TorchSim: High Fidelity Runtime and Memory Estimation for Distributed Training

Sanket Purandare [1 2] **Emma Yang** [1] **Andrew Zhao** [1] **Qitong Wang** [1] **Wei Feng** [2] **Alban Desmaison** [2]
Andrew Gu [2] **Tianyu Liu** [2] **Less Wright** [2] **Gokul Nadathur** [2] **Stratos Idreos** [1]

Abstract

Large AI models unlock powerful applications but are costly and complex to train, primarily due to the challenge of configuring distributed training across GPU clusters. This involves selecting the right combination of techniques based on the model, data, hardware, and performance objectives. In practice, teams often rely on trial and error, leading to high compute costs, cloud spend, and wasted time, without guarantees of success or optimality. We present TORCHSIM, a simulator that eliminates this burden by accurately predicting whether a configuration will succeed (i.e., stay within memory limits) and how long it will take to run, without requiring actual execution or access to the target hardware. Users simply input candidate configurations and choose the best successful one, such as the fastest, avoiding costly and uncertain tuning. TORCHSIM combines analytical and learned models to estimate operator-level runtimes and employs a GPU execution simulator to capture the intricacies of multi-stream parallelism and hardware behavior. Evaluated on both language and vision models across A100 and H100 GPUs, up to 128-GPU scale, with multi-dimensional parallelism and interconnects like InfiniBand and RoCE, TORCHSIM achieves over 90% accuracy in runtime prediction and 99% in memory estimation. It is open-sourced as an extension to PyTorch, with results demonstrated on TORCHTITAN.

1. Motivation and Introduction

Large AI models power a wide range of applications, but their training has become increasingly expensive and complex. Achieving state-of-the-art performance at this scale demands extreme computational investment. For example, Llama 3.1 used 405 billion parameters and 15 trillion tokens, consuming 30.84 million GPU hours across 16,000 H100s (Dubey et al., 2024), while Google's PaLM used 540 billion parameters and 0.8 trillion tokens, requiring 9.4 million TPU hours on 6,144 TPUv4 chips (Chowdhery et al., 2023). These efforts highlight not only the capabilities of modern models but also the prohibitive resource demands, making training cost a major bottleneck for scalable AI. Distributed training is essential for scaling, but no universally optimal strategy exists. For instance, Llama 3.1 employed 4D parallelism, 8-way Tensor, 16-way Context, 16-way Pipeline, and 8-way Fully Sharded Data Parallelism, while PaLM used 3D parallelism, 12-way Tensor, 256-way Fully Sharded Data, and 2-way Data Parallelism, each carefully tailored to model architecture and hardware constraints.

Scaling LLMs requires carefully combining parallelism strategies and system-level optimizations. This includes Data Parallelism (Li et al., 2020; Rajbhandari et al., 2020; Zhang et al., 2022a; Zhao et al., 2023), Tensor Parallelism(Narayanan et al., 2021; Wang et al., 2022; Korthikanti et al., 2023), Context Parallelism (Liu et al., 2023; Liu & Abbeel, 2024; NVIDIA, 2023; Fang & Zhao, 2024), and Pipeline Parallelism (Huang et al., 2019b; Narayanan et al., 2019; 2021; Tang et al., 2024b), often combined with techniques like activation recomputation (Chen et al., 2016; Korthikanti et al., 2023; He & Yu, 2023; Purandare et al., 2023), mixed precision (Micikevicius et al., 2018; 2022), and deep learning compilers (Bradbury et al., 2018; Yu et al., 2023; Li et al., 2024; Ansel et al., 2024b) to maximize efficiency. Moreover, identifying an effective training recipe is highly context-specific, requiring expert intuition and repeated experimentation across a large space of configurations involving parallelism dimensions, sharding strategies, memory trade-offs, and precision modes (Eisenman et al., 2022; Wang et al., 2023; Gupta et al., 2024; Maurya et al., 2024; Wan et al., 2024).

Even when frameworks support advanced optimizations, suboptimal configurations can be extremely costly. For instance, LLaMA 3.1 and PaLM consumed 30.84 million

[1]Harvard University, USA [2]Meta, USA. Correspondence to: Sanket Purandare <sanketpurandare@meta.com>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).

GPU hours and 9.4 million TPU hours, respectively. If trained with configurations just 10–25% slower than optimal (Tazi et al., 2025), they would have required up to 7.71 million additional GPU hours or 2.35 million extra TPU hours, resulting in significant financial and environmental costs.

Crafting an optimal training recipe requires expensive, trial-and-error exploration of a large and sensitive configuration space, making automated and simulation-driven approaches essential for scalable training.

A performance benchmarking study by Huggingface (Tazi et al., 2025) illustrates the high cost and complexity of empirical training configuration exploration for LLaMA models (1.34B to 80B parameters). Fixing the global batch size and sequence length, the study varied key parameters such as node count, degrees of data, tensor, and pipeline parallelism, micro-batch size, gradient accumulation steps, and ZeRO sharding strategies across 3,306 configurations using up to 512 H100 GPUs. Only 1,728 runs completed successfully, while 1,578 failed due to out-of-memory errors, offering little diagnostic value. Results showed training performance and memory efficiency are highly sensitive to configuration choices, with trade-offs between memory usage, recomputation, and synchronization overhead. The total cost of this limited benchmarking effort was approximately **$469K**, highlighting the substantial financial burden of manual exploration. Despite controlling many variables, the study explored only a narrow slice of the configuration space, underscoring the need for automated, simulation-driven methods to guide training configuration selection at scale. (Full details in §B)

The ability to estimate memory and runtime costs before execution would transform large-scale AI training by enabling practitioners to preemptively discard failing setups and select the fastest viable options, eliminating costly trial-and-error. However, achieving accurate end-to-end cost estimation is highly complex due to the interplay of parallelism strategies, hardware heterogeneity, tensor liveness, and dynamic runtime behaviors like asynchronous execution and communication-computation overlap. These factors make purely analytical modeling infeasible and brittle, especially as training frameworks evolve. Instead, robust estimation requires a simulation-based approach that faithfully captures the full complexity of modern distributed training systems.

We introduce TorchSim, a predictive tool that estimates runtime and memory consumption for distributed deep learning training workloads without requiring actual GPU execution. TorchSim combines hardware-aware compute models with topology-sensitive communication models to predict operator-level execution times, using a detailed simulator that replicates multi-stream GPU execution, including compute–communication overlap, synchronization overhead, and exposed communication phases for accurate end-to-end runtime estimation. For memory prediction, TorchSim tracks tensor usage at operator-level granularity, emulates memory consumed by collective operations, and mimics PyTorch's memory management, capturing effects from sharding, activation recomputation, and communication buffering. This unified modeling approach enables users to evaluate training configurations and cluster topologies before execution, supporting principled decision-making and eliminating the need for costly empirical benchmarking.

To support scalable and cost-effective training, TorchSim makes the following key contributions:

1. We design TorchSim as a model-agnostic, non-intrusive simulation framework that integrates with PyTorch pipelines. It supports pluggable compute and communication models across diverse hardware and distributed setups (§ 2).
2. We develop accurate compute and communication estimators using learned models and statistical techniques. TorchSim models operator-level execution, synchronization, and compute–communication overlap, supporting workflows like FSDP, TP, and CP (§ F, § E, § G).
3. We implement an operator-level memory estimator that tracks tensor allocations and deallocations, categorizes memory usage (parameters, gradients, activations, optimizer states), and reports per-device memory statistics for optimization and debugging (§ D).
4. We open-source TorchSim as a PyTorch extension and evaluate it using TorchTitan (Liang et al., 2024), along with benchmarking scripts, compute models (A100/H100), and collective communication datasets (InfiniBand and RoCE).
5. We evaluate TorchSim on diverse models (Gemma-2B, CLIP, T5, ViT, LLaMA variants), GPU types (A100, H100), cluster sizes (up to 128 GPUs), and parallelism strategies. TorchSim achieves 99% memory estimation accuracy and $\geq$ 90% runtime prediction accuracy (§ 3).

2. TorchSim Design and Workflow

Thus far, we have established the need for a fast, GPU execution-free, and high-fidelity tool for runtime and memory estimation to effectively explore the performance landscape of distributed training configurations (§1). In this section, we first present the high-level design of TorchSim and introduce its core design principles, which together define the scope and structure of our solution (§2.1). We then provide an end-to-end walkthrough of TorchSim's workflow in §2.2.

2.1. TorchSim High-level Design and Principles

To deliver accurate and generalizable estimates, TorchSim must effectively capture this complexity while remaining

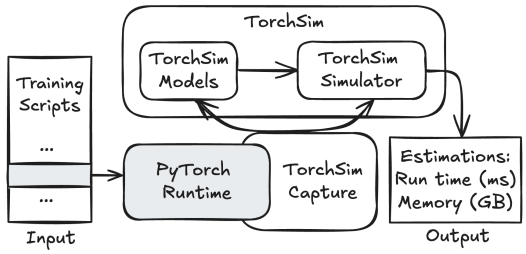


Figure 1. TORCHSIM: High-level System Design

user-friendly and broadly applicable. We now present the high-level system design of TORCHSIM (Figure 1) and introduce the core principles that guide its architecture and implementation.

TORCHSIM comprises three core components: TORCHSIM Capture, TORCHSIM Simulator, and TORCHSIM Models. 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 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.

To ensure generality, accuracy, and usability, TORCHSIM is built around the following design principles: (D1) Model-Agnostic, (D2) Algorithm and Implementation Coverage, (D3) Non-Intrusive, (D4) Accurate End-to-End Estimation, (D5) GPU Execution-Free and Fast, (D6) Insightful, (D7) Modular and Extensible.

2.2. TORCHSIM Workflow

We now present a detailed walkthrough of TORCHSIM's end-to-end workflow, illustrated in Figure 2. The infrastructure underlying TORCHSIM Capture is explained in §C. Corresponding algorithms and models are detailed in subsequent sections: the runtime simulator in §G, compute time modeling in §E, communication time modeling in §F, and memory estimation in §D.

1. **Input.** The input to TORCHSIM is a *train_step* function that receives the model, optimizer, and a sample mini-batch. It executes the forward and backward passes, followed by the optimizer step and gradient zeroing.
2. **MPMD/SPMD Estimation.** For SPMD estimation, TORCHSIM runs a single process under *FakeTensorMode* (Contributors, 2025) with a *FakeProcessGroup* (Contributors, 2024). For MPMD estimation, it launches N processes (one per pipeline stage), each using *FakeProcessGroup*. Memory is estimated per stage independently, while runtime estimation requires coordination among processes, managed by the runtime simulator.
3. **Metadata Capture.** TORCHSIM Capture enables GPU-free simulation by executing the training script while preserving operator behavior. It leverages PyTorch's *FakeTensorMode* to represent tensors using only metadata and uses *FakeProcessGroup* to emulate collectives over virtual device meshes. All operator dispatches are intercepted via *TorchDispatchMode* (He et al., 2022), allowing TORCHSIM to capture inputs, outputs, and metadata such as stream, resource, and synchronization information. Synchronization primitives, including stream waits, event records, and barriers, are captured through dynamic function overrides, ensuring faithful simulation across strategies like FSDP, TP, CP, and PP.
4. **Memory Simulator.** TORCHSIM estimates memory by tracking tensor liveness and allocation metadata captured at operator dispatch time. For each tensor, it records *size*, *device*, and *dtype*, and maintains a live memory snapshot to model allocation and deallocation events. It produces both per-module and global peak memory statistics throughout execution (§ D).
5. **Operator Runtime Estimation.** For each intercepted operator, TORCHSIM classifies it as compute or communication. It then extracts relevant features for estimation: compute ops use metadata such as *input/output shapes*, *dtype*, and backend-specific attributes; collective ops include *data size*, *collective type*, and *process group*. These features are fed into learned or analytical models for latency estimation (§ E, § F).
6. **Runtime Simulator.** The runtime simulator consumes metadata for each operator, including CUDA stream, resource type (compute/comm), dispatch order, and synchronization dependencies. It enqueues each op into its corresponding stream queue and simulates multi-stream GPU execution with accurate compute–communication overlap and synchronization effects (§ G).
7. **Output.** TORCHSIM produces a module-wise breakdown of compute time, communication time, exposed communication time (non-overlapping), and total simulated runtime. It also provides peak memory usage and memory snapshots at various stages of execution.

3. Experimental Results

We now present the results of our distributed training evaluation, demonstrating the effectiveness of TORCHSIM at scale. Specifically, we assess runtime and memory prediction accuracy using the LLaMA 3.1 70B model across 128 GPUs under two settings: 1D Fully Sharded Data Parallel (FSDP) and 2D FSDP with Tensor Parallelism (FSDP+TP). Each machine contains 4 GPUs connected via NVLink, and 16 or

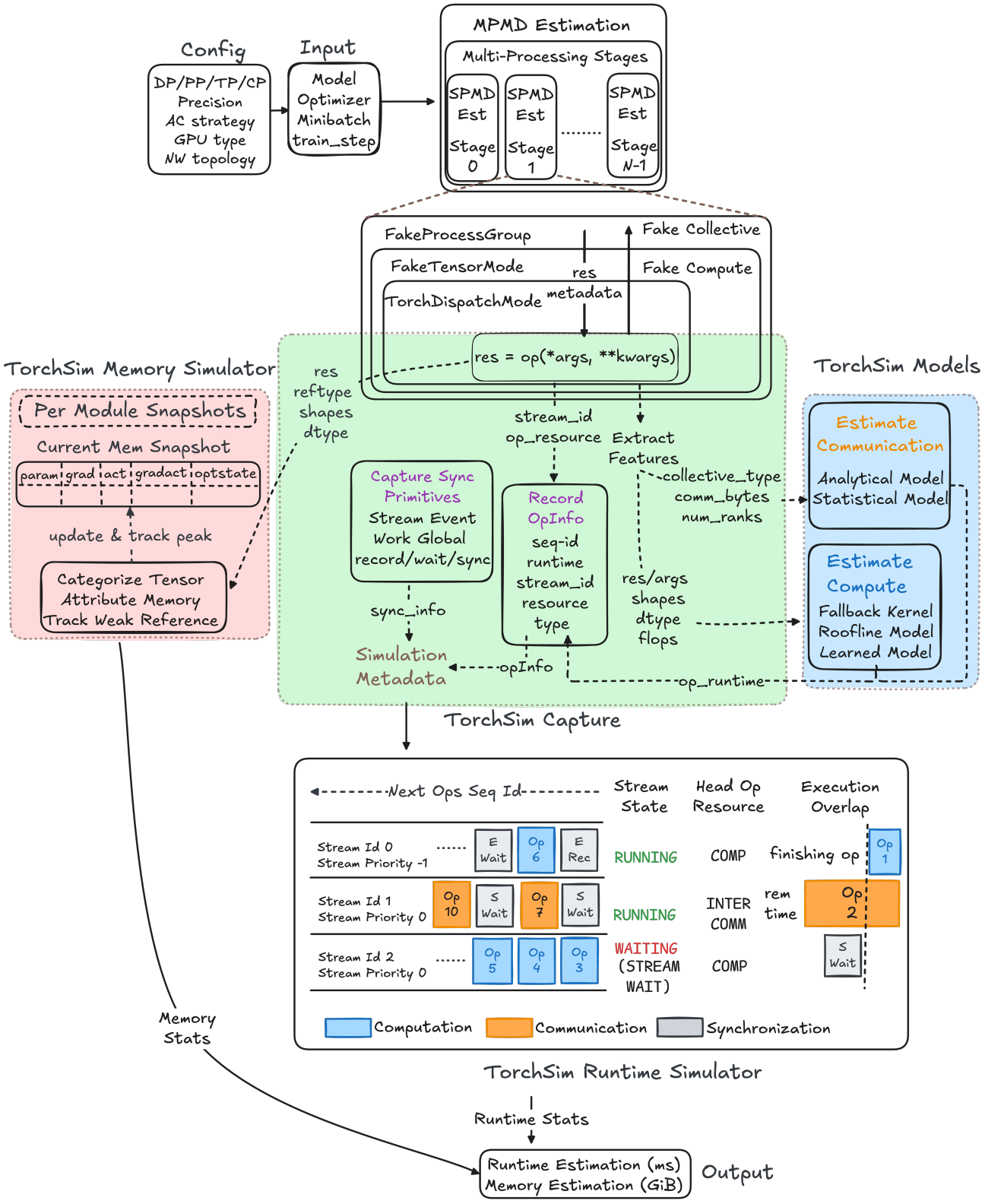


Figure 2. TORCHSIM design internals for capturing tensor, operator, and synchronization primitive metadata to enable precise memory and runtime estimation.

32 such machines are interconnected through InfiniBand.

3.1. Runtime Simulation

Table 17 presents the results for 1D FSDP, while Table 18 shows results for 2D FSDP+TP. In both configurations, TORCHSIM achieves high prediction accuracy (90–91%) with minimal simulation overhead.

Table 1. Runtime prediction accuracy of TORCHSIM for 1D FSDP during LLaMA 3 70B training on 128 GPUs. TORCHSIM achieves over **90%** mean accuracy with minimal prediction overhead.

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 2. Runtime prediction accuracy of TORCHSIM for 2D FSDP during LLaMA 3 70B training on 128 GPUs. TORCHSIM achieves **91%** mean accuracy with minimal prediction overhead.

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

3.2. Memory Simulation

Tables 20 and 21 present memory estimation results for 1D FSDP and 2D FSDP+TP, respectively. Memory Simulator consistently achieves $\geq$99% accuracy even under complex memory patterns introduced by FSDP and TP. All estimations complete in under 30 seconds.

Table 3. Memory Simulator achieves $\geq$99% accuracy for 1D FSDP training of LLaMA 70B on 64 GPUs, with estimations completed in under 30 seconds.

BATCH SIZE	SEQ LEN	AC	EST. (GIB)	ACTUAL (GIB)	ACC	TIME (S)
2	64	SELECTIVE	30.10	30.20	0.995	31.91
2	256	SELECTIVE	30.65	30.97	0.989	31.86
2	1024	FULL	30.50	30.70	0.995	30.36
1	4096	FULL	32.15	32.28	0.996	31.55
1	8192	FULL	40.13	40.18	0.998	31.10

Table 4. Memory Simulator achieves $\geq$99% accuracy for 2D FSDP+TP training of LLaMA 70B on 128 GPUs, with estimations completed in under 30 seconds.

BATCH SIZE	SEQ LEN	AC	EST. (GIB)	ACTUAL (GIB)	ACC	TIME (S)
2	64	SELECTIVE	12.78	12.87	0.992	29.57
2	256	SELECTIVE	12.78	12.87	0.992	29.63
2	1024	FULL	12.79	12.88	0.992	28.42
1	4096	FULL	12.76	12.88	0.990	28.28
1	8192	FULL	13.00	13.12	0.991	28.45

A detailed evaluation of our compute and communication models, along with extensive results for single-GPU memory and runtime estimation across diverse models and hardware, is provided in §H.

4. Related Work

Existing runtime estimation techniques are largely limited to simplified single-GPU or kernel-level settings (Geoffrey et al., 2021; Lee et al., 2025b; 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., 2025a; 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 FSDP, TP, PP, and 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, 2025b), 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. Extended related work is presented in §I.

5. Conclusion

In conclusion, we present TORCHSIM, a principled and practical framework for estimating memory and runtime in large-scale distributed training without GPU execution. TORCHSIM combines a modular design, operator-level memory tracking, and a high-fidelity simulator that models execution, synchronization, and compute–communication overlap. It achieves high accuracy by integrating learned models for compute and statistical models for communication. We show that TORCHSIM consistently delivers 99% memory estimation accuracy and over 90% runtime accuracy across diverse models, GPU types, cluster sizes, and networks. TORCHSIM is open-sourced with TORCHTITAN integration, along with pre-trained cost models and benchmark datasets, offering a production-ready tool for performance modeling in modern AI training systems.

References

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

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

AMD. Amd radeon gpu profiler, 2025. URL https://gpuopen.com/rgp/. Accessed: April 3, 2025.

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., and Chintala, S. 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, April 2024a. doi: 10.1145/3620665.3640366. URL https://pytorch.org/assets/pytorch2-2.pdf.

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., and Chintala, S. 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, 2024b. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., and Huang, J. A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–20, 2025a. ISSN 2326-3865. doi: 10.1109/tkde.2025.3554028. URL http://dx.doi.org/10.1109/TKDE.2025.3554028.

Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., and Huang, J. A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–20, 2025b. doi: 10.1109/TKDE.2025.3554028.

Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., and Huang, J. A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–20, 2025c. doi: 10.1109/TKDE.2025.3554028.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost, 2016. URL https://arxiv.org/abs/1604.06174.

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

CloudPrice.net. Aws p5.48xlarge specs and prices, 2025. URL https://cloudprice.net/aws/ec2/instances/p5.48xlarge. Accessed: February 21, 2025.

Contributors, P. Port fakeprocessgroup to cpp, 2024. URL https://github.com/pytorch/pytorch/pull/118426. Accessed: 2025-03-31.

Contributors, P. Fake tensor, 2025. URL https://pytorch.org/docs/stable/torch.compiler_fake_tensor.html. Accessed: 2025-03-31.

DeepSeek-AI. Deepseek-v3 technical report, 2025.

Desmaison, A. Pytorch hooks part 1: All the available hooks, June 2021a. URL https://dev-discuss.pytorch.org/t/pytorch-hooks-part-1-all-the-available-hooks/246. Accessed: April 1, 2025.

Desmaison, A. Pytorch hooks part 2: nn.module hooks, June 2021b. URL https://dev-discuss.pytorch.org/t/pytorch-hooks-part-2-nn-module-hooks/250. Accessed: April 1, 2025.

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

Eisenman, A., Matam, K. K., Ingram, S., Mudigere, D., Krishnamoorthi, R., Nair, K., Smelyanskiy, M., and Annavaram, M. 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, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/eisenman.

Fang, J. and Zhao, S. Usp: A unified sequence parallelism approach for long context generative ai, 2024. URL https://arxiv.org/abs/2405.07719.

Gao, J., Ji, W., Chang, F., Han, S., Wei, B., Liu, Z., and Wang, Y. A systematic survey of general sparse matrix-matrix multiplication. *ACM Comput. Surv.*, 55(12), March 2023. ISSN 0360-0300. doi: 10.1145/3571157. URL https://doi.org/10.1145/3571157.

Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., and Yang, M. Estimating gpu memory consumption of deep learning models. In *ESEC/FSE 2020*, pp. 1342–1352. ACM, November 2020. URL https://www.microsoft.com/en-us/research/publication/estimating-gpu-memory-consumption-of-deep-learning-models/. 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., and Pekhimenko, G. Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 503–521, 2021.

Gupta, T., Krishnan, S., Kumar, R., Vijeev, A., Gulavani, B., Kwatra, N., Ramjee, R., and Sivathanu, M. 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, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3650085. URL https://doi.org/10.1145/3627703.3650085.

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

He, H., Desmaison, A., Yang, E., and Zou, R. What (and Why) is torch dispatch?, 2022. URL https://dev-discuss.pytorch.org/t/what-and-why-is-torch-dispatch/557. Accessed: 2025-03-31.

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

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

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

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 341–353. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/80083951326cf5b35e5100260d64ed81-Paper-mlsys2023.pdf.

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

Lee, S., Phanishayee, A., and Mahajan, D. 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, February 2025a. ACM.

Lee, S., Phanishayee, A., and Mahajan, D. 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, 2025b.

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., and Lavery, D. 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, 2024. doi: 10.1109/CGO57630.2024.10444871.

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

Li, Y., Sun, Y., and Jog, A. 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, 2023.

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

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., and Idreos, S. Torchtitan: One-stop pytorch native solution for production ready llm pre-training, 2024. URL https://arxiv.org/abs/2410.06511.

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

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

Maurya, A., Underwood, R., Rafique, M. M., Cappello, F., and Nicolae, B. 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, 2024. Association for Computing Machinery. ISBN 9798400704130. doi: 10.1145/3625549.3658685. URL https://doi.org/10.1145/3625549.3658685.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training, 2018. URL https://arxiv.org/abs/1710.03740.

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., and Wu, H. Fp8 formats for deep learning, 2022. URL https://arxiv.org/abs/2209.05433.

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

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. 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, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. 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, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

NVIDIA. Megatron Core API Guide: Context Parallel, 2023. URL https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html. Accessed: 2023-09-25.

NVIDIA. Nvidia nsight systems, 2025. URL https://developer.nvidia.com/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., and Chintala, S. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Purandare, S., Wasay, A., Idreos, S., and Jain, A. μ-TWO: 3 Faster Multi-Model Training with Orchestration and Memory Optimization. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 541–562. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/a72071d84c001596e97a2c7e1e880559-Paper-mlsys2023.pdf.

weakref — Weak references. Python Software Foundation, 2025. URL https://docs.python.org/3/library/weakref.html. Accessed: April 1, 2025.

Autograd Mechanics: Backward Hooks Execution. PyTorch, 2025a. URL https://pytorch.org/docs/stable/notes/autograd.html#backward-hooks-execution. Accessed: April 1, 2025.

Understanding CUDA Memory Usage. PyTorch, 2025b. URL https://pytorch.org/docs/main/torch_cuda_memory.html. Accessed: April 1, 2025.

PyTorch Team. Introducing Async Tensor Parallelism in PyTorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487, 2024a. PyTorch Forum Post.

PyTorch Team. Training with zero-bubble Pipeline Parallelism. https://discuss.pytorch.org/t/distributed-w-torchtitan-training-with-zero-bubble-pipeline-parallelism/214420, 2024b. PyTorch Forum Post.

PyTorch Team. 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-pytorch-using-context-parallel/215082, 2025. PyTorch Forum Post.

Qi, P., Wan, X., Huang, G., and Lin, M. Zero bubble (almost) pipeline parallelism. In *ICLR*, 2024.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pp. 8748–8763. PMLR, 2021.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21 (140):1–67, 2020.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: memory optimizations toward training trillion parameter models. SC '20. IEEE Press, 2020. ISBN 9781728199986.

Shi, A. and DeVito, Z. Understanding gpu memory 1: Visualizing all allocations over time, December 2023. URL https://pytorch.org/blog/understanding-gpu-memory-1/.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Steiner, A., Kolesnikov, A., Zhai, X., Wightman, R., Uszkoreit, J., and Beyer, L. How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*, 2021.

Su, Q., Yang, J., and Pekhimenko, G. 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, 2024. Association for Computing Machinery. ISBN 9798400706318. doi: 10.1145/3656019.3676950. URL https://doi.org/10.1145/3656019.3676950.

Tang, D., Jiang, L., Jin, M., Zhou, J., Li, H., Zhang, X., and Pei, Z. Adaptive blockwise task-interleaved pipeline parallelism. 2024a.

Tang, D., Jiang, L., Zhou, J., Jin, M., Li, H., Zhang, X., Pei, Z., and Zhai, J. Zeropp: Unleashing exceptional parallelism efficiency through tensor-parallelism-free methodology, 2024b. URL https://arxiv.org/abs/2402.03791.

Tazi, N., Mom, F., Zhao, H., Nguyen, P., Mekkouri, M., Werra, L., and Wolf, T. Ultrascale playbook, February 2025. URL https://huggingface.co/spaces/nanotron/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. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Wan, B., Han, M., Sheng, Y., Lai, Z., Zhang, M., Zhang, J., Peng, Y., Lin, H., Liu, X., and Wu, C. Bytecheckpoint: A unified checkpointing system for llm development, 2024. URL https://arxiv.org/abs/2407.20143.

Wang, S., Wei, J., Sabne, A., Davis, A., Ilbeyi, B., Hechtman, B., Chen, D., Murthy, K. S., Maggioni, M., Zhang, Q., et al. 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, 2022.

Wang, Z., Jia, Z., Zheng, S., Zhang, Z., Fu, X., Ng, T. S. E., and Wang, Y. 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, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613145. URL https://doi.org/10.1145/3600006.3613145.

Won, W., Heo, T., Rashidi, S., Sridharan, S., Srinivasan, S., and Krishna, T. ASTRA-sim2.0: Modeling hierarchi-

cal networks and disaggregated systems for large-model training at scale. 2023.

Xiong, D., Chen, L., Jiang, Y., Li, D., Wang, S., and Wang, S. Revisiting the time cost model of AllReduce. 2024.

Yang, E. Z. Deeply rework weakidkeydictionary, December 2022. URL https://github.com/pytorch/pytorch/pull/90825. Accessed: April 1, 2025.

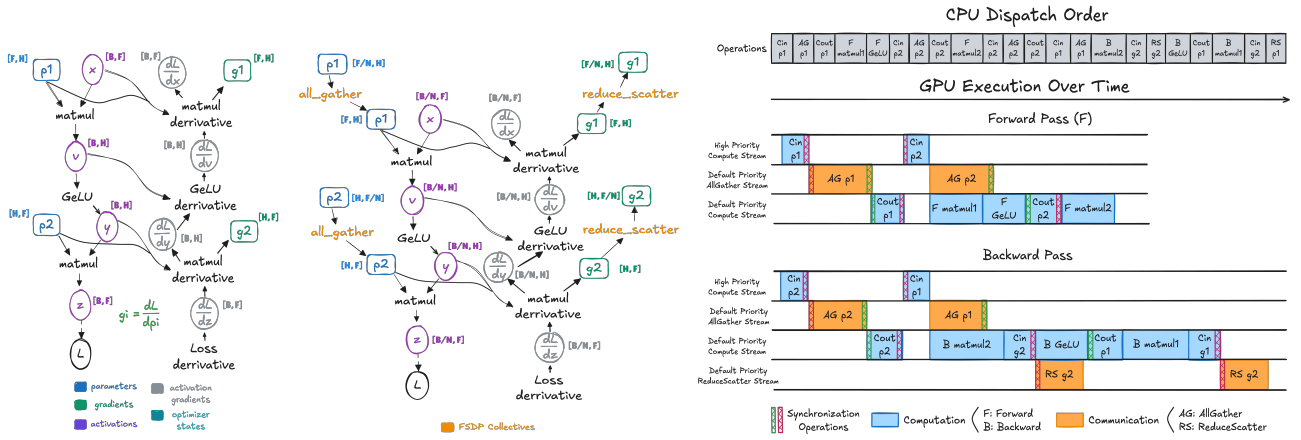
Yu, C. H., Fan, H., Huang, G., Jia, Z., Liu, Y., Wang, J., Zheng, Z., Zhou, Y., Shen, H., Shao, J., Li, M., and Wang, Y. Raf: Holistic compilation for deep learning model training, 2023. URL https://arxiv.org/abs/2303.04759.

Yu, G. X., Grossman, T., and Pekhimenko, G. 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, 2020. Association for Computing Machinery. ISBN 9781450375146. doi: 10.1145/3379337.3415890. URL https://doi.org/10.1145/3379337.3415890.

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., and Wen, E. Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction, 2022a. URL https://arxiv.org/abs/2203.11014.

Zhang, L. L., Han, S., Wei, J., Zheng, N., Cao, T., and Liu, Y. nn-meter: Towards accurate latency prediction of dnn inference on diverse edge devices. *GetMobile: Mobile Computing and Communications*, 25(4):19–23, 2022b.

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., and Li, S. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, aug 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611569. URL https://doi.org/10.14778/3611540.3611569.



(a) Parameters, gradients, optimizer states and activations retained in memory during single GPU training.

(b) 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.

(c) The operators (blue) and communication collectives (orange) dispatched by the CPU across multiple GPU streams for the example in 3b. 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.

Figure 3. Complex semantics of distributed training on GPUs

A. Background: Nuances of Distributed Training Systems

For TorchSim to accurately predict runtime and memory consumption for a single training configuration, a detailed understanding of distributed training methodologies, memory management mechanisms, and GPU execution semantics at every step of execution is essential. This section explores these fundamental concepts, highlighting the complexities that TorchSim effectively addresses.

A.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 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.

Figure 3a illustrates a complete training iteration on a single GPU using a Multilayer Perceptron (MLP) model. In the forward pass, given an input x, it is first multiplied by the parameters of the first layer (p_1), passed through a GeLU activation, and then multiplied by the second layer's parameters (p_2) to produce the output z. The loss is computed by comparing this output with the target. During the backward pass, gradients are computed via backpropagation starting from $\frac{\partial \mathcal{L}}{\partial z}$, applying the chain rule to derive gradients for each parameter (g_1 for p_1, g_2 for p_2). Parameters and intermediate activations (v and y) must be retained throughout this process, although temporary activation gradients ($\frac{\partial \mathcal{L}}{\partial z}$, $\frac{\partial \mathcal{L}}{\partial y}$ and $\frac{\partial \mathcal{L}}{\partial v}$) can be freed immediately after use to reduce memory usage. The backward pass is typically followed by an optimizer step, which updates parameters using the computed gradients and a specified *learning rate*, while maintaining per-parameter optimizer states (e.g., momentum). Parameter gradients are usually released after this step.

A.2. 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, 2024a), 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, 2024b; 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.

Using FSDP as a case study, we adapt the MLP example from Figure 3a to a distributed training setup shown in Figure 3b.

Let $p_1 \in \mathbb{R}^{F \times H}$ and $p_2 \in \mathbb{R}^{H \times F}$ be the parameters of the two layers, and $x \in \mathbb{R}^{B \times F}$ the input, where F is the feature dimension, H the hidden dimension, and B the batch size. The parameters p_1 and p_2 are sharded along the feature dimension and distributed across N GPUs, while the input x is split along the batch dimension. This reduces the memory footprint of parameters, gradients, and optimizer states on each GPU by a factor of N.

During the forward pass, parameters are unsharded using an *all_gather* operation before computation and then resharded immediately after to minimize memory usage. The backward pass follows a similar pattern: gradients are computed using unsharded parameters and then redistributed across GPUs using a *reduce_scatter* operation, effectively sharding the gradients. This strategy ensures that full parameters and gradients are only materialized when required, enabling memory-efficient distributed training.

A.3. Distributed GPU Training Execution Semantics in PyTorch

To accurately estimate memory usage and execution time, it is crucial to closely match the execution and memory behavior of the underlying training system. Figure 3c illustrates the execution flow of a model training iteration in PyTorch (Paszke et al., 2019) using a GPU accelerator.

PyTorch Fundamentals. PyTorch provides *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 multiplication, 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 reference counting. Because operations within a stream execute sequentially, reassigning freed memory to a subsequent operation after its last use ensures correctness.

Example. Figure 3c 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 execution order. When memory is allocated for an operation,

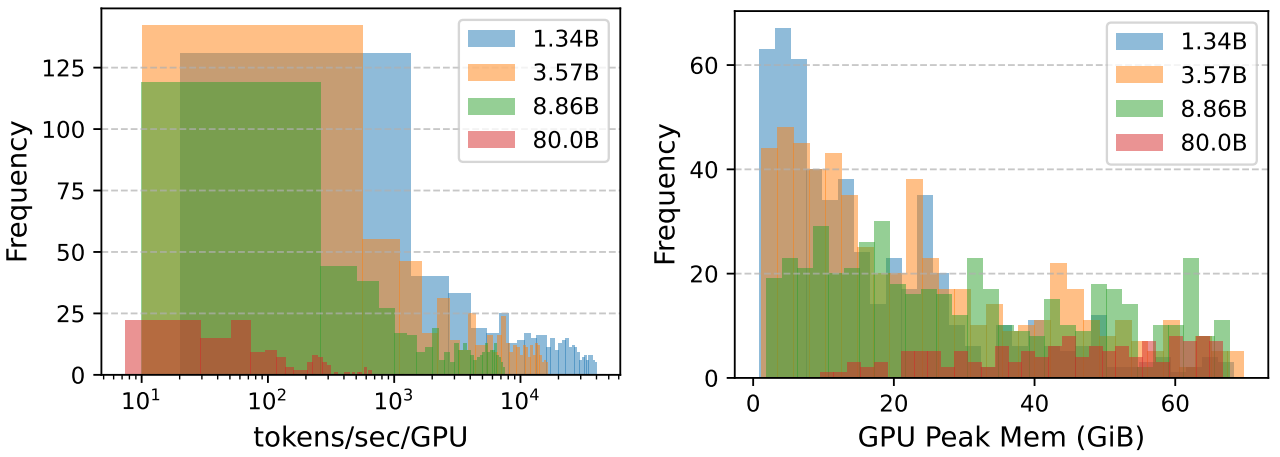


Figure 4. Huggingface performance benchmarking of 1,728 out of 3,306 training runs reveals significant variance in throughput and peak memory consumption, highlighting the impact of training configurations.

Table 5. 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

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.

To estimate end-to-end runtime and memory consumption, TorchSim closely emulates the execution semantics of the GPU stream model. Other distributed training techniques such as TP, CP, and PP are outlined in § **??**.

A.4. Distributed Model Training Paradigms

Distributed training strategies such as FSDP, TP, and CP follow the *Single Program, Multiple Data* (SPMD) paradigm, where each device runs the same program but operates on a distinct portion of the data or model. For instance, FSDP shards parameters and processes different microbatches across devices; CP partitions sequences; and TP distributes identically shaped model shards, such as attention heads or matrix blocks. Since all devices execute identical code in parallel, estimating memory and runtime for a single SPMD process is sufficient to infer system-wide behavior.

In contrast, PP adheres to the *Multiple Program, Multiple Data* (MPMD) paradigm. The model is split into sequential stages, each assigned to a different device or device group, processing distinct subsets of the model and data. When combining parallelism strategies, PP is typically applied first, with each stage internally executing its own SPMD workflow using FSDP, TP, or CP. In this setup, memory and runtime can be estimated by analyzing one SPMD process per pipeline stage. However, accurate runtime prediction requires modeling inter-stage interactions such as communication delays and scheduling dependencies between SPMD processes.

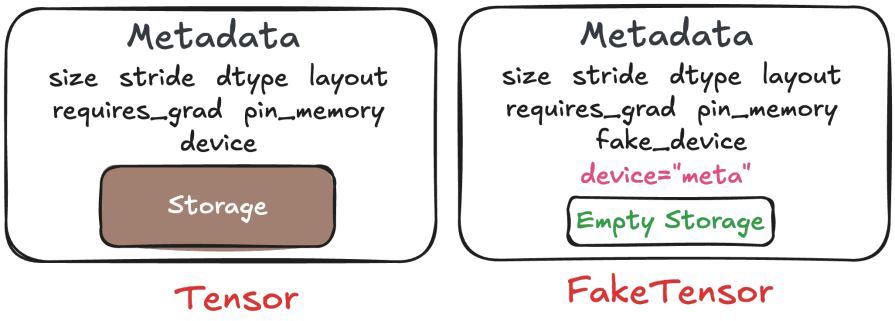


Figure 5. Actual and Fake Tensor Representation.

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

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 4, 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[1] and a runtime of five minutes per experiment. Table 5 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.

C. 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.

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

[1]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.

Storage object contains the raw data and allows for efficient memory sharing among tensors, as illustrated in Figure 5.

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 6 and 7. 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 `Tensor`s, 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 6. 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 8, 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 8 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.

D. 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 § D.1 and then explain how it achieves memory attribution and categorization in § D.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 § D.3, followed by elaboration of Memory Simulator's execution flow in §D.4.

D.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, 2025), 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.

SyncInfo	
sync_action	The action type (from SyncAction).
release_event_id release_seq_id sync_id	Identifiers for event-based, op-based, or global synchronizations.
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 priority	CUDA stream ID and priority corresponding to the queue.
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 registry	Maps a tensor's underlying storage to the list of OpInfo objects that 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 event_record_ids	Track which event IDs have been waited on or recorded, respectively.
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 7. 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 8. Synchronization primitives with PyTorch semantics and detailed actions captured by TORCHSIM.

D.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; PyT, 2025a). 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.

D.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 6.

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, gradient, 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.

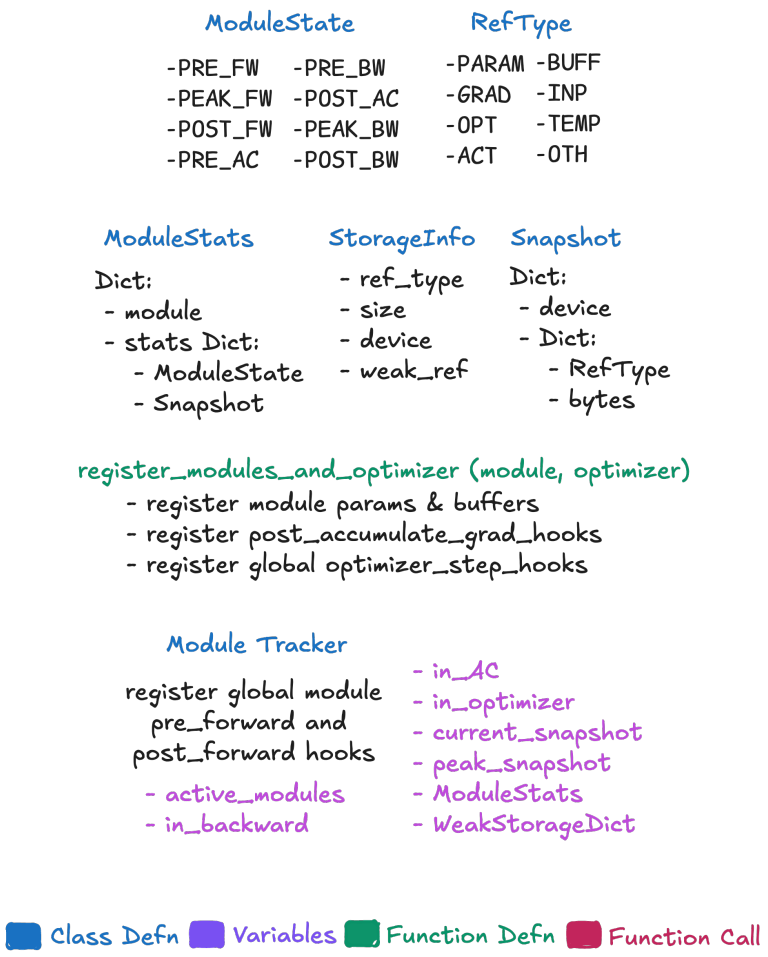


Figure 6. Memory Simulator's data structures and functions for estimating, tracking, attributing and categorizing memory usage

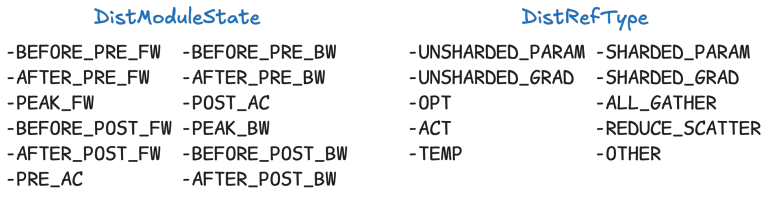


Figure 7. Extending *RefTypes* and *ModuleStates* for distributed training

(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 7) to enable more fine-grained statistics and enrich the estimation and debugging insights.

D.4. Detailed Memory Simulator execution

We first describe the functionality of Memory Simulator's core functions as shown in Figures 8a and 8b:

• *Module Hooks*: In addition to tracking the module execution and liveness as explained in § D.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*.

• *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 8c. 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*.

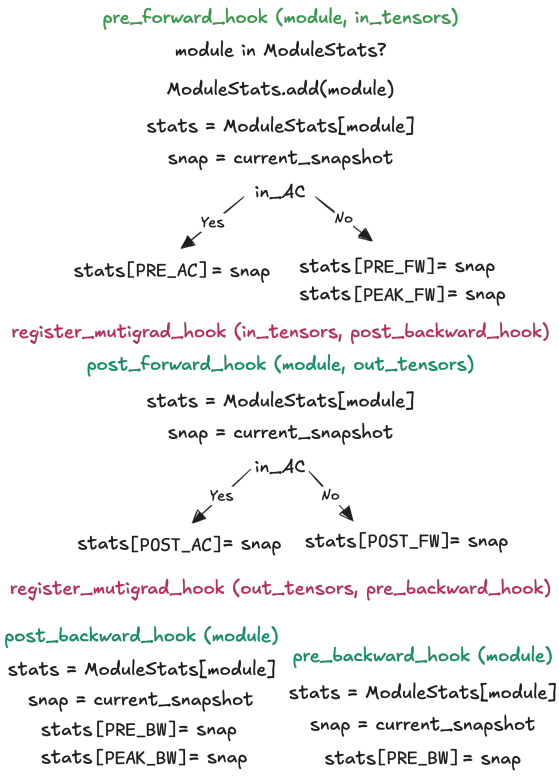
E. TorchSim Compute Time Estimation Models

In this section, we present a comprehensive modeling framework that predicts the runtime of PyTorch neural network operators.[2]

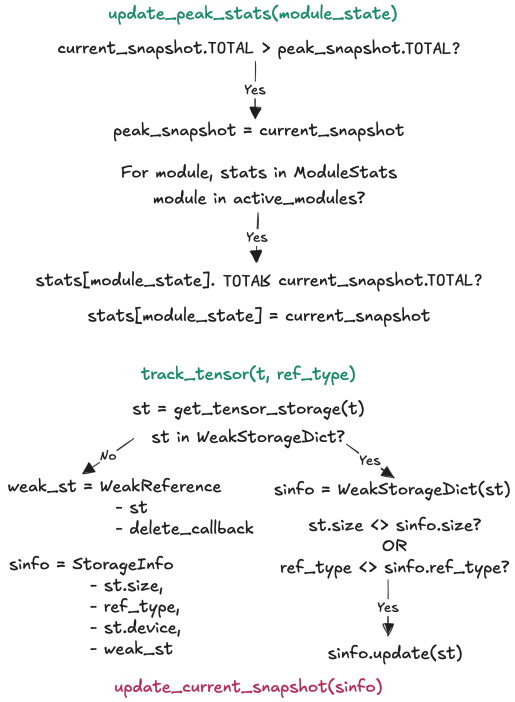
Modeling Methodology. Our predictors model a wide range of operator configurations and runtime behaviors across NVIDIA A100s and H100s and require minimal fine-tuning, limited domain expertise, and no hardware profiling. For example, for scaled dot-product attention (sdpa), we vary batch sizes, sequence 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.

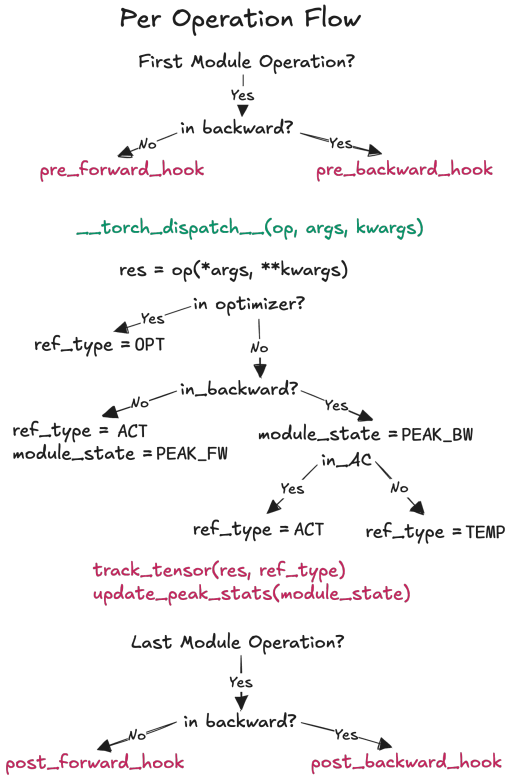
[2]Operator runtime refers to the runtime of the kernel dispatched to compute the operator.



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

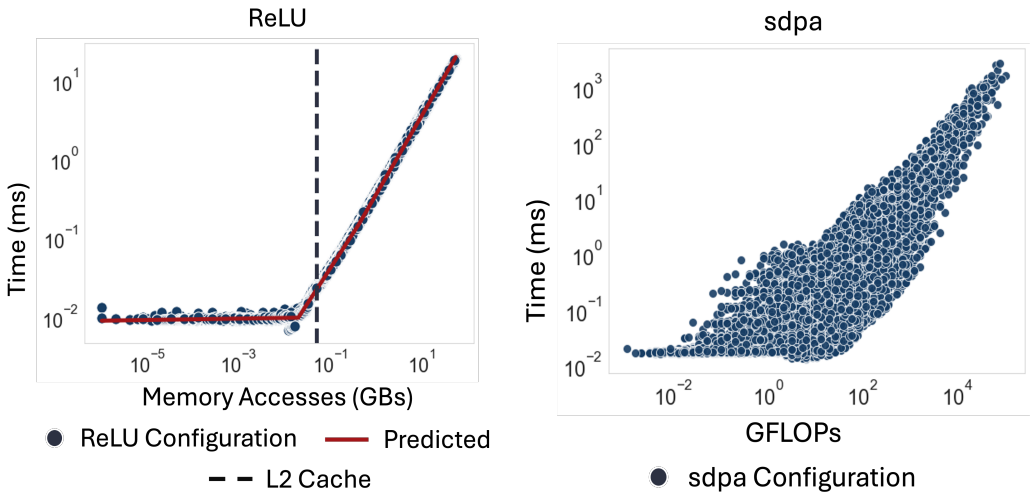


(b) Functions to robustly track Tensor storages and maintain accurate global and per-module peak statistics.



(c) The core per-operation dispatch logic

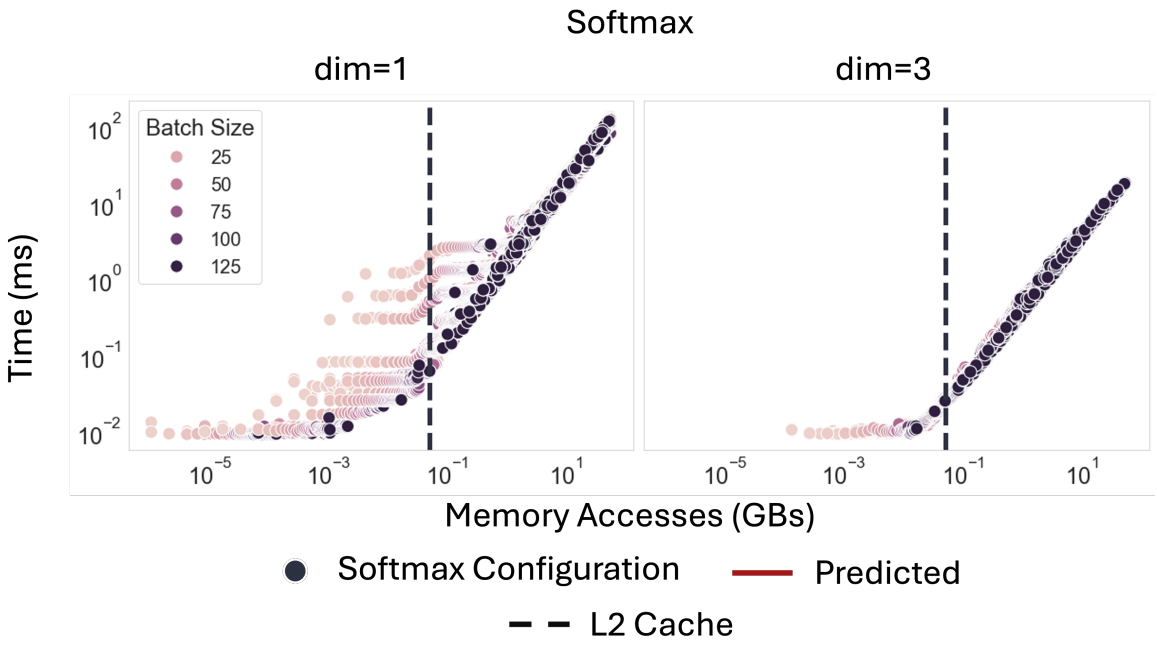
Figure 8. Memory Simulator Workflow



(a) For elementwise operators such as ReLU, their runtimes are piecewise linear in the combined input and output sizes on a log-log scale

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

We observe the elbow point denotes the transition from using only L2 cache to needing GPU memory.



(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 9. The Runtimes for Different Operator Categories

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 9a, 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

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

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 9b, 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.

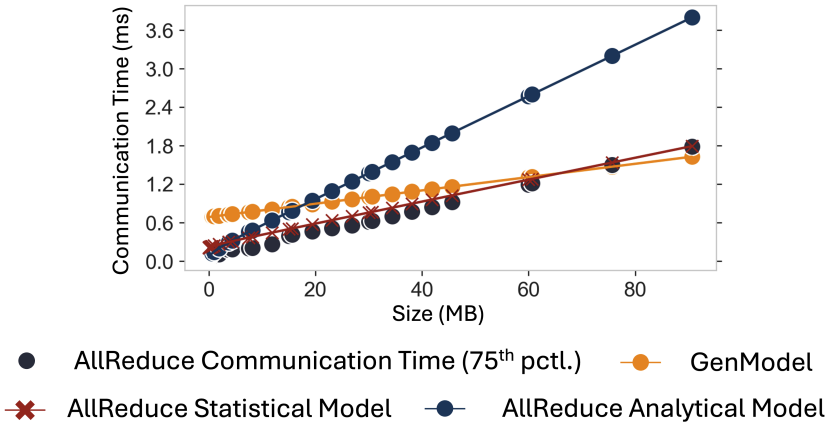
F. TorchSim Communication Time Estimation Models

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.

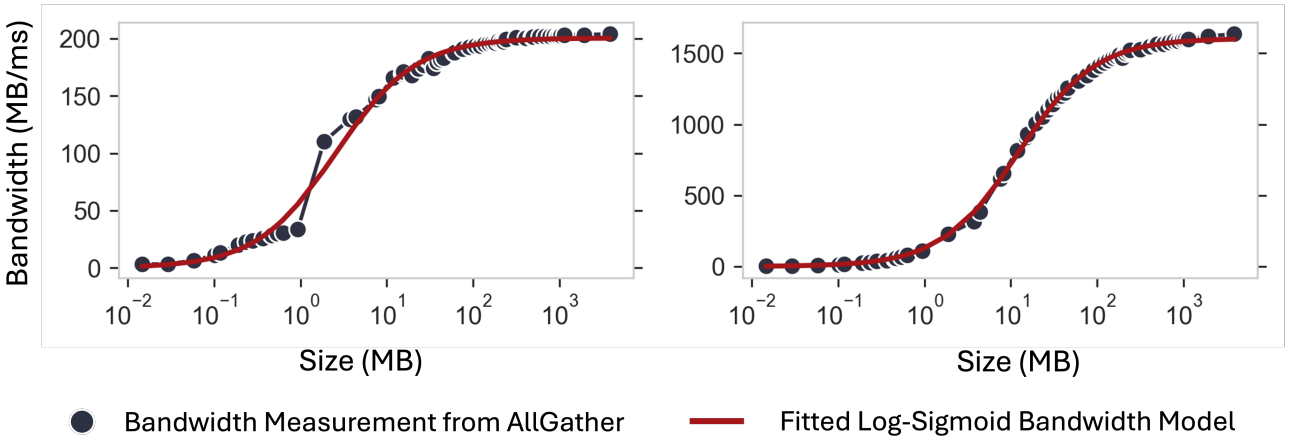
Modeling Inter-Node and Intra-Node Bandwidth. While existing cost models, including those used by NCCL and GenModel, use a single bandwidth measurement for modeling communication time, we empirically show that bandwidth varies as a function of data size. An example is shown in Figure 10b. Furthermore, 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 10b, 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 10b) to the intra-node and inter-node measurements separately:

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



(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 10. Insights from benchmarking inter-GPU communication

Table 9. 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 10. 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	$\frac{(P-N-1) \cdot S}{(P-N)\, B_C(S)} + \frac{(N-1) \cdot S}{N\, B_T(S)}$ $+N \cdot \ell_T + (P/N - 1) \cdot \ell_C$

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:

$$b \in (-\infty, -1] \quad L, k, D_0 \in (-\infty, \infty). \tag{2}$$

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 different 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 10.

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

$$t(S) = \frac{S}{l \cdot B(D_S)}. \tag{3}$$

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

$$t_{\text{AllReduce}}(S) = \frac{2(n-1) \cdot S}{l \cdot B(D_S)} \tag{4}$$

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

$$t_T(S) = \frac{2(N-1)\cdot S}{(N-1)\cdot B_T(D_S)} = \frac{2S}{B_T(D_S)} \tag{5}$$

$$t_C(S) = \frac{2(P-N)\cdot S}{(P-N)\cdot B_C(D_S)} = \frac{2S}{B_C(D_S)} \tag{6}$$

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

$$t_T(S) + t_C(S) = \frac{2S}{B_T(D_S)} + \frac{2S}{B_C(D_S)} \tag{7}$$

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

$$t_{\text{AllReduce}}(S) = t_T(S) + t_C(S) + \ell \tag{8}$$

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:

$$t_{ReduceScatter}(S) = \frac{(P-N-1)\cdot S}{(P-N)\,B_C(S)} + \frac{(N-1)\cdot S}{N\,B_R(S)} + \ell \tag{9}$$

where

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

and

$$t_{AllGather} = \frac{(P-N-1)\cdot S}{(P-N)\,B_C(S)} + \frac{(N-1)\cdot S}{N\,B_R(S)} + \ell \tag{10}$$

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.

$$\begin{aligned}\hat{t}_{\min} \sim \beta_0 + \beta_1 \cdot t + \beta_2 \cdot \ell + \beta_3 \cdot P \\ + \beta_3 * S + \beta_4 \cdot (t \cdot S) \\ + \beta_5 \cdot (t \cdot P) + \beta_6 \cdot (\ell \cdot S)\end{aligned} \tag{11}$$

where the symbols are as defined in table 9, 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 ($O(10\ \mathrm{KB})$), 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:

$$\hat{t}_{min} = \begin{cases} \hat{t}_{min} & \hat{t}_{min} > 0 \\ t & \text{otherwise} \end{cases} \tag{12}$$

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 $t_{\min}$.

$$\widehat{t_{75}/t_{\min}} \sim P \cdot \log(S) \tag{13}$$

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 re-deployed 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 β. 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 β' as follows:

$$\tilde{\beta}' = \tilde{\beta} - \eta(\beta \cdot \mathbf{x} - \mathbf{y}) \tag{14}$$

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.

G. 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 7 outlines the Queue and Runtime Simulator class definitions. We describe the algorithms for Queue management in § G.1, followed by the Simulation loop in § G.2.

G.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 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 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 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 4 checks whether all queues have completed their work. A queue is considered complete if it is in the `READY`

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

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 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.

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 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.

G.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 11 illustrates the runtime simulator in action.

Table 11 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 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

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 ∉ 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 ∉ 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 ∉ 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 11. Process_Sync_Events for a given syncInfo and Q

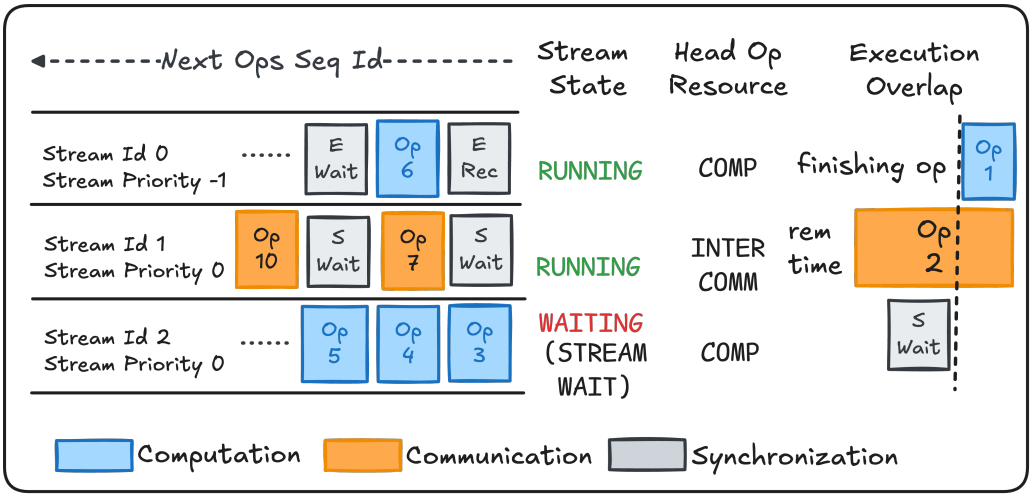


Figure 11. TORCHSIM Runtime Simulator in action.

	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 12. 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 13. Regression and Accuracy Results for the Learned Model on A100s.

H. Experimental Results

H.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 12 and Table 13, we find that our models have good test evaluation results.

In Figures 12 and 13, we see that most of the differences between predicted time and measured 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.

H.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

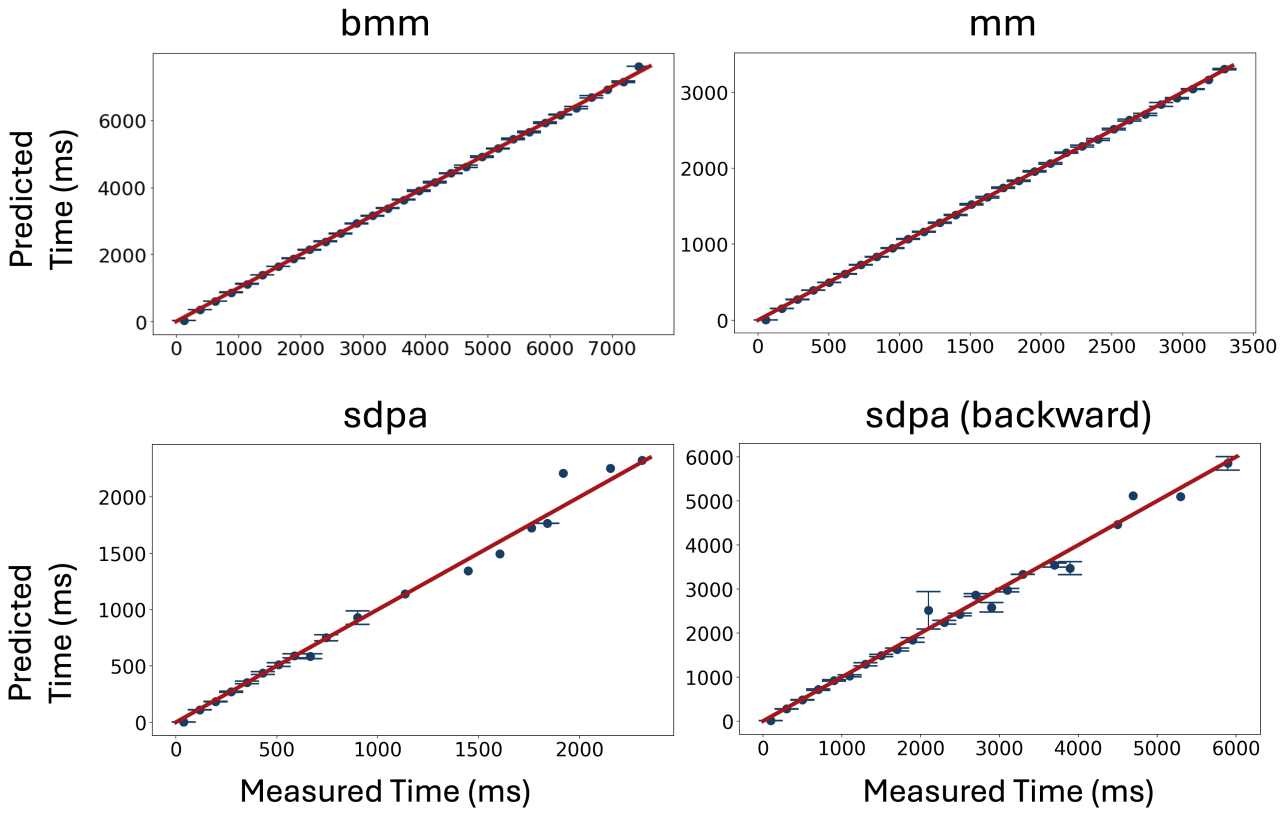


Figure 12. The predicted runtime plotted against the measured runtime for the operators on NVIDIA A100s. The red line denotes the line $y = x$. We binned measured runtimes and took a mean per bin.

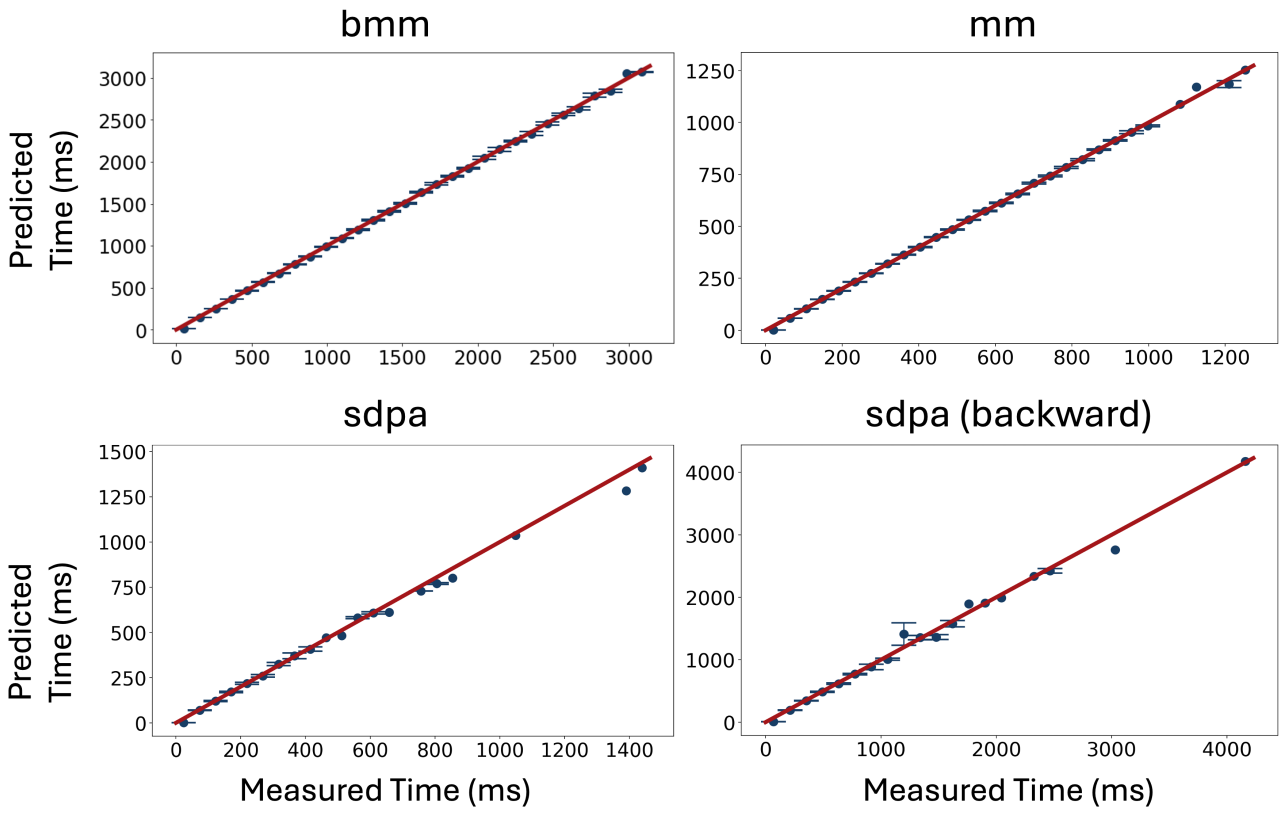


Figure 13. The predicted runtime plotted against the measured runtime for the operators on NVIDIA H100s. The red line denotes the line $y = x$. We binned measured runtimes and took a mean per bin.

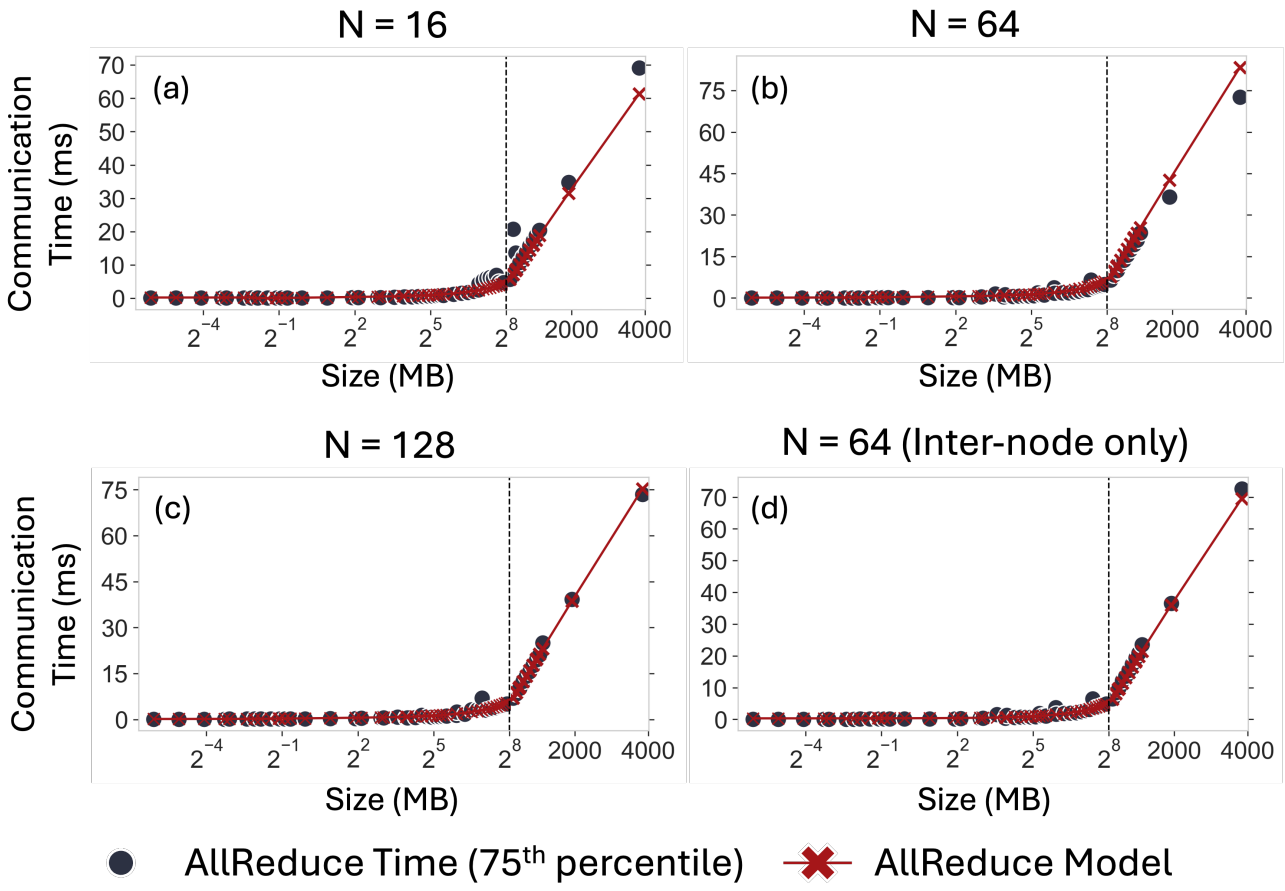


Figure 14. 75th percentile AllReduce communication time from benchmarking data and predictions by the collective communication prediction model for each data size on cluster X. (a)–(c) show data from world sizes of $N = 16, 64, 128$ for 1D parallelism whereas (d) shows data for 2D parallelism for $N = 64$.

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 $\mathbf{6}\times$ of improvement in RMSE for predicting collective latency compared to GenModel. A full table of model performance statistics is shown in Table 14.

Figures 14, 15, and 16 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 17 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.

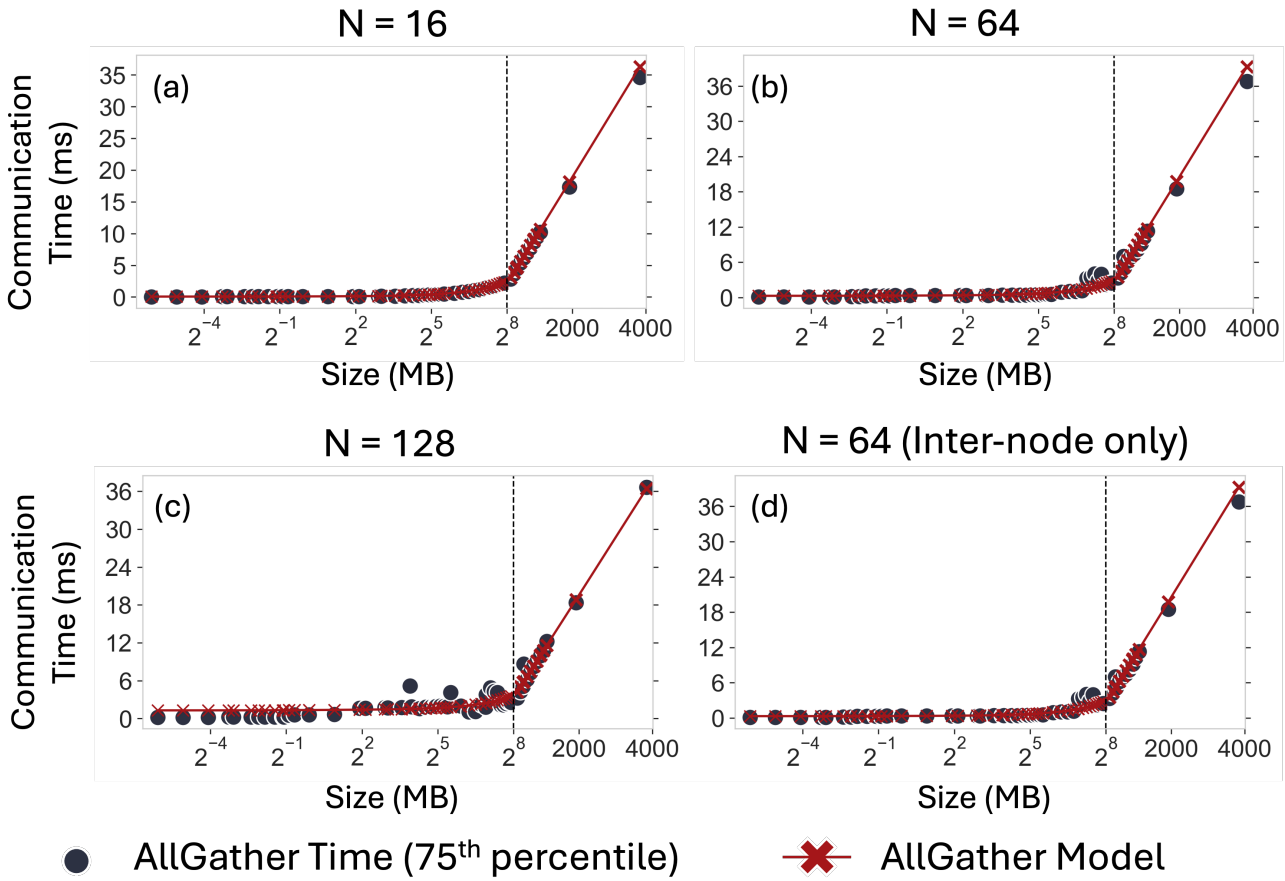


Figure 15. 75th percentile AllGather communication time from benchmarking data and predictions by the collective communication prediction model for each data size on cluster X. Data size refers to the size of the output tensor in the AllGather operation, since the communication overhead scales with the 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 $N = 64$.

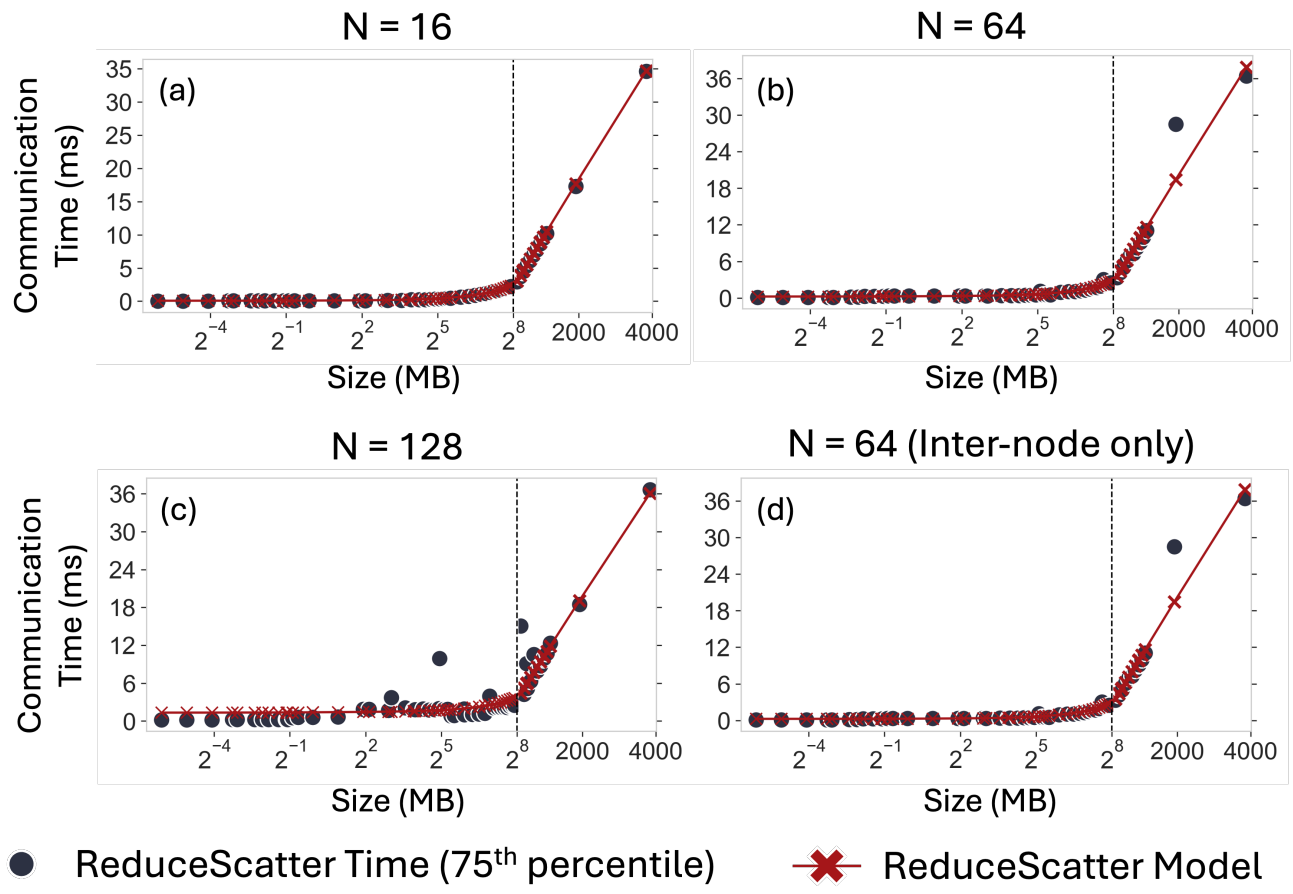


Figure 16. 75th percentile ReduceScatter communication time from benchmarking data and predictions by the collective communication prediction model for each data size on cluster X, with the same log-linear scale as Figure 14. (a)–(c) show data from world sizes of $N = 16, 64, 128$ for 1D parallelism whereas (d) shows data for 2D parallelism for $N = 64$.

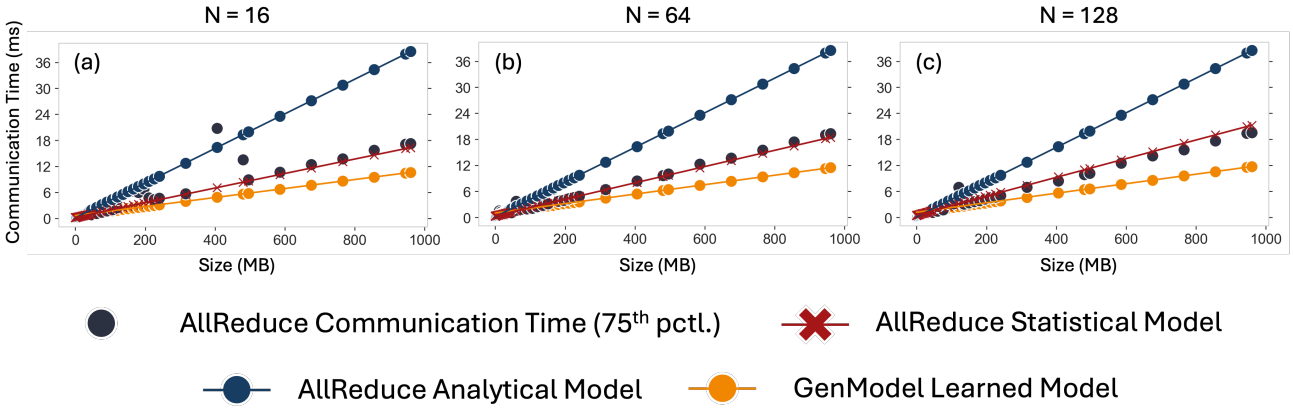


Figure 17. 75th percentile AllReduce collective times on cluster X with predictions by the statistical communication model and the GenModel baseline predictions for 2D parallelism on $N = 16, 64, 128$ GPUs.

Table 14. 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	-	-

H.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 15. Model configurations used in single device simulator experiments.

H.3.1. EXPERIMENTAL SETUP

We evaluate on 4 state-of-the-art models Google Gemma 2B (Team et al., 2024), Meta Llama 3.2 1B(Dubey 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 experiments(Dubey 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.

H.3.2. 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 16 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.

H.3.3. 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 17, shows our results for training with FSDP (1D Fully Sharded Data Parallel) training and Table 18 shows our results while applying FSDP+TP (2D Fully Sharded and Tensor Parallel) parallelism.

H.4. Memory Simulation

H.4.1. 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 1B(Dubey et al., 2024), Open AI CLIP (Radford et al., 2021), Google T5 (Raffel et al., 2020), 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 experiments(Dubey et al., 2024). We vary the batch-size, sequence-length (for large-language models) and image-size (for vision models). 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.

Table 16. 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 17. 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 18. 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).

H.4.2. 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 19 shows the effectiveness of our approach. We get close to 100% accuracy in almost all settings.

H.4.3. 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 20, shows our results for training with FSDP (1D Fully Sharded Data Parallel) training and Table 21 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.

I. Extended Related Work

I.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., 2025b). 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., 2025b). 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.

I.2. Memory Estimation

Current real-time memory tracking tools (Shi & DeVito, 2023; PyT, 2025b), 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

Table 19. 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 20. 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 21. 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

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 D.

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.

I.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., 2025b; 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., 2025b). 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.

I.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., 2025a; 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 10b, 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.

J. Future Directions

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

J.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.

J.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.