LegoAI: Auto-Scaling Large Model Training

A DISSERTATION PRESENTED
BY
SANKET JAYANT PURANDARE
TO
ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2025

Dissertation advisor: Stratos Idreos Sanket Jayant Purandare

LegoAI: Auto-Scaling Large Model Training

Abstract

Training large AI models is computationally intensive. State-of-the-art language and vision models (LLMs and VLMs) often require thousands of GPUs and weeks or even months of training. As models scale to meet the demands of modern applications, efficient distributed training becomes essential, yet remains highly complex. No single distributed training configuration (or *training recipe*) works across all combinations of model architectures, hardware platforms, and data modalities. Practitioners must explore a vast configuration space through costly trial and error, often building and tuning implementations manually. Even then, out-of-memory errors and sub-optimal performance are common. This complexity is further compounded by the difficulty of synthesizing efficient implementations for selected configurations. Existing frameworks are fragmented across disparate libraries, lack interoperability, and are difficult to maintain, making the development, evaluation, and reuse of training recipes a significant engineering burden.

This thesis introduces LegoAI, a system that transforms distributed AI training into an automated, scalable, and modular process. Given a model, dataset, and hardware configuration, LegoAI automatically selects the optimal distributed training configuration and generates a production-ready implementation that scales to thousands of GPUs. At its core, LegoAI serves as a synthesis engine: it decomposes state-of-the-art training strategies into modular, composable design principles and unifies them within a single coherent framework. In doing so, LegoAI exposes a vast configuration space that comprises not only existing state-of-the-art algorithms but also entirely new designs

beyond them. Through high-fidelity simulation, it predicts memory usage and runtime without requiring execution, enabling fast and safe exploration of the configuration space. Finally, for the empirically optimal configuration, it synthesizes an efficient and scalable implementation. In addition to exploring, comparing, and deploying state-of-the-art algorithms, LegoAI enables full-stack research by analyzing and synthesizing entirely new training algorithms derived from the design space through the composition of existing design principles.

We evaluate LegoAI across diverse models, GPU types (A100, H100), and interconnects (InfiniBand, RoCE), demonstrating strong scalability, accurate simulation, and effective policy synthesis. LegoAI achieves speedups of 65.08%, 12.59%, and 30% over optimized baselines on LLaMA 3.1 models at 128, 256, and 512 GPU scales, respectively. It predicts runtime with over 90% accuracy and memory usage with 99.9% accuracy across hardware configurations. To demonstrate LegoAI's research capabilities, we synthesize new memory-efficient training algorithms based on recomputation that reduce overhead by up to 90% compared to baselines, while achieving superior compute-memory trade-offs by matching ILP-optimal solutions and running over 100$\times$ faster.

Thus, LegoAI is the first system to unify the synthesis, simulation, and deployment of distributed training strategies, significantly reducing cost, complexity, and uncertainty while enabling broader and more efficient exploration of the large-scale AI training design space.

Contents

Listing of figures

List of Tables

Dedicated to my grandparents

Acknowledgments

First and foremost, I want to thank my advisor, **Stratos Idreos**, for giving me the space to explore and the structure to stay grounded. His trust let me chase ideas I cared about, and his clarity helped shape them into something real. I will always remember our late-night meetings and the conversations that went beyond research, touching on what it means to become a well-rounded researcher and person. I have learned a lot from how he thinks, and even more from how he motivates and instills unwavering belief. No matter how many things were close to deadlines, including paper submissions, presentations, and defense preparation, he never discouraged me. Instead, he showed even more enthusiasm and pushed me to do better.

I am also grateful to my thesis committee: **Prof. David Brooks**, **Prof. H. T. Kung**, and **Prof. Todd Zickler**. Thank you for your time, thoughtful questions, and constructive feedback. Your insights helped make this work more rigorous, complete, and grounded.

To my labmates and collaborators, thank you for the whiteboard debates, the bug hunts, and the shared momentum. This work reflects your input more than you know. **Andrew, Emma, Sophie, Costin, Alicia, Glen, and Vlad**, your energy, curiosity, and intelligence constantly inspired me, and I learned a lot from our collaboration. **Brian, Utku, Qitong, Wasay, Subarna, Subhadeep, Suvam, and Arpita**, thank you for mentoring me, guiding me, and spending countless hours in brainstorming sessions and technical deep dives. Our random philosophical and geeky conversations reminded me that I was in the right place with the right people and made research genuinely fun.

To my collaborators at PyTorch, **Animesh, Gokul, Xuan, Wanchao, Tianyu, Less, and Wei**, thank you for helping push my research into production and for working with me to scale up prototypes into full systems.

To my friends, thank you for your patience, encouragement, and the late-night conversations that had nothing to do with research. You helped me reset when I needed it and made space for this work when I could not. To my friends in Boston, **Greta, Shivam, Charuvi, Moulshree, Radhika, Veronica, Vanessa, Ricky, and Javi**, thank you for making the city feel like home, listening to my rants and complaints, cooking for me, and making Boston winters not only bearable but joyful. I will always remember our Thanksgiving celebrations and hope we continue the tradition. To my PhD compatriot and best roommate **Harsh**, thank you for being patient, wise, and mature, and for throwing the best parties. You were a true one-stop solution for all my worries.

To the friends and their partners I met during my time at IISc, **Stanly, Ajinkya, Bhushan, Vysakh, Swapnil, Bhagyashree, and Divya**, thank you for supporting me through personal ups

and downs, always believing in me, hyping me up, and being there on long video calls whenever I needed you. Even after moving away, I never felt like I left you. We have grown together from naive college graduates to industry engineers and researchers. The journey has been incredible to share with you.

To **Rucha**, thank you for motivating me to pursue a PhD and for giving me an early glimpse into the world of research. You taught me to aim higher and helped instill ambition in me when it mattered most.

To **Prof. Jayant Haritsa**, **Prof. Matthew Jacob**, and **Prof. Bhavana Kanukurthi**, thank you for your guidance during my time at IISc. You helped me build a solid foundation and taught me how to think and do research. **Prof. Jayant**, you were instrumental in guiding me towards my master's dissertation, which served as a stepping stone toward securing a PhD position with Stratos.

To **Dr. Karthik Ramachandra**, with whom I spent an amazing six months at Microsoft Research, thank you for guiding me through my first research paper, helping me secure my first patent, and giving me a thorough exposure to industry research.

To my friends **Vivek, Laura, Tejashri, and Leena**, thank you for filling my life with love, laughter, vacations, incredible dinners, and for helping me find joy beyond academia. Traveling, hiking, working out, and camping with you brought me balance and perspective. Thank you for helping me navigate real-life decisions, for your advice, and for giving me a safe space to open up.

To my mentor **Rajesh Singh Vats**, who coached me for the GATE examination, thank you for helping me secure a spot at IISc, which completely transformed my career trajectory. I would not have the opportunities I have today without your support.

To my undergraduate thesis advisor and mentor **Prof. Tanuja Sarode**, thank you not only for guiding my thesis but also for nurturing leadership abilities and soft skills that have helped me throughout my career.

To my parents, thank you for being steady through it all. You never needed to understand the details to believe in the effort. Your belief in me, your sacrifices, and your strength meant I never had to look elsewhere for motivation.

To my sister **Manasi**, thank you for your unconditional support and for being my best friend, mentor, and second mother all at once.

To my cousin and best friend **Archit**, thank you for being my wise little brother. You reminded me to take life one step at a time and always told me what I needed to hear, that everything will be fine.

To my cousins and their partners **Mayuri, Sagar, Neeraj, and Seema**, thank you for giving me family abroad and for being a constant source of warmth and support.

To my cousins in India, **Yashodhan, Vidya, Risha, and Rutvi**, thank you for always being there for me and staying connected despite the distance.

And to my aunts and uncles, thank you for your encouragement, love, and belief in me over the years. Your presence, near or far, has always mattered.

1 Introduction

1.1 Motivation

AI models are the driving force behind a wide range of applications. Large Language Models (LLMs) (Devlin, 2018; Liu et al., 2019; Radford et al., 2019; Chowdhery et al., 2023; Anil et al., 2023; Achiam et al., 2023; Dubey et al., 2024; Jiang et al., 2024; Abdin et al., 2024) have been the driving force behind the advancement of natural language processing (NLP) applications spanning

language translation, content/code generation, conversational AI, text data analysis, creative writing and art, education, and research, etc. At the same time Large Vision Models (LVMs) (Ilharco et al., 2021; Rombach et al., 2021; Li et al., 2024a; Meta AI, 2024; Labs, 2024) have been central to progress in image recognition, visual alignment, multimodal interaction, and prompt-based image generation.

Training AI models is computationally expensive. Achieving state-of-the-art performance in large language models (LLMs) requires extreme computational scale and investment. For example, Llama 3.1 was trained with 405 billion parameters on 15 trillion tokens, consuming 30.84 million GPU hours across 16,000 H100 GPUs (Dubey et al., 2024), while Google's PaLM used 540 billion parameters and 0.8 trillion tokens, requiring 9.4 million TPU hours across 6,144 TPUv4 chips (Chowdhery et al., 2023). These models showcase remarkable capabilities in language understanding and generation, but also highlight the steep memory, compute, and time requirements needed to train at this scale. As models continue to grow in size and complexity, these costs extend across domains, often reaching millions of GPU hours and significant financial overhead. Reducing the time, resource usage, and cost of training has become a critical challenge for sustainable and scalable AI development.

Distributed training is essential to scale large models across thousands of GPUs, with no single recipe being universally optimal. For example, Llama 3.1 (405B) was trained using 4D parallelism comprising 8-way Tensor, 16-way Context, 16-way Pipeline, and 8-way Fully Sharded Data Parallelism across 16,000 GPUs. In contrast, PaLM (540B) was trained on 6,144 TPUs using 3D parallelism with 12-way Tensor, 256-way Fully Sharded Data, and 2-way Data Parallelism. Despite their similar scale, the two models used distinct strategies due to differences in architecture, size, and hardware.

Efficient distributed training requires stacking several techniques, often with complex trade-offs. Training large language models (LLMs) at scale is a daunting task that requires a delicate

balance of parallelism, computation, and communication, all while navigating intricate memory and computation trade-offs. The massive resources required for training make it prone to GPU failures, underscoring the need for efficient recovery mechanisms and checkpointing strategies to minimize downtime (Eisenman et al., 2022; Wang et al., 2023; Gupta et al., 2024; Maurya et al., 2024; Wan et al., 2024). To optimize resource utilization and achieve elastic scalability, it is crucial to combine multiple parallelism techniques, including Data Parallel (DDP, HSDP and FSDP) (Li et al., 2020; Rajbhandari et al., 2020; Zhang et al., 2022a; Zhao et al., 2023), Tensor Parallel (TP) (Narayanan et al., 2021; Wang et al., 2022; Korthikanti et al., 2023), Context Parallel (CP) (Liu et al., 2023; Liu & Abbeel, 2024; NVIDIA, 2023; Fang & Zhao, 2024), and Pipeline Parallel (PP) (Huang et al., 2019b; Narayanan et al., 2019, 2021; Tang et al., 2024b). By stacking these parallelisms with memory and computation optimization techniques, such as activation recomputation (Chen et al., 2016; Korthikanti et al., 2023; He & Yu, 2023; Purandare et al., 2023), mixed precision training (Micikevicius et al., 2018, 2022), and deep learning compilers (Bradbury et al., 2018; Yu et al., 2023; Li et al., 2024b; Ansel et al., 2024b), it is possible to maximize hardware utilization.

Discovering an effective training recipe requires expert intuition and extensive experimentation. While existing systems support a broad range of distributed training techniques, they also expose a large number of configuration options, including parallelism dimensions, sharding strategies, precision modes, and memory trade-offs. Choosing the right combination of techniques, referred to as a training recipe, for a given model, data batch, hardware setup, and performance objective is highly non-trivial. This process is highly context-dependent and typically demands expert intuition, deep system knowledge, and substantial iterative experimentation.

Training under sub-optimal configurations can significantly increase training time and resource consumption. Even when frameworks support a wide range of optimization techniques, using them ineffectively can be extremely costly. Consider our earlier examples: LlaMA 3.1 (405B) and PaLM (540B), which consumed 30.84 million GPU hours and 9.4 million TPU hours, re-

Table 1.1: Cost and experiment counts per model size (in k$) for 3,306 experiments, resulting in a total estimated cost of $469.17K.

Model (B)	Cost (k$)	Experiments per node config						
		1	2	4	8	16	32	64
1.34	107.34	65	120	127	149	158	111	78
3.57	119.78	63	117	126	148	176	118	94
8.86	119.95	61	116	122	145	176	122	93
80.0	122.10	57	87	125	150	185	122	95

spectively. Hypothetically, if these models were trained using configurations that are just 10 to 25% slower than an optimally tuned setup Tazi et al. (2025), this would result in an additional 3.08 to 7.71 million GPU hours or 0.94 to 2.35 million TPU hours. These inefficiencies translate directly into substantial financial costs and increased environmental impact due to unnecessary energy consumption.

Crafting the best training recipe requires costly exploration of the performance space. Identifying an optimal training recipe, which consists of a combination of parallelism strategies, memory optimizations, and precision settings for a given model, dataset, and hardware configuration, is a complex and expensive process. The configuration space is large and filled with highly sensitive parameters that must be tuned to balance cost, accuracy, communication overhead, and computational efficiency. Even with modern tooling, this remains an iterative trial-and-error process that requires multiple full or partial training runs on GPU clusters. Each run involves job scheduling, execution, metric collection, and debugging. Frequent runtime failures, such as out-of-memory errors, further increase overhead. As models grow and training scales to hundreds or thousands of GPUs, this level of empirical exploration becomes increasingly impractical and motivates the need for automated and simulation-driven approaches.

Case Study: Exhaustive benchmarking for training configuration selection is prohibitively expensive.

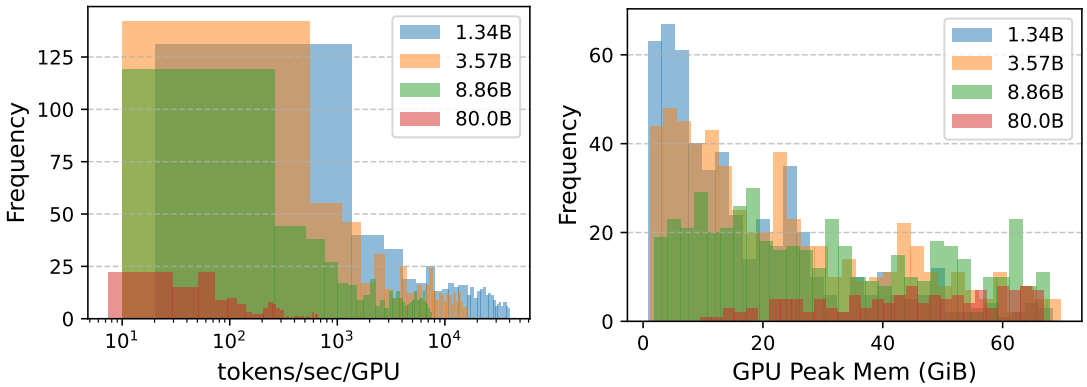


Figure 1.1: Huggingface performance benchmarking of 1,728 out of 3,306 training runs reveals significant variance in throughput and peak memory consumption, highlighting the impact of training configurations.

To illustrate the cost and complexity of empirical exploration, we highlight a performance benchmarking study conducted by Huggingface (Tazi et al., 2025), which evaluated training configurations for LLaMA models of various sizes (1.34B, 3.57B, 8.86B, and 80B parameters). The study maintained a fixed global batch size of 256 and sequence length of 4096, and systematically varied key parameters such as the number of nodes (ranging from 1 to 64), degrees of data parallelism (1 to 256), tensor parallelism (1 to 32), pipeline parallelism (1 to 128), gradient accumulation steps (1 to 256), micro-batch size (1 to 256), and ZeRO sharding strategies (stage 0 and 1). A total of 3,306 configurations were benchmarked on a cluster with up to 512 NVIDIA H100 GPUs (8 GPUs per node), with 1,728 runs completing successfully and 1,578 failing due to crashes or out-of-memory (OOM) errors.

The results reveal three major insights. First, training performance and memory efficiency are highly sensitive to configuration choices. As shown in Figure 1.1, throughput and peak memory usage vary significantly across configurations. Trade-offs are inherent; for example, activation checkpointing reduces memory usage at the cost of recomputation, while tensor or fully shared data parallelism improves memory distribution but increases synchronization overhead. Inefficient overlap of computation and communication can further degrade performance, making configuration selection

highly non-trivial.

Second, failure modes such as OOM errors are frequent and offer little diagnostic value. Out of 3,306 runs, 1,578 failed due to memory exhaustion, resulting in substantial waste of resources and no actionable insights for tuning future configurations.

Third, the financial cost of empirical benchmarking is substantial. The study incurred an estimated total cost of $469.17K, assuming a cost of $98.5 per node-hour* and a runtime of five minutes per experiment. Table 1.1 provides a detailed breakdown by model size and node count.

Despite fixing multiple parameters, including batch size, sequence length, precision mode, and activation checkpointing strategy, the study required thousands of runs to explore only a narrow slice of the full configuration space. Expanding this search to include additional models and optimization techniques would drastically increase both cost and complexity, reinforcing the need for automated and simulation-based approaches to training configuration selection.

Current Distributed Training Frameworks Fall Short: Gaps in Composability, Breadth, Flexibility, and Efficiency. Exacerbating the challenge of discovering an optimal training recipe is the difficulty of engineering an efficient implementation for a given configuration. Despite advances in distributed training techniques, existing frameworks struggle to support their full breadth and composability. Most systems are non-composable, making it hard to stack multiple parallelism strategies alongside memory and compute optimizations, which limits both efficiency and design space exploration. Their architectures often lack flexibility and modularity, impeding integration of new techniques, hardware targets, and evolving software optimizations. Many frameworks underutilize advanced hardware features, offer limited support for customizable checkpointing, and exhibit sub-optimal GPU efficiency. In production, they lack scalable distributed checkpointing, robust failure recovery, and effective debugging tools. Additionally, many rely on poorly main-

*Cost estimate based on AWS EC2 p5.48xlarge (8×H100) on-demand pricing (CloudPrice.net, 2025). The total expenditure was estimated using the formula: Cost = Number of Experiments × Nodes × 98.5 × $\left(\frac{5}{60}\right)$, assuming five minutes per experiment.

tained external dependencies and fail to fully leverage PyTorch's native optimizations, compiler infrastructure, and kernel support, leading to inefficiencies and compatibility challenges across the stack.

1.2 Thesis Problem and Thesis Statement

1.2.1 Problem

Given a specific training context consisting of a model architecture, dataset, and hardware platform, automatically and efficiently identify the empirically optimal distributed training configuration, which spans parallelism strategies, memory optimizations, and precision settings, and generate a corresponding implementation that maximizes throughput while satisfying hardware constraints.

1.2.2 Statement

This thesis introduces LegoAI, a system that transforms distributed AI training into an automated, scalable, and modular process. Given a model, dataset, and hardware configuration, LegoAI automatically selects the optimal distributed training configuration and generates a production-ready implementation that scales to thousands of GPUs. At its core, LegoAI serves as a synthesis engine: it decomposes state-of-the-art training strategies into modular, composable design principles and unifies them within a single coherent framework. In doing so, LegoAI exposes a vast configuration space that comprises not only existing state-of-the-art algorithms but also entirely new designs beyond them. Through high-fidelity simulation, it predicts memory usage and runtime without requiring execution, enabling fast and safe exploration of the configuration space. Finally, for the empirically optimal configuration, it synthesizes an efficient and scalable implementation.

LegoAI consists of two core subsystems:

1. **TorchTitan**, a unified and production-grade distributed training framework that supports modular and composable four-dimensional parallelism with elastic scaling. TorchTitan consolidates advanced parallelism strategies into a single abstraction that simplifies implementation, benchmarking, and the development of new training algorithms.

2. **TorchSim**, a simulation-based predictor that estimates memory usage and runtime for arbitrary training configurations. TorchSim combines analytical models with learned cost functions to emulate GPU execution at the operator level, generalizing across model architectures, parallelism schemes, hardware types, and interconnect topologies.

To demonstrate the extensibility of LegoAI and its capability to support new research, we introduce Auto-SAC, a scalable algorithm that uses TorchSim 's fine-grained predictions to generate optimal selective activation checkpointing (SAC) strategies. Integrated into TorchTitan, Auto-SAC highlights LegoAI 's ability to unify simulation, synthesis, and deployment within a single system.

1.3 TorchTitan: A Unified, Modular, Composable, and Scalable Distributed Training Framework

Although distributed training techniques have progressed significantly, current frameworks fall short of generating efficient implementations for arbitrary distributed training configurations. They are limited in integration, usability, and extensibility, constraining their utility in both research and production workflows.

1.3.1 Existing systems struggle to support full breadth and composability for distributed training

Several frameworks provide APIs for building distributed training workflows, including Megatron-LM (Narayanan et al., 2021), DeepSpeed (Rasley et al., 2020), veScale (Inc., 2024), Slapo (Chen et al., 2023), and PyTorch Distributed (Paszke et al., 2019; Meta Platforms, Inc., 2024). However, these systems exhibit significant limitations in flexibility, integration, and scalability. Megatron-LM requires intrusive model modifications to work with TransformerEngine, lacks seamless integration of Fully Sharded Data Parallel (FSDP) with tensor and pipeline parallelism, and does not support advanced pipeline schedules to reduce computation overhead. DeepSpeed depends on Megatron-LM for tensor and context parallelism and offers only limited support for FSDP and advanced pipeline scheduling. veScale does not support FSDP, context parallelism, selective activation checkpointing, Float8 training, or the PyTorch `torch.compile` backend, and provides only three pipeline schedules in contrast to the six available in TorchTitan. Slapo introduces a schedule language to express training optimizations such as three-dimensional parallelism and supports progressive application of high-level transformations, but still lacks full integration with diverse parallelism and optimization strategies. These limitations restrict composability and hinder systematic exploration of the training configuration space, both of which are critical for scalable and efficient model training.

1.3.2 Toward Composable Distributed Training: DTensor and DeviceMesh as First-Class Primitives

Root cause: Lack of unified tensor and device abstractions across the stack. A central reason for the non-composability and rigidity of existing distributed training frameworks is the absence of unified tensor and device abstractions that span the entire software stack. Without such foundational

components, parallelism strategies, checkpointing mechanisms, and performance optimizations remain fragmented and ad hoc. This fragmentation limits modularity, restricts scalability, and complicates extensibility, making it difficult to build systems that can flexibly combine multiple training techniques or adapt to evolving hardware environments.

Using DTensor and DeviceMesh as foundational building blocks. To address this, we adopt PyTorch's Distributed Tensor (DTensor) and DeviceMesh as core primitives for structuring distributed computation. The DeviceMesh provides a logical organization of the compute cluster by arranging devices into a multi-dimensional grid, where each dimension corresponds to a distinct parallelism strategy. It manages process group creation and device communication across these dimensions in a unified and scalable manner.

DTensor is a distributed tensor that is sharded along one or more dimensions of the DeviceMesh and encodes its sharding specification. It supports sharding propagation, which automatically carries sharding metadata through tensor operations. This enables the composition of multiple forms of parallelism without requiring manual tracking or user intervention. When a sharded tensor is used as input, output tensors inherit the correct sharding behavior by default. DTensor also supports collective operations in sharded contexts, preserving semantic correctness and enabling reliable, deterministic execution across devices.

1.3.3 Solution: Redesigning Distributed Training from First Principles

TorchTitan 's key contribution is the unification of distributed parallelism and optimization techniques within a cohesive and extensible framework. By building on and extending PyTorch's DTensor and DeviceMesh abstractions (PyTorch Community, 2023b), TorchTitan provides a unified representation of distributed training that simplifies the composition of parallelism strategies and preserves consistent single-device semantics through principled sharding primitives. In contrast to prior systems that rely on rigid or task-specific implementations, TorchTitan offers a

systematic foundation for distributed execution. This enables rigorous exploration of configuration options, robust benchmarking of existing methods, and the discovery of new strategies across the broader design space.

TorchTitan is a full-fledged distributed training system for large language models (LLMs), not merely a collection of isolated techniques. Its modular and extensible architecture supports seamless integration of four-dimensional parallelism, advanced optimization techniques, and scalable checkpointing mechanisms, all while leveraging native PyTorch capabilities. The system is designed for production-scale training on thousands of GPUs, while also reducing integration complexity and accelerating experimentation, setting a new standard for scalable, composable, and flexible distributed training.

Finally, TorchTitan serves as an experimental testbed that enables users to curate, benchmark, and compare multiple training recipes for the same model and hardware configuration. Its strength lies in its ability to comprehensively capture the distributed training configuration space, allowing principled evaluation across a broad spectrum of strategies to identify the most effective ones.

1.4 TorchSim: High Fidelity Runtime and Memory Estimation for Distributed Training

While TorchTitan allows users to curate and evaluate the training recipes by mixing and matching different configurations, users still need to find the best training configuration for launching their training run.

1.4.1 The Holy Grail for AI Training at Scale: Accurate Runtime and Memory estimation.

Accurate Cost Estimation as a Foundation for Training Recipe Search If it were possible for machine learning practitioners to estimate the memory consumption and runtime of a training configuration before execution, it would fundamentally change how large-scale model training is conducted. First, accurate memory estimation would allow immediate detection of configurations likely to fail due to out-of-memory errors, preventing wasted time and resources. Second, accurate runtime prediction would enable practitioners to select the fastest configuration among those that satisfy memory constraints. Together, these capabilities would eliminate the need for trial-and-error experimentation, allowing researchers and engineers to assess feasibility and performance upfront, before committing expensive compute resources, time, and cloud budget.

The complexity of end-to-end training cost estimation at scale. Despite its transformative potential, accurate cost estimation is extremely challenging due to the intricate nature of distributed training systems. First, training performance depends heavily on the selected parallelism strategy, such as data, tensor, pipeline, or hybrid parallelism, each introducing different communication patterns and synchronization overheads that influence runtime in non-trivial ways. Second, hardware heterogeneity further complicates prediction. GPU architecture, memory bandwidth, interconnect topology, and hardware-specific scheduling behaviors all impact both computational throughput and communication efficiency.

Third, memory usage is governed by tensor liveness, which varies significantly across different operator sequences and training strategies. Constructing a general analytical model that accounts for memory lifetimes across arbitrary training configurations is infeasible. Finally, modern training frameworks introduce dynamic runtime optimizations such as overlapping communication and computation through asynchronous execution streams. These behaviors evolve rapidly with

framework updates, making theoretical performance modeling brittle and unreliable at scale. As a result, practical and accurate cost estimation requires a simulation-based approach that can emulate real-world execution while accounting for the full complexity of the training stack.

1.4.2 Limitations of Existing Runtime and Memory Estimation Approaches

Existing runtime estimation techniques are largely limited to simplified single-GPU or kernel-level settings Geoffrey et al. (2021); Lee et al. (2025a); Zhang et al. (2022b); Li et al. (2022), and fail to capture the complexity of distributed training. Communication models often neglect critical system factors, including multi-tier network topologies, collective communication algorithms, and the impact of straggler delays Lee et al. (2025b); Won et al. (2023); Mohammad et al. (2017). Moreover, accurate end-to-end performance prediction requires modeling the *computation–communication overlap* introduced by advanced distributed training strategies such as Fully Sharded Data Parallel (FSDP), Tensor Parallelism (TP), Pipeline Parallel (PP), and Context Parallelism (CP). To the best of our knowledge, no existing work faithfully simulates these algorithms, leaving accurate runtime estimation an unsolved problem.

Similarly, memory estimation tools are primarily profiling-based and operate *post hoc* Shi & DeVito (2023); pyt (2025a), offering no predictive insights into the memory impact of training configurations or the ability to prevent Out-of-Memory (OOM) errors proactively. Analytical techniques for estimating peak memory usage Gao et al. (2020); Narayanan et al. (2021) are difficult to maintain and often inaccurate due to the opaque and evolving internals of modern training frameworks. Other single-GPU tools Yu et al. (2020); Su et al. (2024) require actual execution and do not generalize to distributed contexts; they also lack detailed memory attribution and breakdown. To the best of our knowledge, no existing method provides accurate, predictive memory estimation for full-scale distributed training.

1.4.3 Solution: Unified Modeling of Large-Scale AI Training Hardware-Conscious Learned and Analytical Estimation and Distributed GPU Execution Simulation

We introduce TorchSim, a predictive tool for estimating runtime and memory consumption in distributed deep learning training workloads without requiring GPU execution.

TorchSim combines hardware-aware compute models with communication models that are sensitive to topology, algorithmic structure, and collective communication patterns to accurately predict operator-level execution times. It employs a detailed simulator that closely replicates the multi-stream GPU execution model, capturing compute–communication overlap, exposed communication phases, and synchronization overheads to support accurate end-to-end runtime estimation.

For memory prediction, TorchSim tracks tensor memory usage at operator-level granularity without allocating memory, and emulates memory consumed by distributed collective operations. By mimicking PyTorch's memory management and execution semantics, TorchSim can simulate realistic training behavior, including effects from sharding, activation recomputation, and communication buffering.

This capability allows users to evaluate training configurations and cluster topologies before execution, enabling principled design decisions and eliminating the need for costly empirical benchmarking. TorchSim serves as the foundation for system-level optimization in LegoAI, making accurate performance prediction a first-class component of the distributed training pipeline.

1.5 Auto-SAC: Enhancing the Compute–Memory Efficiency Trade-Off in Distributed Training

To demonstrate the extensibility of LegoAI and its ability to drive principled system innovation, we develop Auto-SAC, a principled and scalable algorithm for generating selective activation

checkpointing (SAC) policies. SAC is a widely adopted technique for reducing peak memory usage by selectively recomputing intermediate activations during the backward pass. While existing approaches are either manually configured or use limited automation, they often fail to support fine-grained control or adapt to realistic hardware constraints. Auto-SAC addresses these challenges by leveraging TorchSim 's operator-level runtime and memory estimates, not only for simulation, but to empirically construct memory–compute trade-off curves and guide memory-aware optimization.

Auto-SAC operates in two stages. At the global level, we formulate a memory-constrained Integer Linear Program (ILP) that determines which modules should apply SAC and how much activation memory to discard. This optimization is driven by piecewise-linear approximations of memory–recomputation trade-off curves, generated from TorchSim 's operator-level performance predictions. At the local level, we generate fine-grained SAC policies for each selected module using one of three algorithms: a greedy heuristic, a knapsack-based dynamic programming solver, or a local ILP for exact operator-level optimization. These local solvers balance recomputation cost with memory savings, enabling precise activation discard decisions under module-level constraints.

The key insight behind Auto-SAC is to decompose the SAC policy generation problem into a two-level hierarchy. The global module-level ILP enables tractable coordination under memory budgets across the model, while the local per-module optimization enables fine-grained control over which activations to retain or recompute. This hierarchical approach ensures that SAC policies are both globally coordinated and locally optimized.

We integrate Auto-SAC into TorchTitan to highlight the extensibility of our system and enable rigorous, apples-to-apples comparisons with existing SAC strategies. This integration reinforces TorchTitan 's role not only as a production-grade distributed training framework, but also as a full-stack research platform for prototyping and benchmarking memory–compute trade-offs. Auto-SAC composes with all parallelism strategies handled by TorchTitan, including Fully Sharded Data Parallel (FSDP), Tensor Parallelism (TP), and Context Parallelism (CP). Its heuristic

solvers match the quality of ILP-based solutions while running several orders of magnitude faster, making Auto-SAC practical for real-world, large-scale training deployments.

1.6 Thesis Contributions

This thesis introduces LegoAI, a system that automatically and efficiently identifies the empirically optimal distributed training configuration for a given model, dataset, and hardware platform, and generates a corresponding production-ready implementation. It comprises two core subsystems: TorchTitan, a unified and scalable distributed training framework, and TorchSim, a high-fidelity simulator for runtime and memory estimation. To demonstrate the extensibility and optimization capabilities of LegoAI, we develop Auto-SAC, a proof-of-concept system for generating memory-aware selective activation checkpointing policies.

The key contributions of this thesis are organized as follows.

1.6.1 Development and deployment of a Production-Grade, Unified, Modular, Composable, and Scalable Distributed Training Framework

To enable automated generation and deployment of training recipes, we develop TorchTitan, a production-grade framework for composable four-dimensional parallelism and scalable training. Our contributions include:

1. We advance DTensor by extending its sharding to support n-dimensional parallelism, enabling compatibility with `torch.compile` for compiler-level optimization, and supporting efficient checkpointing via state dict serialization. We also resolve critical bugs to improve its production readiness.
2. We demonstrate how to compose diverse parallelism strategies, data, tensor, pipeline, and context parallelism, under a unified abstraction, enabling multi-dimensional configuration space exploration for large language model training (§3.2).

3. We enable hardware-software co-optimization by exploiting advanced GPU features, supporting customizable activation checkpointing strategies, and integrating `torch.compile` for compiler-assisted memory and communication efficiency (§3.3).

4. We build production-ready support into TorchTitan via scalable distributed checkpointing for fault recovery, integration of debugging tools (e.g., Flight Recorder), and detailed logging for metrics and tracing (§3.4).

5. We evaluate TorchTitan on the LLaMA 3.1 family across 1D to 4D parallelism at scales from 8 to 512 GPUs. We demonstrate training accelerations of 65.08% (LLaMA 3.1 8B, 128 GPUs), 12.59% (LLaMA 3.1 70B, 256 GPUs), and 30% (LLaMA 3.1 405B, 512 GPUs), and highlight the role of 4D parallelism in enabling long-context training on H100 GPUs (§3.5.2).

6. We provide systematic training recipes and practical guidelines to help users navigate configuration decisions across model sizes and cluster topologies (§3.6).

These contributions of TorchTitan position LegoAI as a platform that unifies implementation, deployment, and exploration of the distributed training configuration space.

1.6.2 Building a High Fidelity Runtime and Memory Estimation Simulator for Distributed Training

To support configuration selection without costly trial-and-error, we develop TorchSim, a simulator that predicts runtime and memory usage without requiring actual GPU execution. Key contributions include:

1. We design TorchSim as a model-agnostic, non-intrusive simulation framework that integrates with PyTorch training pipelines. It allows pluggable compute and communication models and supports diverse hardware and distributed configurations (§ 4.1 and § 4.2.1).

2. We build accurate compute and communication estimators using learned models and sta-

tistical methods. TorchSim models operator-level execution, synchronization, and compute–communication overlap to simulate advanced workflows such as FSDP, TP, and CP (§ 4.5, § 4.4, and § 4.6).

3. We implement an operator-level memory estimator that tracks tensor allocations and deallocations, categorizes memory usage (parameters, gradients, activations, optimizer states), and provides per-device memory statistics for optimization and debugging (§ 4.3).

4. We open-source TorchSim as part of TorchTitan Liang et al. (2024), and release accompanying data: benchmarking scripts, compute models (A100/H100), and collective communication datasets (InfiniBand and RoCE).

5. We evaluate TorchSim across diverse models (Gemma-2B, CLIP, T5, ViT, LLaMA variants), GPUs (A100 and H100), cluster sizes (64–512 GPUs), and parallelism techniques. TorchSim achieves 99.9% accuracy in memory estimation and ≥90% accuracy in runtime estimation (§ 4.7).

These capabilities establish TorchSim as LegoAI's foundation for predictive configuration selection, enabling automated exploration of distributed training strategies with high fidelity.

1.6.3 Advancing the state-of-the-art to deliver Superior Compute-Memory Trade-off in Distributed Training

To demonstrate the extensibility of LegoAI and the utility of TorchSim in policy synthesis, we develop Auto-SAC, a system for automatically generating optimal selective activation checkpointing (SAC) policies. Key contributions include:

1. We build a simulation-based estimator using TorchSim that captures operator-level runtime and memory statistics, enabling empirical construction of memory–compute trade-off curves (§ 5.3).

2. We formulate a global mixed-integer linear program (MILP) that selects modules for checkpointing and assigns discard budgets to minimize recomputation while satisfying a user-defined memory budget (§ 5.4).

3. For each checkpointed module, we generate fine-grained SAC policies using one of three solvers: a greedy heuristic (MSPS), a knapsack-based dynamic programming solver, or a local ILP for exact optimization (§ 5.5.1, § 5.5.2, and § 5.5.3).

4. We integrate Auto-SAC into TorchTitan, enabling support for FSDP, TP, CP, and `torch.compile` models. This facilitates end-to-end deployment and benchmarking of memory-aware training policies.

5. We evaluate Auto-SAC on LLMs and multimodal models including LLaMA and Stable Diffusion. Auto-SAC reduces recomputation overhead by up to 90% over naïve checkpointing and matches ILP-level performance with heuristics that are orders of magnitude faster (§ 5.6).

These results position Auto-SAC as both a demonstration of LegoAI 's extensibility and a state-of-the-art method for memory-constrained training optimization.

1.6.4 Summary

Through its core subsystems, TorchTitan for scalable synthesis and execution, and TorchSim for predictive simulation, LegoAI enables the efficient identification and deployment of high-performance training configurations across diverse models, hardware platforms, and parallelism strategies. By integrating Auto-SAC as a proof of concept, LegoAI further demonstrates its extensibility in driving advanced memory-aware optimizations. Together, these components establish LegoAI as a comprehensive solution for automating the design and scaling of large-scale AI training workloads.

1.7 Thesis Organization

This thesis is organized into five chapters. Chapter 2 introduces foundational concepts in distributed training, covering multi-dimensional parallelism, GPU execution semantics, and activation checkpointing. Chapter 3 presents TorchTitan, a unified and modular training framework that composes 4D parallelism, integrates compiler and hardware-aware optimizations, and supports scalable, production-grade deployment. Chapter 4 introduces TorchSim, a high-fidelity simulator that predicts runtime and memory usage by modeling operator-level execution, communication, and tensor liveness without requiring GPU execution. Chapter 5 demonstrates the extensibility of LegoAI through Auto-SAC, a system that uses TorchSim 's predictions to generate optimized activation checkpointing policies under memory constraints, enabling precise and efficient training at scale. Finally, Chapter 6 summarizes the thesis contributions and outlines future research directions in automated distributed training, simulation-based optimization, and large-scale AI systems.

2

Preliminaries

This chapter presents the foundational concepts necessary for understanding distributed training algorithms. We begin with single-GPU training and incrementally build toward multi-dimensional parallelism, examining how various parallelization strategies are composed and analyzing their trade-offs in runtime, memory usage, and communication overhead in large-scale model training. In addition, we uncover the execution semantics of modern deep learning frameworks and provide a detailed understanding of the intricate multi-stream GPU execution model used to maximize hardware utilization and performance. Finally, we explain activation checkpointing, a widely used algorithm for navigating compute memory trade-offs.

2.1 Single-GPU Model Training

At its core, a deep learning model is a composition of differentiable functions, called operators, designed to model data distributions for prediction or generation tasks. A single training iteration, also called as a *train step*, consists of three phases: a forward pass that generates the outputs for all operators, a backward pass that generates the gradients, and a optimizer step that modifies the parameters based on the gradients.

We depict a complete single-GPU training iteration for a Multilayer Perceptron (MLP) model in Figure 2.1.

During the forward pass, the input data propagates through the model to generate predictions. Given an input x, it is first multiplied by the parameters of the first layer (p_1), passed through the GeLU function, and then multiplied by the second layer's parameters (p_2) to produce the output z. The loss is computed by comparing the predicted output with the target.

In the backward pass, gradients of the loss with respect to model parameters are computed using backpropagation. Starting from the derivative of the loss, $\frac{\partial \mathcal{L}}{\partial z}$, the chain rule is applied to compute gradients for each layer's parameters (g_1 for p_1 and g_2 for p_2). Throughout this process, parameters

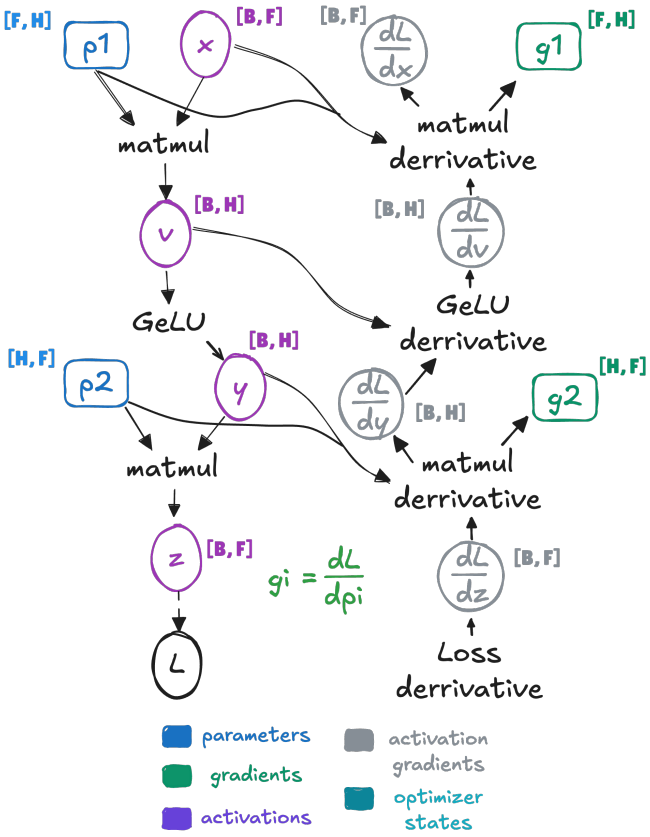
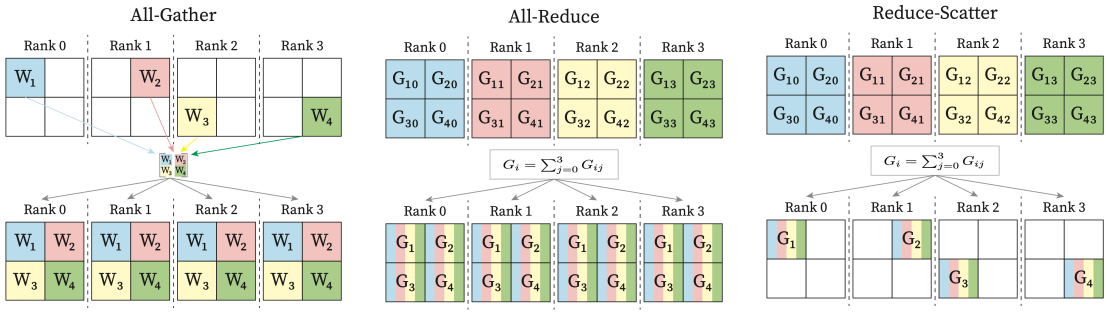


Figure 2.1: Parameters, gradients, optimizer states and activations retained in memory during single GPU training.

and intermediate activations must be retained for gradient computation. However, while parameter gradients are necessary for updates, temporary gradients with respect to activations can be freed immediately after use to optimize memory consumption.

An optimizer step typically follows the backward pass (optional in the case of gradient accumulation), updating parameters using gradients and a specified *learning_rate* while maintaining per-parameter optimizer states (e.g., momentum). Gradients are usually freed after the optimizer step.

2.2 Communication Collectives



(a) All-Gather. Each GPU starts with a shard of data (e.g., W_i) and ends with the full concatenated data across all ranks.

(b) All-Reduce. Each GPU contributes partial values (e.g., G_{ij}) to a global reduction. All ranks receive the full reduced result.

(c) Reduce-Scatter. Each GPU sends data to a global reduction and receives a distinct shard of the reduced output.

Figure 2.2: Visual illustration of core collective communication primitives used in distributed training Kempner Institute (2025). These operations are fundamental to model parallelism and efficient synchronization across GPUs.

Efficient distributed training relies on a set of fundamental communication collectives to synchronize data, parameters, and gradients across GPUs. The most commonly used primitives are **All-Gather**, **All-Reduce**, and **Reduce-Scatter** as illustrated in Figure 2.2*. Each serves a distinct role in managing the movement and aggregation of data in parallel training workflows:

*Figures sourced from Harvard Kempner Institute's Computing Handbook (Kempner Institute (2025))

- **All-Gather**: In an all-gather operation, each GPU begins with a disjoint partition of a tensor and ends up with the concatenated result of all partitions. For example, if each GPU holds a weight shard W_i, the all-gather operation ensures that every GPU receives the full tensor $[W_1, W_2, W_3, W_4]$. This is typically used to reconstruct sharded activations or parameters before computation.

- **All-Reduce**: In an all-reduce operation, each GPU contributes its local data (e.g., partial gradients) to a reduction (such as summation), and all GPUs receive the full reduced result. Mathematically, for a local shard G_{ij} on GPU j, the result $G_i = \sum_j G_{ij}$ is computed and broadcasted to all GPUs. This is commonly used to synchronize parameter gradients in data-parallel training.

- **Reduce-Scatter**: In a reduce-scatter operation, all GPUs contribute data to a global reduction, but each GPU receives only a portion of the reduced result. Continuing the gradient example, after computing $G_i = \sum_j G_{ij}$, each GPU receives only one shard G_i. This is especially useful when full replication is unnecessary, such as in sharded gradient aggregation.

These collectives are essential to building scalable and memory-efficient distributed training systems. They underpin the parallel semantics in data parallelism, tensor parallelism, and pipeline parallelism by enabling consistent synchronization of parameters, gradients, and intermediate activations.

2.3 Distributed Model Training

As models and datasets scale, distributed training techniques such as Distributed/Fully Sharded Data Parallel (DDP/FSDP) (Li et al., 2020; Rajbhandari et al., 2020; Zhao et al., 2023), Tensor Parallel (TP) (Shoeybi et al., 2019; Narayanan et al., 2021; Wang et al., 2022; PyTorch Team, 2024b),

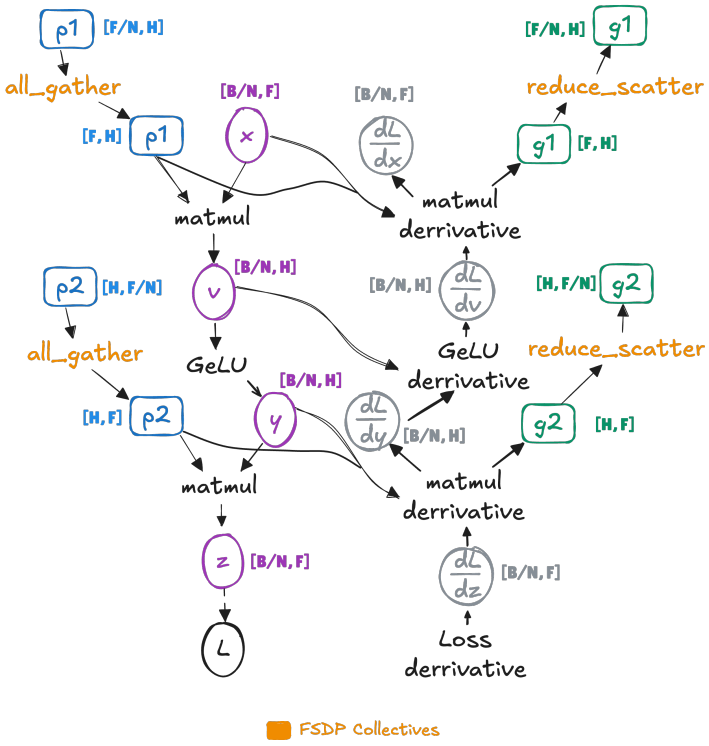


Figure 2.3: FSDP shards the parameters, gradients and optimizer states across multiple GPUs. It reconstructs the parameters dynamically and averages and redistributes the gradients dynamically using the *all_gather* and *reduce_scatter* collective operations respectively.

Context Parallel (CP) (Liu et al., 2023; NVIDIA, 2023; PyTorch Team, 2025), and Pipeline Parallel (PP) (Huang et al., 2019a; Narayanan et al., 2019, 2021; Lamy-Poirier, 2023; Qi et al., 2024; Tang et al., 2024a; PyTorch Team, 2024d; DeepSeek-AI, 2025) are employed to efficiently distribute workloads across multiple GPUs. These methods partition the model's data, parameters, gradients, optimizer states, and activations, reducing the memory required on any single GPU and parallelizing computation.

2.3.1 Fully Sharded Data Parallel (FSDP)

We modify the MLP example in Figure 2.1 to illustrate FSDP in Figure 2.3.

p_1 has shape $[F, H]$, p_2 has shape $[H, F]$, and the input x has shape $[B, F]$, with F as the feature dimension, H as the hidden dimension, and B as the batch size. Using FSDP as a case study, parameters p_1 and p_2 are partitioned along the feature dimension and distributed across N GPUs, while the input x is split along the batch dimension B, as illustrated in Figure 2.3. This reduces the memory footprint for parameters, gradients, and optimizer states on each GPU by a factor of N.

During the forward pass, full parameters are temporarily reconstructed (unsharded) using an *all_gather* operation before computation. Once complete, they are redistributed (resharded) to minimize memory usage. The backward pass follows a similar process, where unsharded gradients are computed and then averaged and sharded across GPUs using a *reduce_scatter* operation. This approach ensures that full parameters and gradients only occupy memory when necessary, significantly optimizing memory usage during training.

2.3.2 Tensor Parallel (TP) for MLP

For Tensor Parallel (TP), the sharding is done as follows: The parameters p_1 and p_2 are statically partitioned across M GPUs along the hidden dimension: p_1 is split column-wise and p_2 is split row-wise as shown in Figure 2.4. This also reduces memory usage for parameters, gradients, and optimizer states by a factor of M. Unlike FSDP, the input x is sharded along the feature dimension F. Before each forward pass, the input is reconstructed using the *all_gather* operation, and after the first linear and activation (GeLU) operations, the intermediate activations remain sharded along H. The second linear operation is applied on these sharded activations, producing a partial result, which is then aggregated and sharded back using a *reduce_scatter* operation along the feature dimension across M devices. During the backward pass, a similar process occurs. This allows TP to save memory not only for the parameters and gradients, but also for the activations throughout the computation.

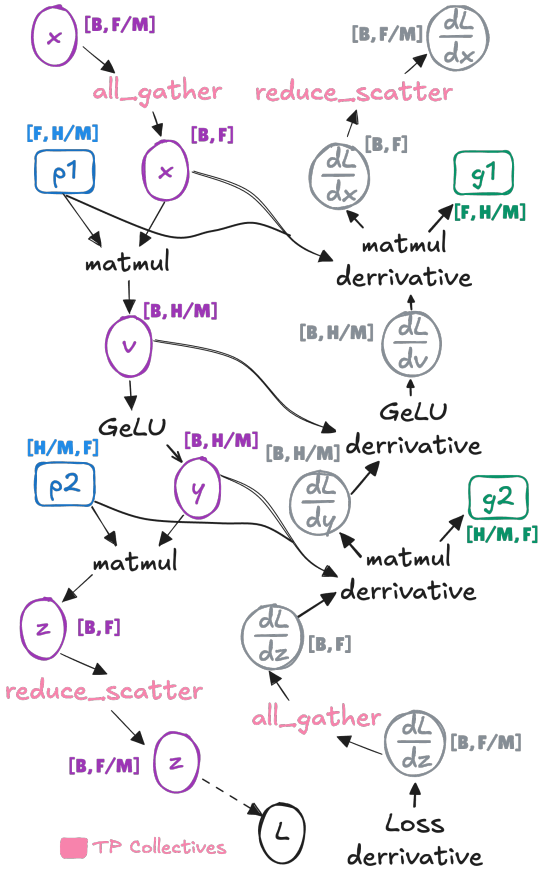


Figure 2.4: Tensor Parallelism (TP) partitions the model parameters across GPUs along the hidden dimension: p_1 is split column-wise and p_2 row-wise. The input x is sharded along the feature dimension and reconstructed using *all_gather* before the forward pass. Intermediate activations remain sharded, and the final output is redistributed using *reduce_scatter*. TP reduces memory usage for parameters, gradients, and activations, while avoiding full *all_reduce* overheads.

2.3.3 Tensor Parallel (TP) for Multi-Head Attention

In *Tensor Parallelism* (TP), the parallelization is applied across the multiple heads of a self-attention mechanism. Consider a single multi-head attention layer where the input tensor x has shape $[B, S, D]$, with B as the batch size, S as the sequence length, and D as the hidden dimension. Suppose the model uses H attention heads, each with dimensionality $D_H = D/H$. TP distributes the computation of these H heads across M GPUs.

Each attention head uses three learned projection matrices W_Q, W_K, W_V of shape $[D, D_H]$, and an output projection W_O of shape $[D, D]$. In TP, the H heads are divided evenly across devices, so each GPU computes H/M heads. Correspondingly, the projection weights are sharded: W_Q, W_K, W_V are partitioned along the output dimension with shape $[D, D/M]$, and W_O is sharded as $[D/M, D]$.

The input $x \in [B, S, D]$ is replicated across all M devices. Each device computes its local projections:

$$Q_i = x \cdot W_Q^{(i)}, \quad K_i = x \cdot W_K^{(i)}, \quad V_i = x \cdot W_V^{(i)} \quad \text{where } Q_i, K_i, V_i \in [B, S, D/M]$$

Each device then computes scaled dot-product attention over its assigned heads:

$$\text{Attn}_i = \text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{D_H}}\right) V_i \in [B, S, D/M]$$

These partial outputs are aggregated using `all-reduce` across GPUs to produce the complete attention output $A \in [B, S, D]$, which is then passed through the sharded output projection. Each device computes a slice of $A \cdot W_O^{(i)}$, and the final result is assembled via `reduce-scatter`.

In the backward pass, gradients are computed locally per head and aggregated using collective

operations. This approach reduces memory usage for projections and intermediate tensors by a factor of M, while enabling parallel computation across attention heads.

2.3.4 Context Parallel (CP) for Multi-Head Attention

In *Context Parallelism* (CP), the parallelism is applied across the sequence dimension. Given input $x \in [B, S, D]$, CP partitions the sequence length S across N GPUs. Each device receives $x_i \in [B, S/N, D]$, corresponding to a different slice of the input sequence. This strategy is especially beneficial for long-context models.

Each device computes local projections:

$$Q_i = x_i \cdot W_Q, \quad K_i = x_i \cdot W_K, \quad V_i = x_i \cdot W_V \quad \text{where } Q_i, K_i, V_i \in [B, S/N, D]$$

To compute global attention, an `all-gather` is used to collect the full K and V across devices, forming $K \in [B, S, D]$ and $V \in [B, S, D]$. Each device computes attention for its local Q_i:

$$\text{Attn}_i = \text{Softmax}\left(\frac{Q_i K^T}{\sqrt{D_H}}\right) V \in [B, S/N, D]$$

The local attention output is passed through the shared output projection $W_O \in [D, D]$. Optionally, a `reduce-scatter` operation is used to partition the final output among devices if required by the downstream layers.

During backpropagation, gradients with respect to K and V are distributed back via `reduce-scatter`, and gradients with respect to Q are used locally. Projection gradients are aggregated with `all-reduce`.

CP reduces per-device memory usage for attention maps and allows distributed training over long sequences. It is particularly well-suited for autoregressive and retrieval-based models where

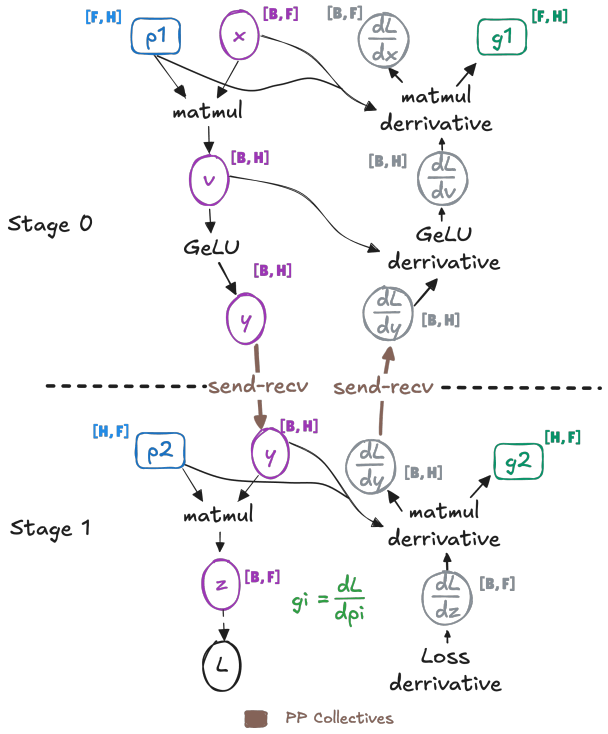


Figure 2.5: Pipeline Parallelism (PP) partitions the model into sequential stages, each assigned to a different GPU or GPU group. Intermediate activations and gradients are transferred between stages using *send/recv* collectives. Each stage executes forward and backward computations independently, enabling concurrent microbatch execution and efficient scaling across deep models.

context length dominates computational cost.

2.3.5 Pipeline Parallel (PP)

In Pipeline Parallelism (PP), the MLP model is divided across two pipeline stages as shown in Figure 2.5. Stage 0 holds the first linear layer $p_1 \in [F, H]$, and Stage 1 holds the second linear layer $p_2 \in [H, F]$. The input tensor $x \in [B, F]$ is fed into Stage 0, where it is multiplied with p_1, passed through a GeLU activation, and produces an intermediate activation $y \in [B, H]$. This activation is transmitted to Stage 1 using `send/recv` collectives. In Stage 1, the second linear transformation $z = y \cdot p_2 \in [B, F]$ is computed. During the backward pass, the gradient of the loss with respect to

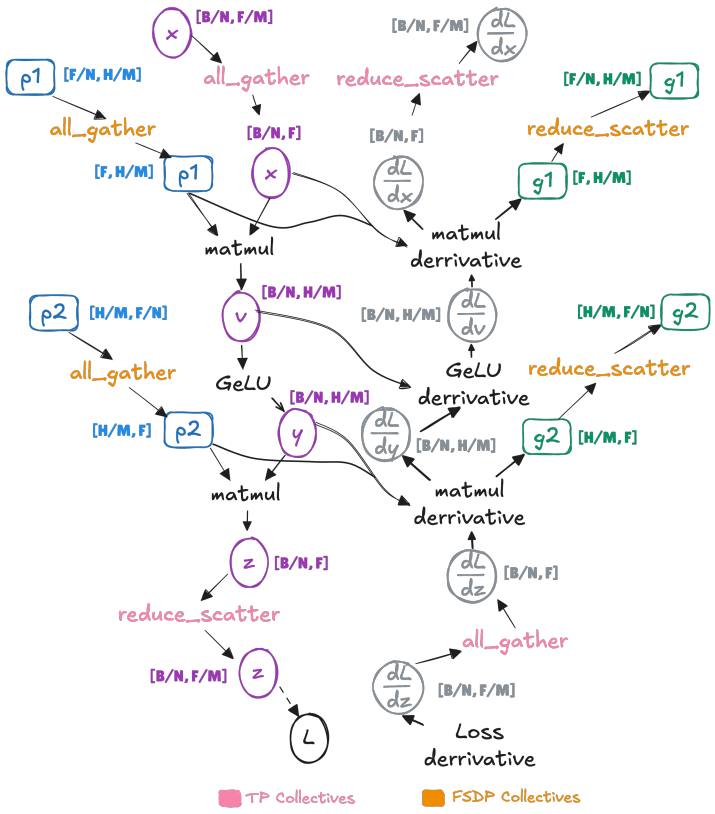


Figure 2.6: 2D Parallelism combines Fully Sharded Data Parallelism (FSDP) and Tensor Parallelism (TP) to reduce memory and distribute computation. Parameters and optimizer states are sharded across FSDP groups, while TP partitions the model layers along the hidden dimension. Inputs are sharded across the feature dimension and reconstructed using *all_gather*, while outputs are redistributed using *reduce_scatter*. Gradients are locally reduced via FSDP and TP collectives.

z, $\frac{\partial \mathcal{L}}{\partial z}$, is used to compute local gradients for p_2, and the gradient with respect to y, $\frac{\partial \mathcal{L}}{\partial y}$, is sent back to Stage 0. There, it is used to compute gradients for p_1. This setup allows forward and backward passes to proceed in a pipelined fashion, improving throughput and reducing per-device memory usage.

2.3.6 Multi-dimensional Parallelism

2D Parallelism applies both Fully Sharded Data Parallelism (FSDP) and Tensor Parallelism (TP) to the MLP example as depicted in Figure 2.6. Let $p_1 \in [F, H]$ and $p_2 \in [H, F]$ be the parameters for the two linear layers. FSDP shards parameters, gradients, and optimizer states across N GPUs along the batch dimension, such that each GPU holds a slice of $[F/N, H]$ or $[H, F/N]$. Within each FSDP group, TP further shards each tensor across M GPUs along the hidden dimension. Specifically, p_1 is split column-wise into $[F, H/M]$, and p_2 is split row-wise into $[H/M, F]$. The input $x \in [B/N, F]$ is sharded along the feature dimension across the TP group and reconstructed via `all_gather` before the forward pass. The first linear transformation and GeLU activation produce activations $v \in [B/N, H/M]$, which are then passed through the second linear layer to compute $z \in [B/N, F]$. The result is redistributed using `reduce_scatter`. This combination reduces memory usage across both model weights and activations while maintaining efficient compute execution.

In the 3D Parallelism setup, the MLP model is distributed across pipeline stages, tensor parallel groups, and FSDP groups as shown in Figure 2.7. Stage 0 holds the first linear layer $p_1 \in [F, H]$ and Stage 1 holds the second layer $p_2 \in [H, F]$. Parameters are first sharded using FSDP across N GPUs along the batch dimension. Within each FSDP group, TP is used to split the tensors across M GPUs along the hidden dimension: p_1 becomes $[F, H/M]$ and p_2 becomes $[H/M, F]$. The input $x \in [B/N, F/M]$ is reconstructed using `all_gather`, then multiplied with p_1 and passed through a GeLU activation to produce $y \in [B/N, H/M]$. This activation is passed to Stage 1 using `send/recv`. In Stage 1, the output $z = y \cdot p_2 \in [B/N, F/M]$ is computed and redistributed using `reduce_scatter`. During backpropagation, gradients are computed locally and synchronized via FSDP and TP collectives, while pipeline stages exchange gradients via `send/recv`. This configuration enables scalable, memory-efficient training across thousands of GPUs.

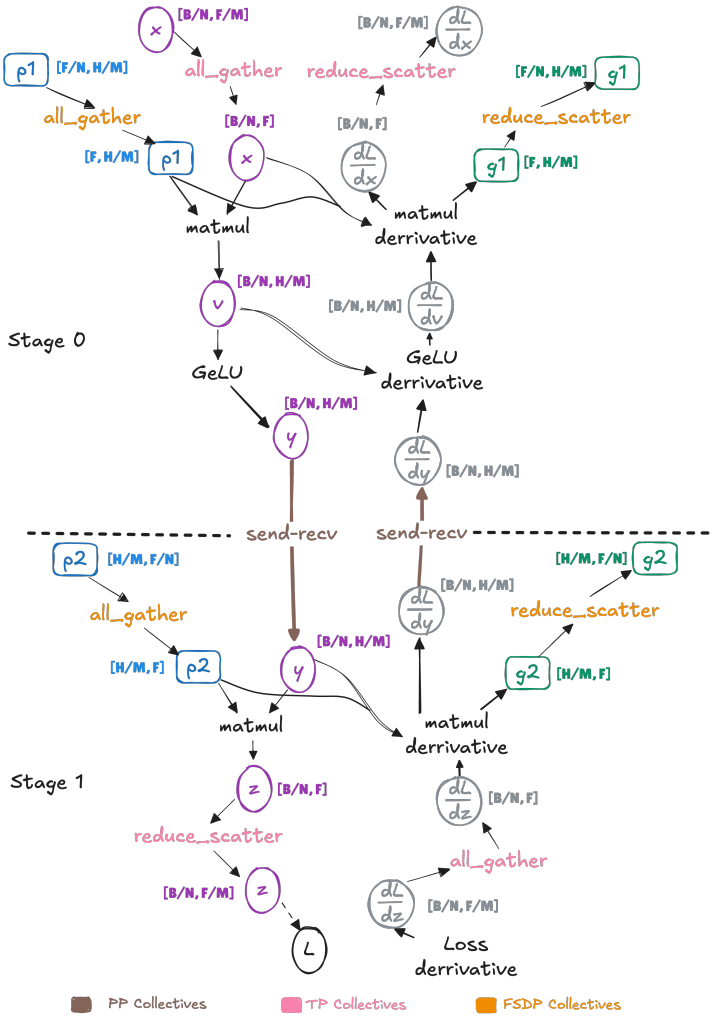


Figure 2.7: 3D Parallelism integrates Pipeline Parallelism (PP), Tensor Parallelism (TP), and Fully Sharded Data Parallelism (FSDP) to enable scalable and memory-efficient training across thousands of GPUs. Each pipeline stage applies TP to shard computation along the hidden dimension, and FSDP to shard parameters within TP subgroups. *send/recv* operations are used for inter-stage communication, while *all_gather* and *reduce_scatter* are used to manage intra-stage computation and memory efficiency.

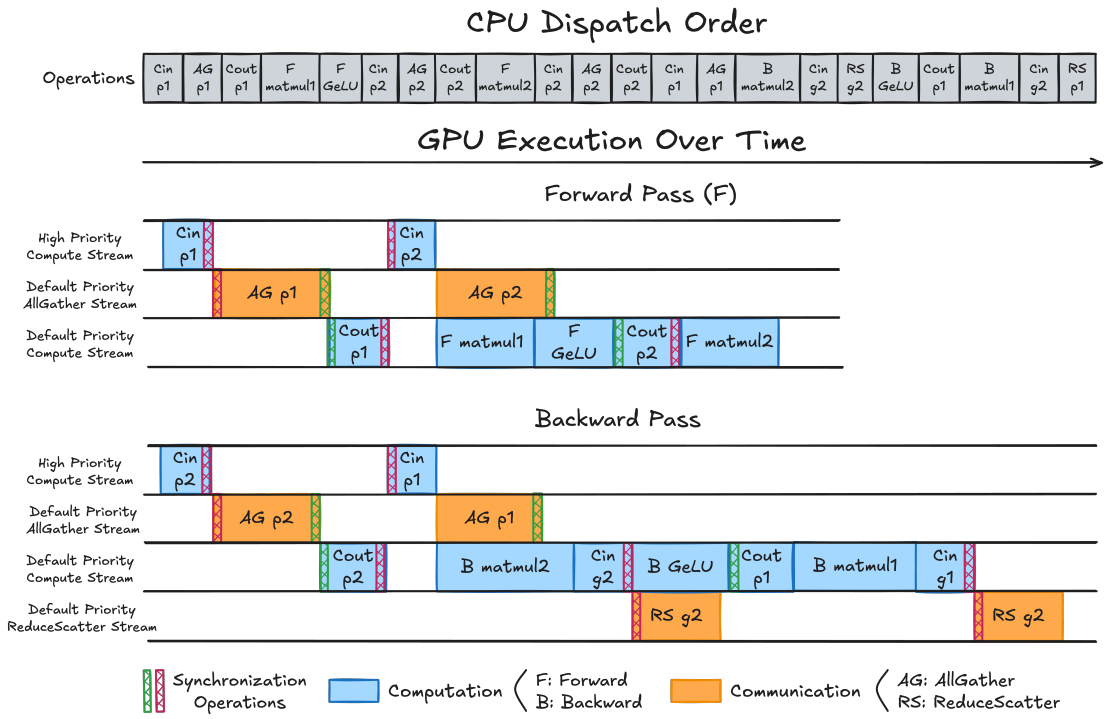


Figure 2.8: The operators (blue) and communication collectives (orange) dispatched by the CPU across multiple GPU streams for the example in 2.3. Although the CPU issues operations sequentially, streams can overlap in execution, e.g., as shown by *Cin_p2* and *AG_p2* in the forward pass, until forced to synchronize, e.g. by *all_gather* (AG) and *reduce_scatter* (RS), as denoted by the red and green vertical lines.

2.4 Distributed Training Execution Semantics

To understand the actual memory usage and execution time, it is crucial to closely inspect the execution and memory behavior of the underlying training system. Figure 2.8 illustrates the execution flow of a model training iteration in PyTorch Paszke et al. (2019) using a GPU GPU.

Framework Fundamentals. Frameworks such as PyTorch provide *modules*, which serve as containers for commonly used deep learning components such as linear, attention, convolution layers, etc. Tensors are the fundamental containers for data such as parameters, gradients, or activations. Each module consists of a well-defined set of operators provided by PyTorch, such as matrix mul-

tiplication, dot product attention, etc., that operate on the tensors. Executing an operator involves launching one or more GPU kernel functions, which are highly optimized parallel implementations of the operator. Thus, performing a forward and backward pass on a module translates to executing a sequence of tensor operations on the GPU.

Goal. Efficient distributed training depends on minimizing memory usage and overlapping communication with computation. This is accomplished by retaining only essential data in memory and performing communication asynchronously with independent compute operations. The *stream* execution model, central to all modern GPUs, enables this high-performance parallel execution.

Execution. A GPU consists of multiple resources, including compute cores, communication engines, and DMA engines. Each stream represents a queue of operations that execute sequentially within the stream but may run concurrently with operations from other streams if they utilize independent resources or do not fully occupy a shared resource. While developers can create multiple streams, operations across different streams may execute in parallel or out of order.

Synchronization. To coordinate execution across streams, *synchronization primitives* such as stream/event waits and barriers are used. If an operation in a high-priority stream depends on data from another stream, explicit synchronization (e.g., via events) is required to enforce the correct execution order. This ensures that while streams enable parallel execution, actual overlap occurs only when operations are assigned to different hardware resources.

Memory. Memory allocation in GPUs follows stream semantics, meaning memory allocated within a stream is returned to the same stream after use. In PyTorch, memory ownership remains unchanged throughout execution unless explicitly cleared or managed by a custom memory allocator. The CPU sequentially dispatches operators for execution on the GPU, assigning them to different streams as determined by the scheduling algorithm. Memory allocation and deallocation are handled by the memory manager on the CPU side. Since each stream owns the memory allocated within it, the CPU can logically free memory assigned to an operator before execution using refer-

ence counting. Because operations within a stream execute sequentially, reassigning freed memory to a subsequent operation after its last use ensures correctness.

Example. Figure 2.8 illustrates the CPU dispatch order and the corresponding multi-stream GPU execution for PyTorch's per-parameter FSDP algorithm, applied to the previous MLP example. The communication collectives (*all_gather* and *reduce_scatter*) are strategically enqueued in separate streams to maximize overlap with independent compute operations whenever possible.

Collective communication operations require *copy-in* and *copy-out* operations for efficiency. Copy-in operations must complete before launching *all_gather*, making them a blocking step. To minimize delays, these operations are enqueued in a high-priority stream. Once *all_gather* finishes, copy-out operations can begin, ensuring that data is available for subsequent computation. Similarly, *reduce_scatter* operations can only start once gradient copy-in operations complete. To enforce these dependencies, *synchronization events* are inserted to maintain correct execution order across streams.

Memory management in this workflow is tightly coupled with the CPU dispatch order. When memory is allocated for an operation, such as the *all_gather* for p_1 during the forward pass, it is logically released as soon as the CPU processes the corresponding copy-out operation, even if the actual operation is still running. This approach allows the memory to be safely reused by the subsequent *all_gather* for p_2, as it is issued on the same stream and is guaranteed to execute only after the copy-out operation for p_1 has completed.

2.5 Distributed Model Training Paradigms

Distributed training strategies such as Fully Sharded Data Parallel (FSDP), Tensor Parallelism (TP), and Context Parallelism (CP) follow the *Single Program, Multiple Data* (SPMD) paradigm. In this model, each device runs the same program but operates on a different portion of the data or model.

For example, in FSDP, devices process different microbatches; in CP, they handle different segments of the sequence; and in TP, they work on separate but identically shaped model weight shards (e.g., attention heads or matrix blocks). Because all devices execute the same code in parallel, it is sufficient to estimate memory and runtime for a single SPMD process to predict system-wide behavior.

In contrast, Pipeline Parallelism (PP) follows the *Multiple Program, Multiple Data* (MPMD) paradigm. The model is partitioned into sequential stages, with each stage mapped to a different device or group of devices, each processing a distinct subset of the model and data. When combining parallelism strategies, pipeline parallelism is typically applied first, and each pipeline stage executes its own internal SPMD workflow using FSDP, TP, or CP across the assigned devices. Because each pipeline stage follows the SPMD model internally, it is sufficient to estimate memory and runtime for one SPMD process per stage to predict system-wide memory usage. However, for accurate runtime estimation, interactions between SPMD processes across pipeline stages, such as communication delays and scheduling dependencies, must be carefully modeled.

2.6 Activation Checkpointing

To reduce memory usage during training, *activation checkpointing* (AC) selectively discards intermediate activations in the forward pass and recomputes them during backpropagation (Chen et al., 2016; Korthikanti et al., 2023; He & Yu, 2023; Purandare et al., 2023). This technique trades additional compute for reduced memory, enabling larger batch sizes and deeper models under fixed memory budgets.

2.6.1 Example: Transformer Layer

Consider a simplified Transformer block, composed of a self-attention sub-layer and a feed-forward network (FFN), interleaved with layer norms and residual connections. Let $x \in \mathbb{R}^{B \times S \times D}$ be the

input tensor.

$$
\begin{aligned}
y_1 &= \text{LayerNorm}(x) \\
y_2 &= \text{SelfAttention}(y_1) \\
y_3 &= x + y_2 \quad \text{(residual)} \\
y_4 &= \text{LayerNorm}(y_3) \\
y_5 &= \text{FFN}(y_4) \\
z &= y_3 + y_5 \quad \text{(residual)}
\end{aligned}
$$

In standard training, all intermediate activations y_1 through y_5 are stored, which incurs high memory overhead.

2.6.2 Full Activation Checkpointing

In full AC, the entire module is treated as a single checkpoint. Only the input x is retained; all intermediate activations are discarded and recomputed in the backward pass. This significantly reduces memory, but the full forward computation is repeated—often introducing high recomputation cost.

2.6.3 Selective Activation Checkpointing (SAC)

SAC generalizes AC by allowing more granular control over which activations are discarded. Rather than checkpointing entire modules, SAC operates at the operator level. For the Transformer example above, a SAC policy might:

- Discard y_1 and y_4 (LayerNorm outputs), which are cheap to recompute,

- Store y_2 (output of attention), which is expensive,
- Retain y_3 (residual), as it is reused.

This fine-grained control provides a better memory–compute trade-off than full AC, but it requires knowing the runtime and memory profile of every operator—an infeasible task for users to manage manually at scale.

3

TorchTitan: A Unified, Modular, Composable, and Scalable Distributed Training Framework

To automatically discover and deploy optimal training recipes, LegoAI must not only identify high-performance configurations but also synthesize scalable and efficient implementations for them. This requires a unified framework that is flexible, composable, and production-ready. To meet this need, we develop TorchTitan, an open-source, PyTorch-native training system designed to execute arbitrary distributed configurations at scale.

TorchTitan supports seamless composition of four-dimensional parallelism, including Fully Sharded Data Parallelism (FSDP), Tensor Parallelism (TP), Pipeline Parallelism (PP), and Context Parallelism (CP), within a unified and modular framework. It enables elastic scaling, efficient memory management, and integration of advanced hardware features such as Float8 training and SymmetricMemory. Designed for both experimentation and deployment, TorchTitan provides detailed logging, robust checkpointing, and debugging capabilities that support reproducibility and fault tolerance.

By scaling LLaMA 3.1 models to thousands of GPUs, achieving speedups of 65.08%, 12.59%, and 30% at 128, 256, and 512 GPUs respectively, and enabling long-context training on NVIDIA H100 clusters, TorchTitan demonstrates its effectiveness as both a systems research platform and a foundation for production-grade training. Through TorchTitan, LegoAI bridges the gap between algorithmic discovery and practical implementation, turning synthesized training recipes into high-performance, executable programs.

3.1 TorchTitan simple and modular end-to-end training pipeline

TorchTitan incorporates various parallelisms in a modular manner to enable easy, user-selectable combinations of multi-dimensional shardings. This composability enables the tackling of difficult scaling challenges by enhancing the ease of exploration for optimizing training efficiencies at scale.

The codebase of TorchTitan is organized purposefully to enable composability and extensi-

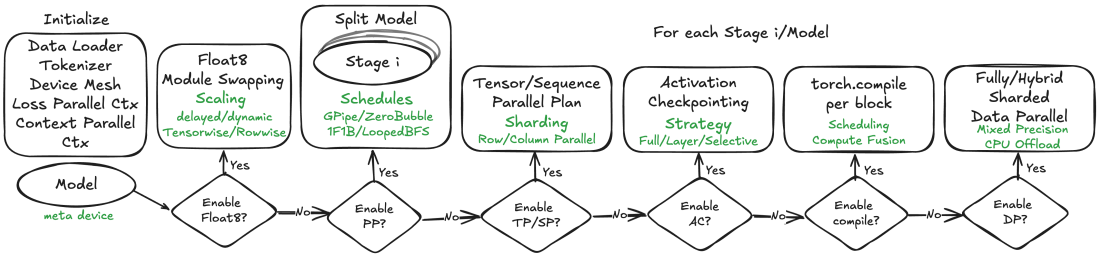


Figure 3.1: Composable and Modular TorchTitan initialization workflow.

bility. We intentionally keep three main components separate and as orthogonal as possible: (1) the model definition, which is parallelism-agnostic and designed for readability, (2) parallelism helpers, which apply parallelisms and training optimizations to a particular model, and (3) a generalized training loop. All these components are configurable via TOML files with command-line overrides, and it is easy to add new models and parallelism techniques on top of the existing codebase.

3.2 Composable N-D parallelism training

In this section, we will walk through the entire regime of scaling model training on large clusters, including meta device initialization and the core composable multi-dimensional parallelisms, to showcase how these techniques can be composed to train LLMs efficiently at increasing scale in TorchTitan. The corresponding code snippets in TorchTitan can be found in Section 3.8.1.

3.2.1 Large-scale model initialization using meta device

As LLMs grow exponentially, scaling challenges arise even before training begins, particularly in instantiating large models for sharding without exceeding CPU or GPU memory limits.

To address this, TorchTitan enables meta device initialization, where the model is first created on a *meta* device that stores only metadata, making initialization ultra-fast. The model is then sharded into Distributed Tensors (DTensors), with the local shard of each parameter residing on the

meta device. Finally, parameter initialization is performed using user-defined functions, ensuring correct DTensor sharding layouts and proper RNG seed usage.

3.2.2 Fully Sharded Data Parallel

The original Fully Sharded Data Parallel (FSDP) (Zhao et al., 2023) is an effective implementation of ZeRO that offers large model training capability in PyTorch. However, the original implementation (FSDP1) in PyTorch suffers from various limitations due to its FlatParameter implementation.

Given these limitations, TorchTitan integrates a new version of Fully Sharded Data Parallel (FSDP2), which uses the per-parameter Distributed Tensor sharding representation and thus provides better composability with model parallelism techniques and other features that require the manipulation of individual parameters.

TorchTitan integrates and leverages FSDP2 as it's default 1D parallelism, benefiting from the improved memory management (often 7 percent lower per GPU memory requirement vs FSDP1) and the slight performance gains (average of 1.5 percent gain vs FSDP1). More details on FSDP2 and usage example are shown in Section 3.8.2. TorchTitan makes it simple to run with FSDP2 by embedding appropriate defaults, including auto-sharding with your world size automatically.

For scaling to even larger world sizes, TorchTitan also integrates Hybrid Sharded Data Parallel (HSDP) which extends FSDP2 by creating 2D DeviceMesh with replica groups. Details are shown in Section 3.8.3

3.2.3 Tensor Parallel

Tensor Parallel (TP) (Narayanan et al., 2021), together with Sequence Parallel (SP) (Korthikanti et al., 2023), is a key model parallelism technique to enable large model training at scale.

TP is implemented in TorchTitan using the PyTorch's `RowwiseParallel` and `ColwiseParallel`

APIs, where the model parameters are partitioned to DTensors and perform sharded computation with it (Figure 3.4). By leveraging DTensor, the TP implementation does not need to touch the model code, which allows faster enablement on different models and provides better composability with other features mentioned in this paper.

Tensor and Sequence Parallel (TP/SP) While TP partitions the most computationally demanding aspects, Sequence Parallel (SP) performs a sharded computation for the normalization or dropout layers on the sequence dimension, which otherwise generate large replicated activation tensors, and thus can be challenging to memory constraints per GPU. See Section 3.8.4 for more details, illustrations, and usage for both TP and FSDP + TP.

Due to the synergistic relationship between TP and SP, TorchTitan natively bundles these two together, and they are jointly controlled by the TP degree setting.

Loss Parallel When computing the loss function, model outputs are typically large, especially with TP/SP, where they are sharded across the vocabulary dimension. Naively computing cross-entropy loss requires gathering all shards, leading to high memory usage.

Loss Parallel enables efficient loss computation without fully gathering model outputs, significantly reducing memory consumption and improving training speed by minimizing communication overhead and enabling parallel sharded computation. Due to these advantages, TorchTitan implements Loss Parallel by default.

3.2.4 Pipeline Parallel

For large-scale pretraining, TorchTitan employs Pipeline Parallelism (PP), which minimizes communication overhead by leveraging P2P communications. PP divides the model into S stages, each running on a separate group of devices. Typically, each stage represents a model layer or a group of

adjacent layers, but can include partial layers. During the forward pass, each stage receives input activations (except stage 0), computes locally, and sends output activations (except stage $S-1$). The last stage computes the loss and initiates the backward pass, sending gradients in reverse order. To improve efficiency, the input batch is split into microbatches, and the pipeline schedule overlaps computation and communication across microbatches. TorchTitan supports various pipeline schedules (Narayanan et al., 2019; Huang et al., 2019b; Narayanan et al., 2021; Tang et al., 2024b). Recently, TorchTitan added support for new schedules including ZeroBubble and 'Flexible-Interleaved-1F1B', making use of pipeline IR to quickly express new schedules as a list of compute actions and rely on compiler passes to insert and optimize communication actions PyTorch Team 2024d.

The PP training loop differs from standard training by creating pipeline stages and executing schedules instead of directly invoking `model.forward()`. Since loss is computed per microbatch, TorchTitan introduces a shared `loss_fn` to unify pipeline and non-pipeline workflows, reducing code divergence.

`torch.distributed.pipelining` also simplifies interactions with data parallelism, ensuring that reductions occur only after the final microbatch and handling shard/unshard operations (e.g., with ZeRO-3), as well as applying gradient scaling transparently within the pipeline schedule executor. For more details on TorchTitan's implementation of PP, see Section 3.8.5.

3.2.5 Context Parallelism

TorchTitan has been extended to incorporate Context Parallelism (CP) (Liu et al., 2023; Liu & Abbeel, 2024; NVIDIA, 2023), enabling 4D parallelism by adding CP as an additional dimension to existing DP, TP, and PP. CP scales model training by splitting the sequence dimension across GPUs, significantly increasing the maximum trainable context length without causing out-of-memory (OOM) errors. For example, on Llama 3.1 8B with 8 H100 GPUs, using CP enabled

training at context lengths up to 262,144 tokens, achieving minor MFU degradation as CP degree increases (PyTorch Team, 2025). For more details on CP integration please refer to Section 3.8.6.

3.3 Optimizing training efficiencies

3.3.1 Navigating compute-memory trade-offs using activation checkpointing

Activation checkpointing (AC) (Chen et al., 2016; He & Yu, 2023; Purandare et al., 2023) and selective activation checkpointing (SAC) (Korthikanti et al., 2023) are standard training techniques to reduce peak GPU memory usage, by trading activation recomputation during the backward pass for memory savings. It is often needed even after applying multi-dimensional parallelisms.

TorchTitan offers flexible AC and SAC options utilizing `torch.utils.checkpoint`, applied at the `TransformerBlock` level. The AC strategies include "full" AC, op-level SAC, and layer-level SAC.

Within a `TransformerBlock`, full AC works by recomputing all activation tensors needed during the backward pass, whereas op-level SAC saves the results from computation-intensive PyTorch operations and only recomputes others. Layer-level SAC works in similar fashion as full AC, but the wrapping is applied to every x `TransformerBlock` (where x is specified by the user) to implement configurable trade-offs between memory and recompute. (Details are in Section 3.8.7.)

3.3.2 Regional compilation to exploit `torch.compile` optimizations

`torch.compile` was released in PyTorch 2 (Ansel et al., 2024b) with TorchDynamo as the frontend to extract PyTorch operations into an FX graph, and TorchInductor as the backend to compile the FX graph into fused Triton code to improve the performance.

In TorchTitan, we use regional compilation, which applies `torch.compile` to each individual `TransformerBlock` in the Transformer model. This has two main benefits: (1) we get a full graph (without graph breaks) for each region, compatible with FSDP2 and TP (and more generally `torch.Tensor` subclasses such as DTensor) and other PyTorch distributed training techniques; (2) since the Llama model stacks identical `TransformerBlock` layers one after another, `torch.compile` can identify the same structure is being repeatedly compiled and only compile once, thus greatly reducing compilation time.

`torch.compile` brings efficiency in both throughput and memory (see Section 3.5.2) via computation fusions and computation-communication reordering, in a model-agnostic way with a simple user interface. Below we further elaborate how `torch.compile` composability helps TorchTitan unlock hardware-optimized performance gain with simple user interface, with the integration of advanced features such as Asynchronous TP and Float8.

3.3.3 Asynchronous Tensor Parallel to maximally overlap communication

By default, TP incurs blocking communications before/after the sharded computations, causing computation resources to not be effectively utilized. Asynchronous TP (AsyncTP) (Wang et al., 2022) achieves computation-communication overlap by fractionalizing the TP matrix multiplications within attention and feed-forward modules into smaller chunks, and overlapping communication collectives in between each section. The overlap is achieved by a micro-pipelining optimization, where results are being communicated at the same time that the other chunks of the matmul are being computed.

PyTorch AsyncTP is based on a `SymmetricMemory` abstraction, which creates intra-node buffers to write faster communication collectives. This is done by allocating a shared memory buffer on each GPU in order to provide direct P2P access (PyTorch Team, 2024b).

With TorchTitan's integration of `torch.compile`, AsyncTP can be easily configured in TorchTi-

TAN to achieve meaningful end-to-end speedups (see Section 3.5.2 for details) on newer hardware (H100 or newer GPUs with NVSwitch within a node). Usage details are in Section 3.8.8

3.3.4 Boosting throughput with mixed precision training and Float8 support

Mixed precision training (Micikevicius et al., 2018) provides both memory and computational savings while ensuring training stability. FSDP2 has built-in support for mixed precision training with basic `torch.dtype`. This covers the popular usage of performing FSDP all-gather and computation in a low precision (e.g. `torch.bfloat16`), and perform lossless FSDP reduce-scatter (gradient) in high precision (e.g. `torch.float32`) for better numerical results. See Section 3.8.9 for usage details.

TorchTitan also supports more advanced mixed precision training with Float8, a derived data type, applied selectively to linear layers (available on newer hardware like NVIDIA H100), achieving substantial performance gains while ensuring training stability (reported in Section 3.5.2). The Float8 feature from `torchao.float8` supports multiple per-tensor scaling strategies, including dynamic, delayed, and static (see Micikevicius et al. (2022); PyTorch Community (2023a), Section 4.3 for details), while being composable with other key PyTorch-native systems such as autograd, `torch.compile`, FSDP2 and TP (with Float8 all-gather capability) (PyTorch Team, 2024a).

3.4 Production ready training

To enable production-grade training, TorchTitan offers seamless integration with key features out of the box. These include (1) efficient checkpointing using PyTorch Distributed Checkpointing (DCP), and (2) debugging stuck or crashed jobs through integration with Flight Recorder.

3.4.1 Scalable and efficient Distributed Checkpointing

Checkpoint save/load are crucial in training large language models for two reasons: they facilitate model reuse in applications like inference and evaluation, and they provide a recovery mechanism in case of failures. An optimal checkpointing workflow should ensure ease of reuse across different parallelisms and maintain high performance without slowing down training. There are two typical checkpointing methods. The first aggregates the state (model parameters and optimizer states) into an unsharded version that is parallelism-agnostic, facilitating easy reuse but requiring expensive communication. The second method has each trainer save its local sharded state, which speeds up the process but complicates reuse due to embedded parallelism information.

DCP addresses these challenges using DTensor, which encapsulates both global and local tensor information independently of parallelism. DCP converts this information into an internal format for storage. During loading, DCP matches the stored shards with the current DTensor-based model parameters and optimizer states, fetching the necessary shard from storage. TorchTitan effectively uses DCP to balance efficiency and usability. Furthermore, DCP enhances efficiency through asynchronous checkpointing by processing storage persistence in a separate thread, allowing this operation to overlap with subsequent training iterations. TorchTitan utilizes DCP's asynchronous checkpointing to reduce the checkpointing overhead by 5-15x compared to synchronous distributed checkpointing for the Llama 3.1 8B model (PyTorch Team, 2024c).

3.4.2 Flight Recorder to Debug Job Crashes

Debugging NCCL collective timeouts at large scales is challenging due to the asynchronous nature of communication kernels. PyTorch's Flight Recorder addresses this by logging the start, end, and enqueue times for all collective and p2p operations, along with metadata like process groups, source/destination ranks, tensor sizes, and stack traces.

This data is invaluable for diagnosing hangs in parallelism code. For PP, it can pinpoint the latest send or recv completed on the GPU, helping debug schedule bugs. For FSDP and TP, it identifies ranks that failed to call collectives, aiding in uncovering issues with PP scheduling or TP logic.

3.5 Experimentation

In this section, we demonstrate the effectiveness of elastic distributed training using TorchTitan, via experiments on Llama 3.1 8B, 70B, and 405B, from 1D parallelism to 4D parallelism, at the scale from 8 GPUs to 512 GPUs. We also share the knowledge and experience gained through TorchTitan experimentation. A walkthrough of the codebase on how we apply (up to) 4D parallelism can be found in Section 3.8.1.

3.5.1 Experimental setup

The experiments are conducted on NVIDIA H100 GPUs* with 95 GiB memory, where each host is equipped with 8 GPUs and NVSwitch. Two hosts form a rack connected to a TOR switch. A backend RDMA network connects the TOR switches. In TorchTitan we integrate a checkpointable data loader and provide built-in support for the C4 dataset (`en` variant), a colossal, cleaned version of Common Crawl's web crawl corpus (Raffel et al., 2020a). We use the same dataset for all experiments in this section. For the tokenizer, we use the official one (tiktoken) released together with Llama 3.1.

3.5.2 Performance

To showcase the elasticity and scalability of TorchTitan, we experiment on a wide range of GPU scales (from 8 to 512), as the underlying model size increases (8B, 70B, and 405B) with a varying

*The H100 GPUs used for the experiments are non-standard. They have HBM2e and are limited to a lower TDP. The actual peak TFLOPs should be between SXM and NVL, and we don't know the exact value.

number of parallelism dimensions (up to 4D). To demonstrate the effectiveness of the optimization techniques introduced in Section 3.3, we show how training throughput improves when adding each individual technique on appropriate baselines. In particular, when training on a higher dimensional parallelism with new features, the baseline is always updated to include all previous techniques.

We note that, throughout our experimentation, memory readings are stable across the whole training process[†], whereas throughput numbers (token per second, per GPU) are calculated and logged every 10 iterations, and always read at the (arbitrarily determined) 90th iteration. We do not report Model FLOPS Utilization (MFU) (Chowdhery et al., 2023) because when Float8 is enabled in TorchTitan, both BFLOAT16 Tensor Core and FP8 Tensor Core are involved in model training, but they have different peak FLOPS and the definition of MFU under such scenario is not well-defined. We note that the 1D Llama 3.1 8B model training on 8 or 128 H100 GPUs without Float8 achieves 33% to 42% MFU.

Our experiments serve multiple objectives:

- **Establish composability and modularity:** TorchTitan demonstrates seamless integration of various parallelisms and optimization techniques.
- **Showcase performance improvements:** Significant speed-ups are observed across parallelisms and optimizations.
- **Validate elastic scalability:** TorchTitan scales effectively with both the model size and the number of GPUs.
- **Ablation studies:** Detailed performance gains for individual techniques are presented.

In particular

[†]Different PP ranks can have different peak memory usages. We take the maximum across all GPUs.

- Table 3.1: Highlights improvements from compiler support over eager execution, followed by further gains with Float8 training.
- Table 3.2: Demonstrates how earlier gains scale as the number of GPUs increases.
- Table 3.3: Shows speed-up achieved by AsyncTP (a HW/SW co-designed technique) over 2D training combined with `torch.compile` and Float8 training.
- Table 3.4: Quantifies the benefits of Interleaved 1F1B scheduling over 1F1B on top of AsyncTP, `torch.compile`, and Float8 training.
- Table 3.5: Demonstrates the effectiveness of CP on enabling long context training, even at small scale.
- Table 3.6: Demonstrate the composability of 4D parallelism, and the effectiveness of CP on enabling long context training at large scale.

For FSDP, the ZeRO-3 variant is used for all experiments except for those involving PP where the ZeRO-2 variant is used. This distinction is due to the inefficiency of ZeRO-3 in PP, where it incurs additional all-gather calls for each microbatch. In contrast, ZeRO-2 gathers parameters only once for the first microbatch and reshards after the last microbatch's backward pass.

Table 3.1: 1D parallelism (FSDP) on Llama 3.1 8B model, 8 GPUs. Mixed precision training. Selective activation checkpointing. Local batch size 2, global batch size 16. (Stats per GPU)

Techniques	Throughput (Tok/Sec)	Comparison	Memory (GiB)
FSDP	6,258	100%	81.9
+ `torch.compile`	6,674	+ 6.64%	77.0
+ `torch.compile` + Float8	9,409	+ 50.35%	76.8

Table 3.2: 1D parallelism (FSDP) on Llama 3.1 8B model, 128 GPUs. Mixed precision training. Selective activation checkpointing. Local batch size 2, global batch size 256. (Stats per GPU)

Techniques	Throughput (Tok/Sec)	Comparison	Memory (GiB)
FSDP	5,645	100%	67.0
+ `torch.compile`	6,482	+ 14.82%	62.1
+ `torch.compile` + Float8	9,319	+ 65.08%	61.8

Table 3.3: 2D parallelism (FSDP + TP) + `torch.compile` + Float8 on Llama 3.1 70B model, 256 GPUs. Mixed precision training. Full activation checkpointing. FSDP degree 32, TP degree 8. Local batch size 16, global batch size 512. (Stats per GPU)

Techniques	Throughput (Tok/Sec)	Comparison	Memory (GiB)
2D	897	100%	70.3
+ AsyncTP	1,010	+ 12.59%	67.7

Table 3.4: 3D parallelism (FSDP + TP + PP) + `torch.compile` + Float8 + AsyncTP on Llama 3.1 405B model, 512 GPUs. Mixed precision training. Full activation checkpointing. FSDP degree 4, TP degree 8, PP degree 16. Local batch size 32, global batch size 128. (Stats per GPU)

Schedule	Throughput (Tok/Sec)	Comparison	Memory (GiB)
1F1B	100	100%	78.0
Interleaved 1F1B	130	+ 30.00%	80.3

Table 3.5: FSDP + CP + `torch.compile` + Float8 on Llama 3.1 8B model, 8 GPUs. Mixed precision training. Full activation checkpointing. Local batch size 1. (Stats per GPU)

Schedule	Sequence Length	Throughput (Tok/Sec)	Memory (GiB)
FSDP 8, CP 1	32,768	3,890	83.9
FSDP 4, CP 2	65,536	2,540	84.2
FSDP 2, CP 4	131,072	1,071	84.0
FSDP 1, CP 8	262,144	548	84.5

Table 3.6: 4D parallelism (FSDP + TP + PP + CP) + `torch.compile` + Float8 + AsyncTP + 1F1B on Llama 3.1 405B model, 512 GPUs. Mixed precision training. Full activation checkpointing. TP degree 8, PP degree 8. Local batch size 8. (Stats per GPU)

Schedule	Sequence Length	Throughput (Tok/Sec)	Memory (GiB)
FSDP 8, CP 1	32,768	76	75.3
FSDP 4, CP 2	65,536	47	75.9
FSDP 2, CP 4	131,072	31	77.1
FSDP 1, CP 8	262,144	16	84.9

3.5.3 Loss convergence

TorchTitan 's design principles have influenced the development of advanced distributed training features such as FSDP2, AsyncTP, PP, and CP in PyTorch's distributed library. Throughout these contributions, we have ensured the loss converging of individual techniques as well as their various combinations of parallelisms and optimizations.

For example, below is a series of loss-converging tests covering both parallelisms and training optimizations. We use notations of "FSDP 8" for an experiment in which the degree of FSDP is 8, "FSDP 8, CP 8" for an experiment on 64 GPUs where FSDP degree is 8 and CP degree is 8, etc. We assume the correctness of FSDP, which can be further verified by comparing it with DDP or even single-device jobs.

Table 3.7: Loss-converging tests setup.

Parallelism	Techniques
FSDP 8 (ground truth)	default
FSDP 8, TP 2, PP 2	torch.compile, Float8, async TP, Interleaved 1F1B
FSDP 8, TP 2, CP 2, PP 2	torch.compile, Float8, async TP, Interleaved 1F1B
FSDP 8, CP 8	default

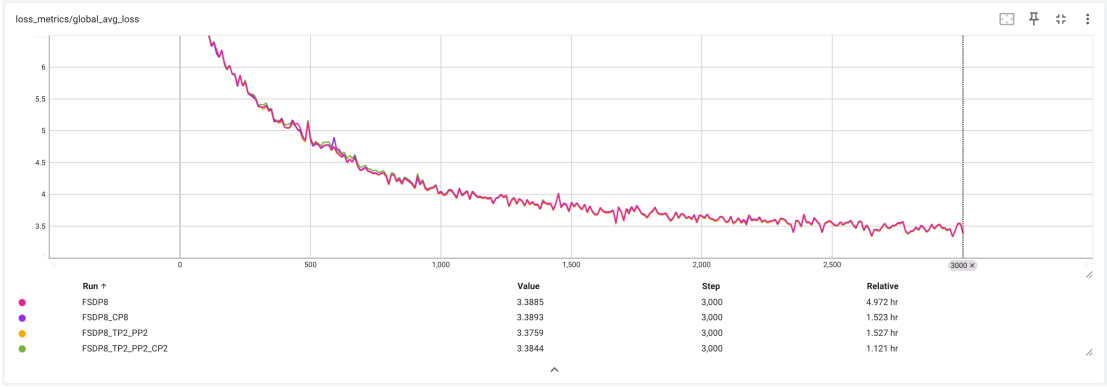


Figure 3.2: Loss converging tests on Llama 3.1 8B. C4 dataset. Local batch size 4, global batch size 32. 3000 steps, 600 warmup steps.

3.6 Scaling with TorchTitan 4D Parallelism

Scaling large language models (LLMs) requires parallelism strategies to handle increasing model sizes and data on thousands of GPUs. TorchTitan enables efficient scaling through composable 4D parallelism. This section highlights key observations and motivations for using TorchTitan 4D parallelism, focusing on a specific combination shown in Figure 3.3.

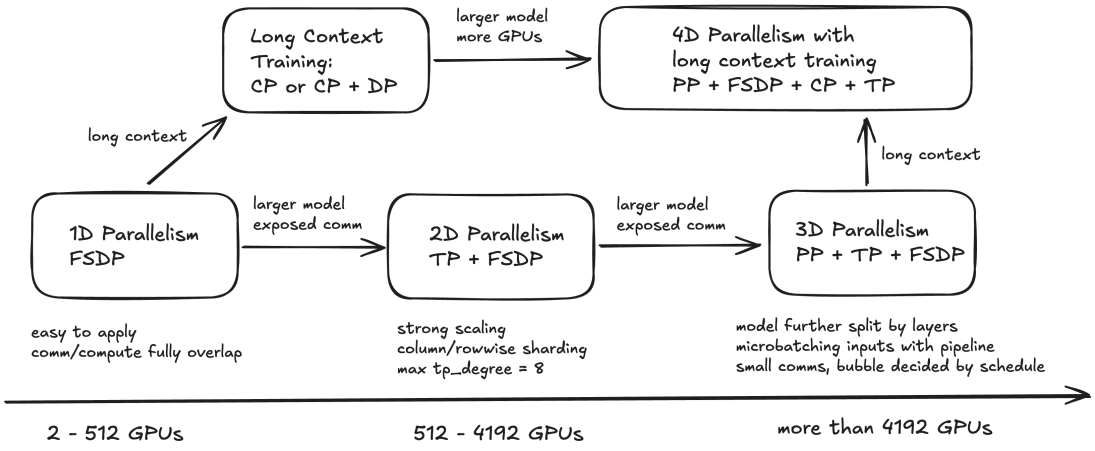


Figure 3.3: Scaling with 4D Parallelism

3.6.1 Scaling with FSDP

FSDP (ZeRO) is a general technique applicable to any model architecture and is often sufficient as the first degree of parallelism when communication is faster than computation (e.g., up to 512 GPUs). However, with larger scales, collective latency increases linearly with the world size, limiting efficiency. To overcome this, model parallelism like TP and PP can be combined with FSDP.

3.6.2 2D Parallelism: TP with FSDP

Tensor Parallelism (TP) reduces collective latency by distributing work across GPUs, enabling smaller effective batch sizes and reducing peak memory usage for large models or sequence lengths. Combining FSDP and TP allows strong scaling with a fixed problem/batch size (Details shown in Figure 3.5). TP also improves FLOP utilization by optimizing matrix multiplication shapes. However, TP introduces blocking collectives and is typically limited to intra-node scaling (e.g., NVLink), with degrees usually capped at 8. Scaling beyond 4192 GPUs requires combining TP with PP.

3.6.3 3D Parallelism: PP with 2D Parallelism

Pipeline Parallelism (PP) reduces communication bandwidth requirements by transmitting only activations and gradients between stages in a peer-to-peer manner. PP is particularly effective for mitigating FSDP communication latency at larger scales or in bandwidth-limited clusters. The efficiency of PP depends on pipeline schedules and microbatch sizes, which influence the size of pipeline "bubbles."

3.6.4 Long Context Training and 4D Parallelism

Context Parallelism (CP) allows ultra long context training by splitting the context (sequence) dimension across GPUs to avoid OOM errors. CP is mainly used for long context training, to give the

model capability to capture more correlations for tokens, thus enhancing the overall model quality. For scaling sequence length, CP can be used alone or together with DP. When training large models or on large number of GPUs, we can combine CP with 3D paralleism, where TP usually keeps the innner-most DeviceMesh dimension, and CP applies in the next outer DeviceMesh dimension.

3.7 Related Work

Existing libraries such as Megatron-LM (Narayanan et al., 2021), DeepSpeed (Rasley et al., 2020), veScale (Inc., 2024), and PyTorch Distributed (Paszke et al., 2019; Meta Platforms, Inc., 2024) offer APIs to support distributed training workflows. However, they often fall short in terms of flexibility, seamless integration, and scalability. In contrast, TorchTitan is designed to overcome these limitations by natively supporting essential features that are missing or insufficiently addressed in current frameworks.

Slapo (Chen et al., 2023) introduces a schedule language to convert a PyTorch model for common model training optimizations such as 3D parallelism, and supports progressive optimization through high-level primitives. In contrast, TorchTitan provides modular and composable APIs built on DTensor and DeviceMesh.

We note that each of these libraries has its own strengths, and TorchTitan is designed to provide foundational components that can be leveraged by all of them. We now present a detailed qualitative comparison, including feature breakdowns and code complexity analysis.

3.7.1 TorchTitan enables new designs

TorchTitan 's extensive feature set and broad design space coverage are driven by its unified design principles i.e. modularity, composability, and extensibility. Leveraging these principles, TorchTitan seamlessly integrates diverse parallelism strategies (FSDP, TP, PP, and CP) and opti-

mizations (e.g., SAC, Float8 training). This unified framework not only supports advanced pipeline schedules and multi-dimensional parallelism but also simplifies the integration of new techniques, making it highly adaptable for cutting-edge research and production-grade deployments.

The following table highlights TorchTitan 's capabilities in context of parallelism, checkpointing and compiler support offerings compared to Megatron-LM, DeepSpeed, and veScale:

3.7.2 Code Complexity and Maintainability

TorchTitan 's design principles also contribute to its significantly reduced code complexity. Despite offering a rich feature set, TorchTitan maintains a compact and modular codebase, making it easier to extend, maintain, and evolve while ensuring high performance. The following table compares the lines of code (LOC) for TorchTitan with Megatron-LM and DeepSpeed:

3.8 Implementation Details

3.8.1 Composable 4D parallelism walkthrough

We have discussed the scaling with TorchTitan 4D parallelism and the motivations to apply different parallelisms to scale training to thousands of GPUs. In this section we will walk through the 4D parallelism code in TorchTitan.

The first step is to create an instance of the model (e.g. the `Transformer` for Llama models) on the meta device. We then apply PP by splitting the model into multiple PP stages according to the `pipeline_parallel_split_points` config. Note that for PP with looped schedules, we may obtain multiple `model_parts` from PP splitting, where each item in `model_parts` is one stage-model-chunk. Next we apply SPMD-style distributed training techniques including TP, activation checkpointing,

[‡]Custom Fusion Kernels

Table 3.8: Comparison of TorchTitan with Megatron-LM, DeepSpeed, and veScale with respect to parallelism, compiler support, activation checkpointing, and model checkpointing.

Features	TorchTitan	Megatron-LM	DeepSpeed	veScale
FSDP-Zero2	Yes	Yes	Yes	No
FSDP-Zero3	Yes	Yes	Yes	No
HSDP	Yes	Yes	No	No
TP	Yes	Yes	No	Yes
Async TP (Micro-pipelining)	Yes	Yes	No	Yes
CP	Yes	Yes	No	No
PP-Gpipe	Yes	Yes	Yes	No
PP-Interleaved (1F1B)	Yes	Yes	Yes	Yes
PP-Looped-BFS	Yes	No	No	No
PP-1F1B	Yes	Yes	Yes	Yes
PP-Flexible-Interleaved-1F1B	Yes	No	No	No
PP-ZeroBubble	Yes	No	No	Yes
(TP+SP)+PP	Yes	Yes	No	Yes
DDP+(TP+SP)+PP	Yes	Yes	No	Yes
FSDP+(TP+SP)	Yes	No	No	No
FSDP+(TP+SP)+PP	Yes	No	No	No
FSDP+(TP+SP)+PP+CP	Yes	No	No	No
MoE	Ongoing	Yes	No	No
Full AC	Yes	Yes	Yes	Yes
Flexible SAC	Yes	No	No	No
DCP	Yes	Yes	Yes	Yes
Float8 Training	Yes	Yes	No	No
`torch.compile`	Yes	No[‡]	Partial	No

Table 3.9: Lines of Code (LOC) comparison across systems.

Lines of Code (LOC)	TorchTitan	Megatron-LM	DeepSpeed
Core Codebase	7K	93K	94K
Total Codebase (Including Utils)	9K	269K	194K

torch.compile, FSDP, and mixed precision training for each model part, before actually initializing the sharded model on GPU.

```
# meta init
with torch.device("meta"):
    model = model_cls.from_model_args(model_config)


# apply PP
pp_schedule, model_parts = models_pipelining_fns[model_name](
    model, pp_mesh, parallel_dims, job_config, device, model_config, loss_fn
)


for m in model_parts:
    # apply SPMD-style distributed training techniques
    models_parallelize_fns[model_name](m, world_mesh, parallel_dims, job_config)
    # move sharded model to GPU and initialize weights via DTensor
    m.to_empty(device="cuda")
    m.init_weights()
```

To apply PP to the model, we run the following code at the high level. `pipeline_llama_manual_split` splits the model into multiple stages according to the manually given `pipeline_parallel_split_points` config, by removing the unused model components from a complete model (on the meta device). Then `build_pipeline_schedule` make the pipeline schedule with various options from `torch.distributed.pipelining`, including 1F1B (Narayanan et al., 2019), GPipe (Huang et al., 2019b), interleaved 1F1B (Narayanan et al., 2021), etc. instructed by the `pipeline_parallel_schedule` config.

```
stages, models = pipeline_llama_manual_split(
   model, pp_mesh, parallel_dims, job_config, device, model_config
)
pp_schedule = build_pipeline_schedule(job_config, stages, loss_fn)
return pp_schedule, models
```

TP and FSDP are applied in the SPMD-style `models_parallelize_fns` function. To apply TP, we utilize the DTensor `parallelize_module` API, by providing a TP "plan" as the instruction of how model parameters should be sharded. In the example below, we showcase the (incomplete) code for sharding the repeated `TransformerBlock`.

```
for layer_id, transformer_block in model.layers.items():
   layer_tp_plan = {
      "attention_norm": SequenceParallel(),
      "attention": PrepareModuleInput(
         input_layouts=(Shard(1), None),
         desired_input_layouts=(Replicate(), None),
      ),
      "attention.wq": ColwiseParallel(),
      ...
   }
   parallelize_module(
      module=transformer_block,
      device_mesh=tp_mesh,
      parallelize_plan=layer_tp_plan,
   )
```

Then, we apply the FSDP by wrapping each individual `TransformerBlock` and then the whole model. Note that the FSDP2 implementation in PyTorch comes with mixed precision training support. By default, we use `torch.bfloat16` on parameters all-gather and activation computations, and use `torch.float32` on gradient reduce-scatter communication and optimizer updates.

```
mp_policy = MixedPrecisionPolicy(param_dtype, reduce_dtype)
fsdp_config = {"mesh": dp_mesh, "mp_policy": mp_policy}

for layer_id, transformer_block in model.layers.items():
   # As an optimization, do not reshard_after_forward for the last
   # TransformerBlock since FSDP would prefetch it immediately
   reshard_after_forward = int(layer_id) < len(model.layers) - 1
   fully_shard(
      transformer_block,
      **fsdp_config,
      reshard_after_forward=reshard_after_forward,
   )
fully_shard(model, **fsdp_config)
```

Independently, we can apply CP by running each training iteration under a Python context manager.

```
optional_context_parallel_ctx = (
   utils.create_context_parallel_ctx(
      cp_mesh=world_mesh["cp"],
```

```
        cp_buffers=[input_ids, labels] + [m.freqs_cis for m in model_parts],
        cp_seq_dims=[1, 1] + [0 for _ in model_parts],
        cp_no_restore_buffers={input_ids, labels},
        cp_rotate_method=job_config.experimental.context_parallel_rotate_method,
    )
    if parallel_dims.cp_enabled
    else None
)
...
with train_context(optional_context_parallel_ctx):
    pred = model(input_ids)
    loss = loss_fn(pred, labels)
```

3.8.2 Fully Sharded Data Parallel

FSDP2 advances the tensor sharding approach by replacing the original FSDP1 FlatParameter sharding. Specifically, parameters are now represented as DTensors sharded on the tensor dimension 0. This provides better composability with model parallelism techniques and other features that requires the manipulation of individual parameters, allowing sharded state dict to be represented by DTensor without any communication, and provides for a simpler meta-device initialization flow via DTensor. For example, FSDP2 unlocks finer grained tensor level quantization, especially Float8 tensor quantization, which we will showcase in the results section.

As part of the rewrite from FSDP1 to FSDP2, FSDP2 implements an improved memory management system by avoiding using record stream. This enables deterministic memory release, and as a result provides lower memory requirements per GPU relative to FSDP1. For example on Llama 2

7B, FSDP2 records an average of 7% lower GPU memory versus FSDP1.

In addition, by writing efficient kernels to perform multi-tensor allgather and reduce scatter, FSDP2 shows on-par performance compared to FSDP1, with even slight performance gains - using the Llama 2 7B, FSDP2 shows an average gain of 1.5% faster throughput.

The performance gains are the result of employing two small performance improvements. First, only a single division kernel is run for the FP32 reduce scatter (pre-dividing the local FP32 reduce-scatter gradient by world size, instead of a two step pre and post divide by square root of world size). Secondly, in TorchTitan, FSDP2 is integrated with a default of not re-sharding the final block in a transformer layer during the forward pass, since it will be immediately re-gathered at the start of the backward pass.

Usage: TorchTitan has fully integrated FSDP2 as the default parallelism when training, and the `data_parallel_shard_degree` is the controlling dimension in the command line or TOML file. Note that for ease of use, the default `data_parallel_shard_degree` is -1, means to simply use all GPUs available, so user do not need to specify the actual world size.

3.8.3 Hybrid Sharded Data Parallel

Hybrid Sharded Data Parallel (HSDP) is an extension of FSDP (Zhang et al., 2022a). In FSDP, communication occurs between all devices within the FSDP group. However, at some point, the FSDP communication overhead exceeds its corresponding computation because the latency of allgather/reduce-scatter communications increases linearly with the number of devices. This results in low MFU and becomes worthless to add more GPUs for scaling.

HSDP obviates this to some degree by creating a 2-D DeviceMesh that contains replica groups on one dimension and shard groups on the other dimension, where each shard group runs FSDP and the replica group runs normal data parallel. This ensures the FSDP communications happen in a fraction of the original world size, with the addition of backward gradient allreduce across replica

groups. HSDP reduces FSDP communication overhead and allows further scaling with data parallel.

Usage: TorchTitan makes it easy to experiment with HSDP by using the two configurable settings: `data_parallel_shard_degree` and `data_parallel_replicate_degree`, which controls the degree of the shard and replica groups we are creating. The product of both replicate and shard degree is the actual data parallel world size.

3.8.4 Tensor Parallel

TP partitions the attention and feed forward network (MLP) modules of a transformer layer across multiple devices, where the number of devices used is the TP degree. This allows for multiple GPUs to cooperatively process the same batch by using the local sharded model parameters, at the cost of adding `all-reduce/all-gather/reduce-scatter` operations to synchronize intermediate activations.

Due to the additional collectives introduced by TP, it needs to happen within a fast network (i.e NVLink). When training LLMs, TP is usually combined with FSDP, where TP shards within nodes and FSDP shards across nodes to create the 2D hierarchical sharding on different DeviceMesh dimensions.

Usage: Because of the synergistic relationship between TP and SP, TorchTitan natively bundles these two together and they are jointly controlled by the TP degree setting in the command line or the TOML entry of `tensor_parallel_degree`. Setting this to 2 for example would mean that 2 GPUs within the node will share the computational load for each transformer layers attention and MLP modules via TP, and normalization/dropout layers via Sequence Parallel. Loss Parallel is implemented via a context manager as it needs to control the loss computation outside of the model's forward computation. It can be enabled via `enable_loss_parallel`.

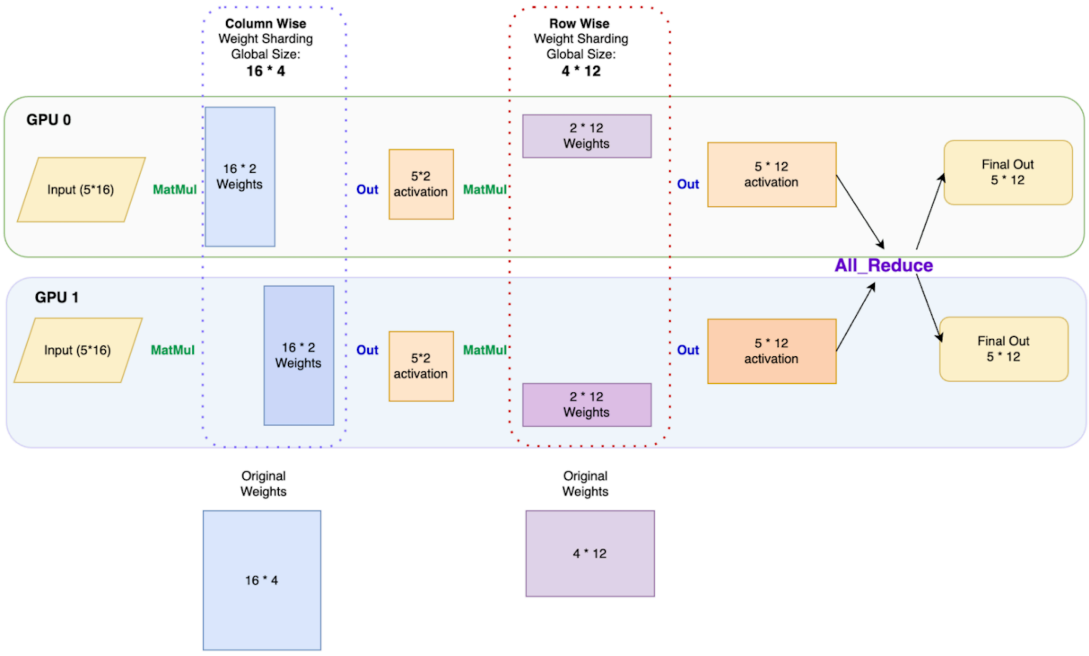


Figure 3.4: Tensor Parallel in detail (2 GPUs, data moves from left to right).

3.8.5 Pipeline Parallel

We expose several parameters to configure PP. `pipeline_parallel_degree` controls the number of ranks participating in PP. `pipeline_parallel_split_points` accepts a list of strings, representing layer fully-qualified-names before which a split will be performed. Thus, the total number of pipeline stages V will be determined by the length of this list. `pipeline_parallel_schedule` accepts the name of the schedule to be used. If the schedule is multi-stage, there should be $V > 1$ stages assigned to each pipeline rank, otherwise $V == 1$. `pipeline_parallel_microbatches` controls the number of microbatches to split a data batch into.

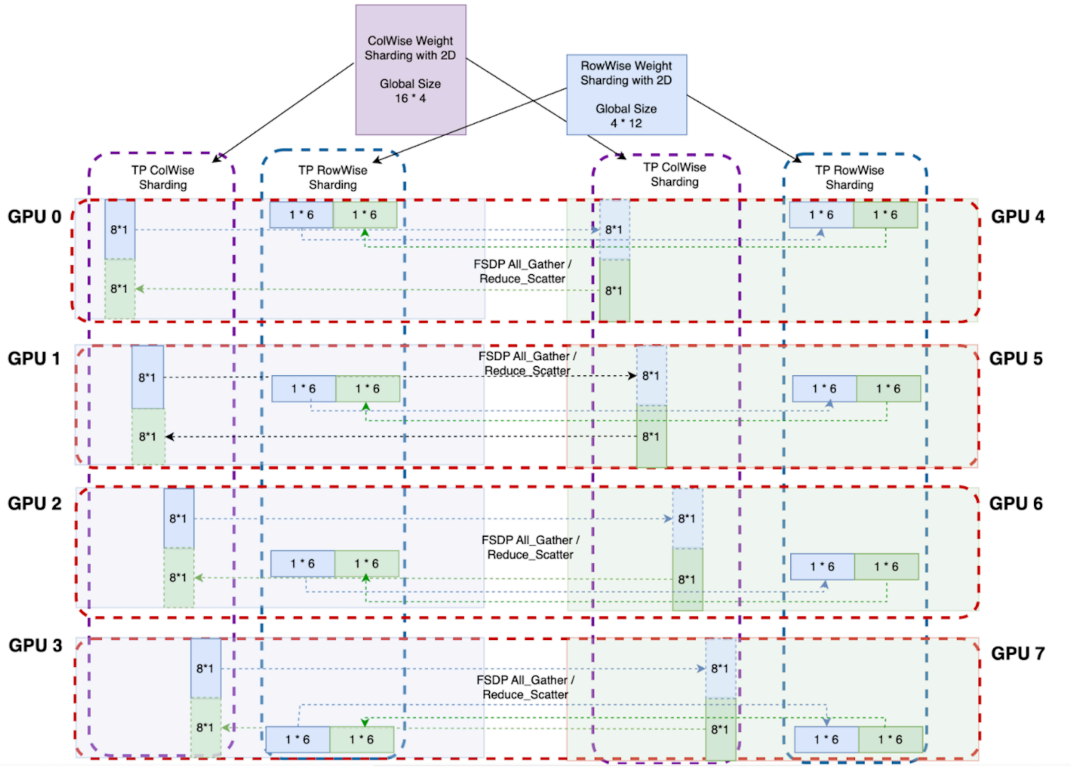


Figure 3.5: FSDP2 + Tensor Parallel (TP degree 4) sharding layout, with 2 nodes of 4 GPUs.

3.8.6 Enabling 4D parallel training: Context-Parallel (CP)

To address context scaling, we have incorporated Context Parallelism (CP) into TorchTitan. Following the principles of modular design of TorchTitan, CP was integrated via a context manager that dynamically replaces calls to attention operators (namely, `scaled_dot_product_attention`) with CP operations, ensuring no changes to the model code are required.

Under the hood, CP shards the DTensor along the sequence dimension across the CP device mesh. It extends the DTensor dispatcher to handle CP-specific operations, such as Ring Attention and causal attention load balancing, ensuring efficient operation. By extending DTensor's capabilities to support CP, TorchTitan ensures that CP is fully compatible with all other parallelisms

(FSDP, TP, PP), optimizations (e.g., activation checkpointing, `torch.compile`), and DCP. This demonstrates the extensibility of TorchTitan 's modular design, which accommodates future optimizations seamlessly while maintaining performance and compatibility.

3.8.7 Activation checkpointing

TorchTitan offers two types of Selective Activation Checkpointing which allow for a more nuanced tradeoff between memory and recomputation. Specifically, we offer the option to selectively checkpoint "per layer" or "per operation". The goal for per operation is to free memory used by operations that are faster to recompute and save intermediates (memory) for operations that are slower to recompute and thus deliver a more effective throughput/memory trade-off.

Usage: AC is enabled via a two-line setting in the command line or TOML file. Specifically, `mode` can be either `none`, `selective`, or `full`. When `selective` is set, then the next config of `selective_ac_type` is used which can be either a positive integer to enable selective layer checkpointing, or `op` to enable selective operation checkpointing. Per layer takes an integer input to guide the checkpointing policy, where 1 = checkpoint every layer (same as full), 2 = checkpoint every other layer, 3 = checkpoint every third layer, etc. Per op(eration) is driven by the `_save_list` policy in `parallelize_llama.py` which flags high arithmetic intensity operations such as matmul (matrix multiplication) and SPDA (Scaled Dot Product Attention) for saving the intermediate results, while allowing other lower intensity operations to be recomputed. Note that for balancing total throughput, only every other matmul is flagged for saving.

3.8.8 AsyncTP

The `SymmetricMemory` collectives used in AsyncTP are faster than standard NCCL collectives and operate by having each GPU allocate an identical memory buffer in order to provide direct P2P

access. `SymmetricMemory` relies on having NVSwitch within the node, and is thus generally only available for H100 or newer GPUs.

Usage: AsyncTP is enabled within the experimental section of the TorchTitan TOML config file and turned on or off via the `enable_async_tensor_parallel` boolean setting.

3.8.9 Customizing FSDP2 Mixed Precision in TorchTitan

Mixed Precision is controlled by the `MixedPrecisionPolicy` class in the `apply_fsdp` function, which is then customized with `param_dtype` as BF16, and `reduce_dtype` defaulting to FP32 by default in TorchTitan. The `reduce_dtype` in FP32 means that the reduce-scatter in the backwards pass for gradient computation will take place in FP32 to help maximize both stability and precision of the gradient updates.

3.9 Summary

TorchTitan is a powerful and flexible framework for LLM training, enabling seamless composability of parallelism techniques (FSDP, TP, PP, CP), memory optimizations (Float8, activation checkpointing), and PyTorch compiler integration for enhanced efficiency. Its modular design supports evolving architectures and hardware, fostering innovation with multi-axis metrics.

Designed for interpretability and production-grade training, TorchTitan offers elastic scalability, comprehensive training recipes, and expert guidance on distributed training strategies. As demonstrated in experiments, it accelerates training by 65.08% on Llama 3.1 8B (128 GPUs, 1D), 12.59% on Llama 3.1 70B (256 GPUs, 2D), and 30% on Llama 3.1 405B (512 GPUs, 3D) over optimized baselines, while enabling long-context training with 4D composability. With its robust features and high efficiency, TorchTitan is an ideal one-stop solution for challenging LLM training tasks.

4

TorchSim: High Fidelity Runtime and Memory Estimation for Distributed Training

In the previous chapter, we introduced TorchTitan, a modular and production-grade training framework that synthesizes efficient implementations for arbitrary distributed training configurations. While TorchTitan enables scalable execution of these configurations across thousands of GPUs, selecting the right configuration remains a key challenge. Given the vast design space of parallelism strategies, memory optimizations, and precision settings, it is essential to evaluate whether a candidate configuration will execute successfully and deliver acceptable performance before committing to full-scale training.

In this chapter, we introduce TorchSim, a simulation-based tool for estimating runtime and memory consumption in distributed deep learning training without requiring actual GPU execution. TorchSim enables users to efficiently navigate the large and complex configuration space exposed by TorchTitan and reason about the performance implications of different training strategies.

TorchSim constructs hardware-aware compute models as well as topology-, algorithm-, and collective-aware communication models to accurately predict operator execution times. It employs a detailed simulator that closely mirrors the multi-stream GPU execution model, capturing compute and communication overlap, exposed communication, and synchronization overheads to estimate end-to-end runtime. For memory estimation, TorchSim tracks tensor lifetimes at operator-level granularity without allocating memory and emulates allocations introduced by distributed collective operations. By mimicking PyTorch's memory management and execution behavior, TorchSim allows users to simulate and compare training and cluster configurations before execution, thereby eliminating the need for costly empirical benchmarking.

4.1 Deriving TorchSim's Design

Thus far, we have established the need for a fast, efficient (GPU execution-free), and high-fidelity runtime and memory estimation tool for comprehensively exploring the performance space of configurations for distributed model training . Furthermore, we have shown that this necessitates faithfully emulating the low-level execution semantics of a distributed training configuration.

In this section, we derive the foundational design principles of TorchSim that comprehensively encompass the problem space in §4.1.1 and present the high-level design of TorchSim in §4.1.2.

4.1.1 Complexity of Distributed Model Training Informs the Design Principles of TorchSim

The following are the core design principles:

(D1) Model-Agnostic. TorchSim must accommodate diverse model architectures (e.g., transformers, state space models, diffusion models) with varying features such as the number of layers, hidden dimensions, and different attention or convolution mechanisms.

(D2) Algorithm and Implementation Coverage. Given the wide variety of distributed training techniques (e.g., FSDP, TP, CP, PP, EP) and optimizations (such as mixed precision training and activation checkpointing), TorchSim must seamlessly support all distributed parallelisms, algorithms, and their various compositions and implementations, faithfully capturing model performance across diverse training setups.

(D3) Non-Intrusive. Since users typically rely on off-the-shelf training recipes, TorchSim must integrate seamlessly without requiring modifications to existing model definitions or training scripts.

(D4) Accurate End-to-End Estimation. TorchSim must deliver reliable runtime and mem-

ory estimates that consider hardware heterogeneity, memory hierarchy, kernel implementations, network topology, and algorithm-specific details, faithfully modeling real accelerator performance while emulating the GPU stream execution model and adhering to stream runtime, memory, and synchronization semantics.

(D5) GPU Execution-Free and Fast. To remain cost-effective, TorchSim should perform execution and memory estimation without requiring access to a GPU cluster. All estimations should run efficiently on a commodity CPU machine with minimal overhead, since exploring the full performance space requires evaluating thousands or even millions of training configurations

(D6) Insightful. Beyond providing aggregate estimates, TorchSim should offer detailed insights such as runtime breakdowns (e.g., exposed versus overlapped communication at module and operator levels) and memory categorization (e.g., distinguishing parameters, activations, gradients, and optimizer states) with clear attribution.

(D7) Modular and Extensible. TorchSim should maintain a modular design to enable seamless replacement or extension of individual components as training techniques, hardware architectures, and execution engines evolve. As the landscape of training methodologies continues to grow in complexity, ensuring ease of integration without introducing unintended side effects becomes increasingly important..

4.1.2 TorchSim High Level Design

TorchSim comprises three core components: TorchSim Capture, TorchSim Simulator, and TorchSim Models, as illustrated in Figure 4.1. TorchSim Capture acts as an extensible wrapper around the PyTorch Runtime Paszke et al. (2019), interpreting the training run as a sequence of operator dispatches. It intercepts these dispatches to collect metadata on operators, memory usage, and synchronization events. TorchSim Models provide runtime predictions at the operator

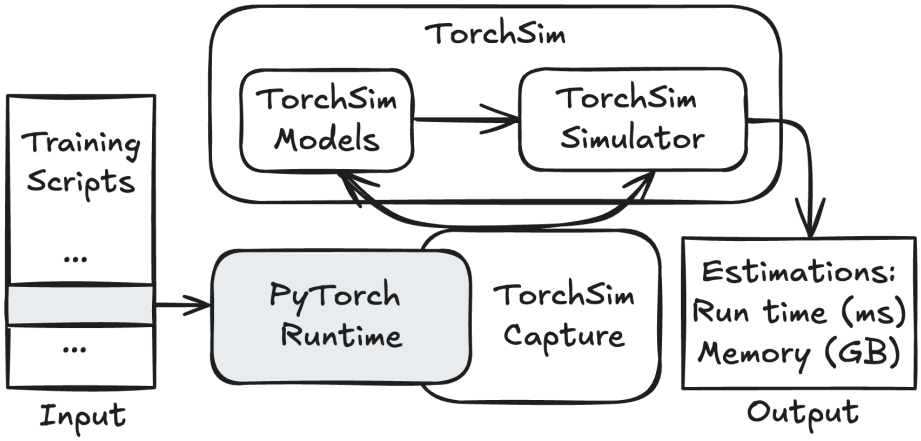


Figure 4.1: TorchSim: High-level System Design

level, which are integrated into the captured metadata. This enriched metadata is then passed to the TorchSim Simulator, which emulates GPU stream execution semantics to produce accurate end-to-end estimates of runtime and memory usage.

4.2 TorchSim Capture and Workflow

In this section, we detail how TorchSim's design realizes the design principles in §4.2.1, and finally, provide an end-to-end walkthrough of TorchSim's workflow in §4.2.2.

4.2.1 TorchSim Capture

We now describe how the internal mechanisms of TorchSim Capture materialize the outlined design principles to enable the generation of comprehensive runtime and memory statistics.

(1) Functionality: Simulating Operator Execution [D5]. To enable GPU-free execution, the training script must run seamlessly while mimicking execution on the target hardware. This allows TorchSim to generate accurate predictions using only inexpensive, commodity hardware.

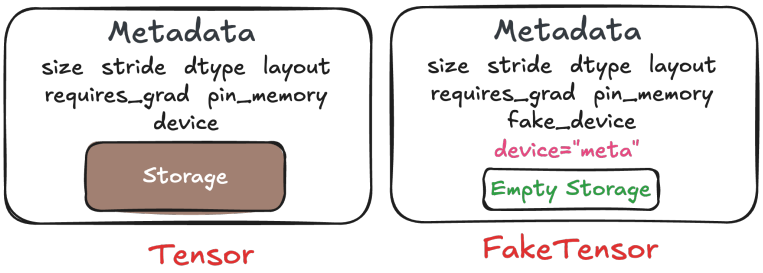


Figure 4.2: Actual and Fake Tensor Representation.

Enabler: *FakeTensorMode*. In PyTorch, each tensor comprises two core components: metadata and storage Paszke et al. (2019). Metadata includes attributes such as *shape*, *dtype*, *device*, *stride*, *layout*, *requires_grad*, and *pin_memory*, while the *Storage* object contains the raw data and allows for efficient memory sharing among tensors, as illustrated in Figure 4.2.

For simulation, only metadata is required, as it determines storage requirements and serves as input to subsequent operations. TorchSim tracks a tensor's *size*, *device*, *dtype*, and *requires_grad* flag. PyTorch's *FakeTensors* support this abstraction by representing tensors without actual data, placing them on an abstract *meta* device while recording their intended execution device via the *fake_device* attribute Contributors (2025).

However, *FakeTensors* alone do not enable full training execution without GPUs. Operators must still execute correctly on them, including proper metadata propagation. *FakeTensorMode* Contributors (2025) enables this by managing fake tensor creation, applying operations, and propagating metadata. TorchSim runs under *FakeTensorMode*, which enables both GPU-free execution and accurate memory estimation.

(2) Functionality: Simulating Communication Collectives [D2, D5]. In distributed training, TorchSim must simulate a dummy collection of devices that mirrors the actual training configuration and its communication groups. This setup allows collective operations issued during an iteration to execute as if real communication had taken place.

Enabler: *FakeProcessGroup*. PyTorch uses the *DeviceMesh* and *ProcessGroup* abstractions to manage communication collectives such as *all_gather* and *reduce_scatter*. A *DeviceMesh* is a multi-dimensional representation of devices, where each dimension corresponds to a specific parallelism strategy. Each *DeviceMesh* is associated with a *ProcessGroup*, and each dimension forms a *Sub-Mesh* with its own *Sub-ProcessGroup*.

Collectives are issued to specific *Sub/DeviceMesh* objects and executed by the corresponding *Sub-/ProcessGroup*. FSDP, TP, and CP follow the *Single Program, Multiple Data* (SPMD) model, where each device runs the same program on different data or model shards. FSDP shards parameters and processes separate microbatches; CP splits sequences; TP partitions model blocks (e.g., attention heads). Since all devices execute identical code, analyzing one SPMD process suffices for memory and runtime estimation. In contrast, PP uses the *Multiple Program, Multiple Data* (MPMD) paradigm, dividing the model into sequential stages across devices. Each stage runs a different code on different data. PP is typically combined with SPMD strategies (FSDP, TP, CP) within each stage. Here, memory/runtime can be estimated per SPMD stage, but accurate timing requires modeling inter-stage communication and scheduling dependencies.

To emulate distributed execution without actual communication, TorchSim replaces *ProcessGroup* with *FakeProcessGroup* Contributors (2024). This fake process group simulates a virtual collection of devices, ensuring that collectives invoked under *FakeTensorMode* return *FakeTensors* and dummy synchronization objects with correct metadata.

(3) Functionality: Intercepting Operator Dispatch [D1, D2, D3]. To ensure that TorchSim remains model-agnostic, independent of specific optimization techniques and implementations, and non-intrusive, it operates at the granularity of individual operators. It interprets a training run as a sequence of operator dispatches, where each operator processes input tensors and produces output tensors. Since tensors are dynamically created—explicitly by users, through operations, or

implicitly by the autograd engine—TorchSim must intercept every tensor operation and capture relevant metadata to estimate runtime and track memory usage.

Enabler: *TorchDispatchMode*. At runtime, PyTorch's Dispatcher routes each operation to the appropriate kernel based on the tensor's *device*, *dtype*, and the operator type. *TorchDispatchMode* is a context manager that enables interception by overriding the *torch.__dispatch__* method, providing access to the operator, its arguments, and results He et al. (2022). TorchSim extends *TorchDispatchMode* to systematically capture all tensor operations for accurate execution analysis.

For memory estimation, TorchSim extracts metadata from the resultant tensors, recording attributes such as *size*, *device*, and *dtype* for every operation encountered in the dispatcher.

For runtime estimation, once an operation is intercepted, it is classified as either a compute operation (e.g., *matmul*, *layernorm*) or a communication collective (e.g., *all_gather*, *reduce_scatter*). TorchSim then extracts relevant features: for compute operations, this includes *dtype*, *input/output shapes*, and backend-specific details; for communication collectives, it includes *data size*, *collective type*, and *process group*.

The extracted metadata is used to populate the simulation data structures defined in Tables 4.1 and 4.2. TorchSim records the CUDA stream, resource type, estimated runtime, and CPU dispatch order for each dispatched operation. It distinguishes between compute and collective operations, using the associated process group to infer resource usage for the latter. Each operation is enqueued into the queue corresponding to the current CUDA stream.

For non-functional collectives that return a `Work` object, the operation metadata (`op_info`) is registered to the work's unique `seq_id` in the `work_registry`, enabling later correlation with `work.wait()`. For functional collectives, which return one or more `Tensors`, the underlying storage of each tensor is mapped to the originating operation in the `wait_tensor_registry`, allowing subsequent `wait_tensor()` calls to synchronize with the correct producer.

Resource	State	SyncAction
INTRA_COMM	WAITING	STREAM_WAIT
INTER_COMM	RUNNING	EVENT_WAIT
COMP	READY	SYNC_WAIT
HOST_TO_MEM	COMPLETE	STREAM_RELEASE
MEM_TO_HOST		EVENT_RELEASE
		WORK_WAIT
		WORK_RELEASE

Table 4.1: Enumeration classes representing the Resources, Queue states and Synchronization actions for primitives.

(4) Functionality: Capturing Synchronization Primitives [D2, D4]. The final requirement for TorchSim is to capture synchronization metadata essential for simulating GPU stream execution. This is critical for supporting comprehensive algorithm and implementation coverage across all distributed training techniques. As summarized in Table 4.3, we identify eight synchronization primitives that are sufficient to emulate these techniques.

Enabler: Dynamic Function Overriding. Because synchronization primitives do not operate on tensors, they are not intercepted by *TorchDispatchMode*. To capture them, TorchSim uses dynamic function overriding to hook into their execution and extract relevant metadata. During simulated execution of the training script, TorchSim incrementally records this metadata as each synchronization primitive is encountered.

Table 4.3 outlines the synchronization primitives intercepted and modeled by TorchSim, along with the logic used to capture their semantics. The left column presents the high-level primitive (e.g., stream waits, event records, global syncs), while the right column describes how TorchSim records the synchronization metadata into its internal data structures. This includes queuing wait and release records linked to stream or event identifiers, initializing global synchronization dependencies, and mapping work- or tensor-based waits to their corresponding producer operations.

SyncInfo	
sync_action	The action type (from SyncAction).
release_event_id	Identifiers for event-based, op-based, or global synchronizations.
release_seq_id	
sync_id	
OpInfo	
seq_id	Unique ID assigned to the op in CPU dispatch order.
stream_id	CUDA stream ID the op is dispatched to.
resource	Set of resources (e.g., compute, communication) used by the op (from Resource).
run_time	Estimated and remaining runtime during simulation.
rem_time	
Queue	
stream_id	CUDA stream ID and priority corresponding to the queue.
priority	
state	Current execution state of a queue (from State).
ops	List of OpInfo objects dispatched to this stream.
sync_infos	Maps op seq_id to a set of SyncInfo records to be applied post-execution.
wait_sync_infos	Tracks currently blocking conditions. A queue can only transition from WAITING to READY when this dict is empty.
[-1] sync seed	Global sync ops pre-injected at key -1 to ensure early synchronizations are honored.
Simulator	
streamid_to_queue	Maps each CUDA stream to its Queue.
seq_id	Global counter assigning unique ID to each dispatched op.
work_registry	Maps the seq_id of a Work object to the OpInfo that produced it.
wait_tensor	Maps a tensor's underlying storage to the list of OpInfo objects that
registry	produced it.
global_sync_infos	Global set of sync ops that apply to all queues and are applied when the queue is first captured.
sync_count	To differentiate individual synchronize() calls.
event_wait_ids	Track which event IDs have been waited on or recorded, respectively.
event_record_ids	
T_sim	Running total of simulated time.
resource_occupancy	Tracks which queues are currently occupying which resources.
completed_ops	Set of operation IDs that have completed execution.
recorded_events	Set of event IDs that have been recorded.

Table 4.2: Classes describing the Synchronization, Operator, Queue, and Simulator Metadata.

Primitive Semantics	Capturing Primitives in TorchSim
Stream Synchronization	
`s1.wait_stream(s2)` Blocks future ops on `s1` until all work in `s2` completes.	If `s2.last_op_seq_id` $\neq -1$: `seqID = s2.last_op_seq_id` Add `STREAM_WAIT(seqID)` to `s1`'s queue. Add `STREAM_RELEASE(seqID)` to `s2`'s queue.
`s.wait_event(e)` Delays ops on `s` until event `e` is recorded.	Get `eventID`, add to `eventWaitIDs`. Add `EVENT_WAIT(eventID)` to `s`'s queue.
`s.synchronize()` Blocks CPU until all ops in stream `s` complete.	If `last_op_seq_id` $\neq -1$: `seqID = s2.last_op_seq_id` Add `STREAM_WAIT(seqID)` to all queues. Add `STREAM_RELEASE(seqID)` to `s`'s queue. Add global sync `STREAM_WAIT(seqID)`.
Event Synchronization	
`e.wait(s)` Delays stream `s` until event `e` is recorded.	Get `eventID`, add to `eventWaitIDs`. Add `EVENT_WAIT(eventID)` to `s`'s queue.
`e.synchronize()` Blocks CPU until all work tied to event `e` is complete.	Get `eventID` from `e`, add to `eventWaitIDs`. Add `EVENT_WAIT(eventID)` to all queues. Add global sync `EVENT_WAIT(eventID)`.
`e.record(s)` Marks event `e` at the current point in stream `s`.	Get `eventID`, add to `eventRecordIDs`. Enqueue zero-runtime `event_record` op. Add `EVENT_RELEASE(eventID)` to `s`'s queue.
Global and Work Synchronization	
`synchronize()` Blocks CPU until all operations across streams complete.	Increment global `sync_count`. Add `SYNC_WAIT(sync_count)` to all queues. Add global sync `SYNC_WAIT(sync_count)`.

`work.wait()`
Blocks until async work completes.
`wait_tensor(t)`
Delays ops on `t` until its producer completes.
	For `work.wait()`:
	Extract `workSeqID`, retrieve `opInfos` from `workRegistry`.
	For `wait_tensor(t)`:
	Extract storage from `t`, retrieve `opInfos` from `waitTensorRegistry`.
	For each `opInfo`:
	`seqID = opInfo.seq_id`
	Add `WORK_WAIT(seqID)` to all queues.
	Add `WORK_RELEASE(seqID)` to originating queue at position `seqID`.
	Add global sync `WORK_WAIT(seqID)`.

Table 4.3: Synchronization primitives with PyTorch semantics and detailed actions captured by TorchSim.

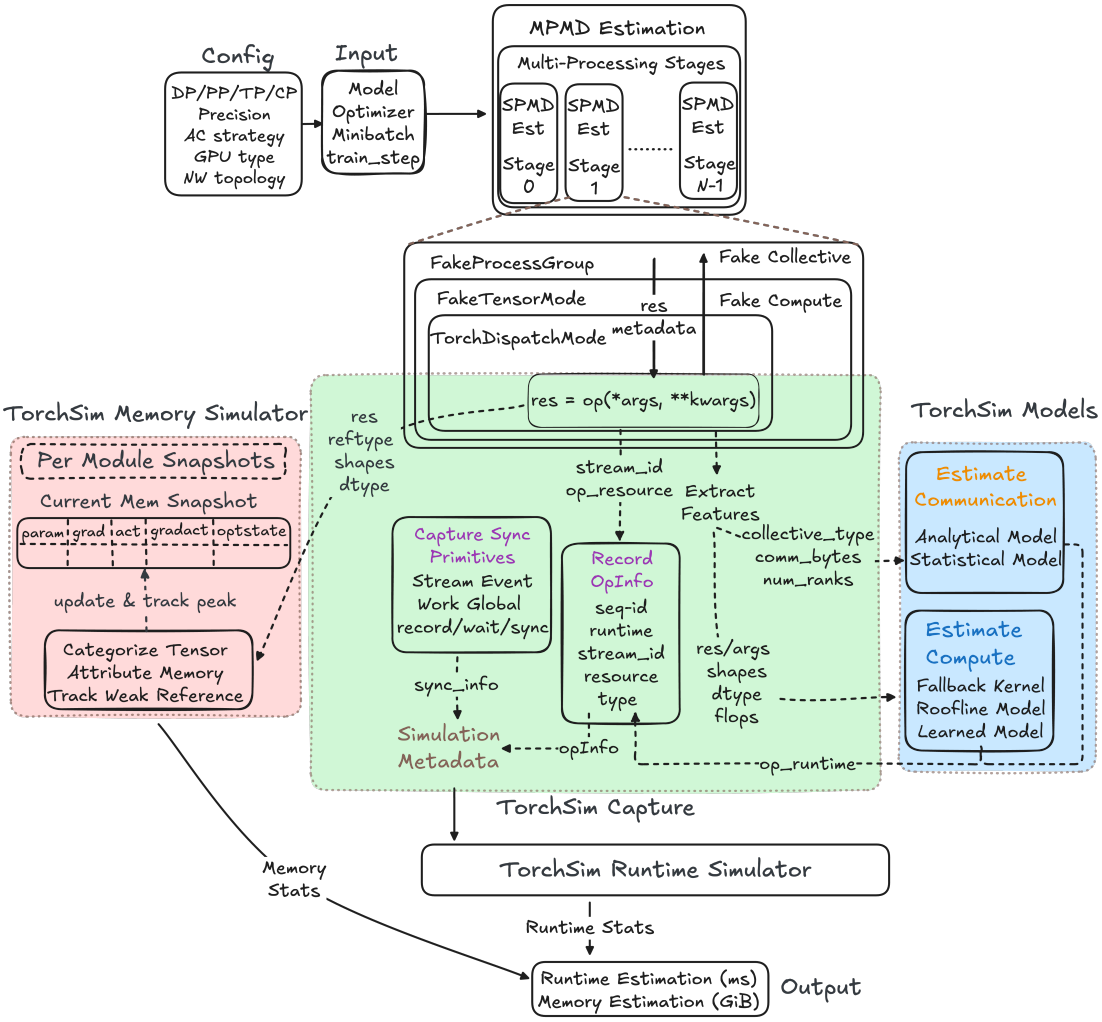


Figure 4.3: TorchSim design internals for capturing tensor, operator, and synchronization primitive metadata to enable precise memory and runtime estimation.

4.2.2 TorchSim Workflow

We now present a detailed walkthrough of TorchSim's end-to-end workflow, as illustrated in Figure 4.3.

1. **Input.** The input to TorchSim is a *train_step* function that receives the model, optimizer, and a sample mini-batch, executing the forward and backward pass, followed by the optimizer step and gradient zeroing.

2. **MPMD/SPMD Estimation.**

For SPMD estimation, TorchSim initiates a single process with a *FakeProcessGroup* and runs execution under *FakeTensorMode*. For MPMD estimation, TorchSim launches N processes with *FakeProcessGroup*, each corresponding to a pipeline stage. Memory estimation is handled independently for each stage, while runtime estimation requires coordination between SPMD processes, managed by the runtime simulator.

3. **Memory Estimation.**

TorchSim estimates memory usage by tracking tensors and their liveness throughout the training workflow. Before executing *train_step*, it extracts tensors from the model's parameters, optimizer states, and inputs. To maintain an accurate view of memory consumption, it dynamically updates a live *current_snapshot* representing the current memory state. Whenever tensor creation, deletion, or resizing occurs, TorchSim updates peak statistics, ensuring that *peak_snapshot* always reflects the maximum observed memory usage. Accurately estimating memory involves tracking tensor liveness, resolving shared storage ambiguities, attributing memory to sources and phases, and categorizing usage, all of which TorchSim addresses in Section 4.3.

4. **Operator Runtime Estimation.**

Compute time is estimated using learned cost models, analytical roofline models, or fallback kernels (Section 4.4), while communication time is predicted using topology- and algorithm-aware

analytical models and statistical learned models (Section 4.5).

5. **Runtime Simulator** The runtime simulator then processes the captured simulation metadata to emulate GPU multi-stream GPU execution model (§ 4.6) to estimate the end-to-end runtime.

6. **Output.** The output includes a module-wise runtime breakdown with estimated compute time, communication time, simulated end-to-end runtime, and exposed communication time (not overlapped with compute). Memory estimation results provide peak memory breakdown and snapshots captured at different stages of a module's execution.

4.3 TorchSim Memory Simulator

Accurate memory estimation involves addressing several critical challenges. How can tensor liveness (creation and deletion) be tracked without interfering with garbage collection? Given that multiple tensors can share underlying storage, how do we prevent over- or under-estimation of memory usage? Beyond identifying peak memory consumption, how can we accurately attribute memory usage to specific sources, determining which module created a tensor and whether the allocation occurred during the forward or backward pass? Additionally, how can memory usage be effectively categorized (e.g., activations, gradients, activation gradients, optimizer states) and quantified per module? Furthermore, how can the impacts of weight, activation, optimizer sharding, prefetching, or distributed training be precisely measured? In this section, we demonstrate how TorchSim addresses these challenges.

We show how Memory Simulator tracks tensor liveness in § 4.3.1 and then explain how it achieves memory attribution and categorization in § 4.3.2. We then deep-dive into the design and implementation of Memory Simulator by introducing the fundamental data structures Memory Simulator uses to maintain statistics in § 4.3.3, followed by elaboration of Memory Simulator's execution flow in §4.3.4.

4.3.1 Tracking Tensor Liveness using *TorchDispatchMode* and *WeakRefs*

While *FakeTensors* allow computation and size estimation without actual data, it does not provide liveness information essential for accurately determining real-time memory usage. Tensors are dynamically created throughout the workflow, explicitly by users (e.g., model initialization), by operations (e.g., matrix multiplication), or implicitly by PyTorch's Autograd engine (e.g., gradients).

Prior to execution, TorchSim extracts metadata for model parameters, optimizer states, and input tensors. During execution, *TorchDispatchMode* robustly intercepts each operation dispatched under its context, retrieving metadata of resulting tensors to track dynamic tensor creations.

Instead of tracking tensor references directly, we monitor references to their underlying storage objects (*UntypedStorage*), as tensors sharing data also share storage objects. To avoid interfering with garbage collection, we utilize *WeakRef* (weak references) Yang (2022); pyt (2025b), allowing storage objects to be collected once no longer in use. *WeakRef* also provides callback capabilities, triggering upon object finalization, enabling precise tracking of memory release and tensor liveness.

4.3.2 Memory attribution and categorization using Module, Tensor, and Optimizer Hooks

While TorchSim is now able to accurately track memory size, which is enough for estimating the peak memory consumption, we haven't attributed the sources of memory consumption. For instance, if a tensor was created, which module created it? To enable memory attribution, we need a couple of features (1) Which modules are active during the execution of an operation? (2) Are we in the backward or forward phase of execution? This essentially boils down to tracking module execution. To track module execution and liveness, we install global module *hooks* on any module that is executed under TorchSim's context. A *hook* is a callback function attached to a specific point in a program's execution flow, allowing custom code to be executed Desmaison (2021a,b);

aut (2025). In particular, we register *pre_fw_hook* to track the beginning and *post_fw_hook* to track the end of the module's forward pass. While PyTorch does provide backward *hooks*, they are tricky to use since they don't work well for *in-place* operations and introduce additional view operations not present in the original execution flow. To circumnavigate this, in the module's *pre_fw_hook* , we install a *multi_grad_hook* on all of the input tensors of the module that require a gradient, this acts as a *post_bw_hook* for the module. Similarly, in the module's *post_fw_hook* we install another *multi_grad_hook* on the output tensors of the module, that acts as a *pre_bw_hook* . Finally, to capture the execution of Optimizer, we install *opt_step_pre_hook* and *opt_step_post_hook* , which capture the phase where the module's parameters are updated using gradients and optimizer states, by setting the flag *in_optimizer*.

Subsequent to memory attribution, the final frontier for TorchSim is memory categorization, which answers questions like, how much activation memory did a module create in the forward pass? Part of memory categorization, especially categorizing tensors as parameters, can be done by enumerating over the parameters of the module in the *pre_fw_hook* . During this phase, we also install *post_accumulate_grad_hook* on the parameters to track their gradients, after they are accumulated in the *grad* attribute of the parameter. All the other tensors generated during the forward pass and retained for the backward pass are categorized as activations, while tensors generated during the backward pass are categorized as temporary memory. To track if we are in backward pass, we query PyTorch Autograd's engine for a non-negative *task_graph_id*. There is one catch, though, during activation checkpointing, the activation tensors are generated in backward. So, how do we mark them as activations? The good news is that the module's *pre_fw_hook* and *post_fw_hook* are called while recomputing the activation tensors in the backward; we make use of this to correctly categorize those tensors as activations in the backward pass.

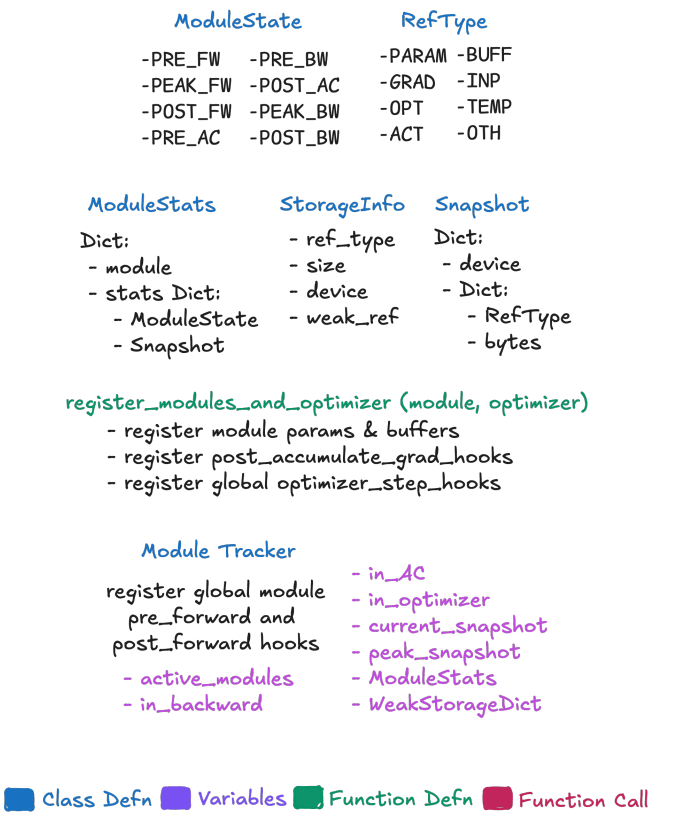


Figure 4.4: Memory Simulator's data structures and functions for estimating, tracking, attributing and categorizing memory usage

4.3.3 Memory Simulator data structures and variables

Memory Simulator uses the following data structures to keep track to memory statistics, module states, tensor and module liveness information, tensor categories, and program states. These are depicted in Figure 4.4.

1. *ModuleTracker*: It is responsible for installing the global *pre_fw_hook* and *post_fw_hook* to track the module execution. It maintains a set of *active_modules* for tracking the currently active modules and a flag *in_backward* to track if we are in the backward phase of execution.
2. *RefType*: Enumeration of tensor memory categories, that are, parameter, buffer, activation, gra-

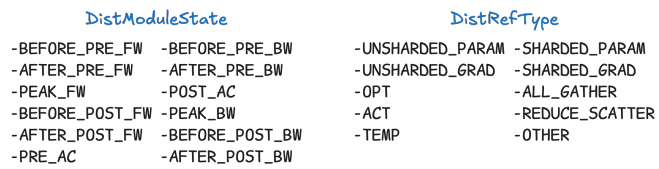


Figure 4.5: Extending *RefTypes* and *ModuleStates* for distributed training

dient, temporary, input, optimize state, or other (user-defined type).

3. *Snapshot*: Snapshot captures the state of memory (occupancy and categorical breakdown) per device. It is a two-level dictionary with *device* as the first-level key and a dictionary as the value. The second-level dictionary is keyed by *RefType* with its value being the amount of memory consumed by each category, essentially the memory breakdown.

4. *ModuleState*: We capture eight module states at different points during the lifetime of its execution, namely, before and after forward, before and after backward, before and after activation checkpointing, and peak memory state during forward and backward.

5. *ModuleStats*: It is a two level dictionary that stores the *Snapshots* for all the modules at each *ModuleState*.

6. *StorageInfo*: This is the primary accounting data structure for storing the metadata of the intercepted tensor's *UntypedStorage*. It preserves the *RefType*, *size, device*, and the *WeakRef* to the original *UntypedStorage* object.

7. *Variables*:

   (a) *in_AC*: A flag that determines if we are in the activation checkpointing region.

   (b) *in_optimizer*: A flag that determines if we are in the optimizer step region.

   (c) *current_snapshot*: Captures and maintains the state of memory at any given point in time.

   (d) *peak_snapshot*: Captures the peak memory state across the training execution.

(e) *WeakStorageDict*: A weak-key dictionary to store references to all the *UntypedStorage* objects alive at any given point in time.

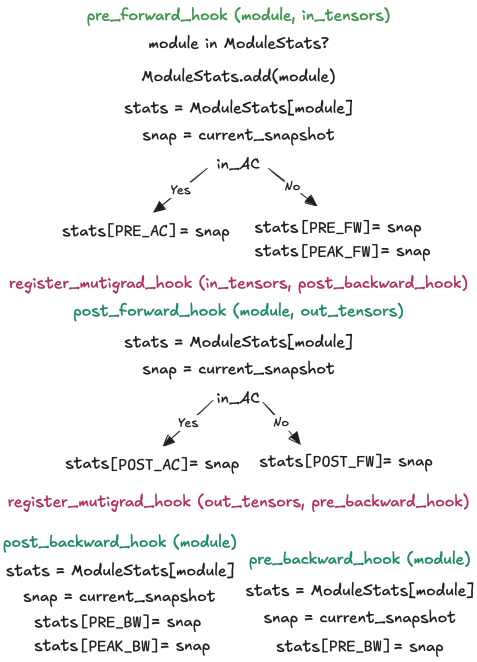
8. *register_modules_and_optimizer*: Registers the module's parameter and buffer storages and installs *post_accumulate_grad_hook* on them. And globally registers *opt_step_pre_hook* and *opt_step_post_hook* for tracking optimizer states.

To extend Memory Simulator's functionality to distributed training workflows, we need to capture additional *ModuleStates*, *RefTypes*. For instance, PyTorch's FSDP Zhao et al. (2023) implementation internally uses *pre_fw_hook* and *pre_bw_hook* for unsharding the parameters using *all_gather* for forward pass computation and gradient computation in the backward pass, respectively. And uses the *post_fw_hook* and *post_bw_hook* to reshard the parameters after the forward computation and gradient computation in the backward pass. Additionally, the gradients are aggregated and sharded in the *post_bw_hook* as well by using *reduce_scatter* operation. Similarly, to capture the state before and after the unsharding and sharding of the parameters and gradients, Memory Simulator extends the *RefTypes* and *ModuleStates* (shown in Figure 4.5) to enable more fine-grained statistics and enrich the estimation and debugging insights.

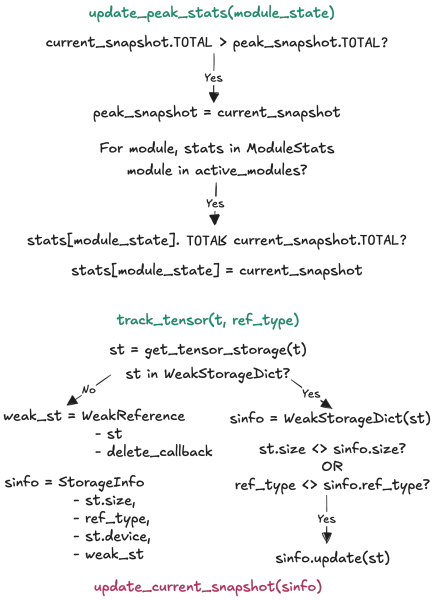
4.3.4 Detailed Memory Simulator execution

We first describe the functionality of Memory Simulator's core functions as shown in Figures 4.6a and 4.6b:

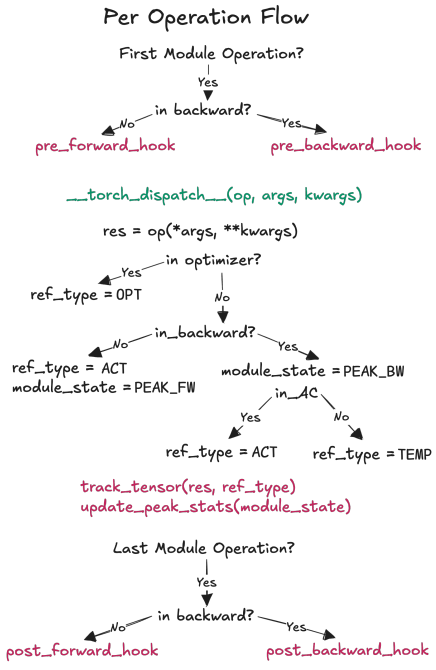
• *Module Hooks*: In addition to tracking the module execution and liveness as explained in § 4.3.2, the module *hooks* serve to capture and initialize *Snapshots* for each *ModuleState*. Specifically, for *pre_fw_hook* and *post_fw_hook* , depending on whether they are called during forward pass or activation checkpointing in the backward pass, *Snapshots* are stored for appropriate *ModuleStates*.



(a) Module and tensor hooks to capture memory snapshots at precise points during execution.



(b) Functions to robustly track Tensor storages and maintain accurate global and per-module peak statistics.



(c) The core per-operation dispatch logic

Figure 4.6: Memory Simulator Workflow

• *track_tensor*: For a given tensor *t* and *ref_type*, it first extracts its storage *st*. If *st* is already being tracked and if its size (due to resize operation) or categorization (due to *hook* trigger) has changed, then its statistics are updated. If it is a new storage to be tracked then a *WeakRef* is created and a *delete_callback* is registered. A new *StorageInfo* object is created and its statistics are populated and tracked in *WeakStorageDict*.

• *update_peak_stats*: Each time the current snapshot is updated, either due to new tensor creation, deletion, or change in its size, the peak statistics of Memory Simulator need to be updated. If the total memory accounted by *current_snapshot* exceeds the *peak_snapshot*, then it is updated to current.

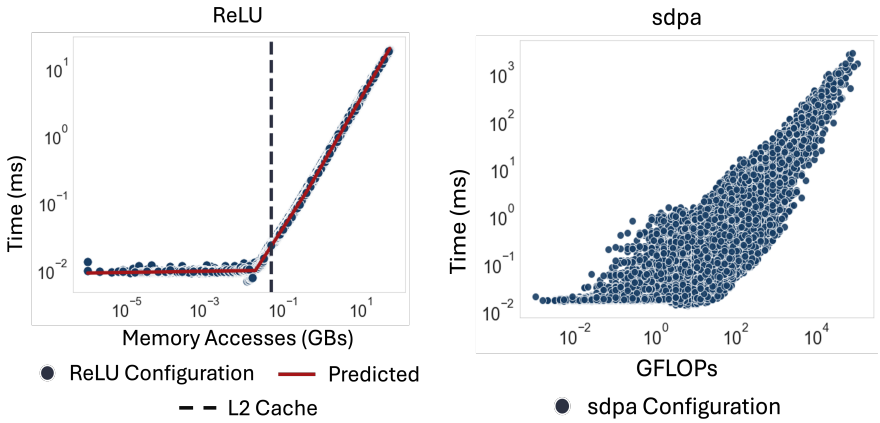
The central execution flow of Memory Simulator is depicted in Figure 4.6c. First, PyTorch's execution engine calls the registered module *hooks* at the start and end of each operation. Just before the operation is dispatched, we intercept it by overriding the *torch.__dispatch__* method of *TorchDispatchMode*. We execute the operation by dispatching the operator with its arguments and obtaining the result. Then, we first check if we are in the optimizer, forward, backward or activation checkpointing region by querying the *in_optimizer, in_backward* and *in_AC* flags respectively and set the correct *ref_type* and *module_state*. For each tensor produced in the result, we call the *track_tensor* function with the set *ref_type*. Finally, we call the *update_peak_stats* function with the set *module_state*.

4.4 TorchSim Compute Time Estimation Models

In this section, we present a comprehensive modeling framework that predicts the runtime of PyTorch neural network operators.*

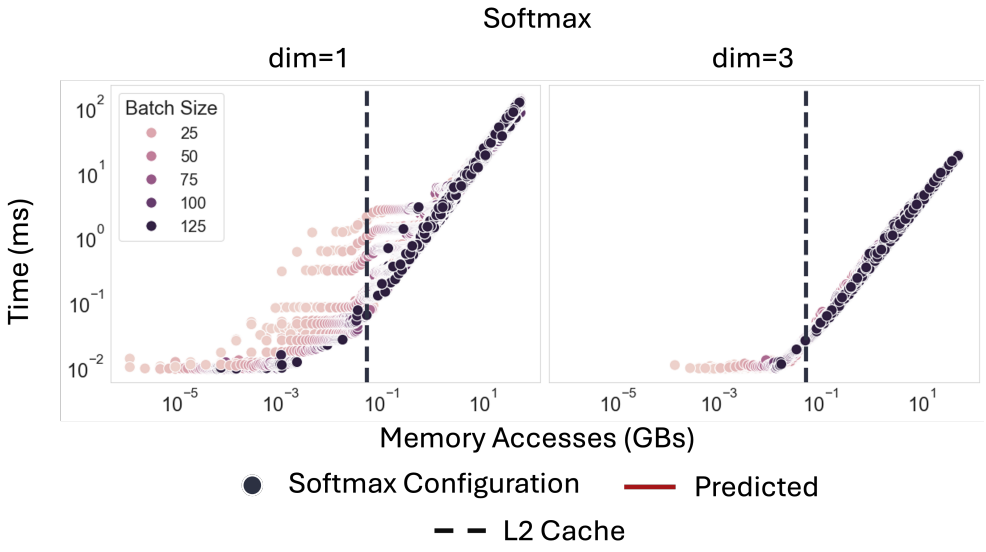
Modeling Methodology. Our predictors model a wide range of operator configurations and runtime behaviors across NVIDIA A100s and H100s and require minimal fine-tuning, limited

*Operator runtime refers to the runtime of the kernel dispatched to compute the operator.



(a) For elementwise operators such as ReLU, their runtimes are piecewise linear in the combined input and output sizes on a log-log scale. We observe the elbow point denotes the transition from using only L2 cache to needing GPU memory.

(b) For compute-bound operators such as sdpa, the runtime varies on different orders of magnitudes even for configurations with the same estimated GFLOPs Ansel et al. (2024a).



(c) For reduction operators such as softmax, their runtimes are affected by striding, including which dimension we take the softmax over, and parallelism, such as the batch size, in addition to memory bandwidth.

Figure 4.7: The Runtimes for Different Operator Categories

domain expertise, and no hardware profiling. For example, for scaled dot-product attention (sdpa), we vary batch sizes, sequence and target lengths, query, key, and value dimensions; if there is a causal mask; and backends (cudnn, efficient, and flash).

Neural Network Operator Categories. Pytorch consists of nearly 2000 neural network operators. Instead of modeling each operator independently, we observe that we can classify them into three categories based on their performance characteristics. Within each class, operators have similar runtime properties, so they can be modeled in the same way. We describe each category and its respective model in detail, along with examples.

Category One: Elementwise Operators. Elementwise operators perform a single computation for each element in the input, e.g., adding two arrays or multiplying two arrays; the performance of elementwise operators is dominated by data movement. As shown in Figure 4.7a, when plotting their runtimes on a log-log scale against the combined input and output sizes, our intuition suggests that the performance can be modeled in three segments. In the first segment, the data size is below the L2 cache capacity of the GPU, so the runtime remains nearly constant. In the second segment, data movement starts saturating memory bandwidth as the data transitions from L2 cache to main memory. In the third segment, the log-log slope approaches one, implying that the runtime scales linearly in the original domain. We formalize this intuition as

$$T(x) = \begin{cases} \exp(a_1 + b_1 \log(x)), & \text{if } x < K_1, \\ \exp(a_2 + b_2 \log(x)), & \text{if } K_1 \leq x < K_2, \\ \exp(a_3 + \log(x)), & \text{if } x \geq K_2, \end{cases}$$

with continuity constraints that enforce smooth transitions:

$$a_2 = a_1 + b_1 \log K_1 - b_2 \log K_1 \quad \text{and} \quad a_3 = a_2 + b_2 \log K_2 - \log K_2.$$

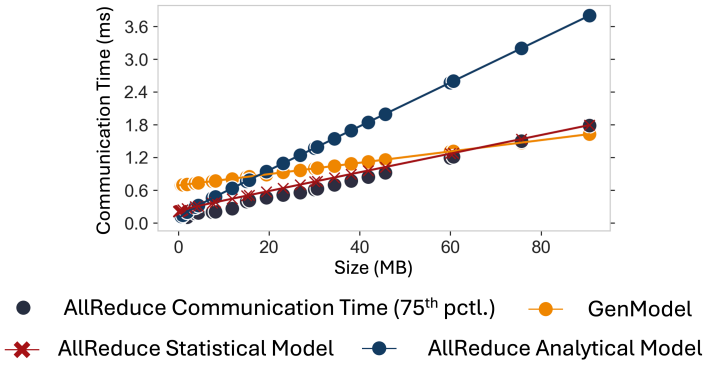
For example, operators in this group include ReLU, cosine, and sine.

Category Two: Reduction Operators. Unlike elementwise operators, reduction operators perform several passes over the data, so they operate in parallel across multiple GPU cores; examples include sum, mean, min, max, and softmax. As shown in Figure 7b, their runtime is affected by parallelism and stride, such as the dimension used. These factors introduce deviations that we model using lightweight learned predictors (a decision tree), with one model for each operator and hardware combination.

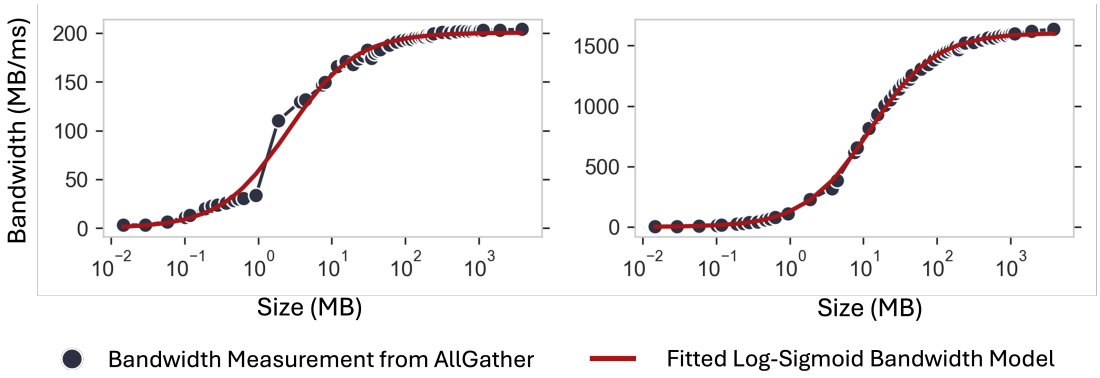
Category Three: Compute-Bound Operators. Unlike memory-bound operators, whose runtime is mainly affected by memory bandwidth and data size, compute-bound operators' runtimes are more hardware sensitive. In our experiments, we consider matrix multiplication (mm), batch matrix multiplication (bmm), scaled dot-product attention (sdpa), and 2D convolution (conv2d). For example, sdpa computes attention using matrix multiplications and softmax; although we expect multiplying two long and skinny matrices to take longer than multiplying two square matrices with the same arithmetic intensity due to cache-friendly tiling, as shown in Figure 4.7b, it is unclear how exactly this affects runtime. Therefore, for each operator and hardware combination, we train a random forest using only operator-level features because they are nonparametric, capture nonlinear interactions, and adapt well to different inductive biases, in that not only do the factors driving the runtime of small configurations differ from those for larger ones, noting that we cover runtimes from 10^{-2} to 10^{6} milliseconds, but also different hardware architectures can affect kernel behavior.

4.5 TorchSim Communication Time Estimation Models

In this section, we present the TorchSim models for predicting communication time. Our models capture the heterogeneous link bandwidths of multi-node network topologies with both inter-node and intra-node interconnects, as well as straggler delay.



(a) The discrepancy between the AllReduce analytical model and the benchmarking data, as well as that between the GenModel prediction and the ground truth, demonstrates the necessity for a topology-aware learned model to capture nonlinearities in the communication time at small data sizes and straggler delay.



(b) Bandwidth measurements from cluster X with inter-node and intra-node AllGather measurements illustrate the log-sigmoidal form with respect to data size and the order-of-magnitude difference between inter- and intra-node bandwidths.

Figure 4.8: Insights from benchmarking inter-GPU communication

Modeling Inter-Node and Intra-Node Bandwidth. While existing cost models, including those used by NCCL and GenModel, use a single bandwidth measurement for modeling communication time, we empirically show that bandwidth varies as a function of data size. An example is shown in Figure 4.8b. Furthermore, inter-node connectivity typically uses Ethernet or Infiniband, while intra-node communication is enabled by NVLink, PCIe, and other higher-speed interconnects. Thus, we fit **separate log-sigmoidal curves** for inter-node and intra-node segments of collective communication as shown in Figure 4.8b, enabling our learned model to handle heterogeneous topologies with different link bandwidths across the cluster.

The ground truth data for fitting the bandwidth function is collected by performing AllGather collectives across a range of data sizes between the GPUs and nodes in the cluster. We then fit a sigmoid curve (as shown in Figure 4.8b) to the intra-node and inter-node measurements separately:

$$\sigma_{\{C,T\}}(D) = \frac{L}{1 + \exp(-k \cdot (\log(D+1) - D_0))} + b \tag{4.1}$$

where D is the data size. The bandwidth functions $\sigma_{\{C,T\}}$ are thus sigmoidal with data size on the log scale, where σ_C denotes intra-node curve and σ_T denotes inter-node curve.

Since we use $B(D_S)$ in the denominator of the time equation in the AllReduce model in the next section, in order to avoid undesirable discontinuities for positive D_S, we constrain the parameters in the following way:

$$b \in (-\infty, -1] \quad L, k, D_0 \in (-\infty, \infty). \tag{4.2}$$

Modeling Communication Time. Given the bandwidth model, we can now create a topology-aware and algorithm-aware analytical cost model for communication collectives. We provide models for three commonly-used collectives in distributed training: AllReduce, AllGather, and ReduceScatter.

The key technical insight behind these closed-form cost models is that NCCL uses two differ-

Table 4.4: List of Symbols for Communication Modeling

Symbol	Description
N	Number of processors
S	Data size per processor
P	Total number of processes
$B_T(S)$	Inter-node bandwidth as a function of data size
$B_C(S)$	Intra-node bandwidth as a function of data size
ℓ_T	Inter-node latency
ℓ_C	Intra-node latency
ℓ	Total latency

Table 4.5: Analytical Cost Models for Collective Communication Algorithms

Collective	Analytical Model
AllReduce	$\frac{2S}{B_T(D_S)} + \frac{2S}{B_C(D_S)}$ $+(\lfloor \log_2(N) \rfloor + 1) \cdot \ell_T + (P/N - 1) \cdot \ell_C$
AllGather	$\frac{(P-N-1) \cdot S}{(P-N)\, B_C(S)} + \frac{(N-1) \cdot S}{N B_T(S)}$ $+N \cdot \ell_T + (P/N - 1) \cdot \ell_C$
ReduceScatter	$\frac{(P-N-1) \cdot S}{(P-N)\, B_C(S)} + \frac{(N-1) \cdot S}{N B_T(S)}$ $+N \cdot \ell_T + (P/N - 1) \cdot \ell_C$

ent algorithms to execute collectives: a chain or a double binary (or two-tree) tree. NCCL uses the double binary tree for **inter-node** segments of the AllReduce operation and the chain for **intra-node** segments of AllReduce. The chain is used for all parts of the AllGather and ReduceScatter algorithms.

The analytical cost models are derived by counting the number of reduction and assignment operations performed across ranks on the same node and between nodes for each collective. The cost model expressions are shown in Table 4.5.

We now show a detailed derivation of the analytical cost models. For some operation over S elements with l links of bandwidth $B(D)$, the time of the operation is

$$t(S) = \frac{S}{l \cdot B(D_S)}. \tag{4.3}$$

where D_S is the data size for an array of S elements.

AllReduce. An AllReduce operation over n ranks requires $n-1$ additions and n assignments. Each of these addition/assignments steps requires a data transfer between two different ranks, except for the first assignment since the rank of the first assignment already performed the last addition and thus already has the complete summation. Therefore, the total communication time for an AllReduce operation is

$$t_{\text{AllReduce}}(S) = \frac{2(n-1) \cdot S}{l \cdot B(D_S)} \tag{4.4}$$

where $B(D_S)$ is the bandwidth function.

We can also decompose the AllReduce time into inter-node communication time (t_T) and intra-node communication time (t_C). Separating these two terms allows us to model each communication time using its own bandwidth function.

Suppose our topology consists of P GPUs evenly distributed across N nodes. Thus, the inter-

node communication is performed over N ranks with $N-1$ links of bandwidth (since there are $N-1$ edges in each binary tree), whereas the intra-node communication is performed over $P-(N-1)$ ranks with $N(P/N-1) = P-N$ links of bandwidth (since each intra-node chain has $P/N-1$ links). Then, the communication times are

$$t_T(S) = \frac{2(N-1)\cdot S}{(N-1)\cdot B_T(D_S)} = \frac{2S}{B_T(D_S)} \tag{4.5}$$

$$t_C(S) = \frac{2(P-N)\cdot S}{(P-N)\cdot B_C(D_S)} = \frac{2S}{B_C(D_S)} \tag{4.6}$$

Thus, the sum of the inter-node communication time and the intra-node communication times are

$$t_T(S) + t_C(S) = \frac{2S}{B_T(D_S)} + \frac{2S}{B_C(D_S)} \tag{4.7}$$

The total AllReduce time is thus the summation above, plus the communication latency ℓ.

$$t_{\text{AllReduce}}(S) = t_T(S) + t_C(S) + \ell \tag{4.8}$$

where

$$\ell = (\lfloor \log_2(N) \rfloor + 1)\cdot \ell_T + (P/N-1)\cdot \ell_C$$

ℓ_T is the inter-node latency and ℓ_C is the intra-node latency.

The logarithmic term of the startup latency follows from the height of the binary tree for inter-node communication, while the linear term follows from the number of links in the chain for intra-node communication.

The models above account for both inter-node and intra-node communication. In the case of 2D parallelism, when there is only inter-node communication, we simply only include the intra-node terms of the analytical model to predict the communication time.

AllGather and ReduceScatter. The AllGather and ReduceScatter operation can be thought of as simply the assignment and addition portions of the AllReduce model, respectively. However, all other NCCL operations other than AllReduce use chains, rather than trees, to connect nodes. Thus, we have the following analytical models for these two collectives:

$$t_{ReduceScatter}(S) = \frac{(P-N-1)\cdot S}{(P-N)\,B_C(S)} + \frac{(N-1)\cdot S}{N B_R(S)} + \ell \tag{4.9}$$

where

$$\ell = N \cdot \text{inter-node latency} + (P/N-1)\cdot \text{intra-node latency}.$$

and

$$t_{AllGather} = \frac{(P-N-1)\cdot S}{(P-N)\,B_C(S)} + \frac{(N-1)\cdot S}{N B_R(S)} + \ell \tag{4.10}$$

where

$$\ell = N \cdot \text{inter-node latency} + (P/N-1)\cdot \text{intra-node latency}.$$

As with AllReduce, the intra-node terms of the cost models above are omitted when considering 2D, inter-node only parallelism.

Modeling Variability in Minimum Completion Time. While the analytical models account for inter-node and intra-node communication, we consider them as explaining the *minimum* time required for a communication collective to complete. However, it is important that a predictive model capture any *straggler delay*, not just a lower bound on the communication time. Furthermore, there is still variability in the minimum completion time of the communication algorithm across ranks that can be explained by the world size and data size, but is not yet captured by the analytical model.

To address this, we turn to a statistical approach to modeling collective communication time and straggler delay. We first use the following linear model to predict the minimum communication

time across ranks for each collective.

$$\begin{aligned}\hat{t}_{\min} \sim \beta_0 &+ \beta_1 \cdot t + \beta_2 \cdot \ell + \beta_3 \cdot P \\ &+ \beta_3 * S + \beta_4 \cdot (t \cdot S) \\ &+ \beta_5 \cdot (t \cdot P) + \beta_6 \cdot (\ell \cdot S)\end{aligned} \tag{4.11}$$

where the symbols are as defined in table 4.4, the β_i terms are coefficients fitted by ordinary least-squares regression, and t is the collective time predicted by the analytical model. The terms in which the coefficient corresponds to the product of two variables refers to an *interaction term*, which allows the linear relationships between the communication time, the analytical model, and other variables in the model to be adjusted based on data size and world size.

For the small data sizes ($O(10\text{ KB})$), the fitted linear model can sometimes yield negative predictions. In these cases, we adopt an adaptive strategy where we fall back to the analytical model:

$$\hat{t}_{min} = \begin{cases} \hat{t}_{min} & \hat{t}_{min} > 0 \\ t & \text{otherwise} \end{cases} \tag{4.12}$$

Modeling Straggler Delay. On top of this statistical model, we then use a second linear model to predict the straggler delay ratio, defined as the ratio of the 75th percentile of communication time across ranks t_{75} to the minimum communication time $t_{\min}$.

$$\widehat{t_{75}/t_{\min}} \sim P \cdot \log(S) \tag{4.13}$$

After fitting these models on collective communication benchmarking data taken at a variety of data sizes and world sizes, we can then derive a statistical estimate of the straggler-included communication time as $\hat{t}_{\min} \cdot \widehat{t_{75}/t_{\min}}$.

Layering these analytical and statistical techniques enables our communication models to predict collective latency at high precision across data sizes and world sizes, **estimating communication time within an RMSE of 3 ms** across collectives and 1D/2D parallelism on two clusters, and achieving up to a **6.8× improvement over GenModel** for predicting 2D AllReduce.

Adaptability to heterogeneous clusters. The proposed communication models account for network topologies with heterogeneous link bandwidths by separately considering inter-node and intra-node bandwidths and steps in the algorithms used for collective communication. This formulation assumes a two-tier topology, with slower inter-node bandwidths and faster intra-node bandwidths, and collective algorithms that operate over such topologies. However, this model is easily extensible to topologies with three or more types of interconnects and collective algorithms that communicate over such topologies by fitting additional nonlinear functions to their bandwidths to the respective benchmarking data, adding corresponding terms to the analytical models, and re-fitting the statistical models. Thus, our approach provides an easily-adaptable framework for modeling collective communication time.

Extensibility to unseen clusters. If an already-fitted communication model needs to be redeployed to a different, unseen cluster, one of two approaches can be taken for adapting the existing model for a new communication time estimation. If the cluster provider is able to re-benchmark the bandwidth and collective communication times offline, then adaptability is simply a matter of re-fitting the bandwidth and statistical collective models. If benchmarking data is entirely or partially unavailable, we propose an *online learning* approach to adjusting the models with data while training a model with a distributed parallelism strategy.

Suppose the weights of the regression model are initialized to $\boldsymbol{\beta}$. During a training run on cluster with N nodes and P GPUs, communication collective times $\mathbf{y}$ can be recorded. We can then compute new weights $\boldsymbol{\beta}'$ as follows:

$$\beta' = \beta - \eta(\beta \cdot \mathbf{x} - \mathbf{y}) \tag{4.14}$$

for known parameters $\mathbf{x} = (N, P, S, D_S)$ as defined above and learning rate η.

As the weights β are updated, the communication model adapt to the properties of the new cluster without a full offline benchmark.

4.6 TorchSim Runtime Simulator

We now explain how the Runtime Simulator estimates the end-to-end runtime by emulating the multi-stream GPU execution on the captured simulation metadata. Table 4.2 outlines the Queue and Runtime Simulator class definitions. We describe the algorithms for Queue management in § 4.6.1, followed by the Simulation loop in § 4.6.2.

4.6.1 Queue State Management and Resource Allocation

Algorithm 1 prepares each CUDA stream queue for simulation by setting up its initial state and synchronization conditions. It iterates over all queues and checks for any synchronization events registered under the special key `-1`, which represents pre-operation dependencies. Depending on the type of synchronization action—such as stream, event, work, or global synchronize wait—it adds the corresponding condition to the queue's wait set. Finally, it sets the state of the queue to `WAITING`, indicating that it is blocked until its dependencies are resolved.

Algorithm 2 identifies queues that are currently in the `WAITING` state but have no remaining synchronization conditions and updates their state to `READY`. This ensures that queues are able to proceed with execution as soon as all their dependencies have been cleared, which is a key part of dynamic dependency resolution in the simulator.

Algorithm 1 Initialize Queue States

Require: None.
Ensure: Every queue is initialized with proper waiting conditions.

for all each queue `Q` in `streamid_to_queue` **do**
  if `Q.sync_infos` contains entries under key `-1` **then**
    for all each sync event `syncInfo` in `Q.sync_infos[-1]` **do**
      if `syncInfo.sync_action` is `STREAM_WAIT` **then**
        Add entry with key (`STREAM_WAIT`, `syncInfo.release_seq_id`) to `Q.wait_sync_infos`.
      else if `syncInfo.sync_action` is `EVENT_WAIT` **then**
        Add entry with key (`EVENT_WAIT`, `syncInfo.release_event_id`) to `Q.wait_sync_infos`.
      else if `syncInfo.sync_action` is `WORK_WAIT` **then**
        Add entry with key (`WORK_WAIT`, `syncInfo.release_seq_id`) to `Q.wait_sync_infos`.
      else if `syncInfo.sync_action` is `SYNCHRONIZE_WAIT` **then**
        Add entry with key (`SYNCHRONIZE_WAIT`, `syncInfo.sync_id`) to `Q.wait_sync_infos`.
      else
        raise error "Unknown sync action with key -1".
      end if
    end for
  end if
  Set `Q.state` ← `WAITING`.
end for

Algorithm 2 _maybe_resolve_waiting_queues

Ensure: Mark each queue as `READY` if it is `WAITING` and has no pending wait conditions.

for all each queue `Q` in `streamid_to_queue` **do**
  if `Q.state` is `WAITING` and `Q.wait_sync_infos` is empty **then**
    Set `Q.state` ← `READY`.
  end if
end for

Algorithm 3 simulates global synchronization mechanisms by incrementally releasing queues that are blocked on a global synchronize event. It continues to iterate while all queues are in the `WAITING` state, increasing a global synchronization counter each round. For each queue, it removes any pending synchronization condition that matches the current round's global sync identifier. When a queue has no remaining synchronization dependencies, it is marked as `READY`. The algorithm guarantees forward progress and avoids deadlock by asserting that at least one sync condition must be cleared in each round.

Algorithm 3 _maybe_resolve_global_sync

Ensure: Resolve global synchronization waits until at least one queue is `READY`.

while all queues in `streamid_to_queue` are in `WAITING` state **do**
    Increment global counter `simulate_sync_count`.
    for all each queue `Q` in `streamid_to_queue` **do**
        Let `key` ← (`SYNCHRONIZE_WAIT`, `simulate_sync_count`).
        if `Q.wait_sync_infos` contains `key` **then**
            Remove `key` from `Q.wait_sync_infos`.
            if `Q.wait_sync_infos` is now empty **then**
                Set `Q.state` ← `READY`.
            end if
        else
            assert that a sync condition exists (else, deadlock).
        end if
    end for
end while

Algorithm 4 checks whether all queues have completed their work. A queue is considered complete if it is in the `READY` state, has no remaining operations, and no pending or unresolved synchronization events. If all queues meet these criteria, the function returns `True`; otherwise, it returns `False`. This serves as the termination condition for the simulation loop.

Algorithm 5 performs resource scheduling by allocating available hardware resources to operations in `READY` queues. It filters the list of queues to only those that are `READY` and have pending

Algorithm 4 Check All Queues Completed

Ensure: Return True if every queue is `COMPLETE`.

for all each queue `Q` in `streamid_to_queue` **do**
    if `Q.state` is `READY` and `Q.ops` is empty and both `Q.sync_infos` and `Q.wait_sync_infos` are empty **then**
        Set `Q.state` ← `COMPLETE`.
    end if
end for
return True if every queue in `streamid_to_queue` has state `COMPLETE`; otherwise, False.

operations. After sorting them by priority and sequence ID, it checks if the operation's required resources are available. If they are, it assigns those resources, marks the queue as `RUNNING`, and updates the resource occupancy table. This ensures fair and efficient use of limited compute and communication resources across multiple queues.

Algorithm 5 Allocate Resources

Ensure: Allocate available resources to operations in `READY` queues.

Identify all queues in state `READY` with pending operations; denote as `Q_ready`.
assert: Each queue in `Q_ready` has at least one op.
Sort `Q_ready` by (priority, sequence ID of the first op).
for all each queue `Q` in `Q_ready` **do**
    Let `op` ← first element in `Q.ops`.
    if any resource in `op.resources` is already occupied (exists in `_resource_occupancy`) **then**
        continue to next queue.
    end if
    for all each resource in `op.resources` **do**
        Update `_resource_occupancy` to map the resource to `Q`.
    end for
    Set `Q.state` ← `RUNNING`.
end for

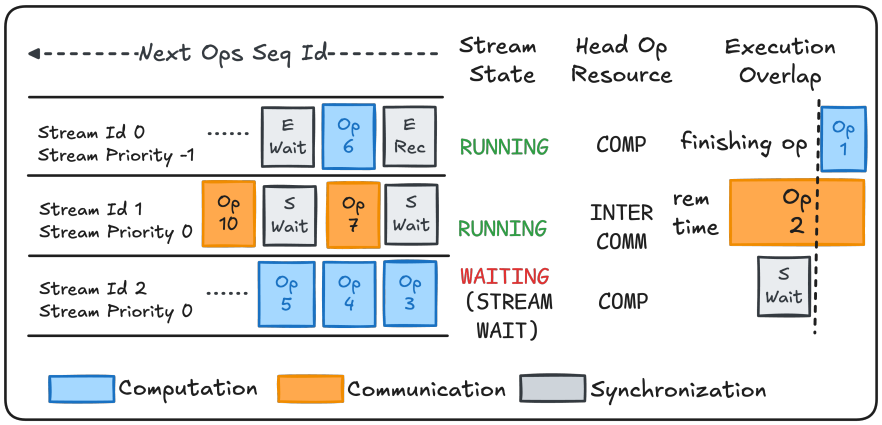


Figure 4.9: TorchSim Runtime Simulator in action.

4.6.2 Simulation Loop

The main simulation loop, outlined in Algorithm 6, advances simulated time and models the execution of all queues until every queue is marked as `COMPLETE`. At each iteration, it determines the minimum remaining execution time among all head operations, advances time by that amount, and updates queue states and resource allocations accordingly. When an operation finishes, it is removed from the queue, its resources are released, and relevant synchronization events are processed. The loop also invokes helper functions to resolve synchronization and queue readiness. The total simulated time is returned at the end. Figure 4.9 illustrates the runtime simulator in action.

Table 4.6 presents the algorithm for processing the synchronization events. It handles all synchronization events associated with a completed operation. Depending on the event type, such as event release/wait, stream release/wait, work release/wait, or global synchronize wait, it updates the global state or other queues' wait conditions accordingly. For release events, it notifies dependent queues so they can proceed; for wait events, it adds new dependencies if the required condition is not yet satisfied. This enables correct modeling of inter-operation dependencies and synchronization across streams.

Algorithm 6 Simulate

Ensure: Total simulated time `T_sim`.

// Pre-check: Verify that all waited events are recorded.
Initialize simulation time: `T_sim` ← 0.
Initialize simulation state:
    `_resource_occupancy` ← empty,
    `_completed_ops` ← empty set,
    `_recorded_events` ← empty set.
Call `Initialize Queue States` to set all queues to `WAITING`.
while not all queues are `COMPLETE` **do**
    Call `Allocate Resources` to mark READY queues as RUNNING.
    Let `resource_independent_queues` be the set of queues currently occupying resources.
    Let `head_ops` be the first operation from each queue in `resource_independent_queues`.
    Determine `min_rem_time` as the minimum `op.rem_time` among `head_ops`.
    Set `Delta_t` ← `min_rem_time`.
    Update simulation time: `T_sim` ← `T_sim` + `Delta_t`.
    for all each head operation `op` in `head_ops` **do**
        Decrease `op.rem_time` by `Delta_t`.
        if `op.rem_time` equals 0 **then**
            Let `Q` be the queue corresponding to `op.stream_id`.
            Remove `op` from `Q.ops`.
            Set `Q.state` ← `READY`.
            for all each resource in `op.resources` **do**
                Remove the resource from `_resource_occupancy`.
            end for
            Add `op.seq_id` to global set `_completed_ops`.
            Call `Process_Sync_Events` with `Q` and `op`.
        end if
    end for
    Call `_maybe_resolve_waiting_queues()`.
    Call `_maybe_resolve_global_sync()`.
end while
return `T_sim`.

SyncAction	Simulator Action
EVENT_RELEASE	Add syncInfo.release_event_id to recorded_events For each queue Q' in streamid_to_queue: If Q'.state is WAITING and Q'.wait_sync_infos contains key (EVENT_WAIT, syncInfo.release_event_id), remove that key.
EVENT_WAIT	If syncInfo.release_event_id $\notin$ recorded_events: Add key (EVENT_WAIT, release_event_id) to Q.wait_sync_infos Set Q.state to WAITING.
STREAM_RELEASE	For each queue Q' in streamid_to_queue: If Q'.state is WAITING and Q'.wait_sync_infos contains key (STREAM_WAIT, syncInfo.release_seq_id), remove that key.
STREAM_WAIT	If syncInfo.release_seq_id $\notin$ completed_ops: Add key (STREAM_WAIT, release_seq_id) to Q.wait_sync_infos Set Q.state to WAITING.
WORK_RELEASE	For each queue Q' in streamid_to_queue: If Q'.state is WAITING and Q'.wait_sync_infos contains key (WORK_WAIT, release_seq_id), remove that key.
WORK_WAIT	If syncInfo.release_seq_id $\notin$ completed_ops: Add key (WORK_WAIT, release_seq_id) to Q.wait_sync_infos Set Q.state to WAITING.
SYNC_WAIT	Add key (SYNC_WAIT, sync_id) to Q.wait_sync_infos Set Q.state to WAITING.

Table 4.6: Process_Sync_Events for a given syncInfo and Q

	Train			Test		
Operator	**RMSE (ms)**	**MAPE (%)**	**ACC (%)**	**RMSE (ms)**	**MAPE (%)**	**ACC (%)**
mm	0.17	1.25	99.7	0.44	2.8	95.7
bmm	1.07	1.47	99.32	2.62	3.3	94.1
sdpa	1.31	0.65	99.8	2.58	1.77	97.5
sdpa backward	2.41	0.8	99.7	5.90	2.13	96.9
conv2d	12.89	4.20	92.9	40.3	11.2	81.0
conv2d backward	48.68	4.08	92.5	125.5	10.96	74.8

Table 4.7: Regression and Accuracy Results for the Learned Compute-Bound Model on H100s

	Train			Test		
Operator	**RMSE (ms)**	**MAPE (%)**	**Accuracy (%)**	**RMSE (ms)**	**MAPE (%)**	**Accuracy (%)**
mm	0.27	1.2	99.7	0.71	2.51	97.7
bmm	0.53	0.9	99.5	1.34	1.76	98.1
sdpa	1.72	0.76	99.6	3.81	2.1	96.6
sdpa backward	3.64	1.25	99.3	9.64	3.32	93.5
conv2d	28.5	4.93	89.93	52.02	9.46	77.5
conv2d backward	105	5.33	88.73	187.6	10.5	74.6

Table 4.8: Regression and Accuracy Results for the Learned Model on A100s.

4.7 Experimental Results

4.7.1 Compute Time Prediction

This section describes how we fitted the learned models for predicting operator runtime using benchmarking data collected for NVIDIA A100s and H100s. We ran two warmup iterations for each operator configuration, and took the median runtime of five iterations. To evaluate our models' performances, we reserve 15% of each dataset as the test set to get root mean squared error (RMSE), mean absolute percentage error (MAPE), and $\pm 10\%$ accuracy.

As shown in Table 4.7 and Table 4.8, we find that our models have good test evaluation results.

In Figures 4.10 and 4.11, we see that most of the differences between predicted time and mea-

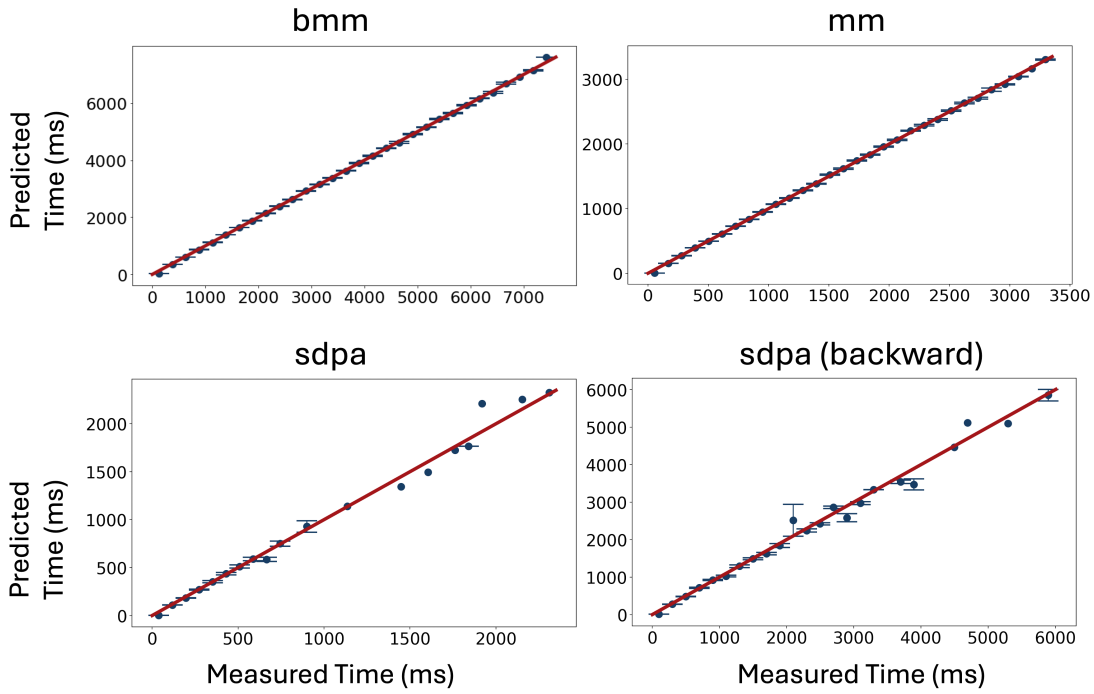


Figure 4.10: The predicted runtime plotted against the measured runtime for the operators on NVIDIA A100s. The red line denotes the line $y = x$. We binned measured runtimes and took a mean per bin.

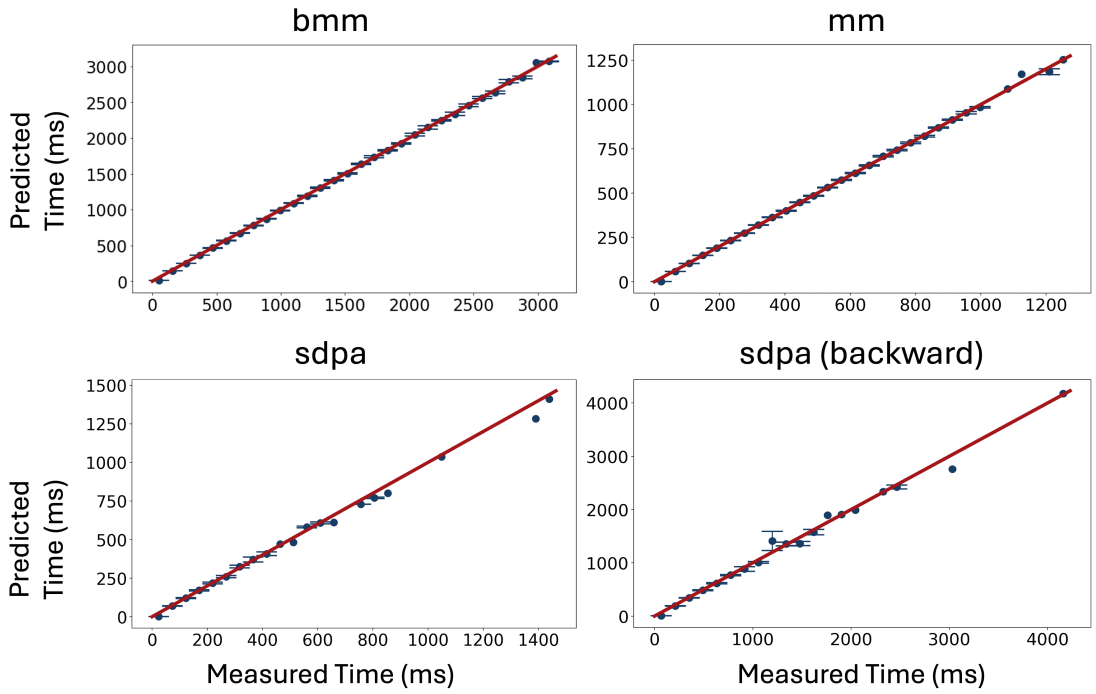


Figure 4.11: The predicted runtime plotted against the measured runtime for the operators on NVIDIA H100s. The red line denotes the line $y = x$. We binned measured runtimes and took a mean per bin.

sured time happen when the measured time is large. This implies that our learned model can predict the runtime of the operators with low RMSE and MAPE and high accuracy, with only access to operator-level features.

4.7.2 Communication Time Prediction

The analytical and statistical models for predicting collective communiation time were fitted and evaluated with benchmarking data collected on cluster X, with NVIDIA H100 GPUs arranged with 4 GPUs per server connected by Infiniband interconnects, and cluster Y, with H100 GPUs with 8 GPUs per server and RoCE support. We benchmarked the AllReduce, AllGather, and ReduceScatter collectives on a series of world sizes ranging from 4 to 32 nodes (16 to 128 GPUs) on cluster X and from 4 to 64 nodes (32 to 512 GPUs) on cluster Y, and data sizes ranging from 15KB to 4GB. The benchmarking experiment for each data size and world size was run 10 times, after a small number of warmup iterations. Furthermore, we benchmarked each collective in two settings: inter-node *and* intra-node communication (1D parallelism) and inter-node only communication (2D parallelism). Including both settings in a predictive model for communication time is relevant for ensuring that our models are performant in both DP-only (1D) configurations and in setups where both DP and TP are used along different dimensions of the topology.

For comparison, we use the GenModel communication model as a baseline for AllReduce latency predictions in 2D parallelism settings. Since the co-located PS benchmarking toolkit used to fit GenModel is not publically available, we approximate the model by using the Recursive Halving-Doubling formula for inter-node only AllReduce, with the learned constants $\alpha, \beta, \gamma, \delta$ in the expression fitted with least-squares regression. We demonstrate that, for predicting AllReduce in 2D parallelism settings, our model demonstrates up to $6\times$ of improvement in RMSE for predicting collective latency compared to GenModel. A full table of model performance statistics is shown in Table 4.9.

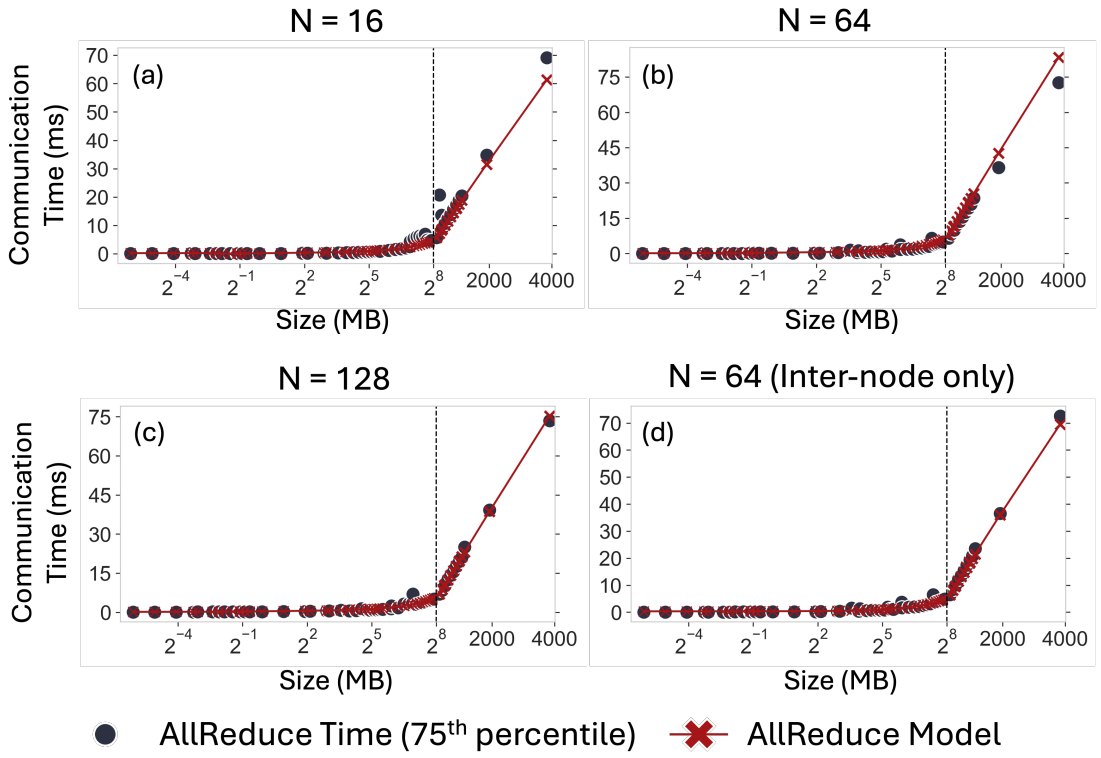


Figure 4.12: 75th percentile AllReduce communication time from benchmarking data and predictions by the collective communication prediction model for each data size on cluster X. (a)–(c) show data from world sizes of $N = 16, 64, 128$ for 1D parallelism whereas (d) shows data for 2D parallelism for $N = 64$.

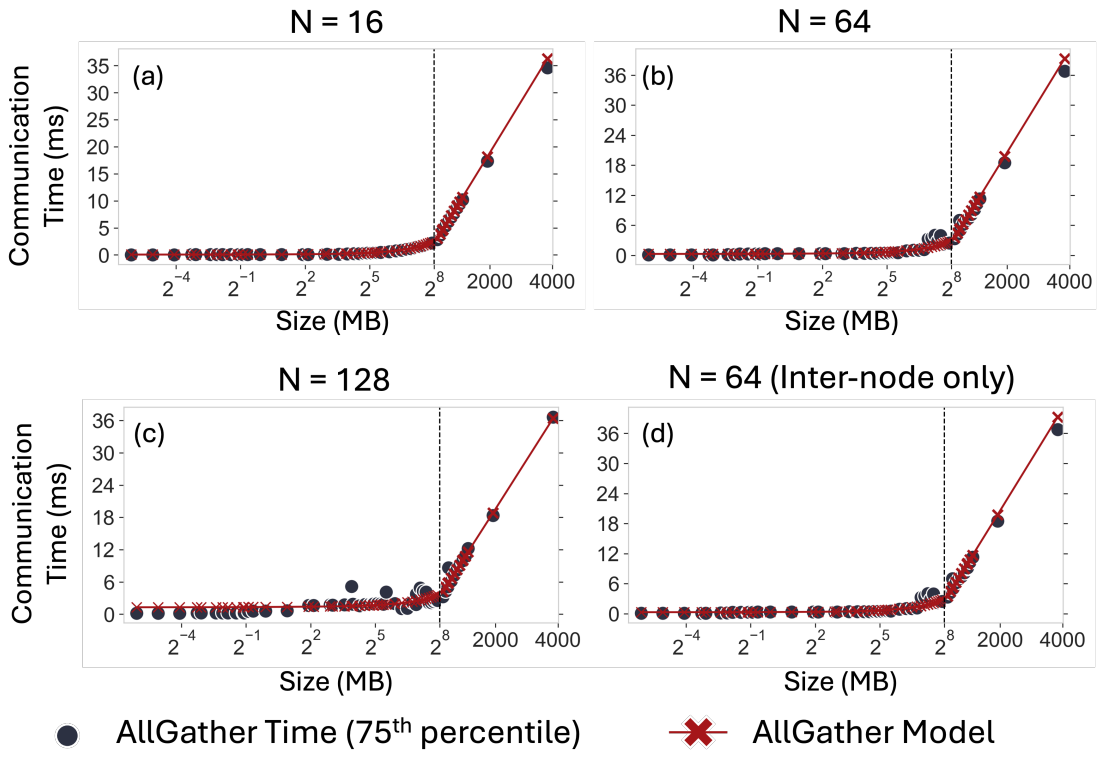


Figure 4.13: 75th percentile AllGather communication time from benchmarking data and predictions by the collective communication prediction model for each data size on cluster X. Data size refers to the size of the output tensor in the AllGather operation, since the communication overhead scales with the output dimensions. (a)–(c) show data from world sizes of $N = 16, 64, 128$ for 1D parallelism whereas (d) shows data for 2D parallelism for $N = 64$.

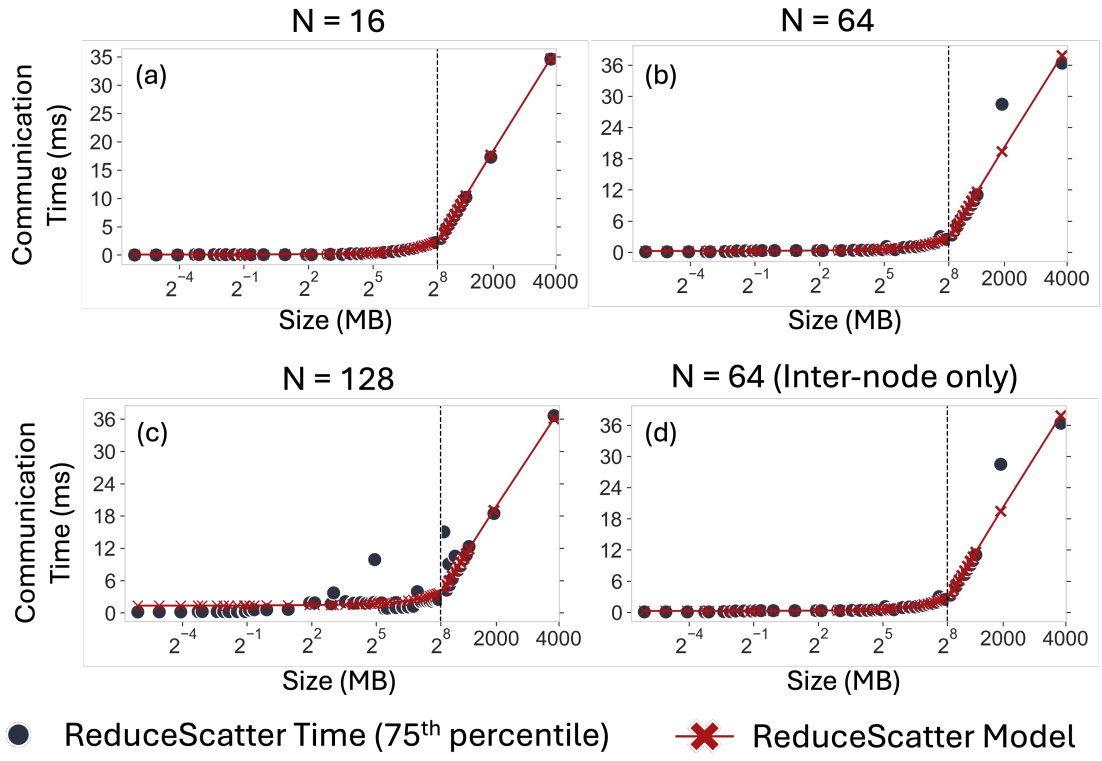


Figure 4.14: 75th percentile ReduceScatter communication time from benchmarking data and predictions by the collective communication prediction model for each data size on cluster X, with the same log-linear scale as Figure 4.12. (a)-(c) show data from world sizes of $N = 16, 64, 128$ for 1D parallelism whereas (d) shows data for 2D parallelism for $N = 64$.

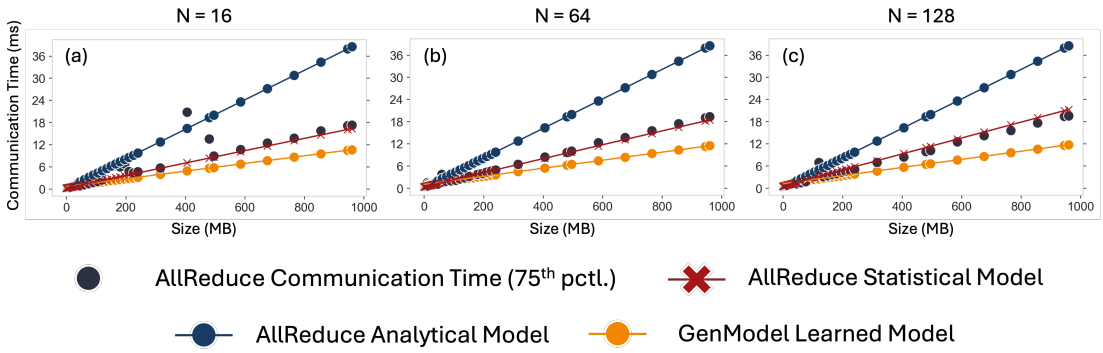


Figure 4.15: 75th percentile AllReduce collective times on cluster X with predictions by the statistical communication model and the GenModel baseline predictions for 2D parallelism on $N = 16, 64, 128$ GPUs.

Figures 4.12, 4.13, and 4.14 demonstrate the performance of the statistical model in predicting AllReduce, AllGather, and ReduceScatter communication time, respectively, on cluster X. With small data sizes ($S < 256$ MB) on the logarithmic scale, we can see that the linear model fits equally well in the non-linear regions of the time-size curve as in the linear regions where data sizes are larger. It is important to note that we only need to fit the communication time and straggler delay ratio models once for each collective; in other words, the model is able to explain the variability in communication time across data sizes and world sizes, thanks to the use of the log-sigmoid bandwidth function, the topology-aware nature of the analytical model and the interaction terms of the learned statistical model.

Figure 4.15 visually compares our proposed statistical model and GenModel's performance at predicting AllReduce communication times in inter-node only communication for 16, 64, and 128 GPUs on cluster X. Not only does GenModel fail to capture the non-linearities in AllReduce time, but it also systematically underestimates the collective time. By capturing straggler delay, non-linear bandwidths, and NCCL algorithms for different topologies as well as supporting both 1D and 2D parallelism, our learned communication models are a significant improvement over existing work.

Table 4.9: Root Mean Squared Error (RMSE) of Collective Communication Model and GenModel on Cluster X and Y. Baseline performance data from GenModel are shown for AllReduce in 2D parallelism only using an approximation of the Recursive Halving-Doubling model, since GenModel can only predict non-hierarchical AllReduce

World Size (N)	Model RMSE (ms)	GenModel RMSE (ms)	Improvement
Cluster X			
AllReduce (2D)			
16	2.363	5.385	2.278×
64	0.756	5.162	6.825×
128	1.302	5.304	4.0725×
AllReduce (1D)			
16	2.374	-	-
64	1.951	-	-
128	0.676	-	-
AllGather (2D)			
16	0.287	-	-
64	0.692	-	-
128	0.973	-	-
AllGather (1D)			
16	0.289	-	-
64	0.691	-	-
128	0.969	-	-
ReduceScatter (2D)			
16	0.128	-	-
64	1.203	-	-
128	1.990	-	-
ReduceScatter (1D)			
16	0.130	-	-
64	1.204	-	-
128	1.987	-	-
Cluster Y			
AllReduce (2D)			
16	0.411	1.176	2.860×
64	0.277	1.234	4.460×
128	0.633	1.782	2.814×
AllReduce (1D)			
16	0.246	-	-
64	0.366	-	-
128	0.467	-	-
AllGather (2D)			
16	0.130	-	-
64	0.266	-	-
128	0.624	-	-
AllGather (1D)			
16	0.174	-	-
64	0.264	-	-
128	0.626	-	-
ReduceScatter (2D)			
16	2.678	-	-
64	0.525	-	-
128	0.808	-	-
ReduceScatter (1D)			
16	2.680	-	-
64	0.542	-	-
128	0.794	-	-

4.7.3 Runtime Simulation

We now demonstrate the accuracy and speed of our tool across diverse models, training configurations, model sizes, architectural features, for single and distributed workflows.

Model	Num. Params	Num. Heads	Head Type	Hidden Dim	Num. Layers
Gemma 2B	2.02 B	8	Grouped-Query Attention	18432	26
TiMM ViT	632 M	16	MLP	1280	32
HF Clip	428 M	16	Standard Self-Attention	4096	24
Llama V3 1B	1.24 B	16	Grouped-Query Attention	4096	24

Table 4.10: Model configurations used in single device simulator experiments.

Experimental Setup

We evaluate on 4 state-of-the-art models Google Gemma 2B Team et al. (2024), Meta Llama 3.2 1BDubey et al. (2024), Open AI CLIP Radford et al. (2021), and PyTorch-Image-Models Vision Transformer Steiner et al. (2021); Alexey (2020) for single GPU training. We utilize Meta Llama 3.1 70B model for our distributed training experimentsDubey et al. (2024). We vary the batch-size, sequence-length (for large-language models) and image-size (for vision models). For measuring actual model execution times, we run three warm-up iterations and measure five actual iterations and use the mean value. For benchmark estimation mode for each operator, we perform 2 warm-up iterations and 3 actual measurement iterations and take the mean. For estimation experiments with our learned and statistical cost-models, we just run one single iteration of training since it is execution free. All our experiments are on the latest NVDIA H100 GPUs. For distributed settings, each machine has 4 GPUs connected with NVLinks, and the 16 machines are connected via Infiniband.

Single-model training

For single model training we experiment with 3 types of precisions Full Precision (FP), Mixed Precision (MP) and Half Precision (HP), to analyze the performance for different data types. We also toggle Activation Checkpointing (AC), to evaluate the recomputation overhead. Our benchmark model, per-form per operator execution to get the final run-time. Our Cost-Model represents the roofline-model that is fine-tuned. Learned model is our approach. We can estimate end-to-end model times within 30 seconds. Table 4.11 shows our results. We achieve a mean accuracy of 90% for Learned model approach against the 76% for Roofline-model and 85% with Benchmark model.

Distributed Training

We evaluate our distributed workflow, to demonstrate the effectiveness of of our communication models and our distributed simulator. We use Llama 3.1 70B model on 128 GPUs. Table 4.12, shows our results for training with FSDP (1D Fully Sharded Data Parallel) training and Table 4.13 shows our results while applying FSDP+TP (2D Fully Sharded and Tensor Parallel) parallelism.

4.7.4 Memory Simulation

Experimental Setup (Hardware, Models and Network)

We evaluate on 7 state-of-the-art models Google Gemma 2B Team et al. (2024), Meta Llama 3.2 1BDubey et al. (2024), Open AI CLIP Radford et al. (2021), Google T5 Raffel et al. (2020b), Open AI GPT Achiam et al. (2023). PyTorch-Image-Models Vision Transformer Alexey (2020) and ConvNextV2 Steiner et al. (2021) for single GPU training memory estimation . We utilize Meta Llama 3.1 70B model for our distributed training memory estimation experimentsDubey et al. (2024). We vary the batch-size, sequence-length (for large-language models) and image-size (for vision models).

Table 4.11: Runtime Simulator Accuracy Across Cost Models for Configurations of Deep Learning Models

Batch Size	Seq Length Img Size	Precision	Activation Checkpointing	Estimation Type	Predicted (ms)	Prediction Time (ms)	Actual (ms)	Accuracy (Actual / Pred)
Gemma 2b								
2	4096	HP	True	Benchmark	460.41	6481.63	424.03	0.92
2	4096	HP	True	Roofline Model	315.87	1342.92	424.03	1.34
2	4096	HP	True	Learned	426.36	7218.43	424.03	**0.99**
4	2048	HP	True	Benchmark	442.71	6372.82	407.84	0.92
4	2048	HP	True	Roofline Model	312.85	1359.28	407.84	1.30
4	2048	HP	True	Learned	409.33	7336.92	407.84	**1.00**
4	1024	FP	True	Benchmark	1648.08	13001.56	1605.04	0.97
4	1024	FP	True	Roofline Model	1068.72	1306.18	1605.04	1.50
4	1024	FP	True	Learned	1579.44	7117.60	1605.04	**1.02**
8	1024	HP	False	Benchmark	390.49	5631.34	356.99	0.91
8	1024	HP	False	Roofline Model	214.02	2426.29	356.99	1.67
8	1024	HP	False	Learned	361.40	6970.04	356.99	**0.99**
Timm ViT								
32	224	FP	False	Benchmark	831.66	8233.67	795.94	0.96
32	224	FP	False	Roofline Model	485.11	3925.25	795.94	1.64
32	224	FP	False	Learned	780.23	6555.14	795.94	**1.02**
64	224	FP	True	Benchmark	1858.68	15154.25	1820.54	0.98
64	224	FP	True	Roofline Model	1176.45	4474.60	1820.54	1.55
64	224	FP	True	Learned	1851.27	7238.46	1820.54	**0.98**
128	224	HP	True	Benchmark	440.07	7414.49	395.18	0.90
128	224	HP	True	Roofline Model	326.36	5610.18	395.18	1.21
128	224	HP	True	Learned	451.91	7221.01	395.18	**0.87**
256	224	HP	True	Benchmark	815.67	11073.03	764.06	0.94
256	224	HP	True	Roofline Model	645.36	5925.16	764.06	1.18
256	224	HP	True	Learned	885.13	7353.14	764.06	**0.86**
HF Clip								
32	20/336	FP	False	Benchmark	999.47	10050.04	936.18	0.94
32	20/336	FP	False	Roofline Model	609.15	3659.49	936.18	1.54
32	20/336	FP	False	Learned	950.12	7161.46	936.18	**0.99**
64	20/336	FP	True	Benchmark	2263.38	18999.18	2193.21	0.97
64	20/336	FP	True	Roofline Model	1474.87	2362.47	2193.21	1.49
64	20/336	FP	True	Learned	2299.78	7894.58	2193.21	**0.95**
Llama								
1	16384	HP	True	Benchmark	649.53	7410.70	616.67	0.95
1	16384	HP	True	Roofline Model	371.86	1185.89	616.67	1.66
1	16384	HP	True	Learned	620.61	6994.24	616.67	**0.99**
2	8192	HP	True	Benchmark	551.96	6701.48	525.06	0.95
2	8192	HP	True	Roofline Model	363.62	1120.93	525.06	1.44
2	8192	HP	True	Learned	513.50	7054.34	525.06	**1.02**
4	4096	HP	True	Benchmark	498.16	6546.15	458.82	0.92
4	4096	HP	True	Roofline Model	353.22	1158.24	458.82	1.30
4	4096	HP	True	Learned	459.63	7090.61	458.82	**1.00**
8	2048	FP	True	Benchmark	3217.58	22517.67	3199.76	0.99
8	2048	FP	True	Roofline Model	2206.64	1700.23	3199.76	1.45
8	2048	FP	True	Learned	3236.30	6945.94	3199.76	**0.99**

Local Batch Size	Seq Len	AC	Est. (ms)	Actual (ms)	Acc. (Est./Actual)	Pred. Overhead (s)
2	64	selective	4953.20	5503.55	0.90	17.78
2	256	selective	4934.07	5423.15	0.91	17.81
2	1024	full	5042.39	5480.86	0.92	18.24
1	4096	full	5477.02	6018.70	0.91	18.01
1	8192	full	9597.48	10663.87	0.90	18.10

Table 4.12: Runtime simulator accuracy for 1D FSDP across 128 GPUs for Llama 3 70B training. We achieve a mean accuracy of **90%** in predicting iteration time while incurring minimal prediction overhead (shown in the final column).

Local Batch Size	Seq Len	AC	Est. (ms)	Actual (ms)	Acc. (Est. / Actual)	Pred. Overhead (s)
8	1024	full	4955.56	5445.67	0.91	31.53
4	4096	full	5383.30	5851.41	0.92	30.37
4	8192	full	10075.91	11195.45	0.90	30.59

Table 4.13: Runtime simulator accuracy for 2D FSDP across 128 GPUs for Llama 3 70B training. We achieve a mean accuracy of **91%** in predicting iteration time while incurring minimal prediction overhead (shown in the final column).

For measuring actual model memory, we run three warm-up iterations and measure five actual iterations and use the max value. For estimation experiments with Memory Simulator, we just run one single iteration of training since it is execution free. All our experiments are on the latest NVDIA H100 GPUs.

Single-GPU

For single model training we experiment with 3 types of precisions Full Precision (FP), Mixed Precision (MP) and Half Precision (HP), to analyze the performance for different data types. We also toggle Activation Checkpointing (AC), to estimate the memory savings. Table 4.14 shows the effectiveness of our approach. We get close to 100% accuracy in almost all settings.

Multi-GPU

We evaluate our distributed workflow, to demonstrate the effectiveness of Memory Simulator at scale. We use Llama 3.1 70B model on 64 GPUs for FSDP configuration (1D Fully Sharded Data Parallel) and on 128 GPUs for FSDP + TP (2D Fully Sharded Data Parallel and Tensor Parallel). Each machine has 4 GPUs connected with NVLinks and the 16/32 machines are connected via Infiniband. Table 4.15, shows our results for training with FSDP (1D Fully Sharded Data Parallel) training and Table 4.16 shows our results while applying FSDP+TP (2D Fully Sharded and Tensor Parallel) parallelism. We achieve $\geq$ 99% accuracy in all cases even with complex memory management of FSDP and TP. We use TorchTitan Liang et al. (2024) to evaluate Memory Simulator.

Table 4.14: Memory usage estimates and actual for various models and configurations. Memory Simulator achieves approximately 99% accuracy in all scenarios across the 7 models with different batch sizes, sequence lengths, precisions and memory optimizations techniques like activation checkpointing.

Model Name	Batch Size	Seq Len/Image Size	Precision	AC	Estimated (GiB)	Actual (GiB)	Accuracy
gemma_2b	8	512	MP	No	59.75	59.81	0.99
	4	1024	FP	Yes	43.34	46.74	0.99
	8	1024	HP	No	66.41	66.47	0.99
	2	2048	FP	Yes	43.38	46.74	0.99
	2	2048	MP	No	59.78	59.84	0.99
	4	2048	HP	Yes	44.96	45.02	0.99
	2	4096	HP	Yes	45.00	45.06	0.99
hf_clip	32	336	FP	No	39.85	39.93	0.99
	64	336	FP	Yes	12.81	12.89	0.99
	64	336	HP	Yes	6.41	6.51	0.99
	64	336	MP	No	47.42	47.50	0.99
	128	336	HP	Yes	10.18	10.29	0.99
hf_GPT2	16	512	MP	No	44.34	44.47	0.99
	8	1024	MP	No	44.34	44.48	0.99
	16	1024	HP	No	49.93	49.95	0.99
hf_T5	6	512	MP	No	32.06	32.20	0.99
	2	1024	FP	Yes	33.70	33.75	0.99
	4	1024	HP	No	49.08	49.14	0.99
	1	2048	FP	Yes	53.50	53.55	0.99
	1	2048	HP	Yes	38.69	38.87	0.99
	1	2048	MP	Yes	44.95	45.00	0.99
llama_1b	4	1024	FP	No	33.04	33.09	0.99
	4	1024	MP	No	31.52	31.58	0.99
	4	2048	HP	Yes	24.94	24.99	0.99
	8	2048	FP	Yes	54.60	54.63	0.99
	8	2048	HP	Yes	42.97	43.02	0.99
	4	4096	HP	Yes	42.97	43.02	0.99
	2	8192	HP	Yes	42.98	43.03	0.99
	1	16384	HP	Yes	38.56	38.61	0.99
convnext	16	224	FP	No	22.70	22.98	0.99
	32	224	FP	Yes	14.46	14.67	0.99
	32	224	MP	No	27.79	28.02	0.99
	64	224	HP	No	33.64	33.91	0.99
	64	224	MP	No	46.91	47.18	0.99
	128	224	FP	Yes	31.45	31.54	0.99
	128	224	HP	Yes	15.74	15.86	0.99
	256	224	HP	Yes	27.38	27.51	0.99
timm_vit	32	224	FP	No	27.45	27.65	0.99
	64	224	FP	Yes	11.29	12.08	0.99
	64	224	HP	No	23.92	24.12	0.99
	64	224	MP	No	31.29	31.59	0.99
	128	224	HP	Yes	7.74	7.82	0.99
	256	224	HP	Yes	11.94	12.00	0.99

Table 4.15: Memory Simulator achieves $\geq$ 99% accuracy with distributed 1D FSDP training and is able to get the estimation within 30 seconds for Llama 70 billion model.

Batch Size	Seq Len	AC	Est.(GiB)	Actual(GiB)	Acc	Time (s)
2	64	selective	30.10	30.20	0.995	31.909
2	256	selective	30.65	30.97	0.989	31.858
2	1024	full	30.50	30.70	0.995	30.360
1	4096	full	32.15	32.28	0.996	31.552
1	8192	full	40.13	40.18	0.998	31.095

Table 4.16: Memory Simulator achieves $\geq$ 99% accuracy with distributed 2D FSDP+TP training and is able to get the estimation within 30 seconds for Llama 70 billion model.

Batch Size	Seq Len	AC	Est.(GiB)	Actual(GiB)	Acc	Time (s)
2	64	selective	12.87	12.98	0.992	29.569
2	256	selective	12.87	12.98	0.992	29.627
2	1024	full	12.88	12.98	0.992	28.423
1	4096	full	12.88	12.99	0.990	28.277
1	8192	full	13.10	14.27	0.991	28.452

4.8 Related Work

4.8.1 Runtime Simulators

Despite the recognized importance of model training simulation, there are few studies due to its inherent complexity Geoffrey et al. (2021); Li et al. (2023); Lee et al. (2025a). Existing approaches do not focus on identifying and capturing the synchronization primitives that are critical for simulating the diverse range of distributed training setups, which involve mixed parallel paradigms and collective primitives. As a result, existing methods are limited to supporting only a few specific simple parallel training paradigms, namely, Gpipe PP, DDP, and ring all-reduce-based TP Lee et al. (2025a). Moreover, their reliance on computational graphs prevents their deployment in real-world model training, as obtaining these graphs in large-scale distributed environments remains an open problem.

In contrast, TorchSim Simulator is the first simulation solution that supports all off-the-shelf parallel paradigms and accurately models the communication-compute overlap, without relying on computational graphs.

4.8.2 Memory Estimation

Current real-time memory tracking tools Shi & DeVito (2023); pyt (2025a), primarily designed to identify Out-of-Memory (OOM) errors and analyze memory usage, have significant limitations. They collect memory profiling statistics during job execution, making the analysis inherently post-hoc. Even if expert users identify memory inefficiencies and adjust configurations, there is no reliable method to estimate the precise impact on peak memory consumption or to guarantee the absence of OOM errors.

Analytical methods for estimating peak memory Narayanan et al. (2021) offer an alternative but

require specialized expertise, detailed knowledge of model architectures, and familiarity with the internal mechanics of automatic differentiation engines like PyTorch Autograd. These methods demand understanding mathematical formulations for each operator and intricate memory allocation policies, which becomes increasingly complicated when considering algorithmic features like prefetching, lazy initialization, dynamic resizing, and scheduling. Typically, researchers skilled in machine learning theory and algorithms lack the complementary systems expertise necessary for effectively utilizing these analytical techniques.

DNN-Mem Gao et al. (2020) depends on analytical formulas, rendering it impractical for maintenance given PyTorch's large number of operators. It also lacks support for eager execution mode and offers minimal support for distributed training, being limited to simple Distributed Data Parallel (DDP) scenarios.

Boom Su et al. (2024) only reports peak memory consumption without offering memory categorization, attribution, or snapshot capturing. It requires source-level modifications to PyTorch for the *FakeMemoryAllocator*, which despite its name, actually performs real memory allocations on a single GPU, providing no demonstrated compatibility with distributed training scenarios.

Skyline Yu et al. (2020) categorizes memory only into weights and activations, omitting critical categories such as activation gradients, weight gradients, and optimizer states. It lacks distributed training results and does not accommodate loop-based workflows common in pipeline parallel training.

Approaches using *TorchDispatchMode* He et al. (2022) and *FakeTensorMode* Contributors (2025) primarily focus on operator-level dispatch without modeling peak memory, categorizing memory usage, or attributing memory to specific modules. Accurate and efficient tensor liveness tracking and maintaining memory usage snapshots demand substantial additional effort, as described in Section 4.3.

In contrast, TorchSim addresses these limitations comprehensively. It extends PyTorch by

adding *FakeTensor* and *FakeProcessGroup* support for all communication and synchronization collectives, essential for accurate simulation of distributed training algorithms. TorchSim accurately categorizes tensors generated through collectives, integrating closely with native PyTorch distributed training techniques such as Fully Sharded Data Parallel (FSDP) and Tensor Parallel (TP). Additionally, TorchSim provides deep integration with tensor subclasses like *DTensors*, *KeyedTensor*, and *JaggedTensors* by appropriately *flattening* and *unflattening* tensors to access local device storage. Furthermore, it tracks heterogeneous device usage, crucial for CPU offloading scenarios, and effectively supports complex parallel strategies like Pipeline Parallel (PP), accurately reflecting variations in peak memory consumption across different pipeline stages and schedules.

4.8.3 Compute Time Prediction

Existing approaches to predicting GPU operator runtime can broadly be categorized into operator-level Justus et al. (2018) and kernel-level Geoffrey et al. (2021); Li et al. (2023); Lee et al. (2025a); Zhang et al. (2022b); Li et al. (2022) methods. Given an AI model, operator-level methods extract all its operators and estimate the compute time of each operator. Kernel-level methods further extract the kernels dispatched to compute each operator, and then estimate the kernel compute time. The predicted compute time for individual operators or kernels are then aggregated to get the model compute time.

Kernel-level predictions have demonstrated high accuracy by directly modeling hardware execution characteristics Zhang et al. (2022b). However, identifying the kernels dispatched from an operator and understanding the orchestration of dispatched kernels for an operator is by itself a challenging research problem Zhang et al. (2022b); Li et al. (2022). In practice, these analyses require unaffordable hardware-specific profiling techniques NVIDIA (2025); AMD (2025) and reverse engineering efforts Geoffrey et al. (2021).

Operator-level predictions rely primarily on hardware-agnostic features, such as batch size and

input dimensions, making it easier to get tested and employed in existing software. However, we notice most existing operator-level predictions only explore a small range of input features for single-node settings Justus et al. (2018); Zhang et al. (2022b); Lee et al. (2025a). This cannot fulfill the emergent demands of predicting cutting-edge AI models Tazi et al. (2025), which have input ranges in 1 to 1e6, depending on the operator types, and compute time in 1e-2 to 1e5 ms. A large range of input features and compute time not only impedes the model convergence by introducing exponentially-increased instability, but also demands dedicated model designs to capture the expanded intricate relationships between the operators and the dispatched kernels.

In TorchSim, we predict at the operator level and encompass a 100x larger input space, catering for all modern AI models. We resolve the technical challenges of increased convergence instability and expanded mapping complexity by embedding the intuition of learning the kernel dispatching implicitly in the choice and design of the models, namely a Random Forest model and a Mixture-of-Experts model. For the first time, we achieve beyond 90% accuracy using the Random Forest model for all compute-bounded operators in PyTorch on all practical input ranges.

4.8.4 Communication Time Modeling

The time for any data to be communicated across a link of bandwidth is typically modeled with the standard $\alpha - \beta$ model, where $\alpha, \beta \in \mathbb{R}^+$, α is the link latency, and β is link bandwidth Lee et al. (2025b); Won et al. (2023); Mohammad et al. (2017). While this model yields sufficient accuracy in predicting communication time in some applications, it falls short in multiple ways for modeling communication in multi-GPU training settings. For instance, the α-β model does not account for the presence of straggler delay in distributed settings involving communication amongst multiple GPUs. The model also assumes that bandwidth β is a scalar quantity–the link bandwidth can vary significantly between inter- and intra-node communication and, as we show in Figure 4.8b, β for a single link is in fact a non-linear function of data size. Furthermore, the model does not account

for the fact that backends such as NCCL use different algorithms for inter-node and intra-node communication in some collectives.

Xiong et al. (2024) builds upon the $\alpha - \beta$ model for AllReduce collectives by first using the $\alpha - \beta - \gamma$ model, where γ represents computational cost of the collective, and adding two additional terms, δ and ε, to capture memory access cost and incast, respectively. By analyzing the computational, communication, and memory access cost of different AllReduce algorithms, the proposed learned model, GenModel, is fitted to data from a co-located Parameter Server-based benchmarking suite as well as two additional microbenchmarks for memory access cost and full mesh communication congestion. While GenModel seeks to be topology-aware and congestion-aware like our proposed models, they have multiple limitations. Firstly, the benchmarking required to fit the model may not always be possible, given that the physical topology of a cluster may not make a Parameter Server (PS) benchmark or full mesh communication possible. Secondly, the algorithms included in GenModel only use a single type of algorithm such as Ring-AllReduce, Recursive Halving Doubling (RHD), or Co-located Parameter Server; however, the vast majority of large GPU clusters today use hierarchical topologies with some combination of these topologies. Finally, GenModel is only evaluated on a PS topology involving up to 15 nodes connected to a single switch with a constant network bandwidth of 10 Gbps and the MPI library backend, limiting its applicability to large scale deployments of inter-GPU communication.

The learned communication models in our solution are a fully topology-aware, algorithm-aware approach to modeling large-scale inter-GPU communication collectives. Our analytical cost models isolate inter-node and intra-node bandwidths, reflecting GPU clusters with hierarchical topologies and different interconnects between GPUs on the same and across nodes. Our models also only require a much simpler, topology-agnostic benchmark to fit our models. We demonstrate the scalability and adaptability of our models on different clusters and across world sizes, evaluating it on collectives between up to 128 GPUs across 32 nodes.

4.9 Future Directions

While TorchSim achieves high accuracy in simulating dense, deterministic training workloads, several promising extensions remain.

4.9.1 Modeling Data-Dependent Computation.

One important next step is extending TorchSim to support data-dependent computation, particularly in architectures such as Mixture-of-Experts (MoE) Cai et al. (2025a,b). These models dynamically route inputs to different expert sub-networks, introducing variability in execution paths and runtime. To simulate this behavior, TorchSim can be extended to sample expert selection patterns from a range of distributions, such as uniform, power-law, or Zipfian, to represent varying degrees of skew in expert activation. For each sampled configuration, runtime can be estimated independently. Aggregated statistics such as the median or higher percentiles can then be used to report end-to-end runtime, providing robust estimates under uncertainty without requiring full enumeration of all possible routing decisions.

4.9.2 Simulating Sparse Computation.

Another direction is supporting sparse computation, which differs from MoE in that execution is not conditional on input routing but on the sparsity pattern of the data itself Cai et al. (2025c); Gao et al. (2023). To handle this, TorchSim can incorporate parametrized cost models that reflect the performance characteristics of sparse kernels. These models can be driven by probabilistic distributions over sparsity levels, enabling runtime estimation as a function of expected sparsity. Similar to the approach for MoE, repeated sampling and aggregation can be used to produce stable performance estimates, while incorporating known overheads and scaling inefficiencies associated with sparse GPU execution.

4.10 Summary

In conclusion, this work presents TorchSim, a principled and practical framework for estimating memory and runtime in large-scale distributed training without requiring GPU execution. TorchSim combines modular design, operator-level memory tracking, and a high-fidelity simulator that models execution, synchronization, and compute-communication overlap. Our approach achieves high accuracy by integrating learned models for compute prediction and statistical models for communication overhead. We demonstrate that TorchSim consistently achieves 99 % accuracy in memory estimation and over 90% accuracy in runtime estimation across a diverse set of models, GPU types, cluster sizes, and networks. We open-source TorchSim with integration into TorchTitan and release pre-trained cost models and benchmark datasets, making it a widely accessible and production-ready solution for performance modeling in modern AI training systems.

5

Auto-SAC: Enhancing the Compute–Memory Efficiency Trade-Off in Distributed Training

In the previous chapters, we introduced TorchTitan as a composable and modular framework for scaling distributed training across thousands of accelerators, and TorchSim as a simulation-based estimator that provides high-fidelity predictions of runtime and memory consumption for training configurations. These capabilities allow users to explore large configuration spaces and reason about performance trade-offs in a principled way.

In this chapter, we extend the utility of TorchSim by using its operator-level runtime and memory estimations to drive automated activation checkpointing (AC), a technique used to reduce peak memory consumption by trading off recomputation during backpropagation. While activation checkpointing is commonly used, existing approaches are manually configured, automated in a non-tractable way, lack fine-grained control, and are not optimized under realistic hardware constraints.

We present Auto-SAC, a fully automated two-stage system for generating optimal SAC policies using TorchSim 's estimations. At the global level, we formulate an Integer Linear Program (ILP) that determines which modules should apply AC and how much memory to discard. This ILP is driven by piecewise-linear approximations of recomputation-vs-memory trade-off curves, constructed using runtime and memory statistics estimated by TorchSim. At the local level, we generate operator-level policies that meet per-module memory budgets using one of three strategies: greedy, knapsack-based dynamic programming, or a module-local ILP.

The key insight behind Auto-SAC is to decompose the SAC optimization problem across two hierarchical levels. The first level operates at the module granularity, where decision space is relatively small, enabling efficient global coordination under memory budgets. The second level applies targeted, operator-level policies within each selected module, allowing for fine-grained control over which activations to retain or discard.

We integrate Auto-SAC into TorchTitan to demonstrate that the system is not only effective, but also general and extensible. This integration supports rigorous, apples-to-apples comparisons against existing checkpointing strategies and highlights TorchTitan 's value as a research platform

for developing memory optimizations.

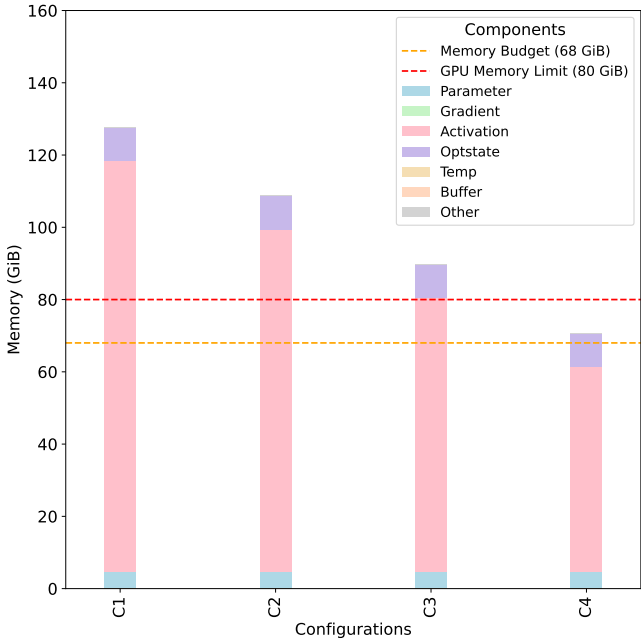
Auto-SAC supports all distributed training strategies handled by TorchTitan, including Fully Sharded Data Parallel (FSDP), Tensor Parallelism (TP), and Context Parallelism (CP). It advances the state of the art in memory-aware training by enabling fine-grained checkpointing decisions grounded in accurate simulation. In evaluations across LLMs and multi-modal generative models, Auto-SAC reduces recomputation overhead by up to 90% compared to naïve checkpointing. Moreover, its heuristic solvers deliver near-optimal performance with orders-of-magnitude lower runtime, making the method practical for large-scale use.

5.1 Motivation and Key Insights

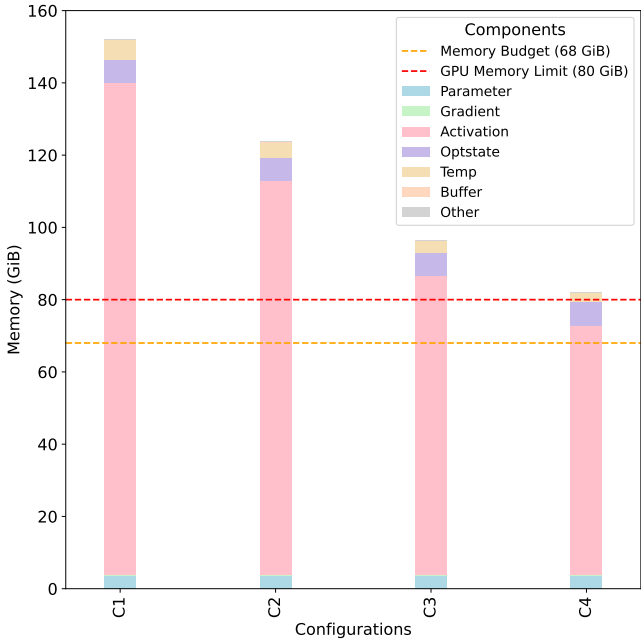
5.1.1 Activation Memory Dominates Training Footprint

Training large neural networks, especially transformer-based and multi-modal models, requires substantial GPU memory. While parameters, gradients, and optimizer states all contribute, a dominant portion of memory is consumed by activations—intermediate tensors stored during the forward pass for reuse in backpropagation.

Figure 5.1 shows the peak memory breakdown for two representative workloads: LLaMA 3.2 (1.2B) and Stable Diffusion 1.5 (1.06B). Across a range of batch sizes and sequence lengths, we observe that activations consistently account for over 90% of peak memory usage. This makes activation memory the primary target for memory optimization, particularly under tight hardware constraints.



(a) LLaMA v3.2 (1.23B)



(b) Stable Diffusion v1.5 (1.06B)

Figure 5.1: Peak memory breakdown across configurations. Activations dominate memory usage in both LLM and diffusion models.

5.1.2 Key Insight Behind Auto-SAC

Selective Activation Checkpointing (SAC) holds the promise of significantly reducing memory consumption with minimal computational overhead. However, deploying SAC effectively remains challenging: it often requires manual tuning, deep knowledge of operator behavior, and lacks systematic support for fine-grained decisions. Most existing checkpointing systems are coarse-grained, unaware of operator-level runtime costs, and do not expose actionable memory–compute trade-off statistics.

The key insight behind Auto-SAC is to decompose the SAC policy generation problem into two hierarchical levels:

- At the **module level**, where the number of decision points is small, we perform global memory allocation using a memory-constrained ILP. This allows efficient coordination across the model while respecting peak memory constraints.
- At the **operator level**, we apply targeted policies within each selected module to decide which specific activations to retain or discard. This enables fine-grained control tailored to each module's computational and memory profile.

Auto-SAC addresses key deployment challenges by:

- Estimating operator-level memory and runtime via simulation,
- Constructing empirical memory–recomputation trade-off curves,
- Automatically generating memory-aware policies under hardware constraints.

By separating global memory budgeting from local checkpointing decisions and grounding both in simulation-driven estimations, Auto-SAC offers a principled, automated, and scalable framework for applying SAC in real-world training pipelines.

5.2 Auto-SAC: High-Level Design and Solution Overview

5.2.1 Design Principles

Auto-SAC is designed to be both effective in optimizing memory and usable in practice. Its architecture is guided by the following principles:

Composability. Auto-SAC works seamlessly with modern distributed training setups, including Fully Sharded Data Parallel (FSDP), Tensor Parallelism (TP), and Context Parallelism (CP). It is fully compatible with `torch.compile`, enabling policies to be applied across graph-transformed models.

Reusability. Auto-SAC integrates with existing AC backends such as XLFormers and AOTAutograd. It acts as a policy generator—users retain their existing model structure and benefit from improved memory efficiency without modifying training code.

Interoperability. It supports diverse hardware and software environments, including multi-GPU and sharded setups, making it suitable for both research and production deployments.

Interpretability. Auto-SAC makes checkpointing decisions transparent. Each policy includes operator-level trade-off metrics, enabling users to inspect and reason about activation retention and recomputation.

Debuggability. Built-in tooling helps users visualize memory–compute trade-offs, validate memory targets, and identify bottlenecks in the activation graph.

5.2.2 User Experience

Auto-SAC is usable out of the box, but also flexible enough for power users:

- **Power Users:** Can access detailed statistics and customize policies manually.
- **Intermediate Users:** Can inspect and tune default policies with minimal configuration.
- **Beginner Users:** Can simply set a memory budget and let Auto-SAC handle the rest.

This layered experience ensures Auto-SAC is broadly usable—whether as a plug-and-play optimizer or a research tool for fine-grained control.

5.2.3 System Overview

Auto-SAC is a two-stage system for automatically generating selective activation checkpointing (SAC) policies that balance peak memory constraints and recomputation cost during model training. It is designed to operate efficiently under modern memory bottlenecks by leveraging accurate runtime and memory estimates from TorchSim, enabling principled decisions across both module and operator levels.

Auto-SAC breaks the problem into two stages:

1. **Global Module-Level Optimization:** Identify which modules should apply SAC and how much memory to discard, while ensuring the peak memory usage stays within a user-specified budget (e.g., 85% of GPU memory). This is solved as a global ILP.
2. **Local Operator-Level Policy Generation:** For each module selected for SAC, compute a fine-grained policy that determines which activations to retain and which to discard to meet the assigned memory discard target.

This decomposition enables global coordination of memory usage and local flexibility in applying SAC policies.

Stage 1: Global ILP Formulation

The first stage determines high-level memory allocation across the model. For each module, Auto-SAC decides whether to enable SAC and how aggressively to discard its activations. The global formulation aims to minimize total recomputation time across the model, while satisfying memory constraints.

To model memory usage, Auto-SAC accounts for:

- Forward and backward activations in each module,
- Gradients and activation gradients,
- Parameter and optimizer memory (sharded and unsharded),
- Retained activations from modules yet to execute,
- FSDP-specific memory components.

Each module's potential recomputation cost is modeled using a **piecewise linear approximation** of its memory–compute trade-off curve, constructed by simulating greedy discard decisions and measuring the cumulative impact on recomputation time. This approximation allows Auto-SAC to remain tractable within an ILP formulation while capturing real recomputation behavior with high fidelity.

The ILP jointly decides which modules to apply SAC to, and how much memory to discard in each, under a global peak memory constraint. The result is a set of module-level targets for discarded activation memory.

Stage 2: Local SAC Policy Generation

In the second stage, Auto-SAC generates operator-level SAC policies that satisfy the memory discard targets produced by the global ILP. This step is critical for adapting to operator-specific compute and memory behavior within each module.

Auto-SAC builds on PyTorch's operator-level SAC infrastructure and extends it with structured analysis from TorchSim, including:

- Per-operator runtime and memory estimates,
- Dependency metadata for view-like, in-place, and random operators,
- Autograd tracking status.

Auto-SAC supports three interchangeable algorithms for local policy generation:

Greedy Strategy. Activations are ranked by their **Memory Saving per Second (MSPS)**—the ratio of activation memory to recomputation time. Operators are greedily selected for recomputation until the desired memory discard target is reached. This approach is fast and provides strong approximations in practice.

Knapsack Approximation. The policy generation problem is framed as a 0/1 knapsack, where operators are items with weights (memory usage) and values (recomputation cost). A dynamic programming solver identifies which operators to retain while staying under the memory budget. This method balances policy quality with runtime efficiency.

Local ILP. For modules with tight budgets or highly sensitive compute behavior, an exact ILP is used to identify the optimal set of activations to retain. Constraints ensure structural correctness

for in-place and random ops. This approach yields the best possible policies at the cost of higher computation time.

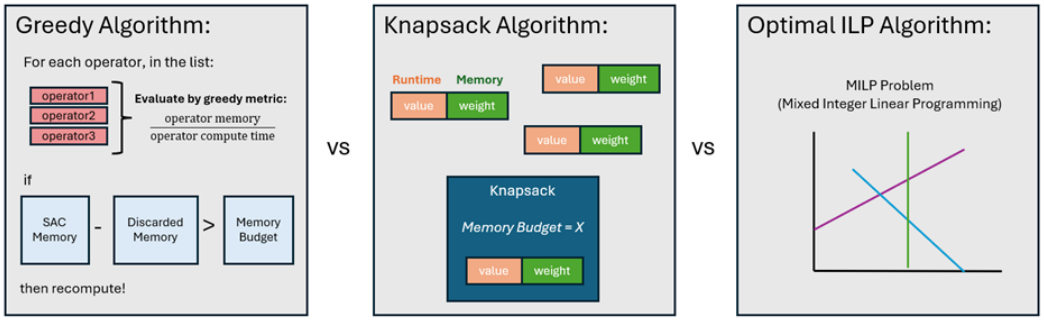


Figure 5.2: Local SAC policy generation strategies: greedy, knapsack, and ILP. All methods take a module-specific memory discard target as input and output an operator-level recomputation policy.

The policies generated by this stage are integrated back into the training graph via PyTorch's SAC APIs, enabling memory-efficient training without user intervention.

5.2.4 Synopsis

By separating global memory allocation from local policy generation, Auto-SAC achieves both system-level coordination and operator-level precision. The use of lightweight estimators, modular solvers, and correctness-aware policy generation makes Auto-SAC effective and extensible for large models, diverse parallelism strategies, and production-scale training environments.

5.3 SAC Estimator

To support automated generation of selective activation checkpointing (SAC) policies, Auto-SAC introduces the `SACEstimator`, a module that collects detailed per-operator memory and runtime statistics and constructs a trade-off profile between memory savings and recomputation overhead.

5.3.1 Functionality Overview

The `SACEstimator` is implemented as a *TorchDispatchMode*-based context manager that operates under PyTorch's *FakeTensorMode*. It observes a simulated forward pass of a module and records metadata for every operator, including:

- Runtime (in milliseconds),
- Activation memory allocated (in bytes),
- Structural properties (e.g., view-like, random, or in-place),
- Whether outputs are retained by PyTorch's autograd engine.

This information is collected into a structured `SACStats` object. Additional logic groups related operators (e.g., in-place or random ops) to ensure checkpointing decisions are consistent with execution semantics.

5.3.2 Outputs and Data Structures

The `SACEstimator` produces three primary outputs per module:

- `SACModStats`: Operator-level statistics including runtime, memory, and type annotations.
- `SACModGreedyOrder`: Precomputed MSPS scores and operator groupings to assist in greedy policy generation.
- `SACModTradeoffStats`: A memory–recomputation trade-off curve, approximated by a piecewise linear function, for use in global optimization.

Module: GPT.transformer.h.0
Total Memory: 1965031936 B Total Runtime: 18.79831815077489 ms Store Random: False

Op Idx	Op Name	Runtimes(ms)	Memory (B)	View-like	Random	In-place
0	native_layer_norm	0.1042	100925440	False	False	None
1	view	0	0	True	False	None
2	t	0	0	True	False	None
3	addmm	3.9662	301989888	False	False	None
4	view	0	0	True	False	None
5	split	0	0	True	False	None
6	view	0	0	True	False	None
7	transpose	0	0	True	False	None
8	view	0	0	True	False	None
9	transpose	0	0	True	False	None
10	view	0	0	True	False	None
11	transpose	0	0	True	False	None
12	_scaled_dot_product_efficient_attention	1.7628	102236672	False	True	None
13	transpose	0	0	True	False	None
14	view	0	0	True	False	None
15	view	0	0	True	False	None
16	t	0	0	True	False	None
17	addmm	1.3221	100663296	False	False	None
18	view	0	0	True	False	None
19	native_dropout	0.117	125829120	False	True	None
20	add	0.156	100663296	False	False	None
21	native_layer_norm	0.1042	100925440	False	False	None
22	view	0	0	True	False	None
23	t	0	0	True	False	None
24	addmm	5.2883	402653184	False	False	None
25	view	0	0	True	False	None
26	gelu	0.4161	402653184	False	False	None
27	view	0	0	True	False	None
28	t	0	0	True	False	None
29	addmm	5.2883	100663296	False	False	None
30	view	0	0	True	False	None
31	native_dropout	0.117	125829120	False	True	None
32	add	0.156	0	False	False	None

(a) Operator-level SAC statistics (`SACModStats`) for a Transformer block in GPT-2. Includes runtime, memory, and operator tags. This metadata drives policy generation under memory constraints.

AC Trade-off for Module: GPT.transformer.h.0 MSPS = Memory/Runtime

Op Id(s)	Op Name(s)	Discarded Mem (%)	Discarded Mem (B)	Recomp time (%)	Recomp time (ms)	MSPS	Always Stored	Always Recomputed
{1}	{'view'}	0	0	0	0	nan	False	True
{2}	{'t'}	0	0	0	0	nan	False	True
{4}	{'view'}	0	0	0	0	nan	False	True
{5}	{'split'}	0	0	0	0	nan	False	True
{6}	{'view'}	0	0	0	0	nan	False	True
{7}	{'transpose'}	0	0	0	0	nan	False	True
{8}	{'view'}	0	0	0	0	nan	False	True
{9}	{'transpose'}	0	0	0	0	nan	False	True
{10}	{'view'}	0	0	0	0	nan	False	True
{11}	{'transpose'}	0	0	0	0	nan	False	True
{13}	{'transpose'}	0	0	0	0	nan	False	True
{14}	{'view'}	0	0	0	0	nan	False	True
{15}	{'view'}	0	0	0	0	nan	False	True
{16}	{'t'}	0	0	0	0	nan	False	True
{18}	{'view'}	0	0	0	0	nan	False	True
{22}	{'view'}	0	0	0	0	nan	False	True
{23}	{'t'}	0	0	0	0	nan	False	True
{25}	{'view'}	0	0	0	0	nan	False	True
{27}	{'view'}	0	0	0	0	nan	False	True
{28}	{'t'}	0	0	0	0	nan	False	True
{30}	{'view'}	0	0	0	0	nan	False	True
{19}	{'native_dropout'}	0.064	125829120	0.0062	0.117029	1.08e+09	False	False
{31}	{'native_dropout'}	0.1281	251658240	0.0125	0.234057	1.08e+09	False	False
{0}	{'native_layer_norm'}	0.1794	352583680	0.018	0.338221	9.69e+08	False	False
{21}	{'native_layer_norm'}	0.2308	453509120	0.0235	0.442385	9.69e+08	False	False
{26}	{'gelu'}	0.4357	856162304	0.0457	0.858487	9.68e+08	False	False
{20}	{'add'}	0.4869	956825600	0.054	1.01452	6.45e+08	False	False
{3}	{'addmm'}	0.6406	1258815488	0.265	4.98077	7.61e+07	False	False
{17}	{'addmm'}	0.6918	1359478784	0.3353	6.30285	7.61e+07	False	False
{24}	{'addmm'}	0.8967	1762131968	0.6166	11.5912	7.61e+07	False	False
{12}	{'_scaled_dot_product_efficient_attention'}	0.9488	1864368640	0.7104	13.354	5.8e+07	False	False
{29}	{'addmm'}	1	1965031936	0.9917	18.6423	1.9e+07	False	False
{32}	{'add'}	1	1965031936	1	18.7983	0	False	False

(b) Trade-off summary (`SACModTradeoffStats`) showing discarded memory vs. recomputation time for candidate policies, along with MSPS and operator-level constraints.

Figure 5.3: SAC Estimator outputs for a Transformer module in GPT-2.

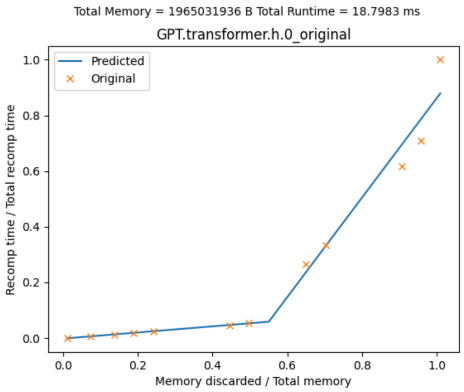
5.3.3 Trade-Off Modeling with Piecewise Linear Functions

To efficiently integrate recomputation costs into the global ILP, Auto-SAC models the relationship between discarded activation memory and recomputation time using a piecewise linear upper bound. This is derived by simulating a greedy discard process ordered by operator MSPS scores.

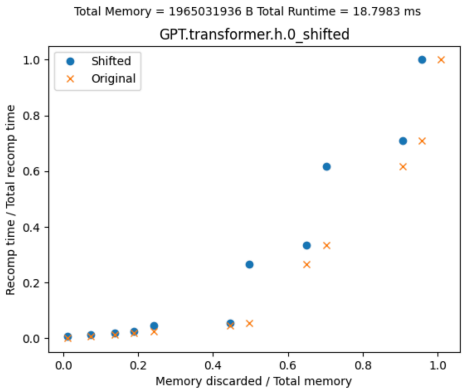
The trade-off curve $\mathcal{T}$ is computed via the following steps:

1. Begin with operators that must always be recomputed (e.g., view-like).
2. Discard additional operators greedily in descending MSPS order.
3. At each step, record the cumulative memory discarded and total recomputation time.
4. Fit a piecewise linear upper bound using the `pwlf` package.

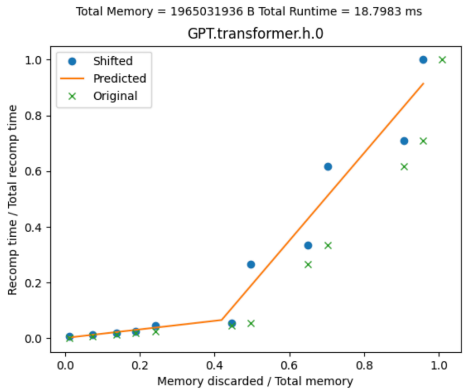
This fitted function approximates recomputation time as a function of memory discarded and is used in the global ILP formulation as a constraint. To ensure conservativeness, the curve is shifted by dropping the initial recomputation point and final memory point.



(a) Original trade-off curve. Each point shows cumulative recomputation time vs. discarded memory.



(b) Shifted trade-off curve. Removes the first y-point and last x-point to upper bound future discard actions.



(c) Piecewise linear upper bound fitted to the shifted curve. This model is used by the global ILP to constrain recomputation time.

Figure 5.4: From raw trade-off data to upper-bound approximation for GPT-2 Transformer module.

5.4 An ILP-Based Global per Module AC Budgeting Algorithm

The first stage of Auto-SAC formulates a global Mixed Integer Linear Program (MILP) to determine per-module activation checkpointing (AC) decisions. Specifically, the ILP solves for:

- Whether to apply SAC to each module,
- How much activation memory to discard in modules where SAC is applied.

The objective is to minimize total recomputation time across the model while ensuring that peak memory consumption remains under a user-specified threshold.

Variables Used

Variable	Description
y_i	Binary indicator: 1 if SAC is applied to module i, 0 otherwise
r_i	Fraction of discardable activation memory selected for module i
d_i	Total discarded activation memory in module i
a_i	Effective backward activation memory for module i
m_i	Total memory usage at module i
$\max_m$	Peak memory usage across all modules
rcp_i	Fractional recomputation time for module i
rct_i	Absolute recomputation time for module i (in ms)
ACM_i	Total discardable activation memory for module i
IA_i	Activation memory implicitly saved without SAC
TA_i	Total forward activations in module i
AG_i	Activation gradient memory for module i
P_i, G_i, O_i	Parameter, gradient, and optimizer memory in module i
f_{act}	Dtype scaling factor for activation memory

Table 5.1: Variables used in the global ILP formulation

5.4.1 Objective

Let $y_i \in \{0,1\}$ indicate whether SAC is applied to module i, and $r_i \in [0,1]$ denote the fraction of discardable activation memory selected. We aim to minimize the total recomputation time:

$$\text{minimize} \quad \sum_i \text{rct}_i$$

where $\text{rct}_i = \text{sac\_runtime}_i \cdot \text{rcp}_i$, and rcp_i is computed via a piecewise linear approximation of the memory–compute trade-off curve.

5.4.2 Constraints

1. Discarded Memory Definition. Each module's discarded activation memory d_i is computed as:

$$d_i = \text{ACM}_i \cdot r_i - (\text{ACM}_i - \text{IA}_i) \cdot y_i$$

This formulation accounts for memory implicitly saved by PyTorch's autograd engine in the default case.

2. Activation Memory Accounting. Effective backward activation memory a_i includes retained activations and gradients, minus discarded memory from prior modules:

$$a_i = \text{TA}_i + \text{AG}_i - \sum_{j<i} d_j$$

3. Module Memory Usage. Total memory usage m_i is the sum of activation memory (scaled by dtype), parameters, gradients, and optimizer state:

$$m_i = a_i \cdot f_{\text{act}} + P_i + G_i + O_i$$

4. Peak Memory Constraint. We track peak memory using an auxiliary variable $\max_m$ and enforce:

$$\max_m \geq m_i \quad \forall i, \qquad \max_m \leq \text{MemoryBudget}$$

5. Recomputation Time Approximation. Recomputation time is modeled via piecewise linear segments (s):

$$\text{rcp}_i \geq \text{slope}_s \cdot r_i + \text{intercept}_s, \quad \forall s \in \text{segments}_i$$

$$\text{rct}_i = \text{sac\_runtime}_i \cdot \text{rcp}_i \quad \text{if } y_i = 1; \quad \text{else } 0$$

6. Logical Constraints.

- SAC is only applied to non-leaf and gradient-requiring modules.
- No nested AC units: if module j is a descendant of i, then $y_i + y_j \leq 1$.
- SAC is not applied to modules overlapping with FSDP subtrees.

5.4.3 Solving the ILP

The MILP is solved using a standard solver (e.g., CBC), subject to time limits and solution gap tolerances. The solver outputs:

- y_i, r_i: Per-module SAC decisions and discard aggressiveness,
- d_i: Memory discarded per module,
- $\max_m$: Estimated peak memory across all modules,
- Total recomputation time across the model.

These values are passed to the second stage to guide operator-level policy generation. If the solver fails to find a feasible solution within budget, Auto-SAC can fall back to relaxed memory targets or heuristic baselines.

5.5 Greedy, Kanpsack and ILP SAC Algorithms

5.5.1 Greedy Algorithm

The greedy algorithm in Auto-SAC constructs an operator-level checkpointing policy for each module that satisfies a given memory budget while minimizing recomputation time. It uses a simple

yet effective heuristic based on the **Memory Saving per Second (MSPS)** metric:

$$\text{MSPS}_j = \frac{a_j}{r_j}$$

where a_j is the memory consumed and r_j the recomputation time of operator j. Operators are ranked in decreasing order of MSPS and are discarded (i.e., marked for recomputation) until the memory budget is met.

Symbol	Description
a_j	Memory used by operator j (in bytes)
r_j	Runtime of operator j (in milliseconds)
MSPS_j	Memory Saving per Second: $\frac{a_j}{r_j}$
$\mathcal{B}$	Module-level memory budget (in bytes)
`stored_ops`	Set of operators that must be stored
`recomputed_ops`	Set of operators pre-marked for recomputation
`policy`[j]	Final decision: 1 = store, 0 = discard

Table 5.2: Symbols used in the greedy SAC algorithm

Handling Special Cases

To maintain correctness, Auto-SAC applies special handling for:

- **In-place operators:** Must be treated together with their parent operator.
- **Random operators:** Always stored to preserve determinism.
- **Autograd-tracked operators:** Must be stored to comply with autograd dependencies.

Algorithm 7 Greedy SAC Policy Generation

Require: `sac_stats`, `sac_greedy_order_meta`, memory budget $\mathcal{B} \in [0, 1]$
Ensure: Binary vector `policy_output`: 1 = store, 0 = discard

Initialize `policy_output` $\leftarrow [1, 1, ..., 1]$
$M_{\text{total}} \leftarrow \sum_j a_j$, $\mathcal{B}_{\text{bytes}} \leftarrow \mathcal{B} \cdot M_{\text{total}}$
Load: `stored_ops`, `recomputed_ops`, `inplace_op_groups`, `random_ops_group`, `msps_meta`
Aggregate `stored_indices` with linked inplace/random groups
$M_{\text{stored}} \leftarrow \sum_{j \in \texttt{stored\_indices}} a_j$
if $M_{\text{stored}} > \mathcal{B}_{\text{bytes}}$ **then**
    return `policy_output` with 1s for `stored_indices`, 0 elsewhere
end if
Initialize `recompute_indices` $\leftarrow$ `recomputed_ops`
$M_{\text{discarded}} \leftarrow \sum_{j \in \texttt{recompute\_indices}} a_j$
$\mathcal{B}_{\text{bytes}} \leftarrow \mathcal{B}_{\text{bytes}} - M_{\text{stored}}$
Sort `msps_meta` by descending MSPS
while $M_{\text{total}} - M_{\text{discarded}} > \mathcal{B}_{\text{bytes}}$ **do**
    if `msps_meta` is empty **then**
        return `policy_output` with 1s for `stored_indices`
    end if
    Pop top candidate j from `msps_meta`
    Add j and any linked inplace/random ops to `recompute_indices`
    Update $M_{\text{discarded}}$
end while
for all $j \in$ `recompute_indices` **do**
    `policy_output`[j] $\leftarrow 0$
end for
return `policy_output`

Execution Flow Summary

Algorithm 7 proceeds in the following stages:

1. **Initialize**: Begin with a policy that stores all activations.

2. **Preserve Required Operators**: Store any operators that are random, in-place, or tracked by autograd.

3. **Check Feasibility**: If required activations alone exceed the memory budget, return early.

4. **Greedy Ranking**: Rank remaining operators by MSPS and discard (i.e., recompute) the highest-value ones until the budget is met.

5. **Generate Policy**: Construct a binary vector where 0 = discard (recompute) and 1 = retain (store).

This approach provides fast, scalable, and near-optimal SAC policies, particularly effective for large modules where exact ILP is too slow.

5.5.2 Knapsack-Based Algorithm

The knapsack-based algorithm in Auto-SAC models per-module SAC policy generation as a classic 0/1 knapsack problem. Each activation is treated as an item with a memory cost and a recomputation benefit. The objective is to retain a subset of activations such that the cumulative memory stays within a given budget $\mathcal{B}$, while minimizing recomputation time for discarded activations.

Symbol	Description
a_j	Memory used by operator j (in bytes)
r_j	Recomputation time of operator j (in ms)
$\mathcal{B}$	Memory budget for the module (in bytes)
w_j	Normalized memory weight of operator j
v_j	Value of operator j, equal to r_j
C	Discretized capacity of the knapsack
`dp`[i][c]	Maximum value using first i ops within capacity c
`policy`[j]	Final decision: 1 = store, 0 = discard for operator j

Table 5.3: Symbols used in the knapsack SAC algorithm

Handling Constraints

As with other policy generators, the knapsack algorithm respects operator constraints:

- **In-place operators** are treated as groups that must be recomputed or stored together.
- **Random operators** are always stored to preserve determinism.
- **Autograd-tracked operators** are automatically marked as stored.

Algorithm 8 Knapsack SAC Policy Generation for a Module

Require: `sac_stats`, `sac_greedy_order_meta`, memory budget $\mathcal{B} \in [0, 1]$
Ensure: Binary SAC policy vector: 1 = store, 0 = discard

Initialize `policy_output` $\leftarrow [0, 0, ..., 0]$
Compute $M_{\text{total}} \leftarrow \sum_j a_j$, $\mathcal{B}_{\text{bytes}} \leftarrow \mathcal{B} \cdot M_{\text{total}}$
Extract `stored_ops`, `inplace_op_groups`, `random_ops_group`, `msps_meta`
Compute `stored_indices` and $M_{\text{stored}} \leftarrow \sum a_j$ for those indices
if $M_{\text{stored}} > \mathcal{B}_{\text{bytes}}$ **then**
    return policy with 1s for `stored_indices`, 0 elsewhere
end if
$\mathcal{B}_{\text{bytes}} \leftarrow \mathcal{B}_{\text{bytes}} - M_{\text{stored}}$
Filter `msps_meta` to exclude ops with $r_j = 0$
Let $n \leftarrow$ `len(msps_meta)`
Normalize $w_j \leftarrow a_j / M_{\text{total}}$, set $v_j \leftarrow r_j$
Set step size $\delta \leftarrow 0.01$, capacity $C \leftarrow \lfloor \mathcal{B}_{\text{bytes}} / M_{\text{total}} / \delta \rfloor$
Discretize $w_j \rightarrow \lceil w_j / \delta \rceil$
Initialize DP table: `dp`[i][c] $\leftarrow 0$ for all i, c
for $i = 1$ to n **do**
    for $c = 0$ to C **do**
        if $w_i \leq c$ **then**
            `dp`[i][c] $\leftarrow \max($`dp`$[i-1][c],$ `dp`$[i-1][c - w_i] + v_i)$
        else
            `dp`[i][c] $\leftarrow$ `dp`$[i-1][c]$
        end if
    end for
end for
Initialize `selected_indices` $\leftarrow \emptyset$, $c \leftarrow C$
for $i = n$ to 1 **do**
    if `dp`$[i][c] \neq$ `dp`$[i-1][c]$ **then**
        Add $j = i-1$ and linked groups to `selected_indices`
        $c \leftarrow c - w_j$
    end if
end for
`stored_indices` $\leftarrow$ `stored_indices` $\cup$ `selected_indices`
for all $j \in$ `stored_indices` **do**
    `policy_output`[j] $\leftarrow 1$
end for
return `policy_output`

Execution Flow Summary

Algorithm 8 proceeds as follows:

1. **Initialize:** Set default policy to discard all activations.
2. **Preprocess:** Store operators required by structure constraints and update budget accordingly.
3. **DP Setup:** Normalize operator memory, define discretized knapsack capacity, and populate DP table.
4. **Backtrack:** Extract the highest-value subset of operators that meet the budget.
5. **Finalize:** Merge selected ops with required ones and return a binary policy vector.

This pseudo-polynomial dynamic programming approach yields high-quality SAC policies, providing a strong trade-off between runtime and recomputation cost compared to greedy or ILP solvers.

5.5.3 ILP-Based Algorithm

The ILP-based approach formulates operator-level SAC policy generation as a Mixed Integer Linear Program (MILP). This method provides an exact solution by determining, for each operator, whether its activation should be stored or recomputed, subject to a module-specific memory budget.

Symbol	Description
x_j	Binary decision: 1 = store activation of operator j, 0 = recompute
a_j	Memory used by operator j (bytes)
r_j	Recomputation time of operator j (ms)
$\mathcal{B}$	Memory budget as a fraction of total activation memory
M_{budget}	Absolute memory budget (bytes)

Table 5.4: Symbols used in the ILP-based SAC algorithm

Problem Formulation

Let:

- $x_j \in \{0, 1\}$: a decision variable indicating whether the activation of operator j is stored (1) or recomputed (0),
- a_j: memory usage of operator j,
- r_j: recomputation time of operator j,
- $\mathcal{B}$: normalized memory budget,
- $M_{\text{budget}} = \mathcal{B} \cdot \sum_j a_j$: absolute memory budget for the module.

The goal is to minimize recomputation time by selecting the best subset of activations to retain. The ILP is posed as:

Objective.

$$\text{maximize} \quad \sum_j x_j \cdot r_j$$

This is equivalent to minimizing the recomputation cost of discarded operators since:

$$\sum_j r_j - \sum_j x_j \cdot r_j$$

is the total recomputation cost.

Constraints

- **Memory Constraint:**

$$\sum_j x_j \cdot a_j \leq M_{\text{budget}}$$

- **View-Like Operators:** Must always be recomputed:

$$x_j = 0 \quad \forall j \in \texttt{view\_like\_ops}$$

- **Random Operators:**
  - If `force_store_random` is enabled:

$$x_j = 1 \quad \forall j \in \texttt{rand\_ops}$$

  - Otherwise, they must be grouped:

$$x_{i_1} = x_{i_2} \quad \forall i_1, i_2 \in \texttt{rand\_ops}$$

- **In-Place Operators:** Must follow their parent:

$$x_{\text{op}} = x_{\text{parent}} \quad \text{or} \quad x_{\text{op}} = 1 \quad \text{if op} = \text{parent}$$

Execution Semantics

This ILP guarantees an optimal subset of activations to store under a strict memory constraint, while preserving structural correctness. Operators are only recomputed when:

- Memory budget permits,
- Their recomputation cost is relatively low,
- They are not part of constrained groups (in-place or random).

Fallback Behavior

If the solver fails to find a feasible solution (e.g., due to a tight budget or rigid constraints), Auto-SAC will raise an error and optionally fall back to relaxed constraints or heuristic strategies like greedy.

Use Case

While this ILP solver is more computationally expensive than greedy or knapsack-based alternatives, it is especially valuable in memory-constrained regimes or when evaluating optimality in experimental benchmarks.

5.6 Experimental Analysis

We conduct a comprehensive evaluation of Auto-SAC across two representative models—LLaMA v3.2-1B and Stable Diffusion v1.5—under varied training configurations. For each configuration, we compare five activation checkpointing strategies:

1. **No AC:** No checkpointing is applied. Peak memory is highest, with zero recomputation.

2. **Full AC:** All foundational modules (e.g., each *DecoderLayer* in LLaMA) are checkpointed, resulting in lowest memory usage but highest recomputation.

3. **Auto-SAC (Optimal, Knapsack, Greedy):** Our proposed system automatically determines which modules to checkpoint and how much memory to discard using a global ILP. The resulting per-module budgets are used to generate operator-level SAC policies via the algorithms described in Section 5.5.

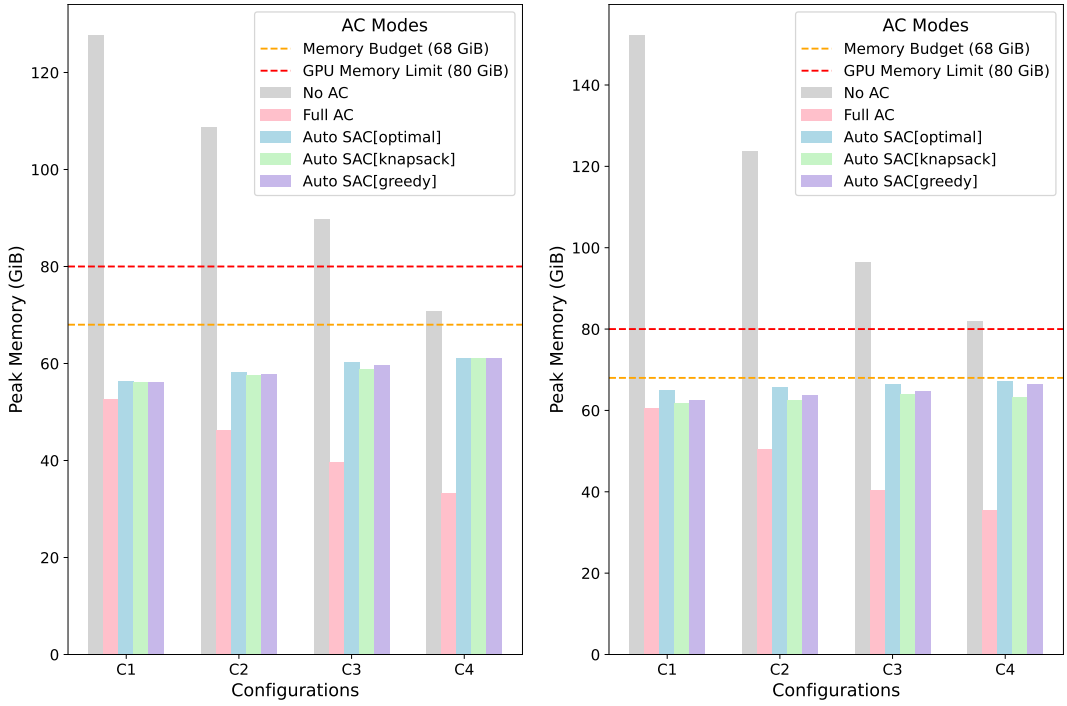
All Auto-SAC runs are constrained to a memory budget of 68 GiB—85% of the total GPU capacity—to account for fragmentation and workspace overhead.

5.6.1 Model Configurations

We define four configurations per model by varying batch size and sequence length (or image size), resulting in memory demands that progressively exceed the available GPU capacity. These configurations are summarized in Table 5.5.

Model	Config	Batch Size	Seq Len	Img Size	Denoising Steps
Stable Diffusion	C1	80	77	512	50
	C2	64	77	512	50
	C3	192	77	256	50
	C4	160	77	256	50
LLaMA v3 1B	C1	6	4096	-	-
	C2	10	2048	-	-
	C3	16	1024	-	-
	C4	12	1024	-	-

Table 5.5: Configuration details for Stable Diffusion and LLaMA v3.2-1B models.



(a) LLaMA v3.2 (1.23B) **(b)** Stable Diffusion v1.5 (1.06B)

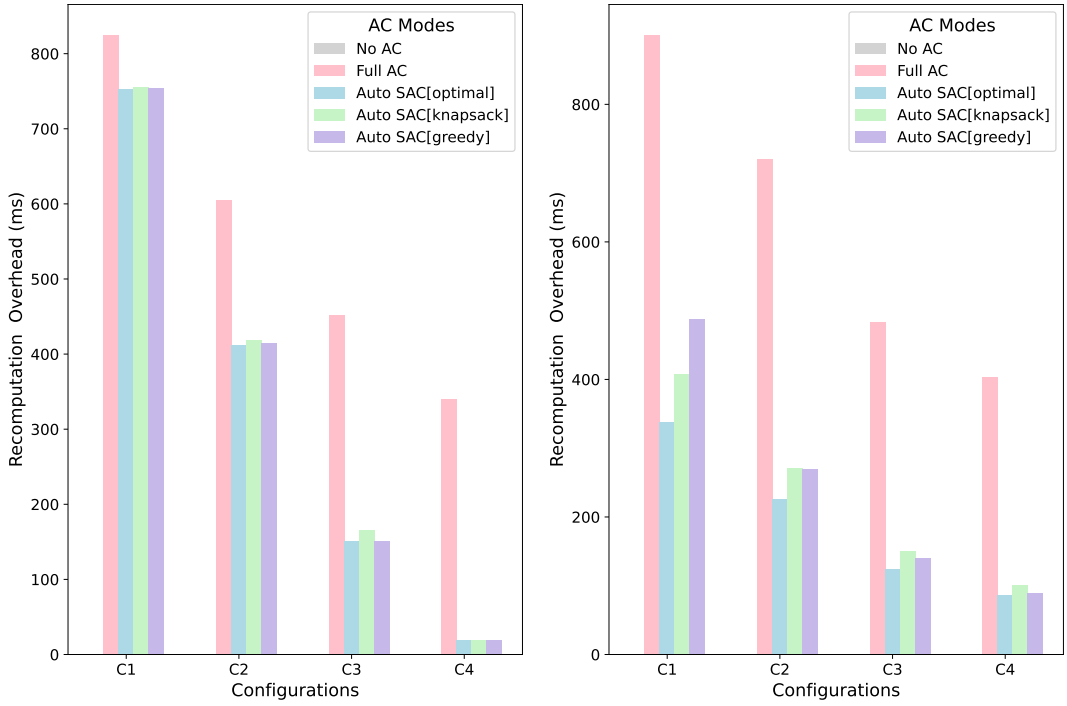
Figure 5.5: Peak memory under various strategies. Auto-SAC nearly saturates the budget while Full AC underutilizes memory due to lack of constraint awareness.

5.6.2 Auto-SAC Maximizes Memory Utilization

Figure 5.5 shows the peak memory usage under different checkpointing strategies. Full AC aggressively discards all activations without regard to the actual memory budget, leading to severe underutilization. In contrast, Auto-SAC tailors checkpointing to the available memory and target configuration, consistently achieving near-maximum utilization.

5.6.3 Auto-SAC Substantially Reduces Recomputation Overhead (6–90%)

As shown in Figure 5.6, Full AC incurs the highest recomputation overhead. In contrast, Auto-SAC algorithms prioritize retaining expensive activations and discarding cheap ones, resulting in up



(a) LLaMA v3.2 (1.23B)

(b) Stable Diffusion v1.5 (1.06B)

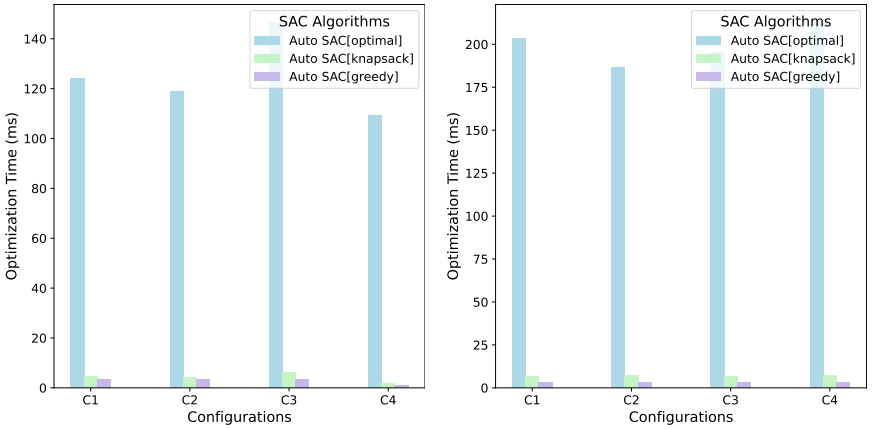
Figure 5.6: Recomputation overhead under each strategy. Auto-SAC consistently outperforms Full AC. Greedy and Knapsack are close to optimal.

to 90% reduction in recomputation time.

The overhead correlates with how much memory needs to be discarded per configuration. Among Auto-SAC variants, the optimal ILP achieves the lowest recomputation, with greedy and knapsack closely matching its performance across all settings.

5.6.4 Greedy and Knapsack are 100× Faster than Optimal

Figure 5.7 compares the time to compute SAC policies across the three solvers. While the ILP offers exact solutions, it is significantly slower than the greedy and knapsack algorithms. Greedy is the fastest, followed closely by knapsack, and both scale well to large models with many operators—demonstrating Auto-SAC's practicality for real workloads.



(a) LLaMA v3.2 (1.23B) **(b)** Stable Diffusion v1.5 (1.06B)

Figure 5.7: Optimization time per solver. Greedy and Knapsack are 100× faster than Optimal (ILP).

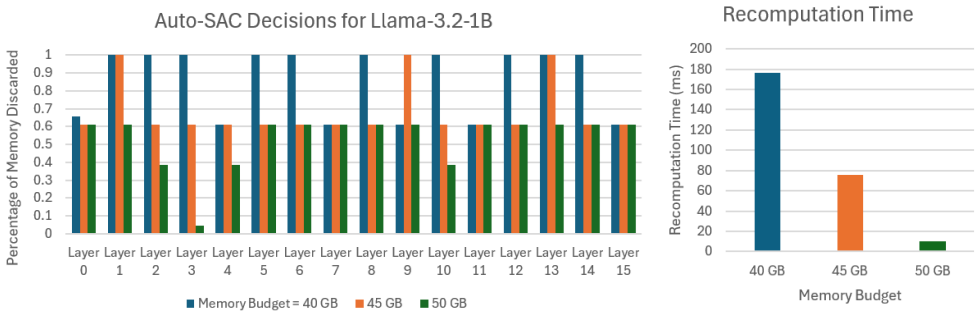


Figure 5.8: Auto-SAC ILP decisions and recomputation times across memory budgets for LLaMA v3.2-1B. As budgets increase, fewer activations are discarded, reducing recomputation.

5.6.5 Case Study: ILP Decisions for LLaMA v3.2-1B

To gain intuition about Auto-SAC's behavior, we analyze ILP-generated SAC policies under three memory budgets (40GB, 45GB, 50GB) for LLaMA v3.2-1B. Figure 5.8 shows both the modules selected for SAC and the corresponding recomputation time under each budget.

We observe the following trends:

1. Higher memory budgets result in lower recomputation times.

2. The amount of discarded memory per module decreases with increasing budgets.

3. Deeper modules are chosen less often for SAC at higher budgets, as more activations can be stored.

5.6.6 Module-Level Analysis

To understand per-module behavior, we analyze SACStats for an individual Transformer block (Layer 15). Table 5.6 shows the runtime and memory of each operator.

Operator Index	Operator Name	Runtime (ms)	Memory (B)
0	pow	0.0267	67108864
1	mean	0.0134	32768
2	add	0.0000	32768
3	rsqrt	0.0000	32768
4	mul	0.0267	67108864
...	...	...	...
82	add	0.0401	0

Table 5.6: Extracted SAC statistics for a Transformer module (Layer 15) in LLaMA v3.2-1B.

We also visualize the module's compute–memory trade-off in Figure 5.9, showing how discarding different portions of memory affects recomputation time.

This analysis reveals that discarding around 60% of activations yields a near-optimal trade-off: significant memory savings with minimal recomputation cost. This aligns with ILP results and illustrates how Auto-SAC targets the "knee" of the trade-off curve to balance efficiency and accuracy.

5.7 Summary

We presented Auto-SAC, a principled and scalable algorithm for automatically generating selective activation checkpointing (SAC) policies that minimize recomputation while adhering to memory constraints. Built on top of high-fidelity runtime and memory estimations from TorchSim, Auto-SAC decomposes the SAC optimization problem into two hierarchical stages: a global ILP

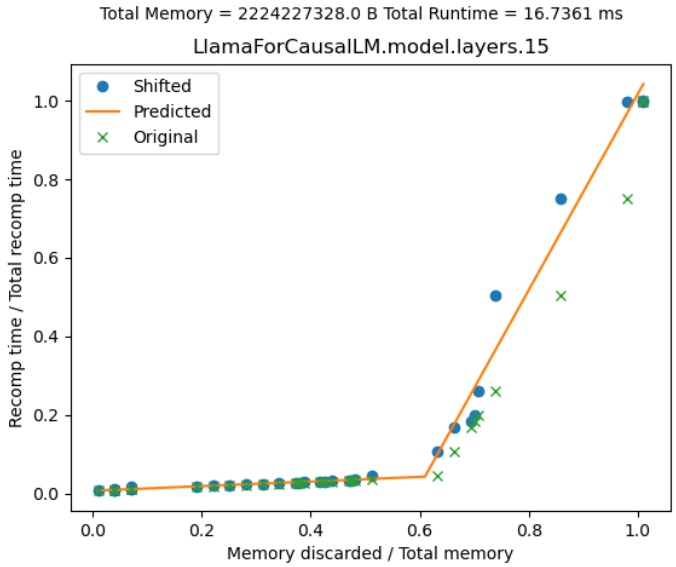


Figure 5.9: Compute–memory trade-off curve for Layer 15 in LLaMA v3.2-1B. Discarding 60% of memory leads to <5% recomputation overhead.

that assigns per-module memory discard budgets, and local operator-level policies that determine which activations to store or recompute.

At the global level, we introduce a memory-aware ILP that uses piecewise-linear trade-off approximations to make module-level checkpointing decisions under a peak memory constraint. At the local level, we provide three policy generation algorithms—greedy, knapsack-based, and ILP—that vary in speed and optimality, allowing users to make trade-offs between policy quality and solver runtime.

We integrate Auto-SAC into the TorchTitan framework, enabling support for all modern distributed training strategies including FSDP, TP, and CP. Our system is compatible with dynamic graph transformations via `torch.compile`, and can be easily adopted across diverse training pipelines.

Through comprehensive experiments on LLaMA and Stable Diffusion models, we show that Auto-SAC consistently maximizes memory utilization, reduces recomputation overhead by up to

90%, and delivers near-optimal checkpointing policies with orders-of-magnitude faster runtimes when using heuristic solvers. Our system also enables fine-grained introspection of operator-level trade-offs and adapts seamlessly across varying memory budgets.

In doing so, AUTO-SAC advances the state of the art in memory-efficient training and demonstrates how simulation-driven estimation and modular optimization can make activation checkpointing more effective, interpretable, and practical at scale.

6

Thesis Summary

Training large AI models requires selecting optimal configurations across a vast space of parallelism strategies, memory optimizations, and precision settings. Given a model, dataset, and hardware platform, identifying and implementing the empirically optimal distributed training configuration remains a time-consuming and error-prone task, often requiring expert intuition, extensive benchmarking, and significant iteration. This thesis introduces LegoAI, a system that automates the synthesis, simulation, and deployment of efficient distributed training strategies. LegoAI au-

tomatically identifies high-performance training recipes tailored to a given context and generates production-ready implementations that scale to thousands of GPUs.

LegoAI comprises two core subsystems. The first, TorchTitan, is a unified, modular, and production-grade training framework that supports composable 4-D parallelism, hardware-software co-optimization, and fault-tolerant deployment. TorchTitan enables seamless integration of Fully Sharded Data Parallelism (FSDP), Tensor Parallelism (TP), Pipeline Parallelism (PP), and Context Parallelism (CP). Across a range of LLaMA 3.1 models, TorchTitan achieves speedups of 65.08% for 8B at 128 GPUs, 12.59% for 70B at 256 GPUs, and 30% for 405B at 512 GPUs over optimized baselines, while enabling long-context training with 4D parallelism on NVIDIA H100 clusters.

The second subsystem, TorchSim, is a high-fidelity simulator for predicting runtime and memory usage without requiring actual GPU execution. It combines learned compute models with analytical communication models and emulates GPU multi-stream execution behavior, capturing operator-level compute-communication overlap and tensor memory lifetimes. TorchSim achieves 99.9% accuracy in memory estimation and over 90% accuracy in runtime prediction across diverse models (e.g., CLIP, T5, LLaMA), hardware platforms (A100, H100), and interconnects (InfiniBand, RoCE), covering FSDP, TP, CP, and hybrid configurations.

To demonstrate the extensibility of LegoAI and its ability to drive optimization, this thesis introduces a principled and scalable system for generating selective activation checkpointing (SAC) policies. It uses TorchSim's fine-grained predictions to construct empirical memory-compute trade-off curves and formulates SAC as a two-level optimization problem: a global ILP selects memory-constrained modules for checkpointing, and local solvers generate operator-level SAC policies using heuristics, dynamic programming, or exact ILPs. Integrated into TorchTitan, this system reduces recomputation overhead by up to 90% compared to naïve checkpointing. Its heuristic solvers match ILP-level performance while running several orders of magnitude faster, making

SAC tractable at scale.

Together, these contributions establish LegoAI as the first system to unify synthesis, simulation, and deployment for distributed training. By eliminating redundant experimentation, optimizing memory-compute trade-offs, and scaling efficiently across models and hardware, LegoAI significantly reduces the cost and complexity of large-scale AI training while expanding the design space of what is possible.

References

(2025). *Autograd Mechanics: Backward Hooks Execution*. PyTorch. Accessed: April 1, 2025.

(2025a). *Understanding CUDA Memory Usage*. PyTorch. Accessed: April 1, 2025.

(2025b). *weakref — Weak references*. Python Software Foundation. Accessed: April 1, 2025.

Abdin, M., Jacobs, S. A., Awan, A. A., Aneja, J., Awadallah, A., Awadalla, H., Bach, N., Bahree, A., Bakhtiari, A., Behl, H., et al. (2024). Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Alexey, D. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv: 2010.11929*.

AMD (2025). Amd radeon gpu profiler. Accessed: April 3, 2025.

Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., & Team, G. (2023). Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Suo, M., Tillet, P., Wang, E., Wang, X., Wen, W., Zhang, S., Zhao, X., Zhou, K., Zou, R., Mathews, A., Chanan, G., Wu, P., & Chintala, S. (2024a). PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*: ACM.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E.,

Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., & Chintala, S. (2024b). Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24 (pp. 929–947). New York, NY, USA: Association for Computing Machinery.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., & Huang, J. (2025a). A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, (pp. 1–20).

Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., & Huang, J. (2025b). A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, (pp. 1–20).

Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., & Huang, J. (2025c). A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, (pp. 1–20).

Chen, H., Yu, C. H., Zheng, S., Zhang, Z., Zhang, Z., & Wang, Y. (2023). Slapo: A schedule language for progressive optimization of large deep learning model training.

Chen, T., Xu, B., Zhang, C., & Guestrin, C. (2016). Training deep nets with sublinear memory cost.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2023). Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240), 1–113.

CloudPrice.net (2025). Aws p5.48xlarge specs and prices. Accessed: February 21, 2025.

Contributors, P. (2024). Port fakeprocessgroup to cpp. Accessed: 2025-03-31.

Contributors, P. (2025). Fake tensor. Accessed: 2025-03-31.

DeepSeek-AI (2025). Deepseek-v3 technical report.

Desmaison, A. (2021a). Pytorch hooks part 1: All the available hooks. Accessed: April 1, 2025.

Desmaison, A. (2021b). Pytorch hooks part 2: nn.module hooks. Accessed: April 1, 2025.

Devlin, J. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. (2024). The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Eisenman, A., Matam, K. K., Ingram, S., Mudigere, D., Krishnamoorthi, R., Nair, K., Smelyanskiy, M., & Annavaram, M. (2022). Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (pp. 929–943). Renton, WA: USENIX Association.

Fang, J. & Zhao, S. (2024). Usp: A unified sequence parallelism approach for long context generative ai.

Gao, J., Ji, W., Chang, F., Han, S., Wei, B., Liu, Z., & Wang, Y. (2023). A systematic survey of general sparse matrix-matrix multiplication. *ACM Comput. Surv.*, 55(12).

Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., & Yang, M. (2020). Estimating gpu memory consumption of deep learning models. In *ESEC/FSE 2020* (pp. 1342–1352).: ACM. The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Industry Track.

Geoffrey, X. Y., Gao, Y., Golikov, P., & Pekhimenko, G. (2021). Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (pp. 503–521).

Gupta, T., Krishnan, S., Kumar, R., Vijeev, A., Gulavani, B., Kwatra, N., Ramjee, R., & Sivathanu, M. (2024). Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24 (pp. 1110–1125). New York, NY, USA: Association for Computing Machinery.

He, H., Desmaison, A., Yang, E., & Zou, R. (2022). What (and Why) is torch dispatch? Accessed: 2025-03-31.

He, H. & Yu, S. (2023). Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. *Proceedings of Machine Learning and Systems*, 5, 414–427.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2019a). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., & Chen, Z. (2019b). *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc.: Red Hook, NY, USA.

Ilharco, G., Wortsman, M., Wightman, R., Gordon, C., Carlini, N., Taori, R., Dave, A., Shankar, V., Namkoong, H., Miller, J., Hajishirzi, H., Farhadi, A., & Schmidt, L. (2021). Openclip. If you use this software, please cite it as below.

Inc., B. (2024). veScale: A scalable and efficient distributed training framework. https://github.com/volcengine/veScale. Accessed: 2024-11-21.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. (2024). Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Justus, D., Brennan, J., Bonner, S., & McGough, A. S. (2018). Predicting the computational cost of deep learning models. In *2018 IEEE international conference on big data (Big Data)* (pp. 3873–3882).: IEEE.

Kempner Institute (2025). Distributed gpu computing. https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/distributed_gpu_computing.html. Accessed: 2025-05-05.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., & Catanzaro, B. (2023). Reducing activation recomputation in large transformer models. In D. Song, M. Carbin, & T. Chen (Eds.), *Proceedings of Machine Learning and Systems*, volume 5 (pp. 341–353).: Curan.

Labs, B. F. (2024). Flux. https://github.com/black-forest-labs/flux.

Lamy-Poirier, J. (2023). Breadth-first pipeline parallelism. In *MLSys*.

Lee, S., Phanishayee, A., & Mahajan, D. (2025a). Forecasting gpu performance for deep learning training and inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (pp. 493–508).

Lee, S., Phanishayee, A., & Mahajan, D. (2025b). Forecasting GPU performance for deep learning training and inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (pp. 493–508). New York, NY, USA: ACM.

Li, F., Zhang, R., Zhang, H., Zhang, Y., Li, B., Li, W., Ma, Z., & Li, C. (2024a). Llava-next-interleave: Tackling multi-image, video, and 3d in large multimodal models. *CoRR*, abs/2407.07895.

Li, J., Qin, Z., Mei, Y., Cui, J., Song, Y., Chen, C., Zhang, Y., Du, L., Cheng, X., Jin, B., Zhang, Y., Ye, J., Lin, E., & Lavery, D. (2024b). onednn graph compiler: A hybrid approach for high-performance deep learning compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (pp. 460–470).

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. (2020). Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*.

Li, Y., Sun, Y., & Jog, A. (2023). Path forward beyond simulators: Fast and accurate gpu execution time prediction for dnn workloads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 380–394).

Li, Z., Paolieri, M., & Golubchik, L. (2022). Inference latency prediction at the edge. *arXiv preprint arXiv:2210.02620*.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., Purandare, S., Nadathur, G., & Idreos, S. (2024). Torchtitan: One-stop pytorch native solution for production ready llm pre-training.

Liu, H. & Abbeel, P. (2024). Blockwise parallel transformers for large context models. *Advances in Neural Information Processing Systems*, 36.

Liu, H., Zaharia, M., & Abbeel, P. (2023). Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*.

Liu, Y., Ott, M., & Goyal, N. (2019). Jingfei du, mandar joshi, danqi chen, omer levy, mike lewis, luke zettlemoyer, and veselin stoyanov. 2019. roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 1(3.1), 3–3.

Maurya, A., Underwood, R., Rafique, M. M., Cappello, F., & Nicolae, B. (2024). Datastates-llm: Lazy asynchronous checkpointing for large language models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24 (pp. 227–239). New York, NY, USA: Association for Computing Machinery.

Meta AI (2024). Introducing llama 3.2: Vision models for mobile and edge devices. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/. Accessed: 2025-05-01.

Meta Platforms, Inc. (2024). PyTorch Distributed. Accessed: 2023-09-26.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2018). Mixed precision training.

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., & Wu, H. (2022). Fp8 formats for deep learning.

Mohammad, A., Darbaz, U., Dozsa, G., Diestelhorst, S., Kim, D., & Kim, N. S. (2017). dist-gem5: Distributed simulation of computer clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*: IEEE.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., & Zaharia, M. (2019). Pipedream: generalized pipeline parallelism for dnn training.

In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19 (pp. 1–15). New York, NY, USA: Association for Computing Machinery.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., & Zaharia, M. (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*, SC '21 New York, NY, USA: Association for Computing Machinery.

NVIDIA (2023). Megatron Core API Guide: Context Parallel. Accessed: 2023-09-25.

NVIDIA (2025). Nvidia nsight systems. Accessed: April 3, 2025.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc.: Red Hook, NY, USA.

Purandare, S., Wasay, A., Idreos, S., & Jain, A. (2023). μ-TWO: 3 Faster Multi-Model Training with Orchestration and Memory Optimization. In D. Song, M. Carbin, & T. Chen (Eds.), *Proceedings of Machine Learning and Systems*, volume 5 (pp. 541–562).: Curan.

PyTorch Community (2023a). Float8 in PyTorch 1.x. PyTorch Discussion Thread.

PyTorch Community (2023b). PyTorch DTensor RFC. GitHub Issue.

PyTorch Team (2024a). Enabling Float8 all-gather in FSDP2. https://discuss.pytorch.org/t/distributed-w-torchtitan-enabling-float8-all-gather-in-fsdp2/209323. PyTorch Forum Post.

PyTorch Team (2024b). Introducing Async Tensor Parallelism in PyTorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487. PyTorch Forum Post.

PyTorch Team (2024c). Optimizing checkpointing efficiency with PyTorch DCP. https://discuss.pytorch.org/t/distributed-w-torchtitan-optimizing-checkpointing-efficiency-with-pytorch-dcp/211250. PyTorch Forum Post.

PyTorch Team (2024d). Training with zero-bubble Pipeline Parallelism. https://discuss.pytorch.org/t/distributed-w-torchtitan-training-with-zero-bubble-pipeline-parallelism/214420. PyTorch Forum Post.

PyTorch Team (2025). Breaking barriers: Training long context llms with 1M sequence length in PyTorch using Context Parallel. https://discuss.pytorch.org/t/distributed-w-torchtitan-breaking-barriers-training-long-context-llms-with-1m-sequence-length-in-pytorc 215082. PyTorch Forum Post.

Qi, P., Wan, X., Huang, G., & Lin, M. (2024). Zero bubble (almost) pipeline parallelism. In *ICLR*.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. In *International conference on machine learning* (pp. 8748–8763).: PMLR.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8), 9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020a). Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1).

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020b). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140), 1–67.

Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). Zero: memory optimizations toward training trillion parameter models. SC '20: IEEE Press.

Rasley, J., Rajbhandari, S., Ruwase, O., & He, Y. (2020). Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. KDD '20 (pp. 3505–3506). New York, NY, USA: Association for Computing Machinery.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2021). High-resolution image synthesis with latent diffusion models.

Shi, A. & DeVito, Z. (2023). Understanding gpu memory 1: Visualizing all allocations over time.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Steiner, A., Kolesnikov, A., Zhai, X., Wightman, R., Uszkoreit, J., & Beyer, L. (2021). How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*.

Su, Q., Yang, J., & Pekhimenko, G. (2024). Boom: Use your desktop to accurately predict the performance of large deep neural networks. In *Proceedings of the 2024 International Conference*

on Parallel Architectures and Compilation Techniques, PACT '24 (pp. 284–296). New York, NY, USA: Association for Computing Machinery.

Tang, D., Jiang, L., Jin, M., Zhou, J., Li, H., Zhang, X., & Pei, Z. (2024a). Adaptive blockwise task-interleaved pipeline parallelism.

Tang, D., Jiang, L., Zhou, J., Jin, M., Li, H., Zhang, X., Pei, Z., & Zhai, J. (2024b). Zeropp: Unleashing exceptional parallelism efficiency through tensor-parallelism-free methodology.

Tazi, N., Mom, F., Zhao, H., Nguyen, P., Mekkouri, M., Werra, L., & Wolf, T. (2025). Ultrascale playbook. Accessed: February 23, 2025.

Team, G., Riviere, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., Hussenot, L., Mesnard, T., Shahriari, B., Ramé, A., et al. (2024). Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

Wan, B., Han, M., Sheng, Y., Lai, Z., Zhang, M., Zhang, J., Peng, Y., Lin, H., Liu, X., & Wu, C. (2024). Bytecheckpoint: A unified checkpointing system for llm development.

Wang, S., Wei, J., Sabne, A., Davis, A., Ilbeyi, B., Hechtman, B., Chen, D., Murthy, K. S., Maggioni, M., Zhang, Q., et al. (2022). Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (pp. 93–106).

Wang, Z., Jia, Z., Zheng, S., Zhang, Z., Fu, X., Ng, T. S. E., & Wang, Y. (2023). Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23 (pp. 364–381). New York, NY, USA: Association for Computing Machinery.

Won, W., Heo, T., Rashidi, S., Sridharan, S., Srinivasan, S., & Krishna, T. (2023). ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale.

Xiong, D., Chen, L., Jiang, Y., Li, D., Wang, S., & Wang, S. (2024). Revisiting the time cost model of AllReduce.

Yang, E. Z. (2022). Deeply rework weakidkeydictionary. Accessed: April 1, 2025.

Yu, C. H., Fan, H., Huang, G., Jia, Z., Liu, Y., Wang, J., Zheng, Z., Zhou, Y., Shen, H., Shao, J., Li, M., & Wang, Y. (2023). Raf: Holistic compilation for deep learning model training.

Yu, G. X., Grossman, T., & Pekhimenko, G. (2020). Skyline: Interactive in-editor computational performance profiling for deep neural network training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20 (pp. 126–139). New York, NY, USA: Association for Computing Machinery.

Zhang, B., Luo, L., Liu, X., Li, J., Chen, Z., Zhang, W., Wei, X., Hao, Y., Tsang, M., Wang, W., Liu, Y., Li, H., Badr, Y., Park, J., Yang, J., Mudigere, D., & Wen, E. (2022a). Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction.

Zhang, L. L., Han, S., Wei, J., Zheng, N., Cao, T., & Liu, Y. (2022b). nn-meter: Towards accurate latency prediction of dnn inference on diverse edge devices. *GetMobile: Mobile Computing and Communications*, 25(4), 19–23.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., & Li, S. (2023). Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12), 3848–3860.