/atc24/presentation/um

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[43] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 195–210. https://doi.org/10.1145/3694715.3695975

[44] Chunyu Xue, Weihao Cui, Han Zhao, Quan Chen, Shulai Zhang, Pengyu Yang, Jing Yang, Shaobo Li, and Minyi Guo. 2024. A Codesign of Scheduling and Parallelization for Large Model Training in Heterogeneous Clusters. *arXiv preprint arXiv:2403.16125* (2024).

[45] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. 2021. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 269–296. https://proceedings.mlsys.org/paper_files/paper/2021/file/9412531719be7ccf755c4ff98d0969dc-Paper.pdf

[46] Siling Yang, Weijian Chen, Xuechen Zhang, Shuibing He, Yanlong Yin, and Xian-He Sun. 2021. AUTO-PRUNE: automated DNN pruning and mapping for ReRAM-based accelerator. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 304–315. https://doi.org/10.1145/3447818.3460366

[47] Weigang Zhang, Biyu Zhou, Xuehai Tang, Zhaoxing Wang, and Songlin Hu. 2023. MixPipe: Efficient Bidirectional Pipeline Parallelism for Training Large-Scale Models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247730

[48] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12, 3848–3860. https://doi.org/10.14778/3611540.3611569

[49] 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. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin