Mist: Efficient Distributed Training of Large Language Models via Memory-Parallelism Co-Optimization

Zhanda Zhu
University of Toronto, Vector Institute, CentML
zhanda.zhu@mail.utoronto.ca

Christina Giannoula
University of Toronto, Vector Institute, CentML
christina.giann@gmail.com

Muralidhar Andoorveedu
CentML
murali@centml.ai

Qidong Su
University of Toronto, Vector Institute, CentML
qdsu@cs.toronto.edu

Karttikeya Mangalam
SigIQ.ai
mangalam@sigiq.ai

Bojian Zheng
University of Toronto, Vector Institute, CentML
bojian@cs.toronto.edu

Gennady Pekhimenko
University of Toronto, Vector Institute, CentML
pekhimenko@cs.toronto.edu

Abstract

Various parallelism, such as data, tensor, and pipeline parallelism, along with memory optimizations like activation checkpointing, redundancy elimination, and offloading, have been proposed to accelerate distributed training for Large Language Models. To find the best combination of these techniques, automatic distributed training systems are proposed. However, existing systems only tune a subset of optimizations, due to the lack of overlap awareness, inability to navigate the vast search space, and ignoring the inter-microbatch imbalance, leading to sub-optimal performance. To address these shortcomings, we propose Mist, a memory, overlap, and imbalance-aware automatic distributed training system that comprehensively co-optimizes all memory footprint reduction techniques alongside parallelism. Mist is based on three key ideas: (1) fine-grained overlap-centric scheduling, orchestrating optimizations in an overlapped manner, (2) symbolic-based performance analysis that predicts runtime and memory usage using symbolic expressions for fast tuning, and (3) imbalance-aware hierarchical tuning, decoupling the process into an inter-stage imbalance and overlap aware Mixed Integer Linear Programming problem and an intra-stage Dual-Objective Constrained Optimization problem, and connecting them through Pareto frontier sampling. Our evaluation results show that Mist achieves an average of 1.28× (up to 1.73×) and 1.27× (up to 2.04×) speedup compared to state-of-the-art manual system Megatron-LM and state-of-the-art automatic system Aceso, respectively.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords: LLM, Systems for Machine Learning, Distributed training

ACM Reference Format:
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 *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3689031.3717461

1 Introduction

Large Language Models (LLMs) have gained high interest and show remarkable capabilities in various fields like question answering, summarization, problem solving, and more [5, 23, 60]. However, their significantly increased sizes and dataset requirements have escalated computational and memory demands. For instance, training LLaMa-3.1-405B [50] uses a cumulative 30.84M GPU hours of computation on NVIDIA H100 GPUs [51]. While most companies and researchers cannot afford to pre-train such LLMs, continuous pre-training or supervised fine-tuning still costs over 2000 NVIDIA H100 GPU hours per 1B tokens [51]. Therefore, efficient distributed training techniques have been proposed [6, 12, 40–42, 45, 46, 49, 53, 68, 73, 76, 77, 80, 82, 85, 87, 89] to improve system performance during training, since even minor reductions in

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3717461

the training time are translated to significant financial and environmental benefits [22, 52, 74].

Prior works [35, 49, 55, 57, 68, 73, 89] propose various parallelization techniques for distributed training. Data Parallelism (DP) [1, 42] splits input data across devices, with each device processing a portion of the data, while maintaining a full copy of the LLM model. Tensor Parallelism (TP) [39, 57, 73] partitions the parameters of each layer across devices, however introducing inter-device communication over activations to ensure computation correctness. Pipeline Parallelism (PP) [32, 35, 43, 55] divides the model into stages, however a large number of stages may lead to performance inefficiencies, e.g., pipeline bubbles where devices are idle during training. We henceforth refer to the number of partitions in DP, TP and PP methods as DP size, TP size, and PP size, respectively. Gradient accumulation techniques are usually applied, dividing a global batch into multiple microbatches, to reduce the memory pressure of each microbatch and facilitate pipeline parallelism [73]. To alleviate memory pressure in devices, memory footprint reduction techniques [3, 17, 26, 31, 36, 38, 63, 66, 67, 69, 70, 87, 88] have also been proposed. Activation checkpointing (CKPT) [17, 36, 38, 88] reduces memory footprint during training by recomputing activations during backpropagation. ZeRO [66, 87] eliminates model states redundancy by partitioning the optimizer states, gradients, or weights across devices. Higher ZeRO levels partition more model states, thereby providing larger memory footprint savings, however increasing the inter-device communication. Offloading [26, 31, 63, 67, 69, 70] temporarily transfers unused tensors from the GPU to the CPU, freeing GPU memory however increasing the data transfer costs. These memory optimizations often require overlapping data transfers with GPU computation to reduce performance overhead [63, 69, 87].

In this work, we observe that memory footprint reduction techniques, although they have been primarily designed to alleviate memory pressure, they can significantly enhance performance, since they assist in balancing trade-offs between runtime overhead and memory footprint reduction. For instance, applying offloading optimization can free up some memory in GPU devices, which can then be leveraged to reduce the TP or PP size, thereby reducing communication overheads or pipeline bubbles during training. Generally, exploiting memory footprint reduction techniques to release some memory footprint in GPU devices can be leveraged to: 1) reduce the TP size, thus mitigating communication overheads; 2) reduce the PP size, thus eliminating pipeline bubbles; and (3) increase the batch size, improving kernel efficiency. Conversely, applying less aggressively memory footprint reduction optimizations results in higher GPU memory usage, which increases the partitioning across devices, thus incurring higher performance overheads related to parallelism. Therefore, additional GPU memory can be gained by applying more aggressive memory footprint reduction techniques, which come with some added overhead. This memory can then be used to reduce the overhead of other optimizations. As long as the benefit from reducing the overhead outweighs the additional cost incurred by the memory footprint reduction techniques, overall training efficiency improves. See detailed motivational examples in Section 3.1.

Overall, distributed training constitutes an optimization problem that can be formulated as choosing the best combination of all available techniques (both parallelism and memory footprint reduction techniques) to maximize training efficiency, while keeping the memory usage lower than the available hardware memory capacity. Manual distributed training methods such as Megatron-LM [73] and DeepSpeed [68], among others [67, 69, 87], are developed to provide some of the above optimizations. However, these manual methods require users to specify configurations, i.e., the combination of parallelism and memory footprint reduction techniques, for optimal performance. This can be quite challenging even for experienced users and takes lots of engineering efforts [89]. Moreover, as model sizes and device counts increase, tuning complexity increases exponentially [89]. To address this issue, automatic distributed training systems have been proposed [12, 40, 45, 46, 49, 53, 76, 77, 80, 82, 85, 89]. Given LLM model and GPU hardware, they construct the search space of various configurations of the their supported optimizations and automatically find optimal combination of them.

We extensively examine distributed training frameworks and identify a key shortcoming: they fail to comprehensively co-optimize memory footprint reduction techniques alongside parallelism, since they only focus on a subset of the available search space. Specifically, these systems are still inefficient in optimizing training performance, because they (i) either tune parallelism configuration with a fixed pre-defined memory optimization [49, 53, 89], (ii) support only one specific optimization like activation checkpointing [40, 46, 76], or (iii) make strong assumptions, such as applying the same memory footprint strategy across all pipeline stages [85]. These constraints lead to a reduced search space and suboptimal performance, as demonstrated in Section 3.1.

We analyze how prior works tune the training configurations and find that co-optimizing all available memory footprint reduction techniques and parallelism is challenging in these prior existing systems, because they suffer from three limitations. First, existing automatic systems do *not* overlap communication with computation beyond the basic gradient all-reduce, thus missing important opportunities for training efficiency. This can cause severe performance degradation (See Figure 12), which becomes even more severe when all memory optimizations are involved. Second, they are not able to efficiently explore the exploded search space when co-tuning all optimizations. When more memory optimization techniques are incorporated, the search space increases significantly, and existing systems fail to find the best configuration in such a huge search space. Third, they

use the averaged microbatch time to model the pipeline parallelism performance, implicitly assuming uniform microbatch execution time within a pipeline stage. However, we find that this is not the case, since first and last microbatches incur higher communication costs (especially when ZeRO and offloading are involved), as we explain in Section 3.2.

To tackle the aforementioned limitations, we propose Mist, a memory, overlap, and imbalance-aware automatic distributed training system that co-optimizes memory footprint reduction techniques with parallelism. Mist consists of three key ideas: (1) **fine-grained overlap-centric scheduling**, which carefully orchestrates the implementation of both memory footprint reduction techniques and parallelism to maximize the computation-communication overlapping; (2) **symbolic-based efficient performance analysis**, which enables fast exploration of the exploded search space of various configurations by efficiently predicting runtime and memory usage via symbolic expressions and batched value substitutions; and (3) **imbalance-aware hierarchical tuning**, which takes into account the microbatch variability and overlap opportunities in pipeline parallelism, decouples the optimization process into an inter-stage Mixed Integer Linear Programming (MILP) problem and an intra-stage Dual-Objective Constrained Optimization problem, and connects them via Pareto frontier sampling. This third key idea addresses both the search space explosion and the microbatch imbalance in pipeline parallelism.

We extensively evaluate Mist on a wide variety of LLMs, including GPT-3 [9], LlaMa [79], and Falcon [2], across diverse training configurations, i.e., varying the global batch size, model size, and using different kernel implementations such as FlashAttention [19]) and hardware setups (up to 32 NVIDIA L4 [59] and 32 NVIDIA A100 GPUs [58]), and demonstrate that Mist significantly outperforms prior state-of-the-art works [46, 68, 73]. Our evaluation results show that Mist achieves an average of 1.28× (up to 1.73×) and 1.27× (up to 2.04×) speedup compared to the state-of-the-art manual implementation Megatron-LM and the state-of-the-art automatic system Aceso, respectively, across different GPU, model, and training configurations.

To sum up, we make the following contributions:

- We identify the need of comprehensively co-optimizing memory footprint reduction techniques alongside parallelism and propose Mist, a highly efficient easy-to-use automatic distributed training framework for LLMs.
- We propose and implement a symbolic analysis system that generates symbolic expressions for workload characteristics to quickly explore the exploded search space. We design a scheduling method that maximizes computation-communication overlap by carefully coordinating memory optimization and parallelism techniques. We introduce a tuning method that decouples the optimization process into two stages and connects them through Pareto frontier sampling, addressing microbatch variability and leveraging overlap opportunities in pipeline parallelism.
- We evaluate Mist using various large-scale LLMs in both NVLink systems (NVIDIA A100 GPUs [58]) and PCIe systems (NVIDIA L4 GPUs [59]) and compare it to multiple strong baselines. Mist significantly outperforms prior works, up to 2.04×, under various training configurations.

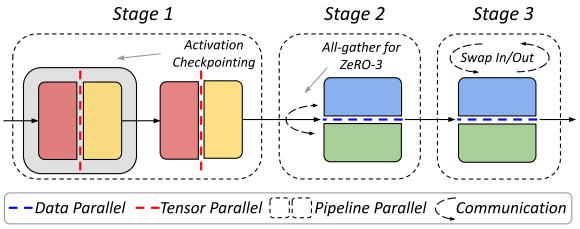


Figure 1. An illustration of optimization configurations.

2 Background

LLMs [9, 78] require excessive computation and memory, leading to significant costs and energy consumption [51]. Consequently, distributed training, i.e., scaling hardware and splitting the model and/or the input data, is the typical solution to train LLMs [68, 73]. To efficiently conduct distributed training, various parallelism techniques [35, 42, 73] and GPU memory footprint reduction methods [17, 26, 63, 66] have been developed. As shown in Figure 1, different parallelism and memory optimizations can be applied in combination.

2.1 Parallelism in Distributed Training

Data Parallelism. To scale training, *Data Parallelism (DP)* [1, 42] distributes input data across GPUs, with each GPU processing its data independently using a model replica. It involves only an all-reduce of gradients per iteration, but requires the entire model to fit within each GPU's memory.

Tensor Parallelism. *Tensor Parallelism (TP)* [57, 73] partitions the parameters in each layer and conducts all-reduce over activations in the forward pass and gradients in the backward pass to maintain computation correctness.

Pipeline Parallelism. *Pipeline Parallelism (PP)* [25, 35, 37, 43, 55, 56] partitions the model into stages, using p2p communication between stages to pass data through the pipeline. Although it only involves small communication overhead to transfer intermediate tensors, the dependency between stages introduces pipeline bubbles, which causes efficiency to suffer as a result of the device idle time.

2.2 GPU Memory Footprint Reduction Techniques

Activation Checkpointing. *Activation checkpointing (CKPT)* (also known as *recomputation*) [3, 17, 36, 38, 88] discards certain activations in the forward pass, while stashing others. Later in the backward pass, the discarded activations are recomputed from the stashed activations, and are then used for gradient computation. This method reduces the memory

Table 1. Comparison of distributed training systems. P, G, O, and A under offloading denote parameter, gradient, optimizer states, activation offloading, respectively. Circle for optimizations represents functionality support and granularity. Circle for tuning represents whether the system can tune all optimizations it supports.

	Parallelism			Offloading				ZeRO-	Auto-Tuning
	DP	TP	PP	P	G	O	A	2/3	Capability
Megatron-LM [73]	✓	✓	✓	○	○	○	○	✗	○
DeepSpeed [68]	✓	✓	✓	○	○	○	○	✓	○
ZeRO-Offload [69]	✓	✓	✗	○	○	◐	○	✓	○
ZeRO-Infinity [67]	✓	✓	✗	◐	◐	◐	◐	✓	○
Alpa [89]	✓	✓	✓	○	○	○	○	✓	◐
Slapo [12]	✓	✓	✓	○	○	○	○	✓	◐
AdaPipe [76]	✓	✓	✓	○	○	○	○	✗	◐
Yuan et al. [85]	✓	✓	✓	○	○	○	◐	✗	◐
Aceso [46]	✓	✓	✓	○	○	○	○	✗	●
Mist	✓	✓	✓	●	●	●	●	✓	●

needed for the saved activations, at the cost of recomputing discarded activations in the backward pass.

Redundancy Optimization. *Zero Redundancy Optimizer* (*ZeRO*) reduces the memory usage by eliminating redundant copies of optimizer states, gradients, and model weights across data-parallel devices [66, 87]. ZeRO operates in three modes: ZeRO-1 (shards optimizer only), ZeRO-2 (shards optimizer and gradients), and ZeRO-3 (shards optimizer, gradients, and weights). While ZeRO-1 introduces no additional communication, ZeRO-2/3 incur communication overhead due to all-gathering and reduce-scattering operations.

Offloading. *Offloading* [26, 31, 63, 67, 70, 81] (also known as *swapping*) involves transferring model states or activations between the GPU devices and the host CPU. This technique helps manage GPU memory constraints by temporarily offloading data, allowing the GPU to accommodate other tasks. The efficiency of swapping significantly depends on overlapping, which allows memory transfers to be hidden outside the critical path.

3 Limitation of Existing Systems

3.1 The Need for Comprehensive Co-Optimization

Distributed training optimization problem involves finding the optimal configuration of parallelism strategies and memory reduction methods to maximize performance, given specific hardware, model, and global batch size. Current distributed training systems, however, lack the ability to comprehensively co-optimize memory footprint optimizations alongside parallelism [46, 85, 89]. As detailed in Table 1, manual methods, such as ZeRO-Infinity [67], support a broader range of memory optimizations but only allow coarse-grained configuration (enabling or disabling offloading) and lack automatic tuning. Automatic methods, such as Alpa [89], AdaPipe [76], and Aceso [12], either support fewer

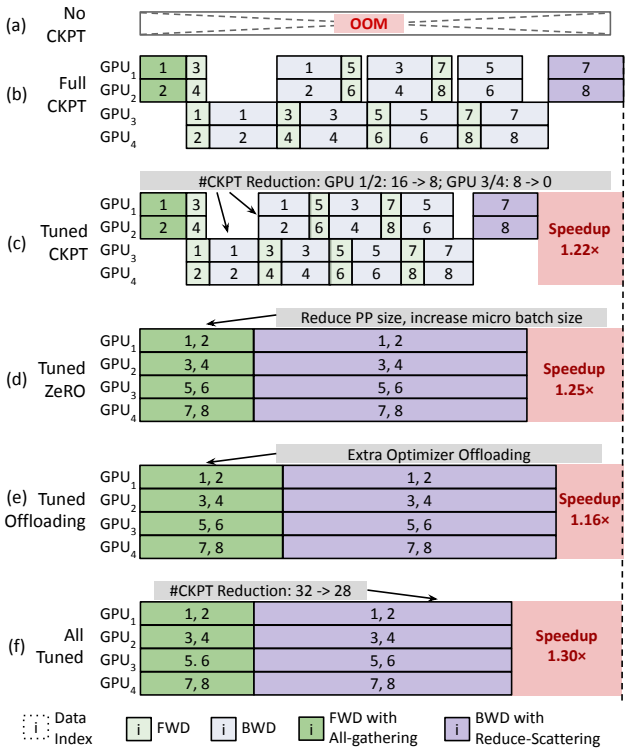
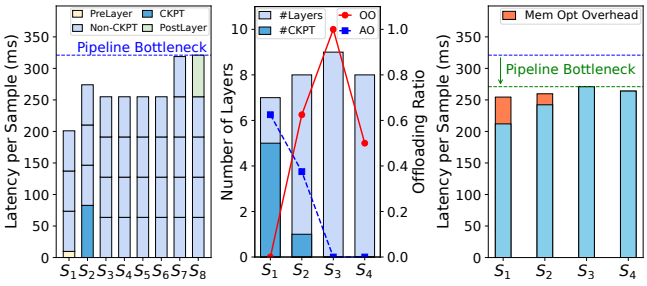


Figure 2. Motivational examples of tuning parallelism with memory optimizations for GPT-3-2.7B on 4 NVIDIA L4 GPUs with $Seq = 4096$, $B_{global} = 8$. Parallelism is always tuned.

optimizations or can only tune a subset of the optimizations they provide. This limited search space leads to sub-optimal configurations and reduced performance.

To illustrate these trade-offs of different optimizations and the impact of co-optimization, we present a motivational example of training GPT-3-2.7B on four NVIDIA L4 GPUs with a global batch size of 8. We manually enumerate all *DP*, *TP*, *PP*, and micro batch size *b* configurations, and when co-tuning memory optimizations, we also enumerate their combinations with the parallelism configurations.

As shown in Figure 2(a), without memory optimization, all parallelism plans result in out-of-memory (OOM) errors. In Figure 2(b), applying full CKPT (all layers being recomputed, as in Megatron-LM [73] and Alpa [89]) reduces memory usage by recomputing activations, avoiding OOM. The best parallelism strategy found is *DP*=2, *PP*=2, *b*=1. In Figure 2(c), if activation checkpointing is tuned (as in Aceso [46] and Adapipe [76]), the number of recomputed layers is reduced from 16 to 8 on the first two GPUs, and from 16 to 0 on the other two, reducing recomputation. During tuning, although another strategy (*DP*=1, *PP*=4, *b*=1) fully eliminates recomputation by using the extra memory from the increased PP size, the added pipeline bubbles outweigh the benefits of reduced recomputation, causing it to under-perform compared to *PP*=2. In Figure 2(d), tuning ZeRO (as in DeepSpeed [68]) enables *DP*=4, *PP*=1, *b*=2 with ZeRO-2, preventing OOM by



(a) Runtime of tuning parallelism with CKPT **(b)** Configurations of comprehensive tuning **(c)** Runtime of comprehensive tuning

Figure 3. Motivational example of showing the speedup source of comprehensive co-optimization for GPT-3-7B on 8 NVIDIA L4 GPUs with $Seq = 2048$, $B_{global} = 512$.

sharding gradients. Similarly, in Figure 2(e), tuning offloading enables the same parallelism with an optimizer offloading ratio of 0.325, avoiding OOM. In both cases, reduced pipeline bubbles and improved kernel efficiency (from the increased batch size) outweigh the memory optimization overhead, increasing training efficiency. These examples show that tuning each memory optimization with parallelism improves training performance, achieving speedups of 1.22×, 1.25× and 1.16× for CKPT, ZeRO, and offloading tuning, respective, compared to the full CKPT strategy.

Building upon these findings, we co-optimize all memory optimizations with parallelism and identify an even better strategy: DP=4, PP=1, b=2 with ZeRO-2 and adjusted activation checkpointing (recomputed layers reduced from 32 to 28), which reduces pipeline bubbles (compared to activation checkpointing tuning only) and recomputation (compared to ZeRO tuning only), leading to a 1.30× speedup while maintaining memory savings.

To further demonstrate the benefits of comprehensive co-optimization, we consider an example of training GPT-3-7B on eight NVIDIA L4 GPUs with a global batch size of 512. When only activation checkpointing is tuned, the best parallelism strategy identified is DP=1, PP=8, b=1, which causes severe pipeline imbalance and hardware idling, as shown in Figure 3(a). However, by comprehensively co-optimizing all techniques, we find a better strategy: DP=2, PP=4, b=2, with adjusted activation checkpointing and optimized offloading ratios, detailed in Figure 3(b), where OO and AO stand for optimizer and activation offloading, respectively. This configuration uses offloading to gain GPU memory, which is then used to reduce PP size from 8 to 4 and eliminate recomputation for the last two stages. As shown in Figure 3(c), co-optimization reduces pipeline stages and device idle time, improving overall performance despite some offloading overhead, as the optimizer offloading overhead is amortized over multiple micro-batches and activation offloading can overlap with computation. Comprehensive co-optimization yields a 1.22× speedup over tuning only parallelism and a 1.11×

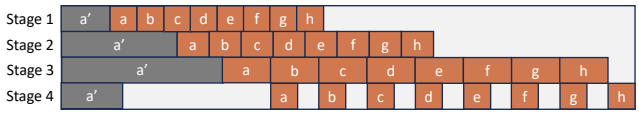


Figure 4. Illustration of pipeline parallelism overlap opportunity and inter-microbatch imbalance. a′ is the extra communication happened in the first microbatch.

speedup over tuning parallelism with activation checkpointing, demonstrating significant performance gains.

However, existing systems lack support for comprehensive co-optimization. For instance, Aceso [46] does not support ZeRO or offloading, Slapo [12] only tunes activation checkpointing within a fixed parallelism plan, and AdaPipe [76] focuses solely on pipeline parallelism and activation checkpointing. It limits their ability to fully leverage the trade-offs between memory reduction and runtime overhead, leading to suboptimal performance.

3.2 Why Existing Auto Systems Fail to Co-Optimize?

We summarize the key shortcomings of existing systems in achieving comprehensive co-optimization:

Shortcoming #1: Lack of overlap awareness. Existing automatic distributed training methods fail to account for computation-communication overlap beyond basic gradient synchronization overlap. This results in significant performance degradation, as seen in our experiments where Aceso underperforms manual implementation Megatron-LM (with overlap) in 6 out of 10 cases despite a larger search space (See Figure 12). Moreover, techniques like ZeRO and offloading add extra communication overheads, requiring overlap with computation or pipeline bubbles to maintain efficiency [66, 67]. In Figure 3(b), Stage 2 shows a 13% overhead if activation offloading is not overlapped, and for Stage 3, offloading optimizer states for a 7B model with a $PP = 4$ takes 7 seconds, resulting in a 40% overhead with a batch size of 64. Additionally, computation-communication interference results in inaccurate performance predictions. For instance, we observe a 7.7% performance degradation for the linear layer in attention module of the motivational example when it is executed concurrently with all-reduce operations, which becomes worse when CPU-GPU communication is also involved. Ignoring overlap leads to mis-estimating optimization configurations and results in sub-optimal strategies.

Shortcoming #2: Unable to navigate the exploded search space. Co-optimizing memory footprint reduction techniques with parallelism significantly expands the search space, making it difficult to efficiently find the best combination. For example, Alpa takes over 40 hours to find the best parallelization strategy for GPT-3-39B on 64 GPUs [89]. As depicted in Figure 5, simultaneously tuning parallelism and memory optimizations further dramatically increases the search space and complexity. Even after applying search space pruning methods, such as inter and intra-stage tuning

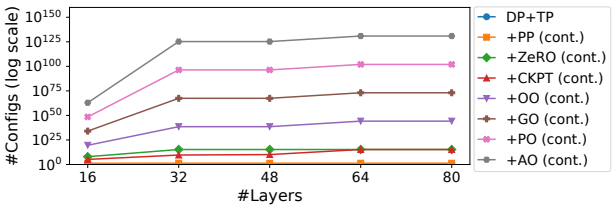


Figure 5. Growth in the number of configurations within the search space as each optimization is incrementally added.

decoupling, the search space remains significantly larger than what existing performance predictors can efficiently handle [21, 34, 48, 71, 84]. For example, Proteus, a fast simulation-based tool that supports the prediction of performance in parallelization and recomputation, requires around 6 seconds to simulate *one* optimization configuration for GPT-2 on 32 GPUs [21]. Despite its speed, this kind of tool is still impractical for effectively exploring the vast search space presented by our problem.

Shortcoming #3: Inaccurate performance prediction due to the lack of inter-microbatch imbalance awareness. Existing automatic distributed training systems often suffer from inaccurate performance predictions when new memory optimizations are involved due to their inability to account for inter-microbatch variability. Automatic parallelism planners [46, 89] typically assume uniform microbatch execution times within a pipeline stage. However, the first and last microbatches take longer due to extra operations like parameter all-gathering, gradient reduce-scattering, and optimizer offloading, as shown in Figure 4. In the motivational example of tuning GPT-3-7B, simply averaging microbatch times leads to performance prediction error of up to 21.55%, depending on the number of microbatches, which may cause up to a twofold performance slowdown. Our evaluation, as shown in Figure 13, ignoring inter-microbatch imbalance leads to about a 9% slowdown compared to optimal solutions. These inaccuracies undermine the effectiveness of tuning process, leading to the selection of sub-optimal strategies.

3.3 Why Simple Heuristics Can Not Address it?

Why manual frameworks with simple heuristics cannot address co-optimization? Manual frameworks struggle to deliver optimal performance in comprehensive co-optimization scenarios, because the best performance can only be achieved with fine-grained configuration for each stage, including layer assignments, DP and TP sizes, recomputed layers, and offloading ratios for each type of model states. The vast and exponentially large search space makes manual exploration impractical, leading users to rely on simple heuristics. For example, a recent study [85] uses a heuristic that applies uniform checkpointed layers and activation offloading ratios across all pipeline stages to reduce tuning search space. However, pipeline parallelism inherently exhibits memory and computation imbalances, making uniform strategies sub-optimal. As shown in Figure 3(b), heterogeneous optimizations per stage are selected. In our motivational examples, uniform heuristics results in 26% and 20% performance degradation for the 2.7B and 7B models, respectively, compared to the optimal strategies achieved through comprehensive co-optimization. As workloads scale, the complexity of tuning grows, making fully automated systems increasingly essential to efficiently handle the expanded search space and deliver optimal performance [46, 76, 89].

Our Goal *is to develop a fully automated distributed training optimization system for LLMs that addresses all the shortcomings above and comprehensively co-optimize memory footprint reduction techniques alongside parallelism.*

4 Mist: Overview and Key Ideas

To accelerate distributed training, we introduce Mist, a memory, overlap, and imbalance aware automatic distributed training system that comprehensively co-optimizes memory footprint reduction techniques with parallelism. Overall, Mist proposes three key ideas:

1. Fine-Grained Overlap-Centric Scheduling and Interference Modeling. Mist proposes an overlap-centric scheduling approach that carefully orchestrates parallelism and memory optimizations to maximize the overlap of computation and communication. By optimizing the order and granularity of these techniques, Mist mitigates memory optimization overhead while maintaining manageable tuning complexity. Additionally, data-driven interference modeling accurately predicts performance when computation and communication kernels run concurrently. This approach addresses Shortcoming #1, the lack of overlap awareness.

2. Symbolic-Based Efficient Performance Prediction. Building upon the scheduling strategy, Mist introduces a symbolic analysis system to significantly enhance the performance prediction efficiency. Unlike traditional methods that require repeated simulations for each optimization configuration [21, 46, 89], Mist symbolizes the model and optimizations, requiring only a single simulation pass to predict runtime and memory usage in the form of symbolic expressions. Subsequent predictions are simplified to value substitutions in these expressions, dramatically reducing redundant simulation and allowing for rapid batched evaluation of multiple configurations. This approach effectively mitigates the Shortcoming #2 search space explosion.

3. Imbalance-Aware Hierarchical Tuning via Pareto Frontier Sampling. Finally, Mist proposes an imbalance-aware hierarchical auto-tuner that decouples tuning into intra-pipeline-stage and inter-pipeline-stage processes, while still considering the microbatch imbalances and overlap opportunities in PP. To find the best intra-stage configurations, our intra-stage tuning uses a dual-objective approach to balance time between imbalanced and stable microbatches.

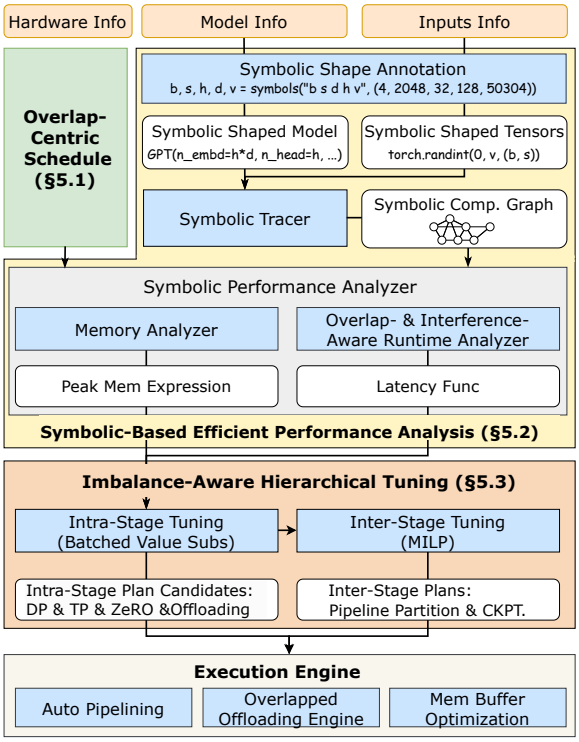


Figure 6. High-level System Overview of Mist.

Inter-stage tuning formulates an MILP problem to determine the best pipeline partitioning using data points sampled from the Pareto frontier of intra-stage tuning. This approach preserves a pruned search space from hierarchical tuning (helping to solve Shortcoming #2) while addressing the lack of inter-microbatch imbalance awareness (Shortcoming #3).

System Overview. Figure 6 presents the high-level overview of Mist. The model and its input data are annotated with symbolic shapes and traced to generate a symbolic shaped computational graph. This graph, on top of the Overlap-Centric Scheduling, is analyzed by our symbolic performance analyzer to derive the peak memory expression and runtime function whose inputs are optimization-related symbols. During intra-stage tuning, Mist evaluates these symbolic expressions with specific optimization values in batches. This evaluation identifies a Pareto-optimal set of parallelism and memory optimization plans for each potential inter-stage candidate, utilizing our Symbolic-Based Efficient Performance Prediction strategy to efficiently navigate the extensive search space. Subsequently, the inter-stage tuning phase formulates a MILP problem using stable microbatch times (T_{stable}) and their deviations (T_{delta}) sampled from intra-stage tuning results. This determines the optimal pipeline partitioning and combination of (T_{stable}, T_{delta}), addressing both inter-microbatch imbalances and inter-stage imbalances in pipeline parallelism. Once the optimal tuning plans are identified, Mist employs an orchestrated execution engine to execute them, including automatic pipeline transformation, overlapped offloading communication, and memory buffer optimizations.

Table 2. Optimization variables in the schedule template.

Name	Value Type	Meaning
G	Integer	Gradient accumulation steps
S	Integer	Number of pipeline stages
L_i	Integer	Number of layers in stage i
b_i	Integer	Micro batch size for stage i
DP_i	Integer	DP size for stage i
TP_i	Integer	TP size for stage i
$ZeRO_i$	One-Hot [0-3]	ZeRO level for stage i
$CKPT_i$	Integer	Number of recomputed layers for stage i
WO_i	Float [0, 1]	Weight offloading ratio for stage i
GO_i	Float [0, 1]	Gradient offloading ratio for stage i
OO_i	Float [0, 1]	Opt states offloading ratio for stage i
AO_i	Float [0, 1]	Activation offloading ratio for stage i

5 Mist: Design Details

5.1 Fine-Grained Overlap-Centric Scheduling

Mist coordinates memory optimizations and parallelism with two primary objectives: balancing optimization effectiveness with manageable tuning complexity and maximizing overlap opportunities to reduce runtime overhead.

Optimizations and Granularity. Mist comprehensively supports various parallelism techniques such as DP, TP, and PP, alongside memory optimizations like fine-grained offloading, flexible activation checkpointing, and different levels of ZeRO optimization. Based on the observation that all transformers are identical and share computational properties within a pipeline stage, Mist adopts stage-wise tuning granularity, meaning all layers within the same pipeline stage use the same parallelism and memory optimization configurations to balance optimization effectiveness and search space. Specifically, for gradient accumulation steps G and the number of pipeline stages S, the combination of (L_i, b_i, DP_i, TP_i, $ZeRO_i$, $CKPT_i$, WO_i, GO_i, OO_i, AO_i) defines the configuration for pipeline stage i (see Table 2 for detailed explanations of the symbols). Notably, swapping strategies are represented as floating-point ratios, enabling fine-grained control of memory offloading and enhancing computation-communication overlap potential.

Overlapped Schedule. Building upon these optimization strategies, the overlapped schedule optimizes the execution order of memory optimizations and parallelism techniques to maximize hardware utilization while maintaining a low GPU memory footprint. As depicted in Figure 7, computation, GPU-GPU communication, and CPU-GPU communication are overlapped. As shown in ❷, during the forward pass, the computation of layer k overlaps with the activation swapping out of layer $k-1$, and the swapping in and all-gathering of parameters for layer $k+1$. Similarly, as shown in ❸, in the backward pass, the computation of layer k overlaps with

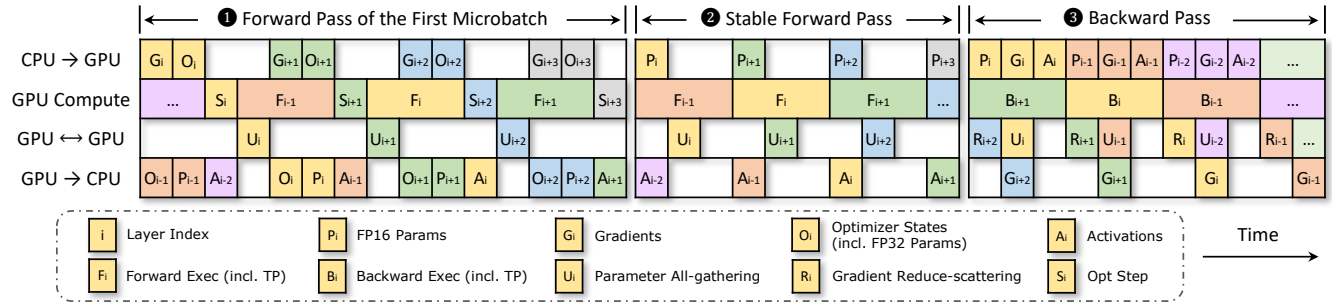


Figure 7. Overlap Schedule Template of Mist

the gradient reduction and the swapping-out of the previous backward layer $k+1$, along with the swapping in of parameters, gradients and activations, and all-gathering of parameters for the next layer $k-1$. This overlap ensures that the computation in layer k is not stalled by data movement or pre-fetching, leading to better hardware utilization. Additionally, Mist supports inter-stage overlap by hiding communications that are independent of previous stages within pipeline bubbles, as shown in Figure 10.

Optimizer Step Decoupling and Repositioning. Furthermore, in scenarios involving ZeRO optimization and offloading, a monolithic optimizer step can lead to increased peak memory usage and redundant communication. Specifically, to perform an optimizer step in the mixed-precision optimizer, the following tensors must be in the GPU device at the same time: FP16 parameters, FP16 gradients, FP32 optimizer states, and FP32 master parameters [66]. When offloading is applied, peak memory during the optimizer step may exceed that of the backward pass, as only partial layer states reside in GPU memory during the forward and backward computations, while a monolithic optimizer step requires all of them for the whole model. Additionally, optimizer steps require rematerializing all states through offloading or all-gathering, which also occur during the forward and backward passes, introducing redundant communication. To address these issues, Mist decouples the optimizer step into multiple steps, repositioning each layer's optimizer step immediately before its first forward pass. Moreover, to eliminate synchronization between pipeline stages for `nan` and `inf` checks, the validate-and-update method from zero-bubble pipeline parallelism can be employed, delaying synchronization and reverting optimizer states if necessary [64].

5.2 Symbolic-Based Efficient Performance Analysis

Performance modeling is crucial for optimizing distributed training as it enables efficient configuration exploration. Existing systems rely on simulation-based performance prediction, running a concrete simulation for each configuration to estimate computation, communication, and memory usage. As shown in Figure 8, a traditional simulator initializes a GPT model with a concrete parallelism configuration of e.g., (DP=2, TP=8, PP=2) on 32 GPUs, applies memory optimizations, and simulates execution to measure performance and peak memory usage. Although each simulation is efficient, it still takes about 6 seconds per configuration [21]. This cost makes exhaustive search impractical for our combined search space. We now demonstrate how Mist's symbolic-based performance analysis helps to accelerate the performance prediction and thus facilitates the traversal over a huge search space.

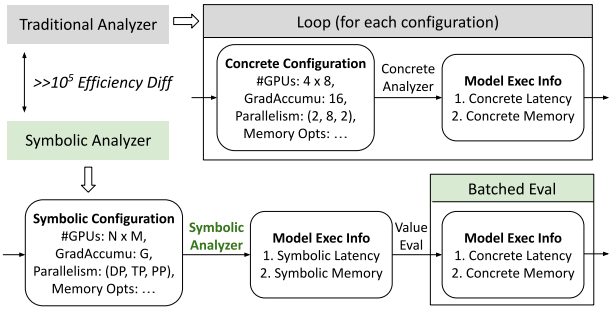


Figure 8. Comparison of the symbolic performance analyzer with the traditional analyzer.

5.2.1 Symbolic Analysis System

Mist overcomes these limitations by employing symbolic-based performance modeling. Symbolic analysis refers to techniques used to analyze systems by reasoning about symbolic representations of data or computations rather than concrete values, used in compiler optimizations [10] and circuit designs [27]. Mist proposes a symbolic analysis system for LLMs, supporting symbolic execution, tracing, and analysis. As shown in Figure 9, users define the model and inputs, simply replacing the concrete dimensions and optimizations with symbols. Then all operations are executed with the information of symbolic shapes. On top of it, Mist traces the computational graph and performs static analysis to derive symbolic expressions for execution time and memory usage. Instead of repeatedly simulating different configurations, Mist only performs a single symbolic simulation pass and later substitutes values into these expressions to quickly evaluate different configurations. This approach enables batched evaluation and compilation optimization, making performance prediction over $10^5\times$ faster than traditional analyzers.

```
from mist import global_symbol_manager as gsm
# Define symbols
b, s, h, d, tp = gsm.symbols("b s h d tp", (4, 128, 12, 64, 8),
      integer=True, positive=True)
# Initialize the model configuration using symbolic parameters
config = GPT2Config(n_embd=h*d, n_head=h, tp=tp, ...)
# Construct the GPT-2 model with symbolic configuration
model = GPT2LMHeadModel(config)
# Create symbolic input tensors
input_ids = torch.randint(0, V, (b, s), dtype=torch.long)
# Execute the model with symbolic inputs
logits = model(input_ids).logits
```

```
>>> logits
symbolic_tensor((b, s, V), concrete_shape=(4, 128, 50257), ...)
```

Figure 9. Example of defining symbolic model configurations and inputs, followed by symbolic execution.

For memory analysis, Mist uses liveness analysis on the symbolic computational graph. It tracks live tensors during execution and determines peak memory usage by identifying the maximum memory allocation at any point. To support pipeline parallelism, Mist performs intra-layer and inter-layer analysis: the intra-layer pass extracts memory statistics (e.g., layer states, saved activations, and intermediate tensors), while the inter-layer pass combines this data to generate stage-wise symbolic memory expressions, enabling efficient estimation across configurations.

For runtime analysis, direct symbolic representation is impractical due to the complex behavior of various GPU kernels. Instead, Mist profiles operator execution dynamically. Computation is estimated using a operator computation database, which benchmarks new operators or unseen input shapes on the current hardware and stores results for future use. Communication is modeled symbolically by dividing communicated bytes by the bandwidth, and overlap is managed via interference modeling, which we introduce below.

The design of our symbolic analysis system is far from trivial and addresses several significant challenges, making it both powerful and widely applicable. First, large models must be run across multiple GPUs due to memory capacity issues, but direct analysis on multi-GPU setups is inefficient. We solve this by using the idea of fake tensors and meta devices, where tensor shapes are represented symbolically but not materialized physically, allowing analysis without needing actual hardware. Second, backward pass memory analysis is difficult due to the absence of an explicit computational graph. We generate a fake backward graph using gradient function properties to track memory during backpropagation. Third, supporting custom kernels like FlashAttention [19] and communication operations required custom symbolic representations, ensuring flexibility. Beyond optimization, our symbolic analysis system offers clear insights into workloads, making it easier to understand how specific parameters and optimizations affect performance, which can be valuable for both practical use and educational purposes.

Algorithm 1: Batched Interference Estimation

Data: $C, G2G, C2G, G2C, params$
Result: Total latency vector T

Function PredINTF($C, G2G, C2G, G2C, params$):
  $X \leftarrow [C, G2G, C2G, G2C]^T$ // Stack features
  $T \leftarrow ZerosLike(C)$ // Initialize output
  for $n = 4$ ***downto*** 2 // The num of concurrent ops
  do
    for $i = 0$ ***to*** $\binom{4}{n} - 1$ // Enumerate all combinations
    do
      // Index pre-defined masks and factors
      $mask \leftarrow IndexPredefinedMask(n, i)$
      $factors \leftarrow IndexFactors(params, n, i)$
      Update($X, T, mask, factors$)
    end
  end
  T += sum(X, axis $= -1$) // Sum remaining time
  return T
Function Update($X, T, mask, factors$):
  $ids \leftarrow \{j \mid (X_j \neq 0)$ matches $mask\}$
  if $ids = \emptyset$ **then return**
  $scaled \leftarrow X[ids] \times factors$
  $overlap \leftarrow \min(scaled, \text{axis} = -1)$
  $X[ids] \leftarrow (scaled - overlap)/factors$
  $T[ids]$ += $overlap$
return

5.2.2 Interference Model To be overlap aware, we integrate an interference model within the symbolic analysis system. Runtime prediction is much more challenging when overlap is involved, such as computation, NCCL (GPU ↔ GPU communication), D2H (GPU→CPU communication), and H2D (CPU→GPU communication) running simultaneously. Mist provides an interference model that predicts the impact of up to four different types of kernels running simultaneously. Instead of using machine learning models like XGBoost [15], which may overfit in this case, we develop a mathematical model with fewer parameters and clearer intuition. In this model, each possible combination of co-running kernels is assigned a set of slowdown factors that quantify the effect of the execution for each participant.

Algorithm 1 implements batched interference estimation, which iteratively applies slowdown factors to update execution times. For each concurrency level ($n = 4$ to 2 operations), it iterates through all $\binom{4}{n}$ combinations, retrieves predefined masks and factors, and invokes `Update`. The `Update` function scales execution time by their respective slowdown factors, computes the scaled overlapping, and updates remaining execution times accordingly. By progressively resolving interference through successive reductions, the algorithm eliminates concurrent operations until only a single component remains. A data-driven approach is used to fit the model, where different shapes and combinations of concurrent kernels are sampled and benchmarked, and the resulting runtime data is used to train the slowdown factors.

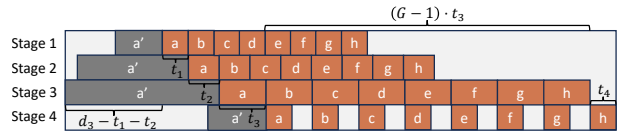


Figure 10. Illustration of runtime of a pipeline considering inter-microbatch imbalance. a' is the extra communication overhead only involved in the first and last microbatches.

5.3 Imbalance-Aware Hierarchical Tuning via Pareto Frontier Sampling

Our tuning problem is defined as, given a model, a global batch size B, and a device mesh (N, M), Mist's auto-tuner outputs the best training plan including the gradient accumulation steps G, layer partitions PP for different pipeline stages, and combination of $[b, DP, TP, ZeRO, CKPT, WO, GO, OO, AO]$ for each pipeline stage, as detailed in Section 5.1.

To efficiently find the best strategy in a huge search space, Mist adopts the idea of hierarchical tuning, decoupling the whole tuning process into intra-stage tuning and inter-stage tuning [89]. Intra-stage tuning aims at finding the best optimization plans for all possible pipeline partitioning candidates, while inter-stage tuning is used to find the best stage partitioning and device assignment. Compared to existing automatic parallelization methods, Mist offers two key improvements: imbalance and overlap awareness.

Inter-Stage Tuning. Inter-stage tuning finds the best layer partition and device assignment, as well as the best number of layers being recomputed. Unlike previous methods that treat all microbatches as the same [46, 89], Mist discovers that using either averaged runtime across different microbatches or simply applying the stable microbatch to tune leads to sub-optimal results. The former approximation might lead to bottleneck drifting, and the latter one fails to consider the extra overhead of optimizations specifically for the first or last micro batch. As Figure 10 shows, Mist considers the inter-microbatch imbalance and proposes a new objective as

$$\min_{\substack{1\le i\le S\\ l_i,ckpt_i,\\ (n_i,m_i)}} \left\{(G-1)\cdot \max_{1\le i\le S}\{t_i\} + \sum_{i=1}^{S} t_i + \max_{1\le i\le S}\left(d_i - \sum_{1\le j<i} t_i\right)\right\} \tag{1}$$

where S means the number of stages, l_i denotes the number of layers, $ckpt_i$ denotes the number of checkpointed layers, and (n_i, m_i) denotes the device assignment, for stage i. For simplicity, we define any microbatch that is neither first nor last as a stable microbatch. t_i means the *stable* microbatch runtime of stage i, and d_i means the runtime *delta* of the first and last microbatches compared to t_i. The first term ensures that Mist correctly identifies the pipeline bottleneck, while the second and third terms account for inter-stage and inter-microbatch imbalances, respectively. The third term also considers the overlap opportunities of hiding communication independent of previous stages in the pipeline bubbles.

Objective 1 can be solved given (t_i, d_i) according to l_i, c_i, and (n_i, m_i). However, t_i and d_i are correlated within a stage. For instance, if optimizer offloading is applied aggressively, the runtime of the first microbatch significantly increases and the runtime of the stable microbatches reduces because of the less intensive memory pressure. This suggests that (t_i, d_i) form pairs along a Pareto frontier within the stage. Thus, we transform the decision variables and obtain:

$$\min_{\substack{1\le i\le S\\ l_i,f_i,(n_i,m_i)}} \left\{(G-1)\cdot \max_{1\le i\le S}\{t_i\} + \sum_{i=1}^{S} t_i + \max_{1\le i\le S}\left(d_i - \sum_{1\le j<i} t_i\right)\right\} \tag{2}$$

$$(t_i, d_i) = IntraStagePareto(i, l_i, (n_i, m_i))\,[f_i] \tag{3}$$

where f_i is the sampled index from the intra-stage Pareto frontier introduced in the next section. We directly combine the checkpointing tuning into the Pareto frontier as it also serves as a trade-off between t_i and d_i. Objective (2) can be reformulated into an MILP problem and solved by the off-the-shelf solver [28].

Intra-Stage Tuning. As Objective (4) shows, given the stage partitioning, device assignment, gradient accumulation steps, and memory budget, intra-stage tuning finds the best data and tensor parallelism, and memory optimization combinations to maximize the throughput and sample the Pareto frontier.

$$\begin{aligned} &\min_{p,z,o}\ \alpha\cdot G\cdot t_{p,z,o} + (1-\alpha)\cdot d_{p,z,o} \\ &i.e.\ \max\left(Mem_{peak}^{(fwd)}, Mem_{peak}^{(bwd)}\right) \le Mem_{Budget} \end{aligned} \tag{4}$$

where a series of $\alpha \in [0, 1]$ are sampled uniformly to construct a Pareto frontier efficiently. And the stable microbatch time t and delta time d of a certain gradient accumulation step G and strategy tuple (parallelism p, ZeRO config z, and offloading configs o) can be obtained from the interference model. The parallelism strategy p includes b, DP, TP. The offloading configuration o consists of OO, GO, WO, and AO.

$$t_{p,z,o} = \mathcal{I}\left(c_{p,z,o}^{stable}, nccl_{p,z,o}^{stable}, d2h_{p,z,o}^{stable}, h2d_{p,z,o}^{stable}\right) \tag{5}$$

$$d_{p,z,o} = \mathcal{I}\left(c_{p,z,o}^{first}, nccl_{p,z,o}^{first}, d2h_{p,z,o}^{first}, h2d_{p,z,o}^{first}\right) - t_{p,z,0} \tag{6}$$

where $\mathcal{I}$ is the interference model proposed before, c means the GPU computation time, $nccl$ means the GPU-GPU communication time, $d2h$ means the device to host copy time, and $h2d$ means the host to device copy time. The superscript $stable$ indicates the time of a stable micro batch, while $first$ indicates the time of the first micro batch.

All the statistics of runtime and memories are reported by our symbolic analyzer. With the help of our symbolic-based performance analyzer, querying single datapoints is extremely fast. Thus, to get the best strategy, we simply search in a brute-force way, which would not miss any optimization possibilities, ensuring the optimal solution.

Table 3. Hardware Specifications.

Platform	GPU	GPU#	Mem.	PCIe Spec	NVLink	Interconnect
GCP	L4	[2, 4, 8, 16, 32]	24GB	Gen3@16x	✗	100Gbps
AWS	A100	[2, 4, 8, 16, 32]	40GB	Gen4@16x	✓	400Gbps

Table 4. Workload Specifications.

GPU	Models	Param# (billion)	Global Batch Size	Seq Len
L4	GPT, Llama, Falcon	[1.3, 2.6, 6.7, 13, 22]	[32, 64, 128, 256, 512]	2048
A100	GPT, Llama, Falcon	[1.3, 2.6, 6.7, 13, 22]	[32, 64, 128, 256, 512]	4096

6 Evaluation

We prototype Mist with ~27K LoC in Python. To support all optimizations, we have implemented it from scratch based on PyTorch [61], supporting symbolic torch tracing and execution, model automatic pipelining, overlapped offloading and ZeRO execution, and memory buffer optimizations.

We evaluate Mist on various training configurations with different hardware, models, and hyper-parameters to demonstrate its ability to effectively find the optimal combination of memory optimizations and parallelism. Our results show that Mist constantly outperforms state-of-the-art distributed training systems. We use training throughput (samples per second) as our primary metric. Since all optimizations applied by Mist are lossless, the fidelity of computation is preserved, ensuring the model convergence is not affected. Additionally, we provide speedup breakdown, sensitivity studies, prediction accuracies, tuning time, and case studies to explore the sources of our speedup and provide insights.

6.1 Methodology

Hardware Settings. To fully study the capabilities of Mist, we evaluate its training performance on both PCIe and NVLink systems, as they offer different combinations of hardware resources. We conduct our major experiments on up to 32 NVIDIA L4 GPUs [59] and 32 NVIDIA A100 GPUs [58]. Detailed hardware specifications are shown in Table 3.

Workloads Setting. We selected three representative types of LLMs, GPT-3[9], LLaMa [23, 78, 79], and Falcon [2]. All are transformer-based models with some variation in their components. GPT-3 consists of typical transformer decoder layers[9]. LLaMa integrates techniques like pre-RMSNorm [86], gated functions [72], rotary embedding [75], among others, to improve performance on tasks involving long-range dependencies [23, 78, 79]. Falcon adopts parallel attention and MLP layers inspired by GPT-J [14] and GPT-NeoX [8], reducing the number of all-reduce operations associated with tensor parallelism from two to one per layer [2]. Following common practice, we scale the number of GPUs and the global batch size with the size of the model. To minimize the impact of different frameworks and kernel implementations, we set the dropout ratio to zero and disable all biases in the linear layers. Table 4 shows the workloads specifications.

Baselines. We compare Mist with three state-of-the-art deep learning distributed training systems: (1) Megatron-LM [73] (`core_r0.4.0`), (2) DeepSpeed [68] (`v0.12.6`), and (3) Aceso [46]. **Megatron-LM** and **DeepSpeed** are state-of-the-art manual implementations. Since they do not support automatic tuning, to achieve the best performance, we perform a grid search over all possible optimization combinations for single-node distributed training. For multi-node cases, we benchmark the best strategies that Mist finds within the same search space as Megatron-LM and DeepSpeed. **Aceso** is the state-of-the-art automatic distributed strategy tuner with search space larger than others [12, 76], which can automatically find the best combinations of parallelism and activation checkpointing plans. We follow its artifact to get its numbers.

In the common practice of training LLMs, FlashAttention [18, 19], a vital kernel for performing the fast and memory-efficient attention mechanism, is applied by default to reduce memory usage and achieve the best performance. When FlashAttention is enabled, we only compare with Megatron-LM and DeepSpeed since the Aceso does not support it. We compare all three baselines on L4 GPUs. For A100 GPUs, we only compare Mist with state-of-the-art manual and automatic methods, Megatron-LM and Aceso as DeepSpeed generally underperforms Megatron-LM in our experiments.

We attempted to compare with Alpa [89], but it fails to find any feasible solutions on L4 GPUs for our workloads. Our conjecture is that Alpa only considers memory usage in the Inter-Op pass by compiling the searched strategy and running it, while its memory-unaware Intra-Op pass likely causes OOM errors for all proposed strategies.

6.2 End-to-End Training Performance

Speedup in Real-World Scenarios. Figure 11 compares the end-to-end throughput of various distributed training frameworks in real-world scenarios where FlashAttention [18, 19] is enabled. We make two key observations. First, Megatron-LM outperforms DeepSpeed in most cases. This is mainly due to the fact that the parallelization plans that work in Megatron-LM cause out-of-memory issues in DeepSpeed, forcing DeepSpeed to choose sub-optimal parallelization strategies. Second, Mist consistently outperforms other distributed training frameworks, achieving an average speedup of 1.32× (up to 1.59×) over Megatron-LM on L4 GPUs, 1.51× on average (up to 1.67×) over DeepSpeed on L4 GPUs, and 1.34× on average (up to 1.72×) over Megatron-LM on A100 GPUs. Specifically for the GPT-3 model, which is the most heavily optimized model in other frameworks, Mist achieves 1.22× speedup on average (up to 1.32×) on L4 GPUs, and 1.20× speedup on average (up to 1.32×) on A100 GPUs, compared with Megatron-LM. The higher speedup for LLaMa model mainly comes from the better RMSNorm kernel implementation and efficient rotary embedding implementation [19]. Overall, we conclude that Mist achieves the best

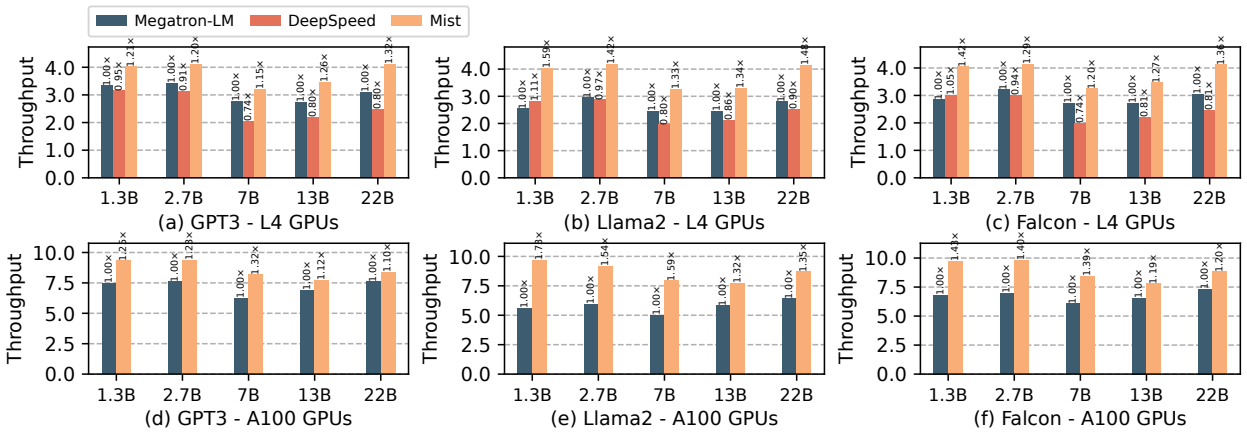


Figure 11. End-to-end training throughput (samples/sec) on L4 GPUs and A100 GPUs, with FlashAttention enabled. Sequence lengths are 2048 for L4 GPUs and 4096 for A100 GPUs. The numbers of GPUs are 2, 4, 8, 16, 32, respectively.

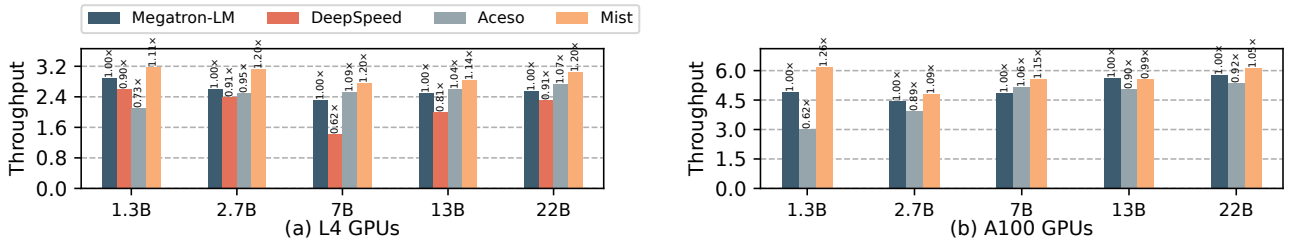


Figure 12. End-to-end training throughput (samples/sec) on L4 GPUs and A100 GPUs, without FlashAttention. Sequence lengths are 2048 for L4 GPUs and 4096 for A100 GPUs. The numbers of GPUs are 2, 4, 8, 16, 32, respectively.

performance over prior state-of-the-art manual implementations across various models and hardware.

Speedup compared with more baselines. Figure 12 compares the throughput of the GPT-3 model with both manual and automatic parallelization frameworks without FlashAttention. Mist still consistently outperforms or is equal to all prior distributed training frameworks. Mist achieves an average of 1.14× speedup (up to 1.26× speedup) compared to Megatron-LM and an average of 1.27× speedup (up to 2.04× speedup) compared to Aceso. When training GPT-3 13B on 16 A100 GPUs, Mist does not achieve better performance but still gets almost the same results, because the naive strategy happens to achieve the best trade-off among all resources. We also find that Aceso does not consistently outperform Megatron-LM even though it has larger search space due to fine-grained activation checkpointing tuning. The root cause is that Aceso does not include sharded data parallelism in the search space and miss several essential opportunities for communication-computation overlapping.

Discussion on the hardware. As shown in Figures 11 and 12, Mist exhibits higher speedup on L4 GPUs than that on A100 GPUs, with following reasons. Large-scale distributed training tasks on L4 GPUs are often limited by smaller memory capacity and the restricted intra-node and inter-node bandwidth. In this scenario, Mist plays a crucial role in striking the best trade-off among various resources to enhance the resource utilization. On the other hand, training tasks on A100 GPUs benefit from larger memory capacity and faster intra-node NVLink and inter-node InfiniBand connections, resulting in much higher resource utilization that approaches the physical limits. This leaves less room for improvement.

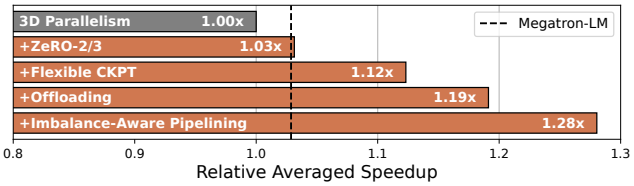


Figure 13. Relative averaged speedup of tuning over different search spaces for GPT model on 8, 16, and 32 L4 GPUs.

6.3 Speedup Breakdown

To understand how each key ideas of Mist contributes to the final performance, we evaluate it by incrementally enlarging the search space proposed by Mist in Figure 13. We normalized the throughput by the baseline search space of Megatron-LM. Three key conclusions are drawn: First, Mist's advantage is not from the better implementation; with the same search space as Megatron-LM, Mist is slightly slower due to implementation overhead supporting other optimizations. Second, activation checkpointing tuning provides a 1.12× speedup on average, with offloading adding an extra 7% speedup, showing their abilities of striking better trade-offs among

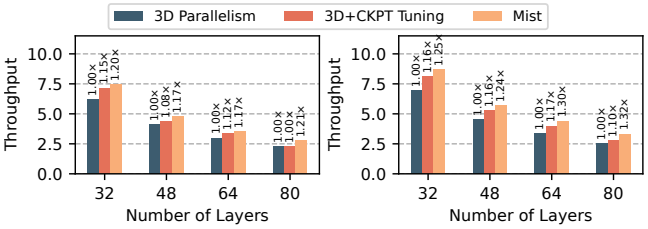


Figure 14. Performance of GPT-3 with different number of layers on 32 L4 GPUs. Left: without FlashAttention; Right: with FlashAttention.

resources. Third, inter-microbatch imbalance-awareness offers an extra 9% speedup upon all prior speedups, as it provides accurate runtime predictions for pipeline parallelism. In summary, all optimizations included in Mist are crucial for improve the system performance.

One key insight we observe is that much of the speedup comes from reducing activation checkpointing and eliminating pipeline bubbles. However, naively disabling checkpointing often leads to OOM errors, and tuning it (as done in Aceso) only partially solves the issue. Further improvements from increasing ZeRO levels or increasing offloading are essential, as they help to further reduce recomputation. As long as these overheads can be overlapped or amortized across multiple microbatches, performance significantly improves.

Additionally, speedups may vary depending on hardware resources and workload intensity. On A100 GPUs with moderate workloads, most speedups come from activation checkpointing tuning. However, when memory pressure is high, combining it with offloading becomes important. For instance, training a 40B GPT-3 model on 32 A100 GPUs, Mist is expected to get 1.10× speedup compared to 1.04× with only activation checkpointing tuning.

6.4 Sensitivity Study

To comprehensively understand the robustness of Mist, we evaluate its performance with different model scales and different global batch sizes.

Robustness over Different Model Scales. As depicted in Figure 14, Mist consistently outperforms the baseline search spaces by up to 1.32× higher throughput, particularly at 80 layers, regardless of whether FlashAttention is enabled. Activation checkpointing tuning is particularly effective for smaller model sizes. However, as model size increases, the speedup from checkpointing alone decreases. With the entire search space enabled, Mist maintains substantial speedups across different model sizes.

Robustness over Different Global Batch Sizes. As shown in Figure 15, Mist always achieves the best performance compared to the baseline search space across different global batch sizes. Notably, Imbalance-Aware Inter-Stage Tuning provides an extra 1.13× speedup on average. One concern is that with larger global batch sizes and potentially more

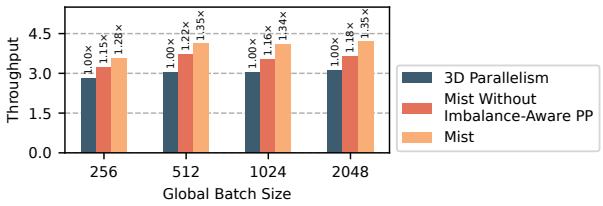


Figure 15. Performance of GPT-3 22B with different global batch sizes on 32 L4 GPUs.

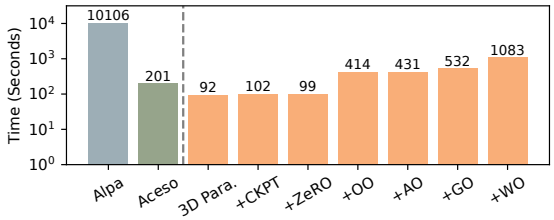


Figure 16. Tuning Time of the GPT-3 22B model on 32 GPUs. Orange bars are Mist with different optimizations applied.

microbatches, the benefit of imbalance-aware inter-stage tuning might diminish, as treating all microbatches as the same might seem sufficient. However, the sub-optimal strategy produced by inaccurate predictions lead to a significant performance gap due to the larger gradient accumulation steps while Mist's inter-microbatch awareness avoids it.

6.5 Tuning Time Comparison

As Figure 16 shows, to understand the tuning efficiency of Mist, we evaluate the tuning time by enabling optimizations one by one and compare it with the tuning time of Alpa [89] and Aceso [46]. For Alpa, we choose 6 data points because it doesn't automatically tune the gradient accumulation steps and layer grouping size. We make three key observations: First, Mist helps to reduce the tuning time a lot compared to Alpa. Second, when Mist is configured to use a similar search space as Aceso, which is branded as an efficient distributed training searching system, Mist can be faster. Third, even when Mist enables more optimization and greatly increases the search space, the tuning time remains reasonable compared to the significantly longer training time. Moreover, since searching over different gradient accumulation steps is independent, Mist's tuning can be parallelized across different CPU cores or machines to make it faster.

6.6 Accuracy of Symbolic Shape Analysis System

The effectiveness of Mist's tuning system relies heavily on the performance prediction accuracy. Therefore, we sample different strategies and benchmark the accuracy of the predicted runtime and memory usage compared with the actual ones. Mist consistently demonstrates a high prediction accuracy for both runtime and memory usage. The averaged runtime error ratio is 1.79%, and average memory footprint error ratio is 2.10%. For runtime, our analysis system focuses more on the magnitude comparison to determine the best

strategy. As a result, some minor times, such as the optimizer step time, are not included. To better understand runtime accuracy, we shift the predicted runtime so that the mean values of predicted and actual runtime match.

7 Other Related Work

DNN Performance Modeling. The closest works to ours in performance modeling are DistSim, Proteus, and dPRO [21, 34, 48], which use simulation-based approaches to predict model performance. Mist differs in that we consider an expanded search space including all memory optimizations in addition to parallelism strategies. This requires additional sophisticated modeling with respect to overlap and interference. Moreover, Mist employs a symbolic modeling first approach that is well-suited for the efficiency that the expanded search space demands. DistIR [71] is a work that also employs an analytical-based approach, but employs a cost-model predictor of operator latencies. Habitat [84] profiles operators in tandem with methods to extrapolate performance to other GPUs for non-distributed scenarios.

Symbolic Analysis. Symbolic analysis examines program behavior through abstract representations of variables and computations rather than concrete values, enabling systematic exploration of optimization spaces. Foundational applications in compilers leveraged symbolic techniques for dependency analysis and loop transformations via constraint solving [10, 24], while analog circuit design adopted symbolic methods for parameter space exploration [27, 29]. Unlike prior works targeting code-level optimizations or hardware verification, Mist adapts symbolic analysis in the area of performance estimation for deep learning models to efficiently explore the joint space of parallelism strategies and memory reduction techniques in distributed training.

Acceleration Techniques. Techniques like tensor compilation and kernel optimizations (e.g., TVM, Hidet, FlashAttention) are largely different than Mist, since they mainly focus on lower-level, static graph optimizations from the graph to hardware level [4, 16, 18–20, 30, 33, 44]. Additionally, approaches such as gradient compression, quantization, and sparsity, and automatic mixed precision [7, 11, 13, 54, 83] are also orthogonal to Mist: they could be integrated into Mist's schedule template as additional optimization techniques that can improve system performance and memory efficiency. Some of these may also be potentially lossy optimizations, i.e., downgrading the accuracy of models, whereas Mist exclusively targets system-level improvements without compromising training accuracy.

8 Discussion

Integration of Other Techniques. Mist is extensible to additional operators, parallelism strategies, and optimizations. New acceleration or memory optimization techniques like quantization [54, 62] or compression [13, 47] are typically implemented using native torch operators or customized kernels. Native torch operators (including communication operators) can be straightforwardly supported by Mist, and customized CUDA kernels can be easily incorporated by registering them in the symbolic analysis system, as demonstrated with FlashAttention [19]. Therefore, these optimizations are directly reflected in the traced computational graph produced by Mist and analyzed by Mist, thus they will be configured intelligently to achieve high performance.

Future Work. Mist relies on a symbolic computational graph, making it less suited for highly dynamic workloads where a fixed computation graph is difficult to obtain. However, for workloads like Mixture of Experts (MoE) with expert parallelism [65], where computation patterns are largely predictable, data-dependent routing can be handled through multiple simulations to obtain an average performance estimate. Another limitation is that, while Mist supports fine-grained overlap of multiple operations, ensuring correctness remains challenging due to potential data races and value inconsistencies. Future work should focus on developing automated mechanisms to manage overlap and prevent execution conflicts. Additionally, although Mist can analyze arbitrary computational graphs, its efficient tuning algorithm assumes identical layers. Extending it to optimize models with heterogeneous architectures is an important direction.

9 Conclusion

We propose Mist, a memory, overlap, and imbalance aware method that enables efficient LLM training by co-optimizing all memory footprint reduction techniques and parallelism strategies in a comprehensive manner. Mist contributes a fine-grained overlap-centric schedule template, an symbolic-based efficient analysis system, and an imbalance-aware hierarchical auto-tuner to allow efficient optimization in a large search space over optimizations with complex interactions. As a result, Mist achieves 1.27× (up to 2.04×) over state-of-the-art distributed training systems such as Aceso. We hope that Mist will be able to help democratize LLM training for machine learning researchers and practitioners alike.

Acknowledgments

We sincerely thank our shepherd, Zhaoguo Wang, and the anonymous reviewers for their valuable feedback. We also appreciate members of the EcoSystem Research Laboratory at the University of Toronto for their discussions and suggestions, with special thanks to Anand Jayarajan, Xin Li, Wei Zhao, Yaoyao Ding, and Jiacheng Yang for their contributions. The authors with the University of Toronto are supported by Vector Institute Research grants, the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. 2023. The falcon series of open language models. *arXiv preprint arXiv:2311.16867* (2023).

[3] Muralidhar Andoorveedu, Zhanda Zhu, Bojian Zheng, and Gennady Pekhimenko. 2022. Tempo: Accelerating Transformer-Based Model Training through Memory Footprint Reduction. *Advances in Neural Information Processing Systems* 35 (2022), 12267–12282.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, David Berard, Geeta Chauhan, Anjali Chourdia, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. (2024). To appear at ASPLOS.

[5] Anthropic. 2024. Claude. https://claude.ai/.

[6] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 472–487.

[7] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*.

[8] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*. https://arxiv.org/abs/2204.06745

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[11] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. 2022. Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models. In *The Tenth International Conference on Learning Representations, ICLR*.

[12] Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. 2023. Slapo: A Schedule Language for Progressive Optimization of Large Deep Learning Model Training. arXiv:2302.08005 [cs.LG]

[13] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. 2021. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *International Conference on Machine Learning (ICML)*.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*.

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

[18] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. (2023).

[19] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.

[20] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[21] Jiangfei Duan, Xiuhong Li, Ping Xu, Xingcheng Zhang, Shengen Yan, Yun Liang, and Dahua Lin. 2023. Proteus: Simulating the Performance of Distributed DNN Training. *arXiv preprint arXiv:2306.02267* (2023).

[22] facebookresearch/llama. 2023. llama/MODEL_CARD.md. https://github.com/facebookresearch/llama/blob/main/MODEL_CARD.md.

[23] facebookresearch/llama. 2024. llama3. https://github.com/meta-llama/llama3.

[24] Thomas Fahringer and Bernhard Scholz. 1997. Symbolic evaluation for parallelizing compilers. In *Proceedings of the 11th international conference on Supercomputing*. 261–268.

[25] 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.

[26] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. 2023. Mobius: Fine tuning large-scale models on commodity gpu servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 489–501.

[27] Henrik Floberg. 2012. *Symbolic analysis in analog integrated circuit design*. Vol. 413. Springer Science & Business Media.

[28] John Forrest and Robin Lougee-Heimer. 2005. CBC user guide. In *Emerging theory, methods, and applications*. INFORMS, 257–277.

[29] Georges Gielen, Piet Wambacq, and Willy M Sansen. 1994. Symbolic analysis methods and applications for analog circuits: A tutorial overview. *Proc. IEEE* 82, 2 (1994), 287–304.

[30] Google. 2022. XLA. https://www.tensorflow.org/xla.

[31] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, et al. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. *arXiv preprint arXiv:2401.08156* (2024).

[32] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. Pipetransformer: Automated elastic pipelining for distributed training of transformers. *arXiv preprint arXiv:2102.03161* (2021).

[33] Horace He and Shangdi Yu. 2023. Transcending Runtime-Memory Tradeoffs in Checkpointing by being Fusion Aware. *Proceedings of Machine Learning and Systems* (2023).

[34] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. 2022. dpro: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training. *Proceedings of Machine Learning and Systems* (2022).

[35] 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).

[36] 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. *Proceedings of Machine Learning and Systems* 2 (2020), 497–511.

[37] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910* (2020).

[38] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations*.

[39] 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).

[40] 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.

[41] 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.

[42] 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).

[43] 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.

[44] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, et al. 2023. Efficient GPU Kernels for N: M-Sparse Weights in Deep Learning. *Proceedings of Machine Learning and Systems* (2023).

[45] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. 2024. {nnScaler}:{Constraint-Guided} Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 347–363.

[46] 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.

[47] Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, et al. 2022. Gact: Activation compressed training for generic network architectures. In *International Conference on Machine Learning*. PMLR, 14139–14152.

[48] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Li Li, Jingwen Leng, and Minyi Guo. 2023. DistSim: A performance model of large-scale hybrid distributed DNN training. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*.

[49] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 553–564.

[50] meta llama. 2024. llama3.1. https://ai.meta.com/blog/meta-llama-3-1/.

[51] meta llama/llama3. 2024. llama/MODEL_CARD.md. https://github.com/meta-llama/models/llama3_1/blob/main/MODEL_CARD.md.

[52] meta llama/llama3. 2024. llama/MODEL_CARD.md. https://github.com/meta-llama/models/llama3/blob/main/MODEL_CARD.md.

[53] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2023. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proc. VLDB Endow.* 16, 3 (2023), 470–479. https://doi.org/10.14778/3570690.3570697

[54] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR*.

[55] 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*. 1–15.

[56] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[57] 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.

[58] NVIDIA. [n. d.]. NVIDIA A100 GPUs. https://www.nvidia.com/en-us/data-center/a100/.

[59] NVIDIA. 2023. NVIDIA L4 GPUs. https://www.nvidia.com/en-us/data-center/l4/.

[60] openai. 2022. ChatGPT. https://openai.com/chatgpt/.

[61] 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).

[62] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, et al. 2023. Fp8-lm: Training fp8 large language models. *arXiv preprint arXiv:2310.18313* (2023).

[63] 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*. 891–905.

[64] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2023. Zero Bubble Pipeline Parallelism. *arXiv preprint arXiv:2401.10241* (2023).

[65] 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.

[66] 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.

[67] 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.

[68] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[69] 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.

[70] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[71] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Andrew Fitzgibbon, and Tim Harris. 2021. Distir: An intermediate representation for optimizing distributed neural networks. In *Proceedings of the 1st Workshop on Machine Learning and Systems*.

[72] Noam Shazeer. 2020. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202* (2020).

[73] 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).

[74] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243* (2019).

[75] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[76] 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.

[77] Jakub Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *Neural Information Processing Systems*. https://api.semanticscholar.org/CorpusID:244711821

[78] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[79] 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).

[80] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[81] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.

[82] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[83] Zhuang Wang, Xinyu Wu, Zhaozhuo Xu, and TS Ng. 2023. Cupcake: A Compression Scheduler for Scalable Communication-Efficient Distributed Training. *Proceedings of Machine Learning and Systems* 5 (2023).

[84] Geoffrey X Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[85] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. 2024. Accelerating the Training of Large Language Models using Efficient Activation Rematerialization and Optimal Hybrid Parallelism. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 545–561. https://www.usenix.org/conference/atc24/presentation/yuan

[86] Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. *Advances in Neural Information Processing Systems* 32 (2019).

[87] 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).

[88] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1089–1102.

[89] 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.

A Artifact Appendix

A.1 Abstract

Mist is an automatic distributed training configuration optimizer designed to tune the optimal configuration for the combination of parallelism strategies and memory footprint reduction techniques. This artifact includes the source code of the Mist prototype along with instructions for evaluating its functionality and reproducing key results.

A.2 Description & Requirements

A.2.1 How to access The Mist system is available at the following repositories: https://github.com/dazz993/mist and Zenodo (DOI: 10.5281/zenodo.14873554)

A.2.2 Hardware dependencies Experiments are conducted on up to four GCP L4 machines, each equipped with 8× NVIDIA L4 GPUs, and up to four AWS p4d.24xlarge machines, each with 8×NVIDIA A100 40GB GPUs. For artifact evaluation, an ideal testbed consists of a machine with 8 × GPUs, each with approximately 24GB of memory, as we can provide configurations suitable for direct evaluation on such hardware. Unless explicitly stated otherwise, the following evaluation workflow assumes this testbed. A general evaluation methodology for other GPUs and multi-node evaluation is also provided in our repository.

A.2.3 Software dependencies We provide a Docker image with NVIDIA GPU support for this artifact. The software environment includes CUDA 12.1, PyTorch v2.1.1, Megatron-LM (git-hash 38879f8), DeepSpeed (v0.12.6), and NCCL v2.18.6.

A.2.4 Benchmarks None.

A.3 Set-up

To install the artifact, users should clone the repository and build the Docker image. For users with GPUs other than L4 GPUs (sm_89), the environment variable `TORCH_CUDA_ARCH_LIST` in the Dockerfile may require modification.

```
git clone https://github.com/Dazz993/Mist.git
cd Mist
docker build -t mist -f Dockerfile .
```

Run the Docker container, mounting the Mist repository home to `/workspace`.

```
docker run --gpus all -it --rm --privileged \
  --ipc=host --shm-size=20G --ulimit memlock=-1 \
  --name "mist" -v $(pwd):/workspace/ mist
```

[Optional] To obtain stable results, especially on L4 machines, fix the GPU frequency accordingly:

```
nvidia-smi -ac 6251,1050
```

A.4 Evaluation workflow.

We provide scripts for reproducing single-node results. Multi-node experiments require large-scale clusters and additional setup and execution time, so end-to-end scripts are not provided for artifact evaluation. However, instructions are available in GitHub repository README. **We recommend users to follow the README as it provides extra explanations.**

A.4.1 Major Claims

- *(C1)*: Mist achieves an average of 1.28× (up to 1.73×) speedup compared to state-of-the-art manual system Megatron-LM. See Section 6.2 and Figure 11 and 12. This claim is validated by E2.
- *(C2)*: Mist demonstrates efficient tuning speed even with a large search space. See Section 16 and Figure 16. This claim is validated by E3.

A.4.2 Experiments

- E1: Kick-the-Tries [10 human-minutes]. This experiment evaluates the functionalities of Mist on Large Language Model analysis, execution, and distributed training optimization. Detailed explanations and expected results are also provided in the repository README.
  [Execution] Given a YAML configuration for running the GPT-3 1.3B model on two GPUs: `test-small-base`:
  - Run model performance analysis

    ```
    cd /workspace/benchmark/mist/analysis/
    python run.py --config-name test-small-base
    ```

  - Execute the model distributed training

    ```
    cd /workspace/benchmark/mist/exec/
    torchrun --nproc-per-node 2 \
        benchmark_one_case.py \
        --config-name test-small-base
    ```

  - Run model tuning (the hyperparameters in this configuration file are specifically tuned for GCP L4 GPUs).

    ```
    cd /workspace/benchmark/mist/tune/
    python tune_one_case.py --config-name test-small-base \
      +output_path=/workspace/benchmark/mist/tune/results/test-small-mist
    ```

    Then execute the optimized configuration:

    ```
    cd /workspace/benchmark/mist/exec/
    torchrun --nproc-per-node 2 \
      benchmark_one_case.py \
      --config-path /workspace/benchmark/mist/tune/results/ \
      --config-name test-small-mist
    ```

  [Results] The executed commands output the analysis results, execution time, and memory usage for the base configuration, as well as the execution time and memory usage for the optimized configurations.
- E2: Run Single-Node Performance Evaluation [4 compute-hours]. This experiment evaluates the performance of Mist on a single node for GPT and LLaMA models.

For the L4 machine, we provide pre-tuned configurations that enable a quick assessment of Mist's speedup compared to baseline models. Additionally, we provide a general process for evaluating on a new cluster. For further details, refer to the GitHub repository README.
[Execution]

- Evaluate Mist performance. Results are summarized in /workspace/benchmark/mist/tuned_configs/l4-24gb/gpt/summary.json and corresponding LLaMA folder.

```
cd /workspace/benchmark/mist/tuned_configs/
bash run_single_node.sh
```

- Evaluate Megatron-LM performance. Results are under /workspace/benchmark/megatron/results.

```
cd /workspace/benchmark/megatron/
bash scripts/tops/l4/gpt2/1_8xl4_node_1_pcie.sh
bash scripts/tops/l4/llama/1_8xl4_node_1_pcie.sh
```

- Evaluate DeepSpeed performance. Results are under /workspace/benchmark/deepspeed/results.

```
cd /workspace/benchmark/deepspeed/
bash scripts/tops/l4/gpt2/1_8xl4_node_1_pcie.sh
bash scripts/tops/l4/llama/1_8xl4_node_1_pcie.sh
```

[Results] We provide a python file to collect the results for easy comparison.

```
cd /workspace/benchmark/
python scripts/collect_single_node_results_v1.py
```

It provides tables with absolute throughput values or relative throughput improvements. Here we provide an example:

```
+----------------------+-------------+-------------+
| SpeedUp              | SpeedUp vs  | SpeedUp vs  |
|                      | Megatron    | DeepSpeed   |
+======================+=============+=============+
| gpt2-1.3b-flash_False | 1.175X      | 1.418X      |
+----------------------+-------------+-------------+
| gpt2-2.7b-flash_False | 1.141X      | 1.384X      |
+----------------------+-------------+-------------+
| gpt2-7.0b-flash_False | 1.222X      | 2.053X      |
+----------------------+-------------+-------------+
```

- E3: Benchmarking Tuning Time [0.75 compute hours]. This experiment reproduces the tuning time results for Mist, as shown in Figure 16.
We demonstrate how tuning time varies as the search space expands incrementally. To evaluate this, we run a GPT-22B model on a 4 × 8 GPU setup. Beyond examining tuning time, this experiment also provides insights into the performance of large-scale distributed training, as we can see the performance improvements when more optimizations are applied.
[Execution]

```
cd /workspace/benchmark/mist/benchmark-tuning-time
python run.py --model=gpt2/22b -n 4 -m 8
```

The results are shown in /workspace/benchmark/mist/benchmark-tuning-time/results/.../summary.json.

A.5 Notes on Reusability

Mist provides a symbolic execution system that represents all tensors with symbolic dimensions. Additionally, it supports tracing, which generates a corresponding symbolic computational graph. This feature can serve as an educational tool, helping users understand shape propagation and how each input dimension is utilized. Furthermore, it serves as a basis for exploring performance estimation in future research.