Moye: Ex-Situ Performance Simulation for Distributed Deep Learning Training

Abstract

Simulation offers unique values for both enumeration and extrapolation purposes, and is becoming increasingly important for managing the massive machine learning (ML) clusters and large-scale distributed training jobs. In this paper, we build Moye to tackle three key challenges in large-scale training simulation: (1) tracing the runtime training workloads at each device in an ex-situ fashion so we can use a single device to obtain the actual execution graphs of 1K-GPU training, (2) accurately estimating the collective communication without high overheads of discrete-event based network simulation, and (3) accounting for the interference-induced computation slowdown from overlapping communication and computation kernels on the same device. Moye delivers on average 8% error in training step—~3x lower than state-of-the-art simulators—for GPT-175B on a 96-GPU H800 cluster with 3D parallelism on Megatron-LM under 2 minutes.

1 Introduction

The unprecedented success of large language models (LLMs) has been driven in large part by the large-scale infrastructures that allow the model and training dataset to scale. Organizations have constructed massive clusters with tens of thousands of GPUs to train models with hundreds of billions or even trillions of parameters [8, 22, 30], in a continuous stride to improve model capabilities as promised by the scaling law [31].

Training LLMs on such massive scales consumes substantial time and financial resources. On the other hand distributed training is inherently complex and rapidly evolving, involving a myriad of algorithm/model innovations (linear attention [32]), parallelism strategies [35, 42, 45, 55], operation and kernel optimization (kernel fusion [18]), and hardware designs [2]. **The complexity is further compounded by emerging architectures like Mixture-of-Experts (MoE)** [17, 37]. As a result, accurate simulation of distributed training has started to garner attention as an essential tool to effectively manage training infrastructures and systems [13, 19, 23, 38, 51].

Broadly, simulation is useful for two main purposes: enumeration and extrapolation. In distributed training context, simulation can be used to enumerate combinations of parallelism strategies, optimization techniques, and hardware configurations that are available in the current cluster, to derive the optimal training plan for a given job and to determine efficient resource allocation schedules across jobs. Simulation is also useful (perhaps more so) to extrapolate beyond what is currently available, which is paramount for strategic decision making such as capacity planning [30, 45] that involve many what-if questions with significant impact. For example, what speed-up can be achieved by scaling the current cluster by a factor of 3, or by increasing the network bandwidth by 2x? This also greatly facilitates the development of new optimizations, which only need to be prototyped on a small scale for the simulator to extrapolate its potential benefits on a large scale quantitatively.

Prior work has made salient progress in training simulation [13, 19, 23, 26, 27, 36, 38, 51, 56]. Yet, they fall short in three key aspects that hinder their capabilities in supporting both enumeration and extrapolation practically.

First, modern large-scale training often employs **parallelism strategies like 3D parallelism, a combination of data parallelism (DP), tensor parallelism (TP), and pipeline parallelism (PP), as well as expert parallelism (EP) for MoE models**, to break down the large models along different dimensions into sub-models that fit into a device's memory and maximize cluster resource utilization. **Ensuring these complex configurations truly "fit" requires accurately predicting peak GPU memory in training. Practitioners often resort to overly conservative memory estimates to preempt unexpected and costly out-of-memory (OOM) errors. This strategy, however, can lead to significant under-utilization of GPU resources.** To accurately simulate **parallelism strategies and their memory implications**, one needs to obtain the training workloads, which encompass the device- or rank-specific execution graphs **detailing the computation, communication, and memory events, along with their dependencies.** Existing work [13, 23, 38, 39, 56] requires a full-scale cluster deployment of the job to trace the training workloads from each and every device at runtime. This is because each device builds its own execution graph independently in parallel during job initialization to most efficiently carry out training of its unique sub-model according to the parallelism strategy setting and various optimization techniques (e.g. kernel fusion). The *in-situ* tracing approach is clearly very costly and sometimes even impractical for extrapolation usecases.

Second, simulating collective communication (CC) (e.g. `all-reduce`) is critical because (1) it pieces all the devices and nodes together according to the various parallelisms, and (2) its performance can often be the bottleneck [20, 24, 30, 40]. However, most existing work [19, 23, 38, 51, 56] rely on a coarse-grained α-β model [47] without considering the actual communication patterns and optimizations in the implementation, resulting in significant simulation errors. On the

other hand, more recent work such as SimAI [13] explicitly simulates each peer-to-peer (P2P) send and receive primitive of a CC kernel using a packet-level event-based network simulator such as ns-3 [43]. This fine-grained approach provides high accuracy at the cost of prohibitive overheads even at a medium scale: simulating a 128-GPU job takes SimAI over 2 hours with an optimized ns-3 implementation (§2.3).

Third, we observe that overlapping communication and computation operations, an optimization widely used to improve efficiency [15, 16, 30, 53], incurs non-negligible slowdowns to the computation due to contention for shared resources (e.g., cache, DRAM bandwidth) [19, 25]. This, however, has largely been overlook in prior work thus far. Simulating interference-induced slowdown is particularly challenging since it depends on many idiosyncrasies ranging from scheduling logic of the ML framework and the underlying library to hardware-specific implementation optimizations. Much of these details are vendor-proprietary and inaccessible.

In this work, we propose Moye, a practical simulation platform for large-scale distributed training. Moye is built with three key design choices. (1) It adopts an *ex-situ* tracing approach to accurately capture training workloads using just a single device, avoiding the need for a full-scale cluster deployment. The key technical idea here is to transform the parallel initialization process at each rank into a sequential one, allowing a single device to act as each rank iteratively to trace the corresponding execution graph and **GPU memory footprints**. (2) It employs white-box modeling to simulate a CC kernel's performance with four parts: connection setup overhead, intra-server and inter-server transmission, and possible data reduction time. The model is supplemented with exhaustive profiling to obtain key parameters (e.g. chunk size which determines number of rounds of transmission for a message) optimized by CC libraries like NCCL for different hardware features and software configurations that affect the individual components above. This hybrid approach strikes a good balance between accuracy and efficiency. (3) It introduces an black-box ML-based slowdown predictor to model the slowdown caused by overlapping operations. The XGBoost-based predictor relies on categorical features such as transmission protocol and channel configuration in NCCL, to numerical kernel-level performance statistics such as streaming multiprocessors (SM) occupancy and DRAM utilization, to capture their complex interactions.

We implement Moye to support mainstream training frameworks: PyTorch, DeepSpeed [41], and Megatron-LM [45]. Our implementation with ~10k LoC also includes a suite of automation tools to facilitate workload tracing and runtime profiling. We evaluate Moye on H800 and A800 clusters across a variety of scenarios and models, achieving an average prediction accuracy of 91.4% in training step time, and is up to 3x better than the state-of-the-art. For 96-GPU training of GPT-175B, Moye achieves 92% accuracy in less than 2 minutes. We plan to open source Moye to the community.

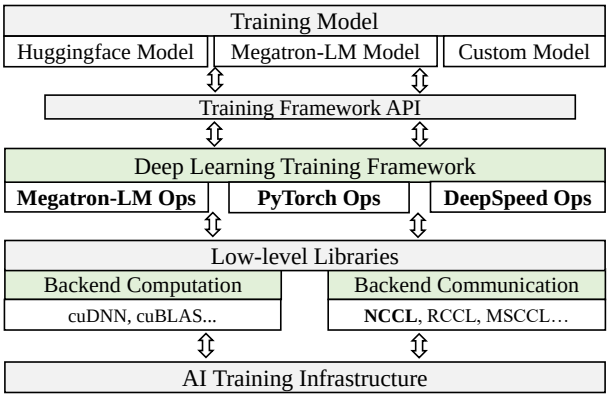


Figure 1. Overview of the technical stack and architecture for large-scale model training.

2 Background and Motivation

2.1 Large-scale Distributed DNN Training

Figure 1 illustrates the layered technical stack underlying large-scale training, from high-level models and frameworks to low-level GPU libraries and networked systems. Today's state-of-the-art models often exceed 100 billion parameters [8, 29], and recent GPU architectures (e.g., NVIDIA Hopper and Ampere) combined with high-speed interconnects (e.g., InfiniBand and RoCE) enable efficient training at the scale of thousands of GPUs. New parallelization techniques—such as 3D parallelism [45], **EP** [17, 37], and communication-computation overlap [15, 49] —further refine resource utilization, accelerating the convergence of these massive models.

ML frameworks. Modern large-scale training frameworks, such as Megatron-LM [45] and DeepSpeed [41], have become de facto standards in production settings at companies like Microsoft, NVIDIA, and Alibaba [13, 41, 45]. These frameworks implement advanced parallelism strategies—such as 3D parallelism (combining DP, TP, and PP) **and EP**—to efficiently scale model training across large GPU clusters. In brief, DP replicates the model and splits the data, TP divides model parameters among GPUs, PP stages model layers for pipelined execution, **while EP distributes the distinct experts of a MoE model across different devices.** Together, these methods accelerate training of hundred-billion-parameter models.

Communication library. Collective Communication Libraries (CCLs) facilitate efficient data transfer across GPUs in distributed systems [50]. Among these, NCCL is the most widely adopted in production environments [13, 41, 45, 50], providing core communication primitives. NCCL serves as the default communication backend for mainstream training frameworks and incorporates numerous optimizations.

Simulator	Training Workload			3D Parallelism	MoE/EP Support	Comm. Modeling	Overlap Slowdown
	Framework-Specific Ops	Ex-Situ Simulation	GPU Memory Footprints				
FlexFlow [27]	✗	✓	✗	✓	✗	$\alpha-\beta$ model	✗
Daydream [56]	✓	✗	✗	✗	✗	$\alpha-\beta$ model	✗
dPRO [23]	✓	✗	✗	✗	✗	$\alpha-\beta$ model	✗
DistSim [38]	✓	✗	✗	✓	✗	$\alpha-\beta$ model	✗
Astra-sim [51]	✗	✗	✗	–	✗	$\alpha-\beta$ model	✗
Lumos [36]	✓	✗	✗	✓	✗	Outsourced	✗
SimAI [13]	✓	–	✗	–	✗	Event-driven	✗
Calculon [26]	✗	✓	Theoretical	✓	✗	$\alpha-\beta$ model	✗
Proteus [19]	✗	✓	Theoretical	✓	✗	$\alpha-\beta$ model	–
Moye (ours)	✓	✓	**Ex-Situ Profiling**	✓	✓	White-box	✓

Table 1. Comparison of key features across simulators. ✓ indicates full support, ✗ indicates no support, and – denotes partial or conditional support. Simulators are evaluated on training workload generation (framework-specific operations, ex-situ simulation, dynamic GPU memory footprints), 3D parallelism, MoE architectures, communication modeling, and handling of overlap-induced slowdowns.

2.2 Design Goals

The overarching goal of this work is to build a practical simulator for large-scale distributed training. Specifically, we aim to achieve the following goals:

- **High accuracy.** The basic goal is to accurately predict the end-to-end training step time (§8).
- **Ex-situ simulation.** If the simulator's input is collected by deploying the job to the target cluster at full scale, i.e. *in-situ* simulation, its utility is fundamentally limited by the available resources. To be able to explore scales and scenarios beyond what is currently available, we desire *ex-situ simulation* design that can simulate a 1k-GPU cluster using a single machine for example (§9).
- **Generality and usability.** The simulator should support **arbitrary ML workloads expressible in mainstream training frameworks, covering diverse model architectures and parallelism strategies.** It should also be easy to use, automating the process as much as possible to minimize manual effort (§8.2).
- **High efficiency.** The simulator should be fast with minimal computational overhead, especially for large-scale settings, **enabling efficient exploration of the vast design space (§8.5 and §9).**

2.3 Challenges

Training simulation has started to attract attention recently [13, 19, 36, 51]. Inspired by prior work, we also separately consider computation and communication, and computation can be relatively easily simulated by offline profiling on a single device. Yet, to achieve the four design goals, we identify three key challenges that prior work has not fully addressed. Table 1 summarizes the comparison between Moye and existing simulators.

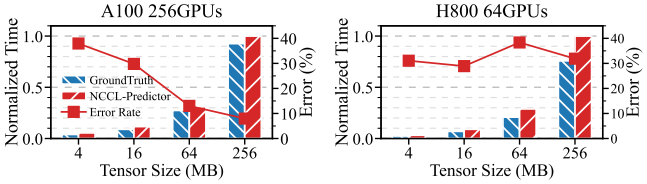


Figure 2. Comparison of predicted running time (by NCCL-Predictor) and ground truth for the `all-reduce` operation.

Challenge 1: *How to obtain the actual training workloads without a full-scale deployment?*

Accurate ML training simulation relies on the so-called *training workload* as the foundation. The *training workload* refers to the execution graph that encodes the dependencies or execution sequence of computation and communication operators, plus other necessary information like tensor shapes and data types. It serves as the blueprint for replaying the training process (on a given device). In large-scale training that employs complex strategies like 3D parallelism and EP, each GPU device (rank) independently initializes its assigned submodel based on TP, PP, and EP, resulting in unique workloads per rank. These workloads are established only when the training framework begins its distributed execution.

Existing simulators largely lack support for prevalent MoE architectures (Table 1). Furthermore, their reliance on overly conservative analytical models for GPU memory prediction results in gross overestimations—causing users to discard optimal configurations under false OOM assumptions (see Sec).

Thus for accuracy, one naturally adopts a runtime tracing-based approach to capture the training workloads [23, 38, 39, 56]: the per-device execution graph is extracted after each device builds its execution graph in parallel. Yet this assumes a full-scale deployment of the job on the cluster, which severely limits the practical utility of the simulator in

large-scale settings and is exactly one of the main motivations for building a simulator. Some systems like Proteus [19], ASTRA-sim [51] and Calculon [26] choose to manually distribute and maintain the operators and generate the training workloads in order to avoid this limitation. However, this hand-crafted approach is inherently imprecise since it fails to accurately represent the nuanced runtime characteristics of *framework-specific operations*—custom or highly optimized operations tightly integrated with a particular framework's execution model (see §8.2). These operations often rely on environment-specific optimizations that emerge only under full-scale deployment conditions. Without capturing these contextual factors, manually approximating operation behavior becomes guesswork, leading to substantial inaccuracies. For instance, we observed that Astra-sim overestimated computation operation by over 100% on A800 GPUs when running Megatron-LM workloads. **These shortcomings extend beyond computation. Existing simulators also largely lack support for prevalent MoE architectures (Table 1). Furthermore, their reliance on overly conservative analytical models for GPU memory prediction results in gross overestimations—causing users to discard optimal configurations under false OOM assumptions (see §9).** How to faithfully capture the training workloads without a full-scale deployment becomes our first challenge.

Challenge 2: *How to accurately simulate collective communication without high overheads?*

Communication operations are critical in distributed training, yet hard to simulate as they involve complex interactions among devices and machines, and are affected by a range of network factors such as topology, latency, congestion control, etc. Existing approaches fall into two categories. Most simulators like ASTRA-sim [51], FlexFlow [27], Calculon [26], Proteus [19], and NCCL-Predictor [10] adopt a $\alpha - \beta$ model [47], which essentially calculates the running time of a communication operation as $tensor\_size/bandwidth + latency$. Figure 2 shows the most advanced NCCL-Predictor greatly overestimates the effective bus bandwidth of `all-reduce` operations. Note NCCL-Predictor already uses additional parameters to account for the impact of hardware, communication protocols (TCP vs RDMA), and algorithms (ring vs tree for `all-reduce`), but the over-simplified $\alpha - \beta$ model is inherently limited in capturing complex interactions across all these factors. Moreover, NCCL optimizes distributed communication by integrating hardware offloading and acceleration techniques, such as GPUDirect RDMA [5] and GDRCopy [6], which lead to variations in internal kernel behavior.

The second, more precise approach involves white-box modeling specifically targeting collective communication [13, 51]. Since NCCL (and other CCLs) implements a communication operation as an orchestrated series of P2P send and receive operations, we can simulate each send and receive to compose the end-to-end result. Similar to the training workloads, this naturally reflects the complex optimizations used by CCLs according to topology, NCCL P2P protocol (LL, LL128 or Simple), network interconnect technologies (IB or RoCE), algorithm (ring or tree for `all-reduce`), etc. To faithfully simulate the send/receive time, prior work [13, 51] uses packet-level discrete-event network simulators like ns-3 [43], which takes the actual topology and profiled link bandwidth as input. Yet packet-level simulation is computationally expensive as it needs to simulate the entire protocol stack at each node in the network: We run the most recent SimAI [13] with an optimized multi-thread ns-3, and observe that simulating a single iteration on a 128-GPU cluster takes over 2 hours (7655s) on a 32-core server. Therefore, the challenge is how to scale the white-box communication simulation efficiently without sacrificing accuracy.

Model	Overlapped op	Slowdown	Kernel Type	Slowdown
VGG19	57.53%	37.67%	GEMM	26.95%
GPT2	38.77%	7.05%	Sum	50.83%
Llama3-8B	26.79%	7.15%	GroupedGEMM	11.78%
Mixtral-8x7B	23.02%	10.55%	Grad	61.37%

Table 2. Statistics about (1) the proportion of overlapped kernels in different models and the slowdown factors in a training step, and (2) slowdown factors of some common computation operations. **Evaluated with PyTorch (VGG19), DeepSpeed (GPT2), Megatron-LM (Llama3-8B), and Flux [11] (Mixtral-8×7B).**

Challenge 3: *How to account for the interference between overlapping computation and communication?* [1]

Overlapping computation and communication is widely used in practice to improve training efficiency and hardware utilization [16, 30, 53]. We find that overlapping introduces non-negligible slowdowns especially to computation operations, even though they are assigned to separate streams on the GPU (contention for shared memory may be a possible cause). Here, we define slowdown as the percentage increase in running time when overlapping communication kernels compared to no overlap. As shown in Table 2, more than 23% computation operations overlap with communication (measured across multiple SOTA training frameworks), leading to a slowdown of up to 1.37x in a training step. We also examine the slowdown to individual kernels. Table 2 (second column group) shows when overlapped with `all-reduce` with 25MB messages, running times of common kernels increase by an average of 37.73%. **The CDF of slowdown factors for all computation kernels of GPT-2 is presented in Figure 3, where slowdown can reach 8 with an average of 1.70.** Some recent work [25] also reported similar slowdowns in production training clusters. Unfortunately, this phenomenon has been overlooked by most existing simulation work despite its salient impact. The only work we know that considers this is Proteus [19]: it simply uses a heuristic factor that varies only with GPU architecture and ML model to

[1] **Experiments run on an NVIDIA A800 cluster mentioned in §8.1.**

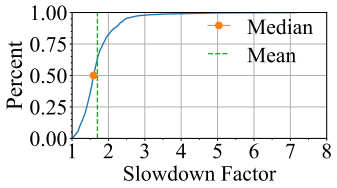


Figure 3. CDF of slowdown factor of various kernels in GPT2.

GPU	Kernel	CUDA	Slowdown
3090	GEMM	11.8	26.4%
	GEMM	12.1	39.3%
	LayerNorm	11.8	18.0%
	LayerNorm	12.1	93.8%
A800	GEMM	11.8	13.4%
	GEMM	12.1	17.8%
	LayerNorm	11.8	9.1%
	LayerNorm	12.1	20.7%

Table 3. Performance comparison of CUDA kernels on different GPU architectures, showing execution slowdown across various CUDA versions.

obtain the slowdown, which is too coarse to be accurate and not generalizable to unseen models.

Simulating the interference-induced slowdown is a daunting task. We ideally would need to model the kernel-level behavior in order to precisely know how different kernels overlap. For example, in the widely-used gradient overlap optimization [53], the simulator must correctly replay the backpropagation computation kernels and schedule communication kernels at precise times to achieve accurate overlap. Further, for a given computation kernel, its slowdown factor is influenced by various factors of hardware and software setup. Table 3 shows the slowdown factors of two kernels on different GPUs and CUDA versions vary a lot. This is because the internal implementation, which varies based on the factors above, ultimately determines how resources are scheduled and shared among concurrent kernels. These proprietary details are extremely inaccessible for us, not to mention the complexity of modeling them. Thus, how to practically simulate the effect of overlapping computation and communication remains a formidable challenge.

3 Design Overview

Building upon the aforementioned observations and motivations, we introduce Moye, a simulator tailored for large-scale distributed training.

Key design choices. We highlight the key choices in building Moye to address the three challenges in §2.3.

- To obtain training workloads without a full-scale deployment, Moye employs a novel ex-situ tracing approach. Generally, ML frameworks support various parallelism in distributed training in the following way: (1) they initialize each rank (typically one rank per device) with its own sub-model based on the parallelism setting, and then (2) each rank builds its own execution graph to start actual training. Moye turns this parallel process into a sequential one, using only one device to act as each rank iteratively to trace the corresponding execution graph and profile the running time of each operation at the same time, achieving faithful workload tracing without requiring the full cluster.
- To achieve efficient communication simulation without high overheads, Moye employs a white-box modeling approach with empirically profiled parameters that strikes a good balance between accuracy and efficiency. Our model is derived closely after NCCL's underlying chunk-based implementation of collective communication capturing critical factors such as per-chunk inter-server transmission time. These factors are profiled offline exhaustively to capture the complex dynamic optimization by NCCL according to hardware features and software configurations.
- To account for the interference between overlapping computation and communication, Moye adopts a black-box method using features related to both computation and communication kernels, such as bucket size, SM occupancy, and cache hit rate.

System architecture. Figure 4 shows Moye's architecture. Its input includes three categories: user-defined settings specifying the training framework, parallelism strategies (DP/TP/PP/**EP** groups); model details defining the model structure and hyperparameters; and hardware configurations detailing device specifics (GPU type), cluster size, network topology, NCCL parameters (e.g., NCCL_TOPO_FILE, NCCL_BUFFSIZE), etc. Given these inputs, the *workload tracer* extracts training workloads for each rank, including the per-rank execution graph and operation running times (§4.1). The *CC estimator* module (§5) leverages the white-box models with parameters corresponding to the given settings to estimate each communication kernel's performance. Then the *timeline composer* module (§4.2) re-constructs the global timelines of the end-to-end training process by assembling all per-rank execution graphs with estimated communication operation performance and inter-rank dependencies. The *validator* module (§6) applies a machine learning model to adjust the running times of overlapping operations, accounting for the interference-induced slowdown, and outputs the final training step time result. Additionally, Moye maintains a *database* for profiling data, reducing redundant measurements across simulations.

4 Workload Tracing

4.1 Per-Rank Workload Tracing

To launch a training job, the ML framework registers the global parallelism parameters (e.g. DP/TP/PP/**EP** groups) given by user into the so-called model parallelism unit (MPU), which maintains all states related to parallelisms for a given rank. Then according to the MPU, each rank concurrently initializes its own part of the model that it needs to train for (Figure 5 ①), and starts training with an execution graph that is optimized for its submodel based on the hardware/-software environment and user configurations.

Moye extracts the per-rank execution graph in an ex-situ fashion by transforming the above parallel process into a

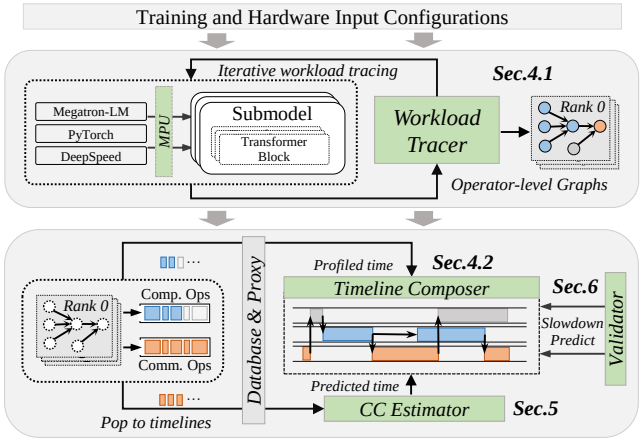


Figure 4. Moye architecture overview. The green components represent the core modules of Moye.

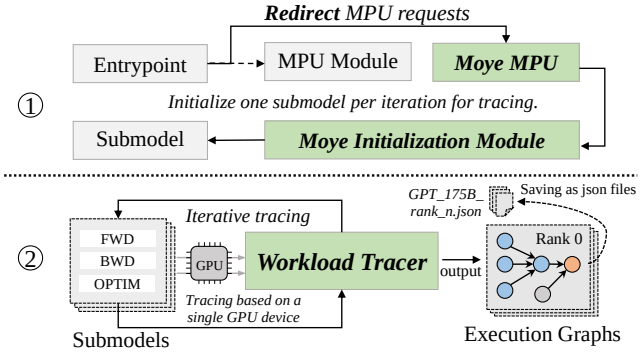


Figure 5. Moye's MPU module and workload tracer.

sequential one (Figure 5 ②). Specifically, Moye hijacks all the calls to the original MPU and re-directs them to its own MPU, and registers the global parallelism parameters as usual. Moye's MPU only manipulates the input arguments to the MPU when necessary; it does not change the original implementation to ensure correctness. Then with a single device, in each iteration i, Moye uses this unique rank ID i to initialize the corresponding submodel and execute one complete training step, so all information regarding the computation operations, including their dependencies (including those w.r.t other ranks) and running times, can be profiled. **Moye faithfully replays the architecture's behavior on CUDA memory, including memory regions allocated for NCCL communication and optimizations such as early deallocation. This enables us to precisely and dynamically trace the memory footprint of each rank during training (see §8.2).**

The caveat here is collective communication: one device obviously cannot execute a communication operation. To resolve this, notice that when the communication operation is launched it also needs to call into the MPU which determines the device's position in the global communication groups for proper execution. This is re-directed to Moye's MPU again allowing it to trace the communication operation's truthful information in the simulated setting (group association, message sizes, etc.); meanwhile, instead of returning results using the simulated (distributed) setting, Moye's MPU returns results corresponding to single device setting, so the communication operation can proceed and training is not blocked. This way Moye traces the actual execution graphs for each rank and running times of computation operations without requiring the full-scale cluster. Implementation details of tracing vary across frameworks which we will discuss in §7.

Note that after per-rank workload tracing, all collective communication operations on the per-node execution graph are placeholders without any running time. The job of Moye's CC estimator and validator is precisely to estimate and refine the communication times, respectively, in the next phase.

4.2 Timeline Composing

Before presenting the communication estimation and validation, we discuss how Moye obtains the end-to-end training step time by composing the global timeline with the per-rank workloads obtained above.

Assuming all communication running times are known, Moye's timeline composer can recover the training process by replaying the per-rank execution graphs. Timeline composer is an event-driven simulator that walks through the per-rank execution graphs and puts all operations (events) onto the global timelines accordingly, including the computation timeline, communication timeline, and memory copy (memcpy) timeline. As shown in Figure 6, a critical piece of information missing from the per-rank graphs is dependencies across ranks; for instance the downstream stages in rank i requires activations from the upstream stages in rank $i-1$ before starting the forward computation in PP. We provide predefined dependencies and matching rules for both inter-stage and intra-stage events for common parallelism strategies, such as 1F1B. To enhance flexibility, this module is exposed as an API, allowing users to input new dependencies and matching rules for custom strategies.

5 Communication Estimation

5.1 Background on NCCL

NCCL provides collective communication (CC) functions as single CUDA kernels [10] to support various training parallelisms. Our models therefore predict performance at the kernel level, a close approximation that ignores negligible launch overheads (the same applies to the slowdown prediction in §6). A key NCCL feature is pipelining messages as smaller equal-sized chunks, which is the basis for our formulation in §5.3. For its all-reduce collective, NCCL supports both ring-based [10] and tree-based [4] algorithms. Other CC kernels use ring-based algorithm only.

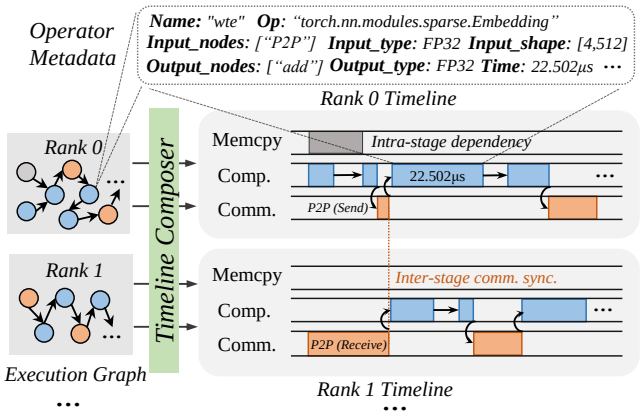


Figure 6. Illustration of how Moye schedules operations in the execution graph to timelines across ranks.

5.2 White-Box Modeling: Synchronization

Moye models a NCCL kernel as two sequential stages: synchronization and execution. Upon invocation, the kernel first checks whether it can start execution based on the readiness of ranks in the current communication group. If not all ranks are ready, the kernel waits for synchronization before it can proceed. Thus, the total running time of a kernel is the sum of synchronization time and execution time. Here we investigate the synchronization time first.

Moye models synchronization time after NCCL's implementation logic. For P2P `send` and `receive`, it waits until its target communicator (i.e., each rank's communicator instance) is ready to proceed. In contrast, for CC kernels, the communicator associated with each rank in the current group must wait until all communicators have been successfully launched. The last communicator to initiate determines the actual start time of this kernel. **Moye's synchronization model captures the impact of stragglers by simulating their effect on communication delays.**

5.3 White-Box Modeling: Execution

Execution time represents the majority of kernel running time, and is the focus of our modeling. We build white-box models to closely model the kernel execution process in NCCL with different algorithms, and profile all the key parameters of the models exhaustively in various hardware/-software settings to achieve accurate prediction without high overheads of event-based simulation.

Basic model. Based on NCCL's GPU-side implementation, the execution time of a CC kernel can in general be broken down into four non-overlapping components, using `all-reduce` as the example:

$$\begin{aligned} T_{comm\_kernel} =& T_{conn\_setup} + T_{intra\_trans} \\ &+ T_{data\_reduction} + T_{inter\_trans}, \end{aligned} \tag{1}$$

where T_{conn_setup} is the connection setup time; T_{intra_trans} and T_{inter_trans} are times to transmit the tensor chunks in the specific algorithm, $T_{data\_reduction}$ is the computation time of the reduce operation over the received tensor chunks with the local tensor. Not all components are present in other CC kernels.

Based on Equation (1), we can formulate the following for all CC kernels with different algorithms:

$$T^{Ring}_{all-gather} = \alpha + \gamma \cdot \eta(N-1), \tag{2}$$

$$\begin{aligned} T^{Ring}_{reduce-scatter} =& \alpha + \eta\big[(N-1) + \gamma \times (N-1)\big] \\ &+ \delta \times tensor\_size, \end{aligned} \tag{3}$$

$$\begin{aligned} T^{Ring}_{all-reduce} =& \alpha + \eta\big[(N-1) + \gamma \times 2(N-1)\big] \\ &+ \delta \times tensor\_size, \end{aligned} \tag{4}$$

$$\begin{aligned} T^{Tree}_{all-reduce} =& \alpha + \gamma(\eta-1) + 2\beta(K-1) \\ &+ 2\gamma \log_2(M) + \delta \times tensor\_size. \end{aligned} \tag{5}$$

Here N is the total number of devices, M number of nodes (servers), and $K = N/M$ number of devices per node. We denote the connection setup time as α, intra-server and inter-server transmission times as β and γ. The reduce time depends on $tensor\_size$, and the reduce throughput δ. Finally, η is the number of rounds or number of chunks for this tensor, which corresponds to NCCL's chunk-based implementation.

Offline exhaustive profiling. The modeling above is explicitly related to nine parameters and implicitly related to an additional parameter, $chunk\_size$. Among them, N, M, K and $tensor\_size$ are user configurations while $\alpha, \beta, \gamma, \delta, \eta$ are what Moye attempts to obtain. However, modeling them explicitly is challenging. First, they obviously depend on $chunk\_size$ and $tensor\_size$. The chunk size is not a predefined constant but dynamically tuned by NCCL [10]. Further, they also vary with hardware configuration and software implementation (e.g. IB vs NVLink, NCCL version). These idiosyncrasies also vary across different CCLs, making it even more difficult to have a general model.

We adopt offline profiling instead to obtain the values of these parameters, since they are fixed when all the hardware/software configurations are settled. First, we modify NCCL to be able to obtain the chunk size during the initialization phase. We enumerate all possible combinations of N, M, $tensor\_size$, and hardware type (IB, TCP, NVLink) to record the corresponding $chunk\_size$ and number of rounds η computed by NCCL. Then for each $chunk\_size$, we profile the corresponding connection setup time, intra-server and inter-server transmission time, and reduce throughput $(\alpha, \beta, \gamma, \delta)$ under the corresponding setting. This does require access to a cluster as the network bandwidth may vary depending on the scale (2-node vs 4-node vs 8-node), but for common Clos-based cluster topologies they remain stable after reaching a small scale due to the symmetry and limited number of tiers.

Taken together, our approach of combining white-box modeling with offline profiled parameters provides a good

Type	Feature	Specification
Comm. details	Protocol	Communication protocol used within NCCL
	Algorithm	Communication topology algorithm applied in NCCL
	Collective	Name of the collective function in the kernel
	Bucket size	Size of the tensor bucket used in collective communication
	Channel number	Number of communication channels in NCCL
Comp. Kernel Metric	Running time	Running time profiled on single-device training
	Compute throughput	Average SM throughput
	Memory throughput	Percentage of cycles where DRAM was active
	DRAM throughput	Peak DRAM throughput
	Achieved occupancy	Average percentage of active warps on each SM
	L1 hit rate	Hit rate for L1 cache
	L2 hit rate	Hit rate for L2 cache

Table 4. Features in Moye's prediction model.

first-order estimation of CC kernel performance with very low overheads. Our approach can be enhanced in several ways, which we discuss in §10.

6 Overlapping Slowdown Prediction

Accurately capturing and modeling the resource contention between overlapping kernels presents substantial challenges in predicting the slowdown factor as discussed in §2.3. To address this, Moye introduces an ML-based slowdown prediction model with a number of hardware- and software-specific features, which we present here.

A GPU devices executes concurrent computation and communication kernels with spatial multitasking, where each kernel is assigned to separate streams with a subset of the SMs. At the same time they also contend for shared resources such as cache and DRAM bandwidth. These factors results in performance interference, and Moye needs to capture them in predicting the slowdown.

Feature extraction. Table 4 summarizes the key features used by our model. We categorize features into two groups: communication-related details and computation kernel metrics. For communication, we consider features such as protocol and algorithm employed by NCCL, CC kernel type, bucket size, and channel configuration. These parameters capture the complexity and overheads of communication: bucket size, for instance, determines when a communication kernel will be launched after accumulating one bucket of data. On the computation side, we profile kernels at a fine-grained level on a single device to measure their baseline performance without overlapping. We collect running times, compute throughput, memory utilization, and cache statistics using NVIDIA Nsight Systems and Nsight Compute. For instance, achieved occupancy and SM activity levels indicate compute intensity, while DRAM utilization and memory throughput—along with L1/L2 hit rates—shed light on memory access latency and potential bottlenecks. Additionally, we classify kernels by their functional types (e.g., matrix multiplication, layer normalization) to further refine our predictions.

XGBoost-based model. We leverage XGBoost [3], a tree-based gradient boosting framework, to predict the slowdown factors caused by overlapping kernels with the above features. XGBoost is well-suited to this task for two key reasons. First, our dataset contains heterogeneous features that are both numerical (e.g., SM throughput, DRAM utilization) and categorical (e.g., GPU type, kernel type), a scenario where tree-based models excel. Second, our target slowdown factors, influenced by the interplay of diverse system conditions, remain bounded within a range well-captured by the training set. As we expand our dataset—incorporating additional profiles from various models, GPU architectures, and cluster scales—the model's generalization ability improves, enabling accurate slowdown predictions across increasingly complex and heterogeneous environments.

7 Implementation

We implement Moye as illustrated in Figure 4. Moye is developed based on PyTorch 2.1, DeepSpeed 0.13.1, and Megatron-LM (commit ID: 53a350ed), and NCCL 2.22.3, comprising ~10K lines of code (LoC). ~2K LoC are dedicated to Megatron-LM, 3K LoC to DeepSpeed and PyTorch, and the remaining 5K LoC for common core modules that underpin the simulation engine, training workload tracing, overlapping induced slowdown dataset collection, and CC parameter exhaustive profiling. **We provide Moye's example API calls for tracing and end-to-end simulation in Appendix.**

Workload tracer. We implement two tracing modules to support execution graph extraction for Huggingface models used by PyTorch and DeepSpeed and Megatron-LM models. For the former, the tracer leverages `torch.fx` to record operations. Since `torch.fx` only captures computation operation during the forward pass, we enhance it to capture operations during backward passes by using tensor gradient functions and registering hooks, as well as optimizer steps and communication operations (handled as placeholders). For Megatron-LM models, a custom tracing module is developed with a series of Python decorators and context managers. Moye directs the handling of custom operations in Megatron-LM to a general template, ensuring that the related function calls are recorded. Finally, the generated execution graph is output as a JSON file and stored in a database for future use. Notably, users can activate Moye's workload tracer by adding a few extra lines of in their existing training scripts.

CC estimator. To profile the key parameters for models in §5.2, we utilize the NCCL profiling tool NPKit [7]. Given that the network topology can impact bandwidth, we empirically profile these parameters for settings up to 4 nodes (32 GPUs) in our clusters, beyond which the effective per-node network bandwidth does not change (and thus these parameters).

Validator/Slowdown predictor. We experimentally find the best hyperparameters for our XGBoost model **(see Appendix)**. We implement an automated pipeline to construct the training dataset. NVIDIA Nsight Compute [1] is used to

profile GPU resource metrics such as SM throughput, memory bandwidth, and cache hit rates. Nsight Systems [12] capture kernel execution traces under overlapping and non-overlapping conditions, exporting the data as SQLite databases.

8 Evaluation

8.1 Experiment Setup

Models. **We evaluate Moye's ability to simulate both dense and MoE architectures. For dense models, we include representative CNNs and transformer workloads: the VGG19 [46] image classification network and GPT-series language models [45] ranging from 13B to 485B parameters. For MoE models, we use Mixtral series [28]. Model configurations are in the Appendix. Since most simulators support only FP32, we evaluate dense models in FP32. For MoE models, we use BF16 precision, with experts executed via CUTLASS GroupedGEMMv1.0 [9].** For ground-truth measurements, all models are trained on real GPU clusters using mainstream frameworks (PyTorch, DeepSpeed, Megatron-LM) with CUDA 12.1, matching the implementation setup in §7.

Clusters. Our evaluation utilizes a 96-GPU H800 cluster (12 servers with 8x 80GB GPUs each) and an 8-GPU A800 cluster. Both configurations use 400 GB/s NVLink for intra-server communication, while the H800 cluster is equipped with 400 Gb/s NDR InfiniBand for inter-server links. Additionally, an RTX 3090 cluster is used for kernel slowdown analysis (§8.4). Appendix provides our additional observations and evaluations on 256-GPU A100 clusters.

Baselines. We compare Moye with three fully open-source simulators:

- **Proteus** [19]: A SOTA simulator for integrated simulation of parallel strategies and runtime behaviors.
- **FlexFlow** [27]: An internal simulator for throughput estimation across different parallelization strategies.
- **Calculon [26]: An analytical performance simulator for hardware-software codesign exploration.**

Other simulators such as ASTRA-sim [51] and dPRO [23] are open source as well, but they suffer from critical limitations: ASTRA-sim natively supports only matrix multiplication kernels, and dPRO does not support ex-situ simulation.

8.2 Computation and GPU Memory Simulation

We first evaluate Moye 's accuracy in simulating computation workloads on both H800 and A800 clusters, using VGG19 (with DeepSpeed), GPT-13B and Mixtral-8×1.75B (both with Megatron-LM) under various PP/EP configurations. Each configuration assigns a distinct submodel to an individual GPU (rank).

Simulation accuracy. Moye reconstructs the computation workload by tracing all computational operations and reproducing their execution sequence based on the execution

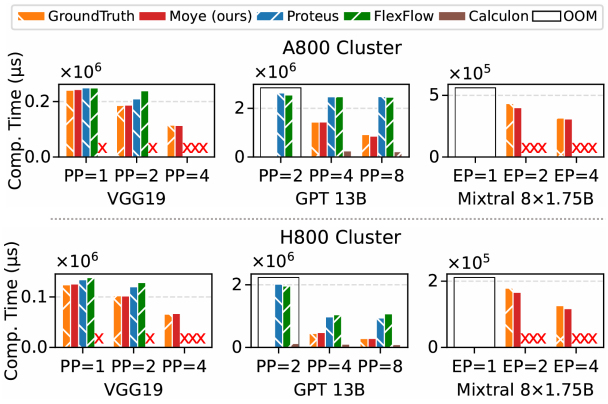


Figure 7. **Computation times of various workloads on A800 and H800 clusters under different parallelization settings. Red crosses denote simulation not supported by the baselines, while black squares mark configurations that encountered OOM errors during practical execution.**

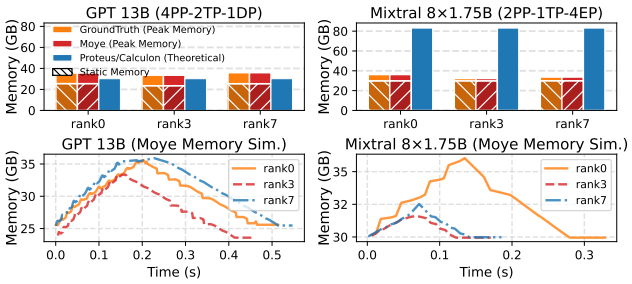


Figure 8. **Validation of Moye's GPU memory simulation for GPT-13B and Mixtral-8×1.75B on an 8-GPU A800 cluster (8 ranks), showing three representative ranks. We compare Moye with Proteus and Calculon (theoretical approaches) and demonstrate Moye's unique ability to capture dynamic per-rank memory usage within a training step (excluding communication time).**

graph. Per-GPU operation running times are derived through Moye 's workload tracer without requiring a full-scale deployment. As shown in Figure 7, Moye achieves a maximum overall error of 8.31%. In contrast, while Proteus and FlexFlow maintain relatively low error rates for VGG19 under DeepSpeed (8.6% and 17.81%, respectively), their errors escalate dramatically for GPT-13B under Megatron-LM, reaching 91.53% and 109.14%, respectively. The root cause of this discrepancy is that Proteus and FlexFlow rely heavily on standard PyTorch operations, failing to incorporate the custom fused operations that frameworks like Megatron-LM employ for performance optimization. Such specialized operations cannot be accurately modeled without extracting their runtime characteristics directly from the target training framework (as we disccusion in §2). **Calculon overestimates computation time in an overly idealized manner (error rate >400%), because its modeling relies on hardware's theoretical FLOPS values, which severely deviate from**

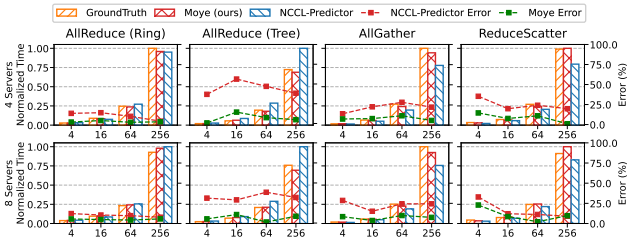


Figure 9. Running time and prediction error for inter-server CC kernels on H800 GPUs across varying tensor sizes and cluster scales. Each subplot is normalized relative to the operation with the longest running time.

real hardware execution due to memory bandwidth constraints, cache hierarchy inefficiencies, instruction pipeline stalls, and tensor core utilization variances. By precisely tracing, Moye captures the actual operation behaviors and achieves substantially higher accuracy.

Generality. **Moye is the only high-fidelity simulator that supports MoE architecture workloads, as shown in Figure 7 (error <6%). It supports simulations across a wide range of training design spaces. In contrast, Proteus and FlexFlow struggle with certain configurations. For example, they cannot simulate VGG19 at PP=4 for dense models due to the requirement for manual, pre-hardcoded workload construction. Calculon, on the other hand, cannot simulate either CNNs or MoE models. In §9, we present a more detailed case study to demonstrate Moye's scaling ability on 8,192 GPUs through ex-situ simulation.**

Memory simulation. **Moye supports high-fidelity GPU memory simulation through an ex-situ profiling approach. Figure 8 (upper) shows the peak memory usage (including both static and dynamic components), where Moye achieves nearly 100% accuracy on GPT-13B and Mixtral-8×1.75B across different GPU ranks. In contrast, theoretical methods can result in an 18.0% error for GPT-13B, and for MoE models such as Mixtral-8×1.75B, the error can increase to 2.58× (a gap of 51 GB). Moye can also reflect dynamic GPU allocation behavior during training, as shown in Figure 8 (lower)—a capability absent in existing simulators. We believe this ability can help engineers identify potential optimization opportunities.**

8.3 Communication Simulation Performance

We now evaluate the effectiveness of Moye 's communication simulation, comparing it against NCCL-Predictor [10], the underlying communication model used by both Proteus and FlexFlow. NCCL-Predictor leverages parameters tuned from production experience and employs an $\alpha - \beta$ model (§2.3).

Simulation accuracy. Figures 9 illustrate the prediction accuracy for collective communication under varying message sizes (4MB to 256MB) and cluster scales. Across most cases, Moye consistently outperforms NCCL-Predictor. Moye's average prediction errors are 12.59% and 13.75% for 4- and 8-server H800 clusters, respectively, outperforming NCCL-Predictor by a margin of 34.4%, 29.56%, and 19.39%. The smallest prediction gap emerges in ring-based `all-reduce`, where Moye is only 4.2% more accurate than NCCL-Predictor. By contrast, for other communication patterns, especially tree-based `all-reduce`, Moye demonstrates a more pronounced accuracy advantage, reducing average prediction errors by up to 33.5%.

These differences can be attributed to NCCL's specialized optimizations for the widely used ring-based `all-reduce`, which align more closely with the assumptions of the $\alpha - \beta$ model. However, NCCL-Predictor's simplifications become problematic for less commonly optimized patterns and smaller message sizes, leading to significant discrepancies. In particular, NCCL-Predictor exhibits an average error rate of 27.6% for 4MB and 16MB messages. By contrast, Moye 's profiling-based white-box approach more accurately captures practical bandwidth utilization and overheads, ensuring consistent accuracy across both small and large message sizes. Notably, Moye 's white-box communication prediction model significantly accelerates simulation speed, as discussed in §8.5.

Tensor size (MB)	4	8	16	32	64	128	256
Chunk size (10^3 bytes)	18	36	72	72	72	144	288
Intra-server Round trip (μs)	57.43	64.36	73.62	74.61	77.46	75.55	99.23
Inter-server Round trip (μs)	54.3	72.32	106.73	112.9	111.25	174.2	278.98
Reduce operation (μs)	50.14	65.86	102.46	219.89	463.43	841.71	1657.72

Table 5. The breakdown of Moye's `all-reduce` running time, profiled in the H800 cluster using 32 GPUs.

Breakdown analysis. Table 5 provides a detailed breakdown of the `all-reduce` kernel runtime in a 32-GPU H800 cluster, including intra-server and inter-server round-trip times and reduce operation computation times for various tensor sizes. Intra-server round-trip times scale with tensor size, increasing from 57.43 µs for 4MB tensors to 99.23 µs for 256MB tensors, reflecting overheads such as scheduling more thread blocks for larger chunks. Similarly, inter-server round-trip times grow with tensor size, reaching 278.98 µs at 256MB. The computation time for the reduce operation also scales nearly linearly with tensor size, increasing from 50.14 µs at 4MB to 1657.72 µs at 256MB. These trends highlight how Moye 's detailed modeling captures realistic overheads across varying message sizes and communication settings.

8.4 Slowdown Prediction

Most simulators overlook the performance slowdown caused by kernel overlap. We compare Moye with Proteus, the only baseline known to model this effect, which does so using a simple heuristic based on GPU architecture and model type.

Solution	GPU Type	5% Error	10% Error	15% Error	RMSE
Moye (ours)	3090	61.48%	73.62%	76.52%	1.0935
Proteus		8.52%	13.62%	19.28%	1.4953
Moye (ours)	A800	61.03%	69.82%	75.01%	0.7147
Proteus		59.04%	65.45%	70.51%	1.5170
Moye (ours)	H800	43.18%	51.59%	57.73%	0.5626
Proteus		34.78%	37.36%	40.18%	1.5493

Table 6. Prediction accuracy for kernel overlap slowdown.

Prediction accuracy. We evaluate slowdown prediction accuracy on a dataset of ~5,000 kernels, with detailed results in Table 6. The findings show Moye consistently provides more accurate predictions than the baseline's heuristic, particularly on modern data-center GPUs. On the H800, 57.73% of Moye's predictions fall within a 15% error margin, a notable improvement over Proteus's 40.18%. This superior accuracy is further highlighted by an RMSE nearly 3x lower than the baseline's (0.5626 vs. 1.5493). Moye demonstrates similarly competitive accuracy on the A800, again with a substantially lower RMSE. The model's robustness is also evident on consumer-grade hardware like the RTX 3090, where it outperforms the baseline by a wide margin across all metrics.

Model-wise evaluation. We further assess the benefits of accurate slowdown prediction on model training simulations. Utilizing the advanced Fully Sharded Data Parallel (FSDP) [55] training mode from PyTorch, which incorporates computation-communication overlapping optimizations, we compare the overall computation time against ground truth and a baseline across different GPU architectures. As depicted in Figure 10, Moye achieves an average slowdown prediction error rate of 4.67%, significantly outperforming Proteus, which has an error rate of 18.83%. Notably, on the RTX3090 GPU, Moye maintains superior prediction accuracy, with its error rate being 8x lower than that of Proteus.

Feature score analysis. To interpret our XGBoost slowdown model, we analyze its feature importance scores across the GPU architectures, as shown in Figure 11. The analysis reveals that the model correctly identifies the primary performance bottleneck for each hardware platform. For the consumer-grade RTX 3090, the model attributes the highest importance to memory throughput (30.3%), aligning with the known bandwidth limitations of its GDDR6 memory. In contrast, for the data-center A800, it prioritizes the L2 hit rate (24.1%), reflecting the architecture's large and efficient cache hierarchy designed to mitigate memory latency. Finally, for the compute-centric H800, the model emphasizes achieved SM occupancy (22.5%) and compute throughput (21.4%), underscoring the Hopper architecture's focus on computational density over memory performance.

8.5 End-to-End Results

Simulation accuracy. We evaluate the overall end-to-end simulation accuracy of Moye on H800 clusters of varying

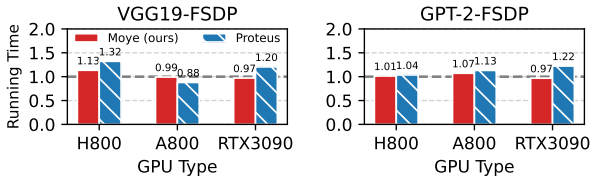


Figure 10. The comparison of model-wise computation running time on different GPUs (H800, A800, RTX3090) in PyTorch FSDP mode for VGG19 and GPT-2 models. Note that all the running times are normalized against the ground truth, represented by the gray line (y=1.0).

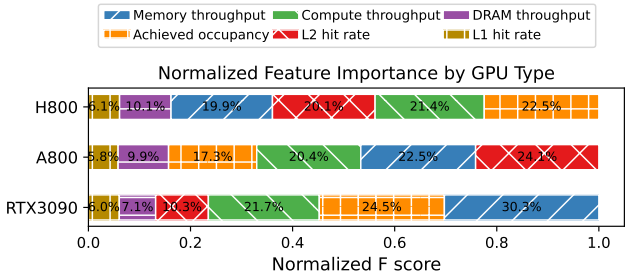


Figure 11. The normalized feature importance of XGBoost across different GPU architectures, highlighting key computational resources.

scales using Megatron-LM. As shown in Figure 12, Moye consistently demonstrates high fidelity across diverse 3D parallelism configurations. On a 96-GPU H800 cluster training GPT-70B ($12 \times 4 \times 2$) and GPT-175B ($12 \times 8 \times 1$), the prediction error is 7% and 8%, respectively. Baseline simulators exhibit larger deviations. Proteus and FlexFlow show errors exceeding 25%. Calculon's predictions are overly optimistic (error rate > 69%) because it relies on idealized analytical models. On average, Moye achieves an error rate of 4.5% for computation workloads and 14.8% for communication operations. Moye's accuracy is enhanced by modeling the slowdown from overlapping kernels—a critical factor that most simulators ignore. In practical tests, this computation-communication overlap constitutes 6.0% to 9.8% of the total step time. Accounting for the interference-induced slowdown would improve its final end-to-end prediction accuracy by 1.2%-3.6%. The importance of this slowdown prediction will become even more pronounced as future training systems more aggressively optimize for kernel overlap.

Simulation time cost. Our evaluation focuses on Moye's simulation time cost, which includes initialization and execution, and excludes the one-time, reusable workload tracing cost. As detailed in Table 7, Moye is highly efficient across different training scenarios. For DDP, simulation is consistently fast, averaging about 10 seconds. The advantage is more pronounced for complex 3D parallelism; by leveraging its white-box communication model, Moye simulates a 128-GPU setup in 83 seconds—a 91.8x speedup over detailed packet-level simulators like SimAI (7655s). This efficiency

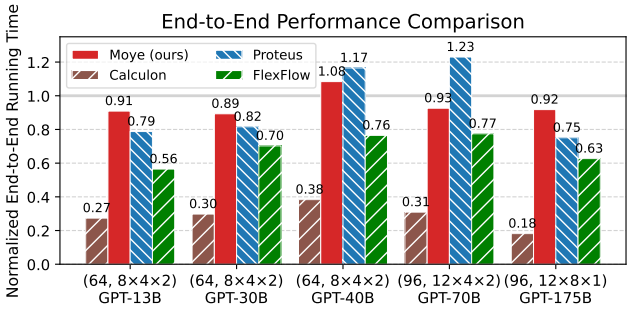


Figure 12. **The end-to-end step time comparison for different GPT model sizes and 3D parallelism configurations (PP × TP × DP) on 64-GPU and 96-GPU H800 clusters. Note that all the data are normalized against the ground truth, represented by the gray line (y=1.0).**

#GPUs	VGG19			GPT 13B~175B		
	Init.	Execution	Total	Init.	Execution	Total
256	0.4	10.3	10.7	8.4	159.2	167.6
1024	0.3	9.7	10.0	35.9	682.8	718.7
4096	0.4	10.2	10.6	149.9	3307.8	3457.7
8192	0.3	10.3	10.5	374.7	4602.2	4976.9

Table 7. The simulation time costs of Moye in seconds. VGG19 is trained using the DDP mode, while GPT models (ranging from 13B to 175B) are trained using 3D parallelism.

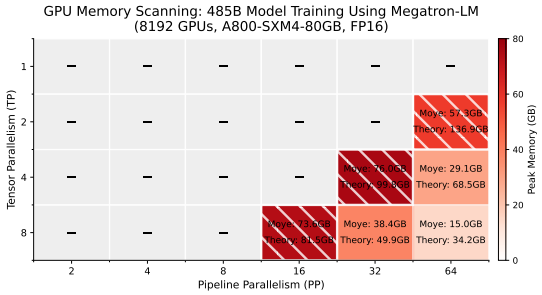


Figure 13. **GPU memory constraint scanning for a 485B model on 8192 GPUs. Each cell represents a unique parallelism strategy defined by its PP and TP sizes, with DP implicitly set to satisfy PP * TP * DP = 8192. Gray cells indicate OOM conditions. The red, hatched cells highlight configurations that are empirically feasible yet are incorrectly identified as OOM by baseline analytical models (Proteus/Calculon).**

scales to large configurations, enabling the simulation of an 8,192-GPU cluster in under 1.4 hours.

9 "What-If" Question Analysis

10 Limitations and Discussions

Moye has limitations and we wish to improve it in at least the following directions.

Communication prediction. We plan to investigate learning based solutions like m3 [34], which efficiently predicts flow-level performance estimation while capturing network-centric factors such as transport protocols. We also wish to extend our modeling to explicitly consider network topology and the potential bandwidth contention, which is currently missing.

Parallelism. **Moye does not yet support sequence/context parallelism**, and memory-efficient ZeRO. We plan to integrate them in subsequent releases.

Setting	Batch size	Peak memory	Throughput	Time (ms)	Cost ($)
(16, 8, 64)	160	73.6 GB	152 TFLOPS	35.2	$1.15
(32, 8, 32)	320	38.4 GB	165 TFLOPS	32.8	$1.28
(64, 8, 16)	640	15.0 GB	171 TFLOPS	31.5	$1.35
(32, 4, 64)	320	76.0 GB	180 TFLOPS	29.9	$1.49
(64, 4, 32)	640	29.1 GB	180 TFLOPS	29.9	$1.49
(64, 2, 64)	640	57.3 GB	195 TFLOPS	27.6	$1.62

Table 8. **Analysis of simulation results of different training configurations at 8,192 GPUs. The red row indicates Moye's optimal choice; the gray row indicates baselines' choice which is suboptimal.**

11 Related Work

We discuss related work other than those covered in §2.

GPU simulation. GPU computation simulators like SCALE-sim [44] and Accel-Sim [33] focus on instruction-level modeling, while other approaches [23, 38, 39, 56] rely on profiling or tracing to capture accurate operator execution times. Moye similarly adopts a trace-based methodology, but crucially does so without requiring a full-scale deployment.

Communication simulation. Discrete event simulators like ns-3 [43] and OMNeT++ [48] are well-known to have high overheads due to their packet-level whole-stack simulation. Other than parallelization optimizations [14, 21], recently learning based approach to predict packet-level or flow-level performance without simulating the whole stack [34, 52, 54] has shown promising results. As mentioned before, integrating them with Moye for training simulation is a promising direction for future work.

12 Conclusion

We presented Moye, a high-fidelity simulator for large-scale distributed ML training that bridges critical gaps in existing approaches. Moye simulates without full-scale deployment with ex-situ workload tracing, estimates CC performance via white-box modeling, and accounts for the slowdown caused by kernels overlapping. Our evaluation shows that Moye not only achieves up to 3x lower simulation error than state-of-the-art baselines but also is highly efficient and practical. We plan to open source Moye to the community.

References

[1] [n. d.]. https://developer.nvidia.com/nsight-compute, title = An interactive profiler for CUDA and NVIDIA OptiX, year = 2025.
[2] [n. d.]. MemryX MX3 Chip. https://memryx.com/products/.
[3] [n. d.]. XGBoost. https://xgboost.readthedocs.io/en/stable/.

[4] 2019. Massively Scale Your Deep Learning Training with NCCL 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[5] 2022. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[6] 2022. GDRCopy. https://github.com/NVIDIA/gdrcopy.

[7] 2023. NPKit: NCCL Profiling Kit. https://github.com/microsoft/NPKit/tree/main.

[8] 2024. https://ai.meta.com/blog/meta-llama-3-1.

[9] 2024. groupedgemm. https://github.com/fanshiqing/grouped_gemm.

[10] 2024. Nvidia Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[11] 2025. A fast communication-overlapping library for tensor/expert parallelism on GPUs. https://github.com/bytedance/flux.

[12] 2025. A system-wide performance analysis tool. https://developer.nvidia.com/nsight-systems.

[13] 2025. SimAI: Unifying Architecture Design and Performance Tunning for Large-Scale Large Language Model Training with Scalability and Precision. In *22th USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.

[14] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. 2024. Unison: A Parallel-Efficient and User-Transparent Network Simulation Kernel. In *Proc. ACM EuroSys*.

[15] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. 2024. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858* (2024).

[16] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling Efficient Scheduling for Communication-Computation Overlap in Large Model Training via Communication Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.

[17] Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Yu Wu, et al. 2024. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066* (2024).

[18] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[19] Jiangfei Duan, Xiuhong Li, Ping Xu, Xingcheng Zhang, Shengen Yan, Yun Liang, and Dahua Lin. 2024. Proteus: Simulating the Performance of Distributed DNN Training. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[20] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, et al. 2024. Efficient Training of Large Language Models on Distributed Infrastructures: A Survey. *arXiv preprint arXiv:2407.20018* (2024).

[21] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. 2023. DONS: Fast and Affordable Discrete Event Network Simulation with Automatic Parallelization. In *Proc. ACM SIGCOMM*.

[22] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[23] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. 2022. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. *Proc. MLSys* (2022).

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

[25] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. 2023. {ARK}:{GPU-driven} Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 87–101.

[26] Mikhail Isaev, Nic Mcdonald, Larry Dennison, and Richard Vuduc. 2023. Calculon: a methodology and tool for high-level co-design of systems and large language models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[28] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[29] Peng Jiang, Christian Sonne, Wangliang Li, Fengqi You, and Siming You. 2024. Preventing the Immense Increase in the Life-Cycle Energy and Carbon Footprints of LLM-Powered Intelligent Chatbots. *Engineering* (2024).

[30] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

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

[32] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*. PMLR, 5156–5165.

[33] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[34] Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2024. m3: Accurate Flow-Level Performance Estimation using Machine Learning. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 813–827.

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

[36] Mingyu Liang, Hiwot Tadese Kassa, Wenyin Fu, Brian Coutinho, Louis Feng, and Christina Delimitrou. 2025. Lumos: Efficient Performance Modeling and Estimation for Large-scale LLM Training. *arXiv preprint arXiv:2504.09307* (2025).

[37] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[38] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Li Li, Jingwen Leng, et al. 2023. DistSim: A performance model of large-scale hybrid distributed DNN training. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 112–122.

[39] Liang Luo, Peter West, Pratyush Patel, Arvind Krishnamurthy, and Luis Ceze. 2022. Srifty: Swift and thrifty distributed neural network training on the cloud. *Proceedings of Machine Learning and Systems* 4

(2022), 833–847.

[40] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2023. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241* (2023).

[41] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {ZeRO-Offload}: Democratizing {Billion-Scale} Model Training. In *USENIX ATC*.

[43] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.

[44] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883* (2018).

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[46] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[47] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[48] Andras Varga. 2019. A practical introduction to the OMNeT++ simulation framework. In *Recent advances in network simulation: the OMNeT++ environment and its ecosystem*. Springer, 3–51.

[49] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. 2022. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 93–106.

[50] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. 2023. xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning. *Journal of Computer Science and Technology* 38, 1 (2023), 166–195.

[51] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. Astra-sim2. 0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 283–294.

[52] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. 2022. DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-Level Visibility. In *Proc. ACM SIGCOMM*.

[53] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *USENIX ATC*.

[54] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. 2021. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proc. ACM SIGCOMM*.

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

[56] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for {DNN} Training. In *USENIX ATC*.

A Additional Observations

We present additional observations on distributed model training in 256-GPU A100 clusters. These cluster consist of up to 256 GPUs (32 servers), with each server equipped with 8 Ampere A100 40GB Tensor Core GPUs. The intra-server connections utilize NVLink, providing 600 GBps of bandwidth, while the inter-server network offers each GPU a dedicated 200 Gb/s InfiniBand connection.

A.1 Training Efficiency Gap

Training time cannot be easily predict. Numerous approaches have been proposed to accelerate training, with practitioners aiming for linear speedup as more devices are added. However, frequent inter-device communication makes achieving this efficiency challenging, as highlighted by extensive research and our own empirical observations. Figure 14 illustrates the performance of three classic DNN models across different cluster sizes and network bandwidths. As cluster size increases, the step time deviates from the expected speedup, with a maximum degradation of 33.1% (Figure 14a). Furthermore, scaling communication time linearly with network bandwidth fails to predict the actual step time (Figure 14b).

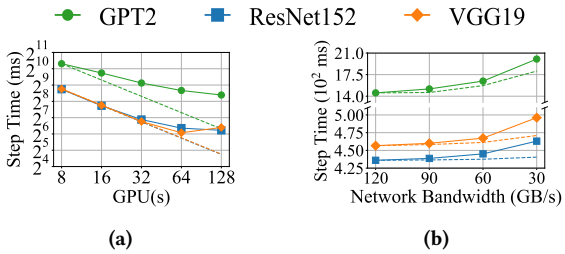
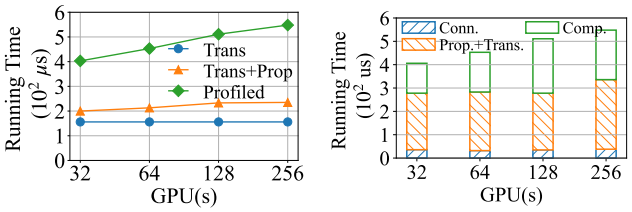


Figure 14. Step time change of three DNN models profiled on A100 cluster. The dashed lines denote the theoretical performance. For network bandwidth, we linearly scale all the communication time and compute the theoretical step time.



(a) Gap with profiled running time. **(b)** Breakdown of running time.

Figure 15. Running time of NCCL `all-reduce` with varying GPU counts, profiled on A100 clusters. We use NCCL-test to launch `all-reduce` kernels and profile the end-to-end running time.

A.2 Communication Analysis

Theoretical model analysis. We empirically show the running time gap between the theoretical transmission delay and the true running time of the all-reduce kernel. We use NCCL-test to launch all-reduce kernels and profile the end-to-end running time. Figure 15a depicts the performance in different cluster size. The value of $tensor\_size/bandwidth$ is a constant while the running time of all-reduce gradually grows with the cluster size. In a cluster with 256 GPUs, the performance gap is up to 251%. Then, we consider another possible solution, Trans. + Prop. in Figure 15a. It incorporates theoretical propagation delay ($distance/propagation\_speed$) because servers propagate reduced tensor to others in the `all-reduce` kernel. However, it cannot simulate correctly either.

NCCL breakdown analysis. We thus perform a breakdown analysis of the `all-reduce` running time in Figure 15b. A complete `all-reduce` kernel involves (a) initiating the connection, (b) transmitting and propagating the message among servers and (c) averaging the results. Empirically, we find that the running time of `all-reduce` kernel is indeed a combination of the transfer time, data reduction time and overhead of connection setup. Here, the transfer time is measured to be the time elapsed between the start of the transmission on the sender and the end of receiving the tensor on the receiver. We notice that an average cost of 352.5 μs is spent on connection setup. Both propagation and computation time for reducing the tensor are non-trivial and gradually increase as the cluster scales up. Apart from the significant contribution of reduce computation, the profiled transfer time is larger than the sum of the theoretical transmission delay and propagation delay.

B NCCL Algorithm Mechanism

Figure 16 illustrates how ring-based and tree-based communication execution work.

Ring-based communication. GPUs are organized into a logical ring. The communication proceeds in a pipelined fashion over data chunks. The workflow consists of two main stages: a `reduce-scatter` followed by an `all-gather`. In the initial `reduce-scatter` stage, each GPU iteratively sends a chunk to its neighbor while receiving a different chunk from its other neighbor, reducing the incoming data with its local copy. This process repeats for $N-1$ steps, after which each GPU holds a fully reduced segment of the final tensor. In the subsequent `all-gather` stage, these reduced segments are circulated around the ring for another $N-1$ steps until all GPUs have a complete copy of the final result.

Tree-based communication. The algorithm leverages a hierarchical tree structure, often a double binary tree that maps to the physical server topology (intra-node and inter-node communication). The operation unfolds in two distinct phases. First, in the `reduce` phase, data chunks are aggregated up the tree from leaf GPUs to the root. Each parent GPU receives chunks from its children, reduces them with its own data, and passes the result to its parent. Once the root contains the final, fully-reduced tensor, it initiates the

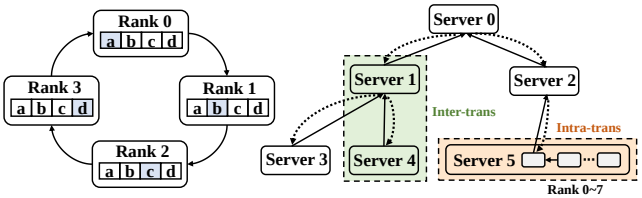


Figure 16. Illustration of ring-based (left) and tree-based (right) communication execution. In the tree-based communication, black solid arrows denote the reduce process, while dashed arrows indicate the broadcast process.

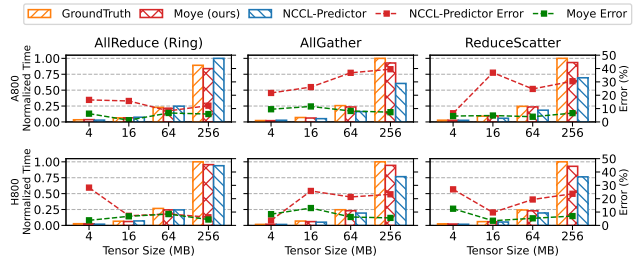


Figure 17. Running time and prediction error for intra-server CC kernels on H800 and A800 GPUs across varying tensor sizes. Each subplot is normalized relative to the operation with the longest running time.

`broadcast` phase, disseminating the result back down the tree to all participating GPUs. This hierarchical approach is particularly effective at optimizing for network topologies with varying bandwidths, such as fast intra-node NVLink and slower inter-node InfiniBand.

Intra-server communication. In Figure 17, we show the running time and prediction error for intra-server CC kernels on H800 and A800 GPUs across varying tensor sizes.

C Model Configurations

The model configurations are summarized in Table 9. Moye can, in principle, **simulate any ML workload**—supporting models of arbitrary sizes, both dense and MoE—provided that the model is supported by a mainstream training framework. In this paper, we focus on simulating runtime and throughput rather than evaluating a task performance, and therefore we use a diverse set of architectures and scales to thoroughly evaluate the generality of Moye.

D Slowdown Predictions

D.1 Hyperparameters Setting

We experimentally find the best hyperparameters for our XGBoost model as shown in Table 11.

D.2 CDFs of Slowdown Predictions

Figure 18 compares the cumulative distribution functions (CDFs) of kernel slowdown prediction error rate predictions

Model	#Params.	#Heads	#Layers	#Hidden	#Seq.
GPT-3 v1	13B	32	40	5120	2048
GPT-3 v2	30B	48	40	7680	2048
GPT-3 v3	40B	72	40	9216	2048
GPT-3 v4	70B	64	80	8192	2048
GPT-3 175B	175B	96	96	12288	2048
GPT-3 v5	485B	160	96	20480	2048

Table 9. Model configurations for transformer models.

Model	#Params.	#Experts	#Heads	#Layers	#Hidden	#FFN Hidden	#Seq.
Mixtral	8×1.75B	8	32	8	4096	14336	4096
Mixtral	8×7B	8	32	32	4096	14336	4096
Mixtral	16×7B	16	32	32	4096	14336	4096
Mixtral	32×7B	32	32	32	4096	14336	4096
Mixtral	8×22B	8	56	56	6144	16384	4096

Table 10. Model configurations for MoE models.

Hyperparameter	Value
Model	XGBRegressor
Objective	reg:squarederror
Max depth	12
Random state	42
Train/Test Ratio	0.8/0.2
K-fold cross-validation	5-fold

Table 11. The hyperparameters of XGBoost model.

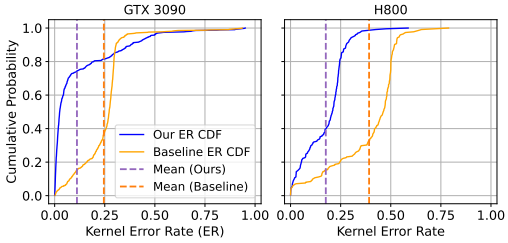


Figure 18. CDFs of kernel slowdown prediction error rate predictions for Moye and the baseline (Proteus) on RTX 3090 and H800 cluster.

made by Moye and the baseline method, Proteus. The comparison is shown for two hardware setups: an RTX 3090 GPU and an H800 cluster. Moye demonstrates a significantly lower kernel error rate across both platforms, with a sharper CDF slope and lower mean error rate compared to the baseline.

E End-to-End Simulation

We evaluate the accuracy of Moye in different training configurations under PyTorch's DDP mode. We adjust the cluster size and network bandwidth. Figure 19 shows the performance of the baseline and Moye compared with the profiled ones. The average prediction accuracy is 97.25%. For language models, the error rate is reduced by at least 2.02x.

Model	Network (/s)	Profiled (ms)	Prediction (ms)	Error (%)
BERT_Large	100Gb	454	470.8	3.70
	1.6Tb	355.8	376.6	5.84
GPT2	100Gb	199.5	206.3	3.40
	1.6Tb	167.4	176.9	5.67
VGG19	100Gb	529.9	521.4	1.60
	1.6Tb	485.6	489.2	0.86
ResNet152	100Gb	351.1	374.3	6.61
	1.6Tb	335.7	354.7	5.65

Table 12. Prediction error rate of Moye in the cluster with 128 A100 GPUs. We change the network interconnection from 1.6Tb/s InfiniBand to 100 Gb/s PCIe.

We find that the Baseline solution performs better in CNN models. We also evaluate Moye by replacing the InfiniBand network with a 100GB/s PCIe interconnection so that it takes a long time to transmit model gradients (Table 12). The average accuracy is 96.17%. Degradation in the accuracy mainly stems from the error in predicting the slowdown factor.

F Moye's API calls

We provide Moye's example API calls for tracing and end-to-end simulation here.

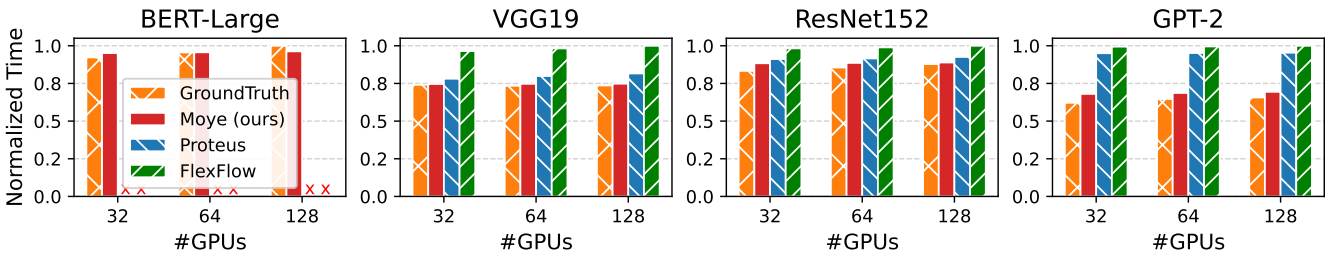


Figure 19. Comparison of prediction accuracy between Baseline and Moye when the cluster size increases.