A Few GPUs, A Whole Lotta Scale: Faithful LLM Training Emulation with CrystalLLM

Shaoke Xi[1,*], ChonLam Lao[1,2,*], Boyi Jia[1,3,*], Jiaqi Gao[1,*,†], Zhipeng Zhang[1], Jiamin Cao[1], Brian Sutioso[2], Erci Xu[3,†], Minlan Yu[2], Kui Ren[4], Yong Li[1], Zhengping Qian[1], Ennan Zhai[1], Jingren Zhou[1]

[1]Alibaba Group, [2]Harvard University, [3]Shanghai Jiao Tong University, [4]Zhejiang University

ABSTRACT

Large language model (LLM) training today runs on clusters spanning thousands of GPUs. While this scale enables rapid model advances, developing, debugging, and performance-tuning, the training framework inevitably becomes complex and costly. This is because engineers often need to reproduce production behaviors to diagnose failures or evaluate optimizations, thereby demanding frequent and even exclusive access to production-scale clusters—which becomes increasingly hard given that the majority of GPUs are already committed to production workloads. Simulation relies on complex performance models that are difficult to maintain, and downscaled experiments often fail to capture scale-dependent behaviors.

We present CrystalLLM to decouple large-scale execution from the need to access large clusters, enabling engineers to run and observe ranks of interest under faithful large-scale behavior using only a few GPUs. CrystalLLM constructs a high-fidelity execution graph via a slicing-based approach that captures computation, communication, and dependencies of the target scale. Then, CrystalLLM performs hybrid emulation where selected ranks execute the original program while the remaining are replayed as virtual participants.

Experiments on large-scale LLM training workloads show that CrystalLLM accurately reproduces performance and memory behavior, achieving only 0.58% average error in iteration time and less than 0.01% error in peak GPU memory usage. CrystalLLM can emulate clusters of up to 8192 GPUs using fewer than 1% of the physical GPUs required by the original deployment.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**;

*These authors contributed equally to this work.

†Jiaqi Gao and Erci Xu are co-corresponding authors.

SOSP '26, September 29–October 2, 2026, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09.
https://doi.org/10.1145/3830418.3843852

1 INTRODUCTION

Training large language models (LLMs) has been scaling at an unprecedented speed. Production systems today routinely involve thousands to tens of thousands of GPUs [7, 13, 27, 31, 46], and emerging frontier models are expected to push this scale even further. While significant research has focused on improving training performance at such levels [5, 6, 36–38, 47, 50], much less attention has been paid for the cost of developing, debugging, and optimizing the systems that enable these workloads.

In practice, LLM training follows a DevOps workflow in which engineers continuously analyze training telemetry, implement optimizations, and deploy updates during ongoing runs. This workflow critically depends on the ability to reproduce and analyze field behavior under realistic conditions. Unfortunately, as the cluster scale grows, the cost of system experimentation increases proportionally. Reproducing a performance issue, validating a design change, or conducting a what-if analysis often requires access to the same production-scale cluster that runs the training job [24, 41]. Allocating thousands of GPUs solely for debugging or experimentation is both expensive and operationally disruptive, making rapid system iteration increasingly difficult.

Two existing solutions partially alleviate the issue, but neither suffices. One option is simulation [2, 14, 19, 26, 35, 42, 44, 51], which lowers hardware cost by replacing real execution with profiled traces and analytical models. But simulation accuracy depends on the fidelity of those models, and maintaining them is increasingly difficult as training stacks evolve rapidly across frameworks, compilers, kernels, network, and hardware generations [5, 6, 20, 32, 37, 47]. The other option is to run real stack at smaller scale. However, downscaling changes exactly the behaviors engineers care about: world size, parallelization strategy, batch size, and scheduling choices all reshape communication structure, memory layout, and bottlenecks [34, 38]. Hence, conclusions drawn from downscaled experiments often fail to transfer to production. In short, we lack a way to study large-scale training behavior faithfully without paying excessive cost.

This paper presents CrystalLLM, a system that enables faithful emulation of large-scale LLM training using limited hardware resources. The key insight behind CrystalLLM is that large-scale interaction structure does not require all

ranks—the per-GPU worker processes of a distributed training job—to execute simultaneously on physical GPUs. Instead, scale can be virtualized by capturing the execution structure of a training job and selectively replaying it while running only a subset of ranks on real hardware. By decoupling the logical training scale from physical GPU allocation, CrystalLLM allows engineers to observe large-scale behavior while using only a small fraction of the original hardware.

CrystalLLM achieves this through a two-phase design. First, CrystalLLM constructs a high-fidelity execution graph that captures computation, communication, and dependency relationships across all ranks of a training job. This graph is generated using a context-switching execution mechanism that multiplexes many logical ranks onto a small set of GPUs. Second, CrystalLLM performs hybrid emulation in which selected ranks execute the real training program on physical GPUs while the remaining ranks are replayed as virtual participants that preserve the communication and synchronization behavior of the original deployment. This design allows engineers to run unmodified training code while observing realistic, large-scale interactions.

In summary, this paper makes the following contributions:

- We identify faithful emulation of large-scale LLM training as a systems problem that neither simulation nor downscaled execution solves, and show that scale can be virtualized: only the ranks of interest need physical GPUs, while the rest participate through replay (§2, §3).
- We design CrystalLLM's two-phase architecture: a graph preparation subsystem that captures a full-scale job's execution graph using only a few GPUs, via context-switched collection and slice-based timing calibration (§4); and a hybrid emulation runtime in which sandbox ranks run the unmodified program while virtual ranks replay the graph to drive their communication, made practical by virtualized initialization and communication pruning (§5).
- We evaluate CrystalLLM on large-scale training workloads of up to 2,048 GPUs: across diverse models and parallelization strategies, it predicts iteration time with 0.58% average error and peak GPU memory with negligible error, using fewer than 1% of the original hardware, and supports practical experimentation and debugging workflows (§7, §8).

CrystalLLM is available at:
https://github.com/AlibabaResearch/CrystalLLM.git.

2 MOTIVATION

2.1 The Dilemma of DevOps in LLM Training

LLM training engineers operate in a DevOps loop: they analyze telemetry and failures from prior or ongoing runs to set optimization targets (e.g., lower step time, reduced memory overhead); implement changes that span Python orchestration, the distributed runtime, communication libraries, and CUDA kernels; verify them on a small number of GPUs; and roll them out to the production job at permitted points such as checkpoint boundaries or restarts, rolling back when regressions appear [41].

This loop faces a fundamental constraint: GPU scarcity. Under the competition of releasing frontier models every 3–6 months [8], the majority of GPUs are occupied by production workloads, leaving only a small fraction (hundreds or even tens) for development, debugging, and validation. Engineers can implement and locally validate changes quickly, but confirmation often arrives only after large-scale deployment.

Small-scale verification provides only partial confidence. Performance is determined by interactions across system layers—GPU microarchitecture, communication, and the memory hierarchy—any of which can become the bottleneck, and optimizations such as communication–computation overlap [3, 49] must be designed against this full picture; problems such as synchronization anomalies, collective contention, and memory pressure often surface only at scale. Meanwhile, the stack keeps moving: kernels are frequently redesigned and fused [5, 6, 37, 47], and new GPU generations arrive roughly every two years [18] with different compute, memory, and interconnect characteristics. Conclusions therefore age quickly, and each shift sends engineers back to the scarce large-scale resources for re-validation.

2.2 Simulating Large-scale Clusters?

Engineers have long sought methods to evaluate the performance of ML training workloads without requiring large-scale GPU deployments. Prior work has explored simulation-based systems [2, 14, 19, 26, 33, 35, 42, 44, 51]. These systems approximate large-scale training behavior by constructing and replaying intermediate representations using techniques such as execution trace collection, operator-level profiling, and analytical modeling, enabling performance estimation without full-scale execution, and are effective for early design exploration.

However, simulation accuracy depends on the fidelity of the underlying models: more detailed models capture more behavior but take more engineering to build and maintain, and this trade-off steepens as the stack grows more complex and evolves faster. SimAI [42] maintains a mocked version of the ML framework alongside the real one; Phantora [33], though simulating at a lower level, still models each CUDA operation and network behavior by hand. Some behaviors—GPU kernel scheduling, thermal throttling—are opaque to the modeler and resist accurate modeling. And as the assumptions embedded in a model drift from the evolving system, profiling and re-implementation recur [33]. For questions that require reproducing production behavior, this modeling cost is paid again with every change to the stack.

2.3 Downscaling the Testing Environment?

Another option is downscaling: running the full stack on a reduced deployment to approximate production behavior, typically by shrinking model parameters, batch sizes, or

parallelism degrees (e.g., DP, PP, TP, and EP) [36, 38] while preserving the original execution paths as much as possible.

Constructing a faithful downscaled environment, however, requires careful reconfiguration. Consider observing the pipeline communication between two middle stages to improve communication–computation overlap: with fewer GPUs than production, the remaining parallelism dimensions (DP, EP, and TP) must shrink to preserve the pipeline configuration, and dependent parameters—the virtual pipeline parallelism (VPP) degree, the global batch size—must be reconfigured accordingly [29]. The result is a configuration that diverges from deployment: for instance, shrinking the batch size below the VPP degree leaves too few microbatches to drive the pipeline schedule, a case that requires special handling yet never arises online.

Even when carefully constructed, downscaled experiments often fail to reflect large-scale behavior. Training performance is scale-sensitive: bottlenecks shift with scale (e.g., from computation to communication and memory capacity) [29], so conclusions may not generalize and can even mislead optimization decisions. CPU activation offloading [34, 36] illustrates this: deciding which activations to offload and when to load them back trades memory savings against interference with ongoing computation and communication; because downscaling alters the memory layout, per-GPU workload, and pipeline schedule, the measured offloading benefit no longer matches production.

2.4 Selective GPU Execution: A Promising Idea

Field development workflows suggest a different intuition. When debugging or analyzing large-scale training runtime, engineers rarely need to inspect the behavior of all ranks simultaneously. Instead, they typically focus on a small subset of ranks of interest (e.g., 8 GPUs), while the remaining ranks simply participate in the distributed execution. For example, engineers often examine the end-to-end step time by observing when the last rank completes an iteration, or inspect the execution path of a specific rank to understand program logic or diagnose failures. Similarly, debugging issues such as out-of-memory (OOM) errors or communication stalls typically requires detailed inspection of only a few ranks rather than the entire deployment.

This observation raises a potential opportunity: *can we execute only the ranks of interest physically while maintaining the rest of the system logically?* Such an approach could preserve deployment-scale interaction structures while significantly reducing hardware requirements. Ideally, we can build a hybrid emulator that decouples logical scale from physical hardware: it first records, offline, how the job executes at scale—what each rank does, and when—so that a few GPUs suffice to reproduce this behavior faithfully. Engineers then specify the ranks of interest, which run the real training program on physical GPUs, while the remaining logical ranks are virtualized to replay the recorded behavior and preserve the interaction structure of the full deployment.

2.5 Envision: A Trace-driven Emulator

Based on our in-production experience, we envision an emulation system that achieves three goals:

- **Low resource footprint.** The system should emulate the large-scale workload with only a few GPUs.
- **High fidelity.** The system should faithfully reproduce the behavior of the ranks of interest as large-scale runs.
- **Code reuse.** The emulation system should directly reuse the current LLM training code base to avoid unnecessary development or maintenance overhead.

A natural approach is record-and-replay where we record execution trace of a program and replay it in the test environment. But, there are two challenges.

Challenge 1: You need scale to see scale. Replay presupposes a faithful representation of large-scale behavior—what runs, in what order, and for how long. Capturing such a representation, however, seems to require running the job at full scale, precisely the cost we aim to avoid; and once ranks are multiplexed onto fewer GPUs, the measured timing no longer reflects the original execution.

Challenge 2: Preserving large-scale behavior at small scale. Even given a faithful graph, the ranks of interest behave realistically only if the remaining thousands of ranks participate: communication must actually arrive, and collectives must complete. Hosting so many virtual participants on a few GPUs risks exhausting memory and contending with the sandbox for the very resources under measurement.

3 CRYSTALLLM DESIGN OVERVIEW

Our goal is to build a ***low-resource-footprint*** and ***high-fidelity*** emulation system that allows engineers to run their ***existing code*** while executing only the ranks of interest in large-scale training, enabling efficient performance evaluation and faithful reproduction of program behavior.

We hence develop CrystalLLM in two phases: *preparation*, which captures a high-fidelity execution graph of the large-scale job on a few GPUs, and *emulation*, which runs the unmodified program on the ranks of interest while replaying the graph around them, as if deployed at full scale.

3.1 Assumptions and Scope

Assumptions. CrystalLLM follows an offline-capture, online-replay strategy: the execution graph is captured on a small test cluster, then replayed to drive emulation. This strategy holds under three assumptions. (1) *Deterministic execution.* Given the same program and training configuration, every run issues the same operation sequence, so graphs captured across runs can be merged and replayed consistently. Once the configuration changes, the graph must be re-captured. (2) *DP symmetry.* Offline capture multiplexes a few GPUs over many logical ranks, so collection time grows with the number of distinct ranks. Asymmetric dimensions

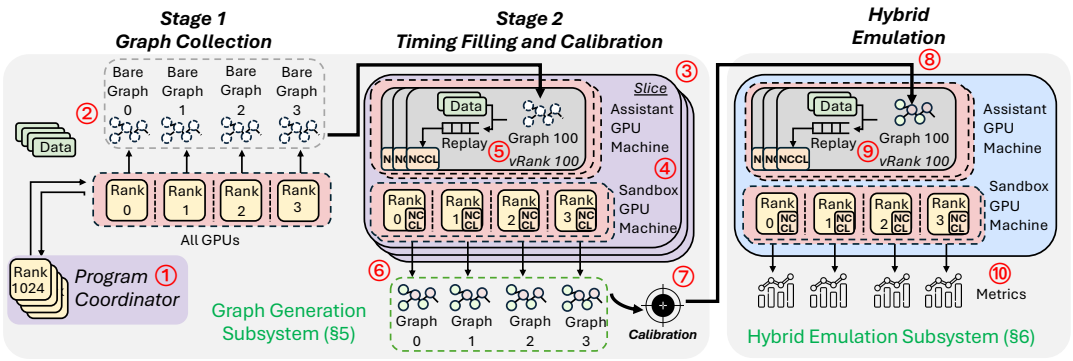


Figure 1. System overview. Phase 1 (graph preparation, §4) collects the bare graph via context switching (Stage 1) and fills and calibrates timing (Stage 2) to produce the execution graph. Phase 2 (hybrid emulation, §5) runs the ranks of interest on sandbox GPUs while virtual ranks replay the graph on assistant GPUs.

(e.g., pipeline stages) must be captured individually, whereas data-parallel (DP) replicas produce structurally identical graphs: capturing a single model replica suffices, and its graph is reused across all replicas (§4.2). This symmetry extends beyond structure: in steady-state pretraining, symmetric ranks exhibit statistically indistinguishable timing, so the captured graph reflects the *normal* (average) behavior of the job at scale, rather than transient outliers on individual replicas (e.g., a random straggler). (3) *Timely data supply.* Online replay must keep pace with the sandbox: communication data of virtual ranks must be GPU-resident before the sandbox reaches the corresponding operation. This holds when the prefetch bandwidth (host-to-GPU PCIe, e.g., 64 GB/s) exceeds the network bandwidth consuming it (e.g., 50 GB/s per NIC) (§5.1).

When CrystalLLM is appropriate. CrystalLLM is appropriate for existing training jobs whose execution and communication graphs are deterministic across runs, including mainstream pretraining stacks such as Megatron-LM [38], DeepSpeed [36], and FSDP [50]. It targets questions that can be answered by observing selected ranks over one or a few iterations.

When not to use CrystalLLM. CrystalLLM is not the right tool in the following cases. (1) When runnable code for the feature being evaluated or the target hardware does not yet exist, simulation is more appropriate because CrystalLLM relies on actual execution of the target software on target GPUs. (2) Fixed-graph replay is invalid when runtime data changes the control flow or communication structure, as in some RL post-training workloads. (3) Behaviors that require genuine computation by all ranks over many iterations—including convergence or numerical drift, congestion created jointly by all traffic, and transient-straggler distributions—require full-scale execution. CrystalLLM may identify an actually faulty sandbox node relative to a normal baseline (§8), but replay does not synthesize rare failures.

Prototype coverage. Our prototype currently supports PyTorch training programs that communicate through NCCL [20] (NVIDIA's collective communication library). Fused compute–communication kernels are not yet supported; extensions to additional stacks or kernels must preserve a single invariant—the sandbox retains the kernel behavior it would have at full scale.

3.2 System Workflow

Phase 1: Graph preparation (§4). The first preparation subsystem constructs a high-fidelity execution representation (i.e., an *execution graph*) for all ranks in the program. CrystalLLM's centralized *coordinator* ① first executes the original program on a small number of GPUs by scheduling only a subset of ranks at a time and coordinating their progress. It multiplexes execution across ranks and records an execution graph for each rank. This process produces a *bare graph* that encodes operations and dependencies without accurate timing ②.

To obtain accurate timing under limited GPU resources, CrystalLLM collects execution graph timings locally within each *slice* ③—a subset of ranks executed at a time on the available GPUs—and then merges and calibrates them to construct a globally consistent execution timeline. The system executes slices one by one. In each slice, the available GPUs are partitioned into two roles: *sandbox GPUs* ④ and *assistant GPUs* ⑤. A subset of ranks runs as real ranks on sandbox GPUs, while the remaining are replayed as virtual ones on assistant GPUs to serve as communication counterparts and preserve correct timing (e.g., responding to collective communication). CrystalLLM iterates over these slices in a round-robin manner (e.g., 0-7, 8-15, …), ensuring that each rank is executed once as a real rank.

After collecting per-slice execution timings ⑥, CrystalLLM performs cross-slice calibration to reconstruct the complete high-fidelity execution graph that captures globally consistent large-scale behavior ⑦.

Phase 2: Hybrid emulation (§5). After graph preparation, CrystalLLM performs emulation in the second subsystem ⑧. In this phase, only the specified ranks of interest are executed using the original program on *sandbox GPUs*, serving as the primary observation points. The remaining ranks are instantiated as virtual ranks and replayed on *assistant*

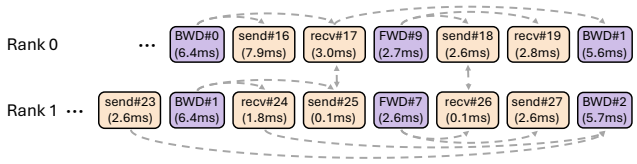


Figure 2. Snippet from CrystalTrace capturing dependencies of 1F1B.

GPUs using the constructed execution graph; these virtual ranks perform real communication with sandbox ranks ⑨. Unlike the previous, this phase runs a single emulation instance where all virtual ranks are replayed using the complete execution graph with accurate timing. This enables CrystalLLM to accurately measure metrics of interest (e.g., end-to-end iteration time and GPU memory usage over time) for sandbox ranks without requiring full-scale hardware ⑩.

The required sandbox size depends on the stage. Collecting the graph structure runs on any number of GPUs, as timing is discarded at this stage (§4.2). For timing-sensitive stages—slice timing filling and hybrid emulation—CrystalLLM provisions the sandbox at node granularity, with enough nodes to host one tightly-coupled parallelism group (max(TP, EP) ranks; e.g., two 8-GPU nodes for EP=16). These groups communicate at per-layer, microsecond cadence (e.g., TP collectives over NVLink), a granularity at which virtual peers risk perturbing the observed timing; meanwhile, a sub-node sandbox saves little, as it still needs an assistant node to carry its inter-node traffic while complicating intra-node replay. In common configurations, emulation thus requires as few as two machines: one sandbox node and one assistant node. The number of assistant GPUs must match that of the sandbox GPUs to ensure sufficient network bandwidth.

4 ESTABLISHING HIGH-FIDELITY GRAPH

To enable replay during emulation, we design CrystalTrace, a replay-oriented graph representation (§4.1) that captures essential information while faithfully preserving large-scale behavior at low cost. We construct this graph using a small number of GPUs (§4.2) and ensure accurate timing (§4.3).

4.1 CrystalTrace Execution Graph

CrystalLLM constructs an execution graph that captures both communication and computation across all ranks, providing a complete view for replay. For each rank, the graph must answer three questions: (1) *what* operations are executed (computation and communication spans), (2) *in what order* they execute (intra- and inter-rank dependencies), and (3) *how long* they take (duration of each span).

A common approach is to use the PyTorch Profiler [32] with tools such as Chakra [39] and Meta HTA [28]. However, PyTorch Profiler incurs significant overhead—we observe up to 20.04% iteration time increase in a 128-GPU Qwen3 training job [46]—because it is not designed for replay. It captures operator-level events that are unnecessary for our purpose. Since our goal is to determine when communication

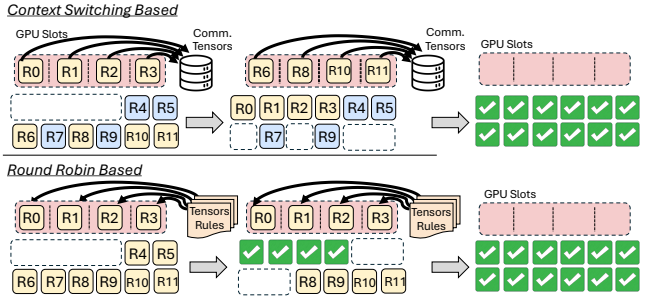


Figure 3. Context switch to generate graph.

should be issued during replay (not to re-execute computation), only communication timing and dependencies need to be preserved; operator-level details are redundant.

Motivated by this insight, we design *CrystalTrace*, a new tracing format that captures only the minimal information required for replay. CrystalTrace operates at a coarser scheduling granularity (e.g., per microbatch), where each scheduling unit defines the minimal dependency boundary across downstream communication.

Figure 2 shows a snippet of the graph. It consists of *nodes* and *edges*: each node represents a computation span or a communication event, identified by a unique `id` and a `duration`, measured on the GPU with CUDA events; each edge encodes a dependency between two nodes (id_1, id_2). We distinguish two dependency types: (1) *directional*, where one operation must complete before another begins (e.g., communication followed by computation within a rank), and (2) *synchronization*, where all participating nodes must reach the operation before any can proceed (e.g., collectives or matched send–receive pairs).

Finally, CrystalTrace records timing from the GPU side only, without tracing explicit CPU-side events. Because actual communication timing is determined by GPU execution—not by when the CPU issues the call—CPU-side timestamps are unnecessary for replay. Nor is the CPU's influence lost: CPU bottlenecks (e.g., data-loading stalls or kernel-launch queuing) manifest as longer gaps between the GPU events that CrystalTrace records, and are thus replayed as-is. By focusing exclusively on execution structure and communication boundaries, CrystalLLM captures only what is necessary for faithful replay, significantly reducing tracing overhead.

4.2 Stage 1: Collecting CrystalTrace

Collecting CrystalTrace for all ranks requires running the full training job at scale, but this conflicts with the requirement of using only a small number of GPUs. To address this, CrystalLLM introduces a context-switching execution system that decouples logical execution from physical scale, enabling graph collection on a small number of GPUs.

At a high level, CrystalLLM maintains a pool of logical ranks, each with its own execution state. At any moment, only N ranks are active on the N available GPUs, while the remaining ranks are paused in CPU memory (or spilled to disk). The CrystalLLM coordinator schedules subsets of

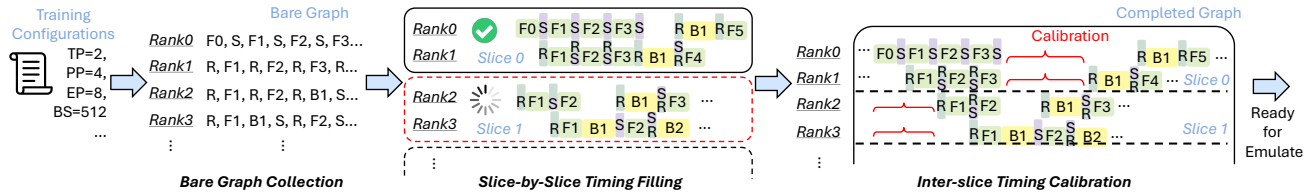


Figure 4. Workflow for generating the execution graph for emulation. Block colors: light green = forward (F), yellow = backward (B), purple = send (S), dark green = receive (R).

ranks onto the GPUs and runs them until they block on a communication point. When a rank blocks, its execution state and communication context are saved, and the system switches to other runnable ranks. By iteratively advancing all ranks, CrystalLLM captures the full execution graph without requiring large-scale hardware. Because each rank executes the original program with real tensor values, this approach guarantees that the program follows the correct code paths and handles value-dependent logic faithfully.

End-to-end workflow. CrystalLLM runs unmodified training programs transparently and manages all ranks through a centralized coordinator. The input can be any large-scale training job (e.g., 256 GPUs with PP=4, TP=4, DP=16), as the parallelism configuration is not relevant.

As shown in Figure 3, at the start of each iteration, the coordinator schedules up to N (e.g., 4) ranks onto available GPUs and begins recording their execution graphs. Each rank runs until it reaches a communication point (e.g., an all-reduce). The coordinator then checks whether all participants of the collective are active. If not, the collective cannot proceed. For example, in Figure 3, ranks R0, R1, R2, R3, R6, R8, R10, and R11 belong to the same collective, but not all are active simultaneously, causing execution to block. The active rank then stores its communication input tensors in host memory, checkpoints its GPU context, and swaps out to free the GPU.

Next, the coordinator schedules ranks that can unblock the collective (e.g., R6, R8, R10, and R11), prioritizing such ranks to reduce context switching (Appendix §A). Once all participants reach the collective and their inputs are available, CrystalLLM executes the collective on the CPU and produces outputs. These outputs unblock stalled ranks, which are later rescheduled and resume execution from where they paused.

This process—execute, block, swap out, gather tensors, resolve communication, and resume—repeats until all ranks complete the iteration. By the end, CrystalLLM captures the full execution graph across all ranks, including operations (*what*) and dependencies (*in what order*). Timing (*how long*) is added later during calibration (§4.3).

User-defined communication input. While context switching correctly handles value-dependent logic, it can incur non-trivial overhead when GPUs are limited. First, storage overhead can be substantial: an A100 GPU has 80 GB of memory, so replaying a 128-GPU job requires storing up to 128 × 80 GB (≈ 10 TB) of data in host memory for swapping; to scale beyond host memory limits, CrystalLLM can spill inactive contexts to disk using CRIU [4]. Second, GPU context switching itself is expensive: checkpointing and restoring a near-full 80 GB context can take more than two minutes using `cuda-checkpoint` [30].

To avoid these overheads, CrystalLLM provides an alternative for advanced users who have prior knowledge of their workload: directly specifying the input and output tensors required for communication, bypassing context-switching entirely. These tensors can be obtained from pre-recorded communication data from prior large-scale runs or generated using user-defined rules. Most tensors, such as weights and gradients, feed numerical computation rather than control flow, and values within a reasonable range typically suffice; user-defined rules matter for the few tensors whose values steer the program's execution logic. Our Megatron-LM workloads involved three: *dataloader statuses*—rank 0 broadcasts its dataloader status after construction, and a failure terminates training, so we inject a "successful" status; *training samples*—embedding-layer ranks receive token indices via broadcast within their TP group, and out-of-vocabulary indices trigger index-out-of-bounds errors, so we inject valid in-vocab values; *MoE dispatch all-to-all splits*—an `allgather` of gating results sizes the subsequent `all-to-all` buffers, and overly large splits cause unintended OOMs, so we inject values that keep the splits within a reasonable range.

With user-provided tensors, each rank can execute independently using its corresponding inputs on the available GPUs, as shown in the lower part of Figure 3. For a 235B MoE model on a 2,048-GPU target collected with 32 GPUs, pure context switching takes about 9 hours, whereas user-specified tensors reduce collection to 64 rounds of roughly one minute each (≈64 minutes). The rounds can be further cut by a factor of $1/N$, where N is the data-parallel (DP) group size, since execution graphs are identical across DP groups: only one model replica (32 ranks) needs collection—a single round, about one minute. Re-collection is needed only when the graph's organization changes, i.e., parallelism settings or collective-communication metadata; compute- or memory-only changes (e.g., kernel fusion, recomputation) reuse the graph and only re-calibrate timing.

4.3 Stage 2: Timing Filling and Calibration

After collecting the bare graph, we know *what* operations execute and *in what order*, but not *how long* they take. This is

because graph collection introduces overheads (e.g., context switching) that distort timing.

To recover accurate timing with limited GPUs, we divide the training job into subsets, or *slices*. We execute slices sequentially, using all available GPUs per slice. In each slice, a subset of ranks runs with real computation and communication, while assistant GPUs emulate the remaining ranks. After all slices are executed, each rank has been run at least once as a real rank, yielding accurate slice-level timing. We then combine slice-level graphs and calibrate their timings to construct a globally consistent execution graph.

1. Slice-by-slice timing filling. We execute the workload in slices. In each slice, a subset of ranks runs the original program on available GPUs, while the rest are replayed as virtual ranks on assistant GPUs using the bare graph (capturing *what* and *in what order*) to provide realistic communication behavior. Native behaviors on real ranks, such as multi-stream communication–computation overlapping, are thus naturally preserved; virtual ranks simply act as communication peers through the same framework stack, supplying the communication data recorded or user-defined in Stage 1. This allows each real rank to observe realistic interactions and enables accurate measurement of computation and communication durations.

For example, in a 1024-rank workload, a slice may execute ranks 0–7 on sandbox GPUs, while ranks 8–1023 are replayed on assistant GPUs. CrystalLLM iterates over such subsets in a round-robin manner (e.g., 0–7, 8–15, …), ensuring every rank is executed once as a real rank.

2. Inter-slice timing calibration. Timing within each slice is locally accurate but not globally aligned. We therefore use dependency information in the execution graph to align timestamps across slices.

As shown in Figure 4, consider a pipeline-parallel job (degree 4) split into two slices. Each slice produces its own execution graph (e.g., slice 0 and slice 1) with locally accurate timing. We then correlate across slices using dependencies: for example, a receive in slice 1 depends on a send in slice 0, so we shift the receive to occur after the send. Propagating such constraints across the graph reconstructs consistent cross-rank timing and removes artifacts from limited-GPU execution. After calibration, we obtain a high-fidelity execution graph that reflects large-scale training behavior.

5 HYBRID EMULATION

After collecting the execution graph, we perform emulation to obtain performance metrics. We next describe how the execution graph is replayed (§5.1). We then present how our design minimizes GPU cost during hybrid emulation by mapping virtual ranks onto physical assistant GPUs with low overhead through virtualized initialization (§5.2) and communication pruning (§5.3).

5.1 Assistant GPU Workflow for Emulation

We execute the real program on *sandbox GPUs* while replaying many virtual ranks on *assistant GPUs*, enabling faithful large-scale behavior with only a few GPUs.

For virtual ranks, CrystalLLM initially instantiates only the subset of them that directly communicate with sandbox GPUs (detailed in §5.2). Once initialized, these ranks participate in replay. During replay, virtual ranks do not perform real computation. Instead, they traverse the pre-collected execution graph using its timing and dependency information, assisting sandbox GPUs in completing communication. The graph contains both computation and communication nodes. When reaching a computation node (e.g., microbatch #4), a virtual rank waits for the recorded duration. When reaching a communication node (e.g., send #3), it performs the actual communication with sandbox ranks.

To support this process, CrystalLLM manages replayed tensors through a runtime prefetch pipeline. The tensor contents come from communication data recorded during graph collection or injected as user-defined communication inputs. Tensors required for upcoming collective operations are prefetched during idle communication periods to avoid contention. We prioritize loading tensors directly from disk into a GPU buffer pool; if GPU memory is full, tensors are staged in a CPU buffer pool until space becomes available. Together, these buffers act as staging caches.

In practice, the GPU buffer pool is typically sufficient. For example, in a 512-GPU job with a single assistant GPU machine, each GPU provides over 100 GB of buffer space for replay (excluding CUDA context usage such as NCCL); with smaller-memory GPUs, the same aggregate buffer instead comes from more assistant GPUs. Moreover, only tensors involved in communication between sandbox ranks and active virtual ranks need to be managed, significantly reducing buffering requirements.

As a result, when a communication node is reached, the required tensor is already resident on the GPU, fulfilling the timely data supply assumption (§3.1): prefetch (PCIe) bandwidth outpaces the network bandwidth that consumes the data. After communication completes, the tensor is evicted. This design allows a single GPU to efficiently serve multiple virtual ranks and communication groups on demand.

5.2 Virtual Ranks Initialization

Virtual ranks must be placed on assistant GPUs to interact with sandbox GPUs, but hosting thousands of them incurs substantial overhead. This overhead primarily stems from two factors. First, large-scale training creates numerous communication groups (e.g., DP, TP, PP). Each virtual rank requires its own communicator and buffer (e.g., 500 MB per group), which can quickly exhaust GPU memory. Second, initializing thousands of these groups on a few physical GPUs causes severe contention for shared resources like

CUDA contexts and driver locks, turning initialization into a partially serialized process that can take hours.

To address this, CrystalLLM introduces *NCCL group reduction*, which reduces overhead by selectively instantiating only the necessary groups and ranks. During group creation, sandbox ranks remain unchanged and are initialized normally. For virtual ranks, we instantiate only the NCCL groups whose members overlap with sandbox ranks. Groups that do not communicate with the sandbox are bypassed at the PyTorch `c10d` layer and are never instantiated, significantly reducing the total number of active groups. For example, when emulating 2,000 virtual ranks, we reduce the number of active NCCL groups from 8,617 to 82.

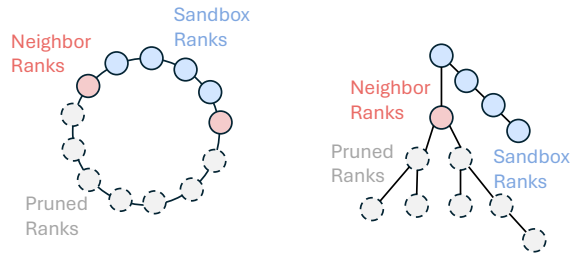
While reducing the number of NCCL groups helps, instantiating all members within each group remains unnecessary and can be further reduced. As shown in Figure 5, direct communication between the ranks of interest and virtual ranks occurs only among neighboring ranks, depending on the collective algorithm (e.g., ring or tree). Therefore, we can safely prune non-neighboring virtual ranks as long as we preserve numerical correctness. This reduces both initialization cost and runtime communication overhead.

Concretely, when all members of a group enter communicator initialization (e.g., `init_process_group`), they participate in a `TCPStore`-based barrier, where each rank increments a counter and waits until all expected ranks have joined. However, since we instantiate only neighboring ranks, the number of active ranks no longer matches the expected group size. To address this, we designate a leader among the assistant ranks to additionally issue requests to `TCPStore` on behalf of all non-neighbor ranks. This effectively accounts for the missing participants and allows the initialization barrier to complete without modifying the world size or user-level code. Furthermore, during NCCL communicator initialization, each machine typically explores its intra-node topology to identify its logical neighbors. We hijack the NCCL initialization control channel to manage this information exchange. Because the emulated cluster is homogeneous, a leader—again chosen among the neighbor ranks—can seamlessly synthesize and exchange the correct intra-node topological data on behalf of all pruned ranks. This effectively establishes the correct NCCL topology and communication channels, successfully virtualizing the backend without the sandbox noticing any missing peers.

5.3 Runtime Communication Pruning

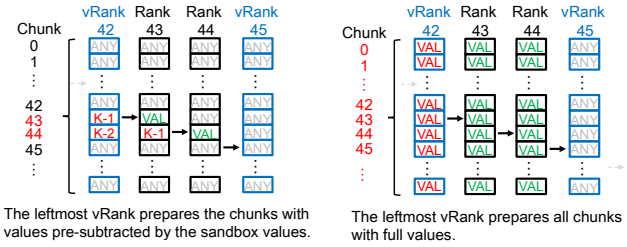
Since we remove inactive, non-neighboring ranks, the original collective communication pattern is changed at runtime (Figure 5). We must therefore ensure that the semantics of collective operations remain correct under this transformation; to this end, we modify NCCL's data transmission logic to guarantee numerical correctness.

Most NCCL collectives build on two primitive patterns: *reduction*, which aggregates values toward a destination,



(a) Ring algorithm. **(b)** Tree algorithm.

Figure 5. Runtime communication pruning: only vRanks adjacent to the sandbox on the ring (a) or tree (b) path participate.



(a) Reduce Stage **(b)** Broadcast Stage

Figure 6. Collective numerical correctness under pruning (ring all-reduce; sandbox Ranks 43–44, neighboring vRanks 42/45, pruned non-neighbors omitted). Each row is a chunk. (a) Reduce: the leftmost vRank pre-subtracts sandbox contributions (K−1, K−2) so sandbox-owned chunks accumulate correct full values (VAL); other chunks carry arbitrary values (ANY). (b) Broadcast: the leftmost vRank supplies full values for all chunks.

and *broadcast*, which propagates values outward [20]; the remaining operations, such as all-to-all and point-to-point send/receive, move data directly without arithmetic. Ensuring correctness for reduction and broadcast therefore covers the collectives in our workloads.

We illustrate our approach using ring all-reduce, which consists of both *reduction* and *broadcast* stages; the same principles apply to other collectives. In the reduction stage, each rank splits its data into multiple chunks and is responsible for aggregating one chunk (e.g., sum, max, or min) from all other ranks. Let $\text{data}_j^{(i)}$ denote the contribution of rank j to chunk i, and let $\text{data}_{\text{full}}^{(i)} = \sum_j \text{data}_j^{(i)}$ denote the fully reduced value. The key property is that each rank only needs to ensure the correctness of its assigned chunk. After one full ring round of $(K-1)$ communication steps, where K is the number of ranks in the ring, each rank holds $\text{data}_{\text{full}}^{(i)}$ for its assigned chunk i. In the broadcast stage, each rank then propagates its completed chunk to all other ranks through another ring round $(K-1)$, ensuring that all ranks obtain the full result.

Figure 6 illustrates how CrystalLLM preserves collective correctness under rank pruning. The example includes neighboring virtual ranks (e.g., *vRank* 42 and *vRank* 45) and sandbox ranks (e.g., *Rank* 43 and *Rank* 44); other pruned non-neighboring ranks are omitted for clarity. In the reduction stage, we leverage the key property of ring all-reduce: each rank is responsible only for its assigned chunk. Therefore,

after pruning, we only need to ensure that sandbox ranks compute their assigned chunks correctly. Once the ranks of interest (e.g., ranks 43 and 44) obtain the correct values for their chunks, i.e., $\text{data}_{\text{full}}^{(43)}$ and $\text{data}_{\text{full}}^{(44)}$, the state after the reduction stage is equivalent to that of the original execution, and the values of other chunks do not affect the outcome.

To achieve this, the leftmost virtual rank prepares adjusted values that compensate for missing contributions from pruned ranks. For each chunk, the goal is to ensure that the value received at the owning sandbox rank matches $\text{data}_{\text{full}}^{(i)}$. For the chunk owned by rank 44, since the ring path reaches rank 44 via rank 43, the leftmost virtual rank prepares $\text{data}_{\text{full}}^{(44)} - \text{data}_{43}^{(44)} - \text{data}_{44}^{(44)}$, so that as the data propagates, rank 43 adds back $\text{data}_{43}^{(44)}$ and rank 44 subsequently adds back $\text{data}_{44}^{(44)}$, reconstructing the correct final value $\text{data}_{\text{full}}^{(44)}$. Similarly, for other sandbox-owned chunks, we subtract the contributions of ranks along the remaining path before reaching the owner (including the owner itself). For chunks not owned by sandbox ranks, their values do not affect the final outcome observed in the sandbox and can therefore be assigned arbitrary values.

In the broadcast stage, each rank propagates its completed chunk to all other ranks. At this point, the leftmost virtual rank needs to provide correct final values for all other chunks not owned by the sandbox ranks. Since sandbox ranks (e.g., ranks 43 and 44) receive the fully reduced values $\text{data}_{\text{full}}^{(i)}$ for their respective chunks during this stage, ensuring correctness for these chunks is sufficient to guarantee overall correctness from the sandbox's perspective.

Tree-based algorithms [20] follow the same principle. NCCL builds hierarchical trees for inter-node communication, with intra-node ranks forming a chain behind a head rank that carries the node's inter-node traffic; correctness is again enforced at the boundary where a sandbox head rank meets a virtual rank. For a given data chunk, the sandbox head plays one of three roles: 1) *Root*: the chunk's final reduction completes here, so its child virtual rank sends a compensated value that, combined with the sandbox's own contribution, yields the correct result; the subsequent broadcast then propagates it downward. 2) *Leaf*: the data it sends upward is reduced elsewhere and does not affect the sandbox, so correctness relies on its parent virtual rank returning the correct final value during broadcast. 3) *Intermediate*: its local reduction is later overwritten by the broadcast, so its child virtual rank can send arbitrary values while its parent supplies the correct final value. As with the ring, these boundary rules preserve numerical correctness while making broad use of arbitrary values.

Beyond numerical correctness, pruning introduces no nondeterminism: sandbox NCCL executes the same number of transmissions with the same volume as the original collective, and the placement of sandbox and virtual ranks is static across runs. Repeated emulations therefore differ only by inherent hardware jitter, comparable to the run-to-run variance of real executions (§7.3).

6 IMPLEMENTATION

CrystalLLM first collects the execution graph using a *centralized program coordinator*, and reuses it to enable *virtual rank replayers* for slice timing collection and emulation.

We implement a coordinator that orchestrates all logical ranks during graph collection. It intercepts unmodified training programs and remaps logical ranks onto a limited set of GPUs via context switching. We patch the PyTorch `c10d` layer to intercept communication and enable CPU-side collective execution without requiring all participants to be active. A CPU collective executor, built on `torch.distributed`, processes inputs once all required tensors are available and produces outputs that are stored and reused when ranks resume, ensuring correctness without full concurrency.

The replayer serves both slice-level timing collection and final emulation. The key difference lies in how the execution graph is used. During slice execution, virtual ranks replay only the graph structure without accurate timing, as this does not affect the timing of real ranks within each slice; any drift is corrected during global calibration. In contrast, during final emulation, virtual ranks execute the calibrated execution graph with complete timing information.

We also extend NCCL to support communication pruning. At initialization, virtual ranks embed a magic number in their `host_hash`, so that ranks recognize each other's identity when exchanging connection information. During execution, if a channel involves no neighbor of sandbox ranks (i.e., not on a path to sandbox ranks), NCCL skips the data transfer and instead generates completion metadata to satisfy the collective, reducing overhead while preserving correctness.

7 EVALUATION

We evaluate CrystalLLM across four dimensions. First, end-to-end results (§7.1) show that CrystalLLM achieves high-fidelity predictions for both iteration time and memory allocation, with average errors of 0.58% and below 0.01%, respectively. Second, we evaluate emulation efficiency and scalability across target configurations (§7.2). Third, we conduct microbenchmarks to assess execution fidelity and quantify the benefits of our key optimizations—initialization acceleration via rank virtualization, transmission reduction via communication pruning, and timing alignment via inter-slice calibration (§7.3). Finally, we compare CrystalLLM with state-of-the-art simulators, including Phantora [33] and SimAI [42], showing that CrystalLLM achieves higher estimation accuracy for complex training workloads (§7.4).

Experimental setup. We conduct experiments on a 2,048-GPU testbed (8 GPUs per node) using the open-source Megatron-LM implementation of Qwen 3 MoE pretraining [1]. We evaluate three model configurations (M.1–M.3) spanning 235B to 1.01T parameters (Table 1).

Table 1. Model structures evaluated.

Model	Params	Layers	Heads	Experts (TopK)
M.1	235B-A22B	94	64	128 (8)
M.2	503B-A20B	62	32	256 (8)
M.3	1.01T-A43B	62	64	256 (8)

Table 2. Parallelization strategies evaluated (Megatron distributed optimizer enabled in all setups).

Strategy	TP	PP	VPP	EP	GA
A	1	4	0	8	8
B	2	4	2	8	16
C	1	16	0	8	32
D	1	8	0	16	16

**TP: Tensor Parallel, PP: Pipeline Parallel, VPP: Virtual Pipeline Parallel (number of virtual stages per pipeline stage; 0 = disabled), EP: Expert Parallel, GA: Gradient Accumulation.*

Table 3. Emulation footprint per target scale (S = sandbox nodes, A = assistant nodes, 8 GPUs each).

Target	M.1/M.2 (all strat.)	M.3-S.C (EP=8)	M.3-S.D (EP=16)
512	1S+1A	1S+1A	2S+2A
1,024	1S+2A	1S+2A	2S+4A
2,048	1S+3A	1S+3A	2S+4A

To cover diverse parallelization strategies under memory constraints, we vary tensor parallel (TP), pipeline parallel (PP), and expert parallel (EP) sizes and gradient accumulation steps, yielding four configurations (S.A–S.D, Table 2). Table 3 lists the emulation footprint used for each target scale, i.e., the number of physical nodes CrystalLLM actually occupies.

7.1 End-to-end Prediction Accuracy

End-to-end iteration time estimation. We evaluate three cluster scales (512, 1,024, and 2,048 GPUs), each with three model configurations and two parallelization strategies. Iteration time is the framework-reported duration of one step, including computation and communication. Each test runs the pretraining code at the full target scale for 85 iterations, and we report the median of the middle 70, excluding warm-up. CrystalLLM emulates the same workload with identical configurations and scripts. Emulation footprints follow Table 3. Simulators have inherent limits on these workloads (e.g., missing memory-allocation simulation or PP and MoE support), so we compare against them separately on the configurations they support (§7.4); downscaled runs are no substitute either, as shrinking the DP degree changes distributed-optimizer sharding and triggers out-of-memory (OOM).

As shown in Figure 7, CrystalLLM tracks the baseline across all scales, with an average error of 0.58% and a maximum of 1.98% under diverse parallelization strategies. The use of actual GPUs for task execution, coupled with the high-fidelity execution graph, empowers CrystalLLM to achieve highly accurate latency estimations.

End-to-end peak memory allocation estimation. We evaluate peak memory at the same three scales using PyTorch's `max_memory_allocated`. Each baseline has 10 warm-up iterations, followed by 3–10 measured iterations.

For MoE models, memory usage varies across devices due to uneven expert workloads after dispatch. We evaluate two cases: (1) *Balanced*, using Megatron Core to ensure uniform distribution, and (2) *Imbalanced*, where we inject precomputed skewed logits via our MoE mock router (Appendix §B) to create non-uniform workloads.

Figure 8 shows that CrystalLLM estimates peak memory with errors consistently below 0.01%. Furthermore, CrystalLLM successfully reproduces OOM errors encountered in the baseline experiments. CrystalLLM achieves high-fidelity memory estimation by accounting for consumption across the entire software stack, ensuring that its peak memory predictions are both comprehensive and accurate.

7.2 System Efficiency

We evaluate CrystalLLM's physical resource requirements and end-to-end emulation time across scales. Driven by the common practice of expanding the PP degree to accommodate larger model states, we scale our M.2 workload from 512 to 8,192 GPUs. Specifically, we proportionally increase the PP size from 4 to 64—the deepest pipeline the model supports—while holding the DP size constant.

To avoid linear overhead as PP grows, CrystalLLM parallelizes emulation by scaling Assistant Nodes with the target scale (red line in Figure 9). A 1:1 Assistant-to-Sandbox Node ratio enables concurrent profiling of pipeline stages. For example, a 512-GPU target (PP=4) requires four sequential steps on one Assistant Node; a 1,024-GPU target (PP=8) retains four steps with two Assistant Nodes, keeping emulation time comparable to the baseline.

This strategy keeps the end-to-end emulation time highly stable (under 80 minutes) up to 8,192 GPUs; graph collection takes only about 4 minutes, constant across scales, since proportionally scaled assistant nodes keep the number of collection rounds unchanged (PP=4 on one node and PP=8 on two nodes both take 4 rounds). Without proportional scaling, a 1,024-GPU ablation with one Assistant Node (second bar) must process all 8 stages sequentially, increasing the time by about 1.6× (125 minutes)—this is how scaling breaks down: as linear time inflation rather than memory or compute saturation. Crucially, CrystalLLM maintains massive efficiency: emulation proceeds at the same per-iteration wall-clock time as a real run (§7.1), yet emulating an 8,192-GPU cluster requires only 32 physical nodes in total (16 Assistant, 16 Sandbox), utilizing $< 0.4\%$ of the target GPUs and achieving $> 99\%$ resource savings. In the implementation of hybrid emulation, CrystalLLM mitigates the overhead associated with redundant virtual rank initialization and inter-rank communication, thereby ensuring high system efficiency.

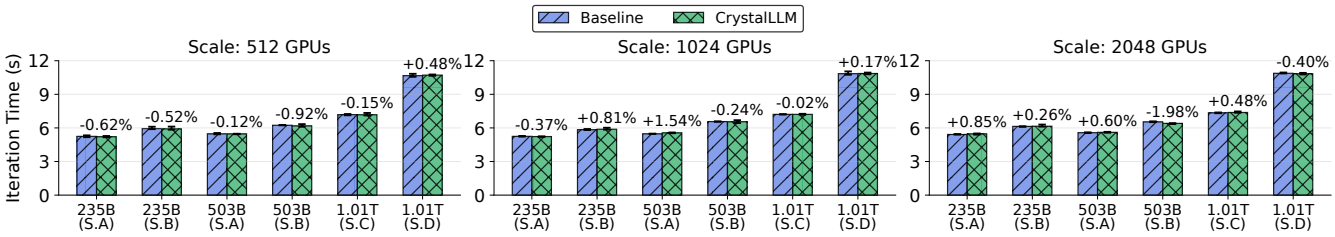
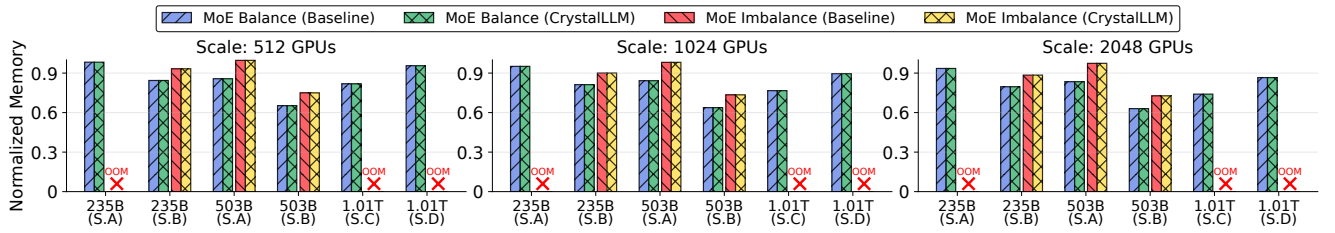


Figure 7. End-to-end iteration time estimation results.



For imbalanced cases, the MoE mock router (Appendix §B) metrics are `br_min`: 0.71, `br_max`: 2.16, `br_avg`: 1.48, `br_std`: 0.37, `br_med`: 1.38, `br_skew`: 0.90.

Figure 8. End-to-end peak memory allocation estimation results. All estimation errors are consistently below 0.01%.

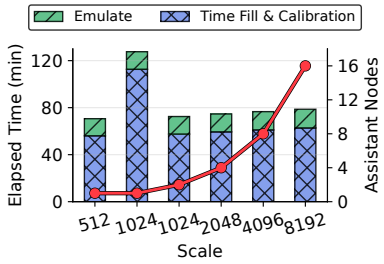


Figure 9. Emulation time breakdown.

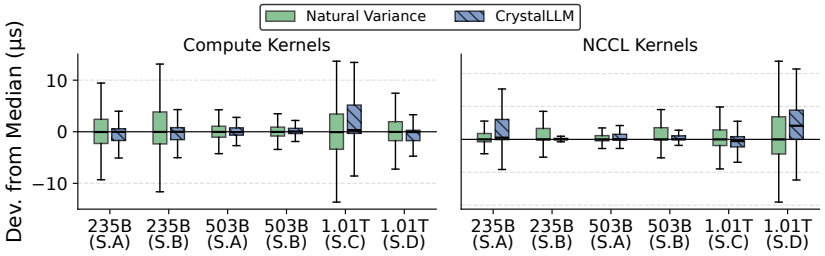


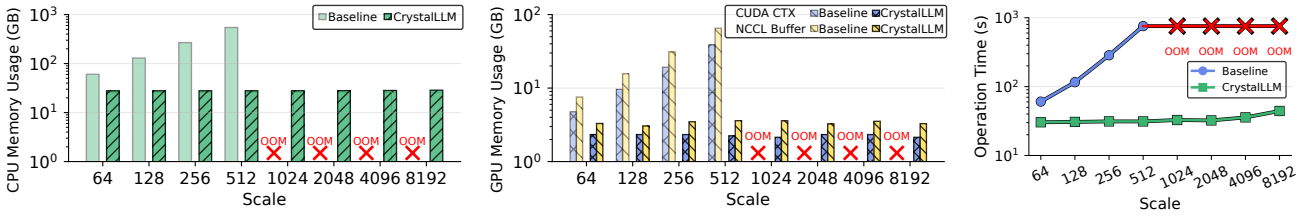
Figure 10. Kernel duration deviation from the baseline median under the 1024-rank scale.

7.3 Micro benchmarks

Execution fidelity. To validate the execution accuracy inside CrystalLLM, we compare its PyTorch Profiler traces against training baselines. We perform a fine-grained, kernel-level comparison using the absolute deviation in kernel duration. For the training baseline, we sample 32 ranks within the same DP group and calculate the median execution duration of each kernel. The inherent hardware jitter—the deviation of actual execution times from this median—is represented by the *Natural Variance* boxes. Then, we compare the emulated kernel durations against the corresponding baseline medians, with the resulting errors shown by the *CrystalLLM* boxes. As shown in Figure 10, for all configurations under the scale of 1024 ranks, the emulated variance falls within the range of natural hardware variance. Beyond duration, we evaluate scheduling fidelity via relative kernel start times (normalized by total iteration time). Figure 12 visualizes the 1024-rank 503B (S.B) case, plotting the natural variance across 32 baseline ranks (gray) against CrystalLLM's deviation from the baseline median (blue). CrystalLLM tightly tracks the median, bounded within the ±0.5% natural hardware jitter. Across all configurations, the maximum observed start time error is 3.64%. This kernel-level fidelity underpins the accurate prediction of configuration changes that alter the kernel composition, such as disabling Flash Attention (Table 4).

Virtual rank initialization optimization. To enable efficient large-scale emulation, CrystalLLM features a customized initialization mechanism. We evaluate its performance by measuring the time and memory required to initialize and execute a global barrier operation across varying cluster scales (64 to 8192 ranks). The testbed consists of two 8-GPU nodes: one acts as the sandbox (running one real rank per GPU), while the other hosts all virtualized ranks. We compare CrystalLLM against a baseline that uses standard NCCL (via shared `NCCL_HOSTID`), where each virtual rank maintains an independent process and CUDA context.

As shown in Figures 11a and 11b, the baseline incurs linearly scaling overheads. At merely 512 ranks, initializing CUDA contexts, establishing communication channels, and allocating NCCL buffers consumes 544.5 GB of CPU memory and 103.7 GB of GPU memory. Consequently, initialization takes 756.0 seconds to complete, and the baseline inevitably encounters OOM errors at scales ≥ 1024 ranks. In contrast, CrystalLLM maintains a strictly *constant* resource footprint regardless of the logical cluster size. By strategically allocating resources only for "active" virtual ranks (i.e., direct topological neighbors of the sandbox nodes), CrystalLLM eliminates massive redundant contexts and buffers. Consequently, CrystalLLM successfully initializes an 8192-rank emulation in just 44.3 seconds, occupying a mere 5.4 GB of GPU memory per physical device.



(a) Overall CPU memory overhead. **(b)** GPU memory overhead per rank. **(c)** Completion time of one barrier.

Figure 11. Memory usage reduction and initialization acceleration of large-scale emulation in CrystalLLM.

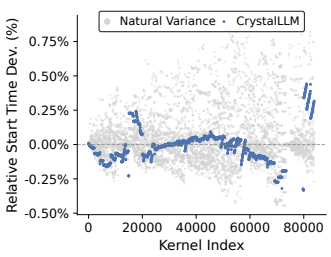


Figure 12. Kernel launch deviation.

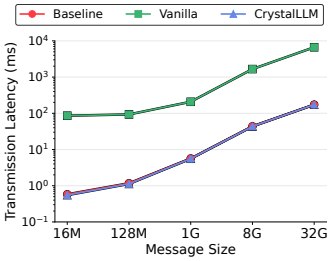


Figure 13. Latency overhead.

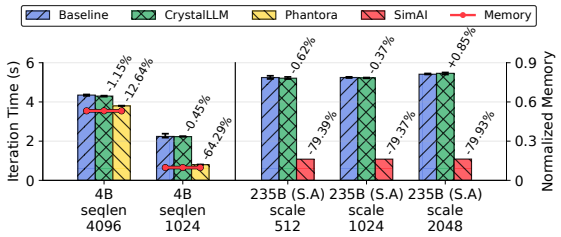


Figure 14. CrystalLLM compared with Simulators.

Virtual rank communication pruning. While the optimization above minimizes *initialization* overhead, CrystalLLM must also mitigate *runtime* resource contention during collective operations. Using the same two-node testbed, we evaluate AllReduce execution. In the Vanilla solution, virtual ranks blindly replay standard NCCL kernels, intensely contending with sandbox ranks for Streaming Multiprocessors and memory/PCIe bandwidth, thereby skewing execution timing. To ensure high fidelity, CrystalLLM prunes redundant data transmissions among virtual ranks, preserving only cross-node RDMA traffic. Figure 13 shows the transmission latency across varying message sizes at a 32-rank scale. Without pruning, the Vanilla approach suffers up to a 148× latency inflation. By applying our pruning techniques, CrystalLLM closely tracks the physical baseline, yielding a maximum error of 0.2% against real executions. Further evaluations up to 128 ranks are in Appendix §C.1.

Inter-slice calibration. To demonstrate the effectiveness of inter-slice timing calibration, we chose the 235B model in strategy S.B and replayed the graph before and after inter-slice calibration. The emulated iteration time before inter-slice calibration quickly drops from 5.7s to 5.13s, demonstrating that without proper alignment, the emulation error can easily go higher than 10%.

7.4 Compared with Simulators

Phantora. Since Phantora [33] does not yet support pipeline model parallelism (PP) or MoE models, we evaluate CrystalLLM against it on small-scale tasks within its supported configurations. We assess their accuracy in estimating iteration time and peak memory allocation under identical model and parallelization configurations. The setup is tailored to a 32-GPU testbed using Qwen 3 4B configuration, with the TP size set to 4 to ensure a fair comparison.

As shown in Figure 14, CrystalLLM outperforms Phantora in both metrics. Notably, while Phantora shows a 12% error in typical cases, its prediction fidelity drops significantly as the sequence length decreases—with the error rising to 64% as the GPU-side computational pressure is decreased. This discrepancy primarily stems from two factors: (1) Phantora approximates actual kernel duration using raw network bandwidth, failing to account for the complex, multi-step nature of collective kernels; and (2) Phantora launches multiple mock processes on CPU to represent individual training ranks, advancing the simulation via inter-process communication with a central simulator. Due to the CPU contention among multiple processes, Phantora could not accurately capture the impact of CPU-side Python interpreter latency on the overall execution duration. As a result, Phantora defaults to omitting these latency components, leading to a systematic underestimation of the total iteration time.

SimAI. To further evaluate the performance of simulators in large-scale MoE training scenarios, we feed the model configurations and parallelization strategies into SimAI [42] and compare its results against CrystalLLM. Notably, SimAI does not support memory allocation simulation and is therefore excluded from the memory-related comparison.

Figure 14 presents the results for the 235B (M.1) model with strategy S.A at different scales; comprehensive results for other configurations are deferred to Appendix §C.2. SimAI deviates substantially from the baseline, with an average error of 77.2%. We attribute this discrepancy to two primary factors. First, while SimAI relies on workload files to describe the training process, the workload-file design fails to capture the inter-rank dependencies and asymmetric behaviors inherent in pipeline model parallel (PP), omitting the communication bubbles associated with PP. Second, SimAI neglects the unique characteristics of MoE models compared to dense ones, ignoring the computational overheads of MoE operations such as gating, permute, dispatch, and combine. These findings underscore

Table 4. Emulation accuracy under different training configurations.

Optimization	Iteration Time (s)		Peak Memory (GB)	
	Baseline	CrystalLLM	Baseline	CrystalLLM
Baseline	5.62 (+0.29/−0.08)	5.70	133.76	133.76
Flash Attention Off	7.74 (+0.24/−0.08)	7.70	135.22	135.22
P2P Overlap Off	5.71 (+0.38/−0.09)	5.82	133.74	133.74
Offload Optimizer	11.98 (+0.51/−0.24)	11.61	112.07	112.07
Recompute	7.07 (+0.37/−0.08)	7.21	98.13	98.13

that simulators often require manual effort to maintain and update simulation models for evolving architectures.

8 USE CASES AND EXPERIENCE

Our production use cases build on two properties of CrystalLLM. First, it faithfully reproduces the *normal* (average) behavior of a training job at scale; this reproduced behavior serves both as the metric for comparing configurations and as the expected baseline against which anomalies stand out. Second, the reproduced environment is isolated from production and repeatable, so heavyweight inspection that production prohibits can be applied freely. The cases below specialize these two properties.

Tuning training configurations. A training configuration is, in essence, an allocation of system resources—SMs, HBM, PCIe, NVLink, host CPU and memory, and the network—which frameworks typically expose through parallelism dimensions (e.g., DP, TP, PP, EP) and memory-strategy interfaces (e.g., recomputation, offloading). Tuning it in production is hard for three reasons. The space is large: several parallelism dimensions and more than a dozen memory knobs interact non-linearly—a raw cross product beyond 10^5 combinations, of which thousands remain viable—while analytical estimates drift as kernels fuse or allocator behavior changes. Trials are expensive: at current model sizes, the smallest complete run takes more than a hundred GPUs—close to an infrastructure team's entire development quota, often forcing downscaled experiments—and one full test (real data, checkpoint loading, reaching steady state) takes 20–30 minutes, so a manual tuning round costs about a day and lands on the optimum within the engineer's experience. The optimum keeps moving: for multimodal workloads, sequence lengths vary by over 10× across training stages, each lasting days to weeks, leaving a narrow tuning window per stage.

CrystalLLM counters each: it searches parallelism and memory configurations at production fidelity (Table 4), surpassing hand-tuned configurations by 3–5% MFU in our practice; it occupies only 16–32 GPUs, fitting the idle fragments of a test cluster; and since memory strategies and sequence lengths rarely change cross-machine communication, one collected graph served 90 strategies in one search. During the search, the sandbox densely samples resource telemetry (see profiling below); when attribution is needed, engineers hold the remaining dimensions fixed and vary a single one, validating each change in isolation instead of waiting for the next full-scale run, where many changes land together and their interactions confound attribution. Such attributions accumulate into reusable knowledge that feeds back into system and model design.

Memory analysis. Memory behavior—allocator reuse, fragmentation, peak usage—emerges from the allocator's interaction with the exact tensor sizes, orders, and lifetimes of a real run, and is hard to predict analytically. CrystalLLM reproduces this behavior in an isolated, repeatable environment, where engineers can take memory snapshots at will. Three cases from our deployment.

MoE memory estimation: token routing is dynamic and shifts across training stages; early-stage imbalance can lengthen iterations by over 5% and swing memory by up to 20 GB, so aggressive memory configurations risk OOM while conservative ones waste GPUs. CrystalLLM allows engineers to plug in a mock router that replays production-observed imbalance features (Appendix §B) and estimates the dynamic memory accurately (Figure 8).

Fragmentation attribution: after adopting a new attention kernel, the maximum trainable sequence length unexpectedly dropped by 13%; snapshots showed that activations became fine-grained, and small tensors occupied segments that previously held large contiguous activations, lowering reuse and raising the reserved watermark by 9% of the per-rank GPU memory.

Memory bugs: in one code version, temporary tensors were not released in time and the reserved-minus-allocated gap grew abnormally; snapshots pinpointed the leak, and the fix shrank the gap from 19% to 8% of the per-rank GPU memory.

Targeted cluster health checks. Gray failures are notoriously hard to pinpoint: benchmark suites [45] hardly cover the full software stack and its corner cases. In one incident, a job launched with an iteration time of 6.38s, against an expected median of 5.63s drawn from historical monitoring—14% slower. The infrastructure team collected the execution graph offline (§4) as the normal behavior, then screened the cluster with CrystalLLM before relaunching the job: machines were paired, one node as sandbox running the real training code and the other as assistant, swapping roles afterwards—each pass costing about 20 minutes.

One sandbox node reported 6.40s versus a 5.70s median across the other pairs; inspection revealed thermal throttling that down-clocked its GPUs to 900MHz. The job recovered after the node was taken offline. Generic suites miss this issue because it manifests only under the target job's sustained execution mix. Thus, rare behaviors are localized as deviations from the reproduced normal baseline rather than emulated directly. CrystalLLM remains the faithful, reproducible environment; its faster fault localization suggests a broader class of applications.

Fine-grained profiling and tool development. Heavyweight or immature observability tools are often prohibited online due to their overhead, pressure on the logging system, or risk of their own bugs. CrystalLLM's isolated environment lifts these constraints: all results in this paper were collected

with unmodified `nvidia-smi`, PyTorch Profiler, and NVIDIA Nsight Systems. The same environment also supports building new tools: in one case, we developed tracing tools on CrystalLLM. Debugging concentrated on injection: CUDA version incompatibilities, and the tuning of hook points and trigger conditions. Downscaled runs could reproduce some of these issues, but the hooks fire only on code paths exercised under the production configuration (e.g., specific sequence-length ranges), so the most stable way to iterate was against the unmodified configuration—which CrystalLLM provides on a few GPUs.

9 RELATED WORK

Answering pretraining questions with fewer GPUs. Planning, tuning, and debugging large-scale training all call for tools that consume far fewer GPUs than the target cluster, but they tolerate different levels of abstraction. Simulators answer questions from a *description* of the system: analytical ones [35, 44] need only operator counts, bandwidths, and topology—usable before any hardware exists—while profiling-driven ones [2, 14, 19, 26, 42, 51] replay kernel timings measured on real GPUs inside a model. At near-zero per-configuration cost, they are the right tool for sweeping design spaces and ranking candidates.

A method can answer only what its input encodes: allocator fragmentation, SM scheduling, and interpreter latency have no counterparts in a description; modeling them faithfully amounts to reimplementing the system (§2.2). Reproducing these behaviors requires the *artifact* itself—real code on real hardware. Downscaled observations may not transfer back because the configuration changes (§2.3); full-scale runs require the target GPU count. CrystalLLM preserves the runnable code and target hardware model, giving up only quantity—the physical concurrency of all ranks: enough to determine whether this stack at scale meets its iteration-time target, fits in memory, and reproduces production failures.

Emulating scale in other domains. Emulating a large system on few machines has a long history; each lineage gives up quantity through a different mechanism. *Time dilation* [16, 17] slows every node's clock so that the same hardware appears to offer $N\times$ the bandwidth or compute—in that setting, scale is a rate that a clock can fake. In pretraining, scale is not a rate: a larger job changes the parallelism configuration and every rank's execution graph, which no rescaling of kernel durations reproduces.

Network emulation [12, 40, 43] multiplexes many lightweight endpoints per machine and synthesizes the fabric between them—the endpoints still execute the real application. A faithfully executing rank cannot be made cheap this way: multiplexed onto one GPU, its kernels no longer run at native speed. CrystalLLM instead stops executing the other ranks and replays them, which works because DP replicas are statistically symmetric and the communication schedule is static; datacenter traffic, emerging from all flows jointly, offers no such representative subset.

Boundary emulation: CrystalNet [25] boots real switch firmware within a change's scope and mocks the rest—the principle behind CrystalLLM's neighbor-rank pruning (§5.3); FireSim [21] and SimBricks [22] push this to cycle level. But each mocked device acts from its own static snapshot, so the surroundings scale by replication; a virtual rank's actions—what to send, and when—follow from the dependency structure of the entire job at target scale, which is why CrystalLLM must first capture the execution graph (Phase 1).

Learned approximation: MimicNet [48] simulates one cluster at full fidelity and approximates the rest with models trained on it; CrystalLLM replaces the approximation with execution—virtual ranks send NCCL traffic at full volume—so a refresh is a minute-level re-collection, not retraining.

Record-and-replay [10, 11, 15] re-injects recorded inputs to reproduce causality at arbitrary speed; CrystalLLM likewise re-injects communication tensors and builds on GPU checkpointing [4, 30], but timing itself is the observable here, so durations are replayed exactly and content is recorded only at the sandbox boundary.

Emulating LLM training. The closest systems differ in how much of the real artifact survives into the loop. Phantora [33] runs the unmodified program on mock CUDA and NCCL backends; execution stops being real at the backend, which explains its growing error as GPU-side pressure drops (§7.4). Echo [9] collects traces on a single device much as Phase 1 does, but replays them into a simulator with modeled communication. Chakra [39] and Mystique [23] replay traces as standalone benchmarks rather than sustaining full-scale interaction around real ranks. CrystalLLM keeps the real program, real GPUs, and real NCCL end to end, replacing only remote ranks' computation with measured durations.

10 CONCLUSION

We present CrystalLLM, a system that faithfully emulates large-scale LLM training using a small fraction of its hardware. By capturing a high-fidelity execution graph on a few GPUs and replaying it around the ranks of interest, CrystalLLM reproduces deployment-scale performance and memory behavior while running unmodified training code. In production, it has turned experimentation that once required a dedicated cluster into a routine that fits idle test nodes. We are extending CrystalLLM toward broader training frameworks and more asymmetric workloads, and we release the system to support future research and industry adoption.

ACKNOWLEDGMENTS

We thank our shepherd and the anonymous SOSP reviewers for their insightful comments and suggestions, which significantly improved this paper. We sincerely thank the Wan pretraining infrastructure team for their invaluable feedback on CrystalLLM.

REFERENCES

[1] 2025. Pai-Megatron-Patch Github Repository. https://github.com/alibaba/Pai-Megatron-Patch. (2025).

[2] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2024. vTrain: A Simulation Framework for Evaluating Cost-Effective and Compute-Optimal Large Language Model Training. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*. IEEE Press, 153–167. https://doi.org/10.1109/MICRO61859.2024.00021

[3] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, et al. 2024. FLUX: Fast Software-based Communication Overlap On GPUs Through Kernel Fusion. (2024). arXiv:cs.LG/2406.06858

[4] CRIU Project Developers. 2026. Github - CRIU: Checkpoint/Restore In Userspace. https://github.com/checkpoint-restore/criu. (2026).

[5] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. (2023). arXiv:cs.LG/2307.08691 https://arxiv.org/abs/2307.08691

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[7] DeepSeek-AI. 2025. DeepSeek-V3 Technical Report. (2025). arXiv:cs.CL/2412.19437 https://arxiv.org/abs/2412.19437

[8] Epoch AI. 2025. "Data on AI Models". (7 2025). https://epoch.ai/data/ai-models/ Accessed: 13 Mar 2026.

[9] Yicheng Feng, Kin Hang Sew, Yuetao Chen, Kaiwen Chen, Jingzong Li, Tianyuan Wu, Zheng Zhou, Peng Cheng, Chuan Wu, Wei Wang, Tsung-Yi Ho, and Hong Xu. 2026. Scalable and Efficient Simulation of LLM Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 26)*.

[10] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2007. Friday: Global Comprehension for Distributed Replay. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*. USENIX Association.

[11] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications. In *2006 USENIX Annual Technical Conference (USENIX ATC 06)*. USENIX Association.

[12] Paulo Gouveia, Jo ao Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, et al. 2020. Kollaps: Decentralized and Dynamic Topology Emulation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys 20)*. Association for Computing Machinery.

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. 2024. The Llama 3 Herd of Models. (2024). arXiv:cs.AI/2407.21783 https://arxiv.org/abs/2407.21783

[14] Fei Gui, Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Ran Zhang, Hongbing Yang, and Dian Xiong. 2025. Accelerating design space exploration for LLM training systems with multi-experiment parallel simulation. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*. USENIX Association, USA, Article 25, 16 pages.

[15] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, et al. 2008. R2: An Application-Level Kernel for Record and Replay. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. USENIX Association.

[16] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. 2008. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association.

[17] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2006. To Infinity and Beyond: Time-Warped Network Emulation. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 06)*. USENIX Association.

[18] Marius Hobbhahn, Lennart Heim, and Gökçe Aydos. 2023. Trends in machine learning hardware. https://epoch.ai/publications/trends-in-machine-learning-hardware. (Nov. 2023). Published online at epoch.ai. Accessed 2026-08-13.

[19] 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. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 623–637. https://proceedings.mlsys.org/paper_files/paper/2022/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf

[20] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. 2026. Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms. (2026). arXiv:cs.DC/2507.04786 https://arxiv.org/abs/2507.04786

[21] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, et al. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA 18)*. IEEE Press.

[22] Hejing Li, Jialin Li, and Antoine Kaufmann. 2022. SimBricks: End-to-End Network System Evaluation with Modular Simulation. In *Proceedings of the ACM SIGCOMM 2022 Conference*. Association for Computing Machinery.

[23] Mingyu Liang, Wenyin Fu, Louis Feng, Zhongyi Lin, Pavani Panakanti, Shengbao Zheng, et al. 2023. Mystique: Enabling Accurate and Scalable Generation of Production AI Benchmarks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA 23)*. Association for Computing Machinery.

[24] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, et al. 2025. Understanding stragglers in large model training using what-if analysis. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation (OSDI '25)*. USENIX Association, USA, Article 27, 16 pages.

[25] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, et al. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)*. Association for Computing Machinery.

[26] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, 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 (CF '23)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/3587135.3592200

[27] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, et al. 2025. Astral: A Datacenter Infrastructure for Large Language Model Training at Scale. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 609–625. https://doi.org/10.1145/3718958.3750521

[28] Meta. 2025. Holistic Trace Analysis. (2025). https://github.com/facebookresearch/HolisticTraceAnalysis GitHub repository, latest release May 28, 2025.

[29] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, et al. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance*

Computing, Networking, Storage and Analysis (SC '21). Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[30] NVIDIA Corporation. 2025. Github - CUDA Checkpoint and Restore Utility. https://github.com/NVIDIA/cuda-checkpoint. (2025).

[31] OpenAI. 2024. GPT-4 Technical Report. (2024). arXiv:cs.CL/2303.08774 https://arxiv.org/abs/2303.08774

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[33] Jianxing Qin, Jingrong Chen, Xinhao Kong, Yongji Wu, Tianjun Yuan, Liang Luo, et al. 2026. Phantora: Maximizing Code Reuse in Simulation-based Machine Learning System Performance Estimation. In *NSDI '26*.

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

[35] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 81–92. https://doi.org/10.1109/ISPASS48437.2020.00018

[36] 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 (KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[37] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. (2024). arXiv:cs.LG/2407.08608 https://arxiv.org/abs/2407.08608

[38] 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).

[39] Srinivas Sridharan, Taekyung Heo, Louis Feng, Zhaodong Wang, Matt Bergeron, Wenyin Fu, et al. 2023. Chakra: Advancing performance benchmarking and co-design using standardized execution traces. *arXiv preprint arXiv:2305.14516* (2023).

[40] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, et al. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 02)*. USENIX Association.

[41] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, et al. 2025. Robust LLM Training Infrastructure at ByteDance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 186–203. https://doi.org/10.1145/3731569.3764838

[42] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, et al. 2025. SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large Language Model Training with Scalability and Precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 541–558. https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-simai

[43] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, et al. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 02)*. USENIX Association.

[44] 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)*. 283–294. https://doi.org/10.1109/ISPASS57527.2023.00035

[45] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, et al. 2024. SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association.

[46] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, et al. 2025. Qwen3 Technical Report. (2025). arXiv:cs.CL/2505.09388 https://arxiv.org/abs/2505.09388

[47] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. 2026. FlashAttention-4: Algorithm and Kernel Pipelining Co-Design for Asymmetric Hardware Scaling. (2026). arXiv:cs.CL/2603.05451 https://arxiv.org/abs/2603.05451

[48] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, Jo ao Sedoc, and Vincent Liu. 2021. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proceedings of the ACM SIGCOMM 2021 Conference*. Association for Computing Machinery.

[49] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, et al. 2025. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. (2025). arXiv:cs.DC/2502.19811

[50] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, et al. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. (2023). arXiv:cs.DC/2304.11277 https://arxiv.org/abs/2304.11277

[51] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 337–352. https://www.usenix.org/conference/atc20/presentation/zhu-hongyu

This appendix contains supplementary material that has not been peer-reviewed.

A COORDINATOR AND PRIORITY-BASED CONTEXT SWITCHING ALGORITHM

The coordinator employs a greedy scheduling strategy that maximizes system-wide progress by prioritizing processes with the highest potential for unblocking other waiting operations. Algorithm 1 presents the core switching logic.

When a process encounters a collective operation that cannot be executed immediately, the coordinator must decide which frozen process to activate on the same GPU. The selection process (lines 3-19) applies several filtering criteria to identify viable candidates. First, it excludes finished processes and those pinned to different GPUs (lines 7-10), ensuring only relevant processes compete for the target GPU. Most critically, it filters out processes whose head-of-line blocking operation is not ready (lines 11-12), as switching to such processes would yield no forward progress.

Among eligible candidates, the coordinator selects the process with the maximum number of pending operations (lines 13-16). This greedy heuristic is based on the insight that processes with more pending work are likely to unblock a larger number of dependent operations when given GPU time, thereby maximizing overall system throughput.

The main coordination loop (lines 21-34) handles incoming collective notifications by first updating the pending operation counts for all waiting ranks (line 23). If all participants in a collective operation are currently resident on GPUs, the operation executes directly without context switching (lines 24-25). Otherwise, the coordinator performs context switching by freezing the requesting process, activating the selected candidate, and storing communication tensors to disk (lines 29-31) for later retrieval when all participants become available.

This approach effectively transforms the limited GPU resources into a larger virtual execution environment, enabling trace generation for large-scale training scenarios using significantly fewer physical GPUs while maintaining correctness through careful dependency tracking and progress-oriented scheduling.

Algorithm 1 Priority-Based GPU Context Switching

```
1:  Input: Trigger rank r_t, Worker status S
2:  Output: Best switching candidate
3:  function SelectSwitch(r_t)
4:      gpu ← S[r_t].gpu_id
5:      max_pending ← 0, best ← NULL
6:      for rank ∈ S do
7:          if S[rank].status = FINISHED then continue
8:          end if
9:          if S[rank].gpu_id ≠ gpu then continue
10:         end if
11:         if S[rank].head_op.status ≠ READY then continue
12:         end if
13:         if S[rank].pending_ops > max_pending then
14:             max_pending ← S[rank].pending_ops
15:             best ← rank
16:         end if
17:     end for
18:     return best
19: end function
20:
21: function HandleCollective(msg)
22:     r ← msg.sender
23:     UpdatePendingOps(msg.remaining_ranks)
24:     if AllParticipantsOnGPU(msg.participants) then
25:         ExecuteDirect(msg)
26:     else
27:         candidate ← SelectSwitch(r)
28:         if candidate ≠ NULL then
29:             FreezeRank(r)
30:             ActivateRank(candidate)
31:             StoreTensors(msg.tensors)
32:         end if
33:     end if
34: end function
35:
36: function UpdatePendingOps(waiting_ranks)
37:     for rank ∈ waiting_ranks do
38:         S[rank].pending_ops ← S[rank].pending_ops + 1
39:     end for
40: end function
```

B MOE MOCK ROUTER

For MoE models, memory allocation typically varies across devices because experts on different devices may process uneven amounts of data after the dispatch phase. The final data distribution is determined by the gating mechanism. After the input data on each rank passes through the gating layer, each rank obtains the probabilities (or logits) of its local tokens being dispatched to each expert. All ranks participating in expert parallelism then perform communications to exchange these local probabilities. By combining these probabilities with the routing strategy (e.g., `top_k`), the final global data distribution is determined.

To control the non-uniform dispatching of MoE models and observe the resulting memory footprint after dispatch, we designed the MoE Mock Router. This component utilizes the Balance Ratio (`br`) to regulate distribution probabilities. The `br` represents the ratio of the actual data volume possessed by a specific rank to the volume it would possess under a perfectly uniform distribution. A `br` > 1 indicates that the rank is over-utilized relative to the average. Since each micro-batch must pass through all layers within a single iteration, multiple gating operations occur, each requiring control via `br`. We characterize the distribution of `br` in one iteration using statistical metrics including `br_min` (minimum), `br_max` (maximum), `br_avg` (average), `br_std` (standard deviation), `br_med` (median), and `br_skew` (skewness). Based on

Table 5. Extended evaluation of AllReduce transmission latency (ms) across different message sizes and cluster scales.

Size	Baseline				Vanilla Emulation				CrystalLLM			
	16	32	64	128	16	32	64	128	16	32	64	128
16M	0.27	0.58	0.41	0.50	0.28	85.92	181.98	639.44	0.27	0.55	0.27	0.97
128M	1.22	1.17	1.56	2.17	1.22	92.98	734.74	981.97	1.18	1.12	1.82	3.97
1G	6.32	5.65	6.40	8.11	6.38	208.17	1070.08	4351.50	6.19	5.56	6.34	12.72
8G	47.72	43.61	44.34	45.04	47.92	1663.51	4256.79	12742.20	47.21	43.34	44.71	67.04
32G	189.09	173.25	177.08	177.62	189.22	6655.51	16950.36	50815.80	187.96	173.11	176.70	267.42

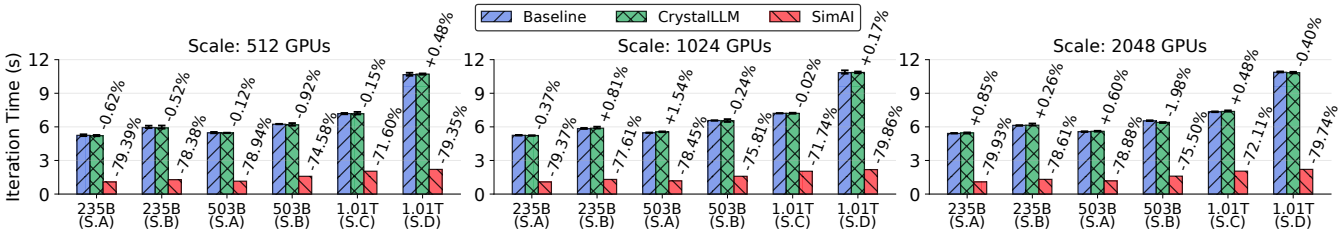


Figure 15. End-to-end iteration time estimation results compared with SimAI.

these statistics, the MoE Mock Router derives the `br` distribution and pre-calculates the logits for all ranks within an expert parallel group. These pre-calculated logits are subsequently injected into the logits tensor generated by the gating mechanism during each invocation.

To ensure that these pre-calculated values do not incur additional GPU memory overhead, they are pinned in host memory. The Mock Router performs an asynchronous in-place copy to the GPU-resident logits tensor only when the gating operation is triggered, effectively overwriting the original values without allocating new device buffers.

C EXTENDED EXPERIMENTAL RESULTS

C.1 Latency Evaluation of AllReduce

Table 5 presents an extended evaluation of the AllReduce transmission latency, detailing performance across varying message sizes (from 16 MB to 32 GB) and expanding the emulation scale up to 128 ranks. The results highlight the severe resource contention inherent in the Vanilla emulation approach, which suffers from exponential latency inflation as the message size and cluster scale grow. In contrast, CrystalLLM's communication pruning strategy consistently minimizes overhead, closely tracking the physical baseline across all tested configurations. However, as the scale reaches 128 ranks, CrystalLLM exhibits noticeable latency deviations for large messages (e.g., 267.42 ms vs. 177.62 ms at 32 GB). This is expected: hosting all virtual ranks on a single physical node saturates its PCIe bandwidth and SMs. As detailed in Section 7.2, CrystalLLM gracefully resolves this hardware bottleneck by scaling out virtual ranks across multiple assistant nodes to restore emulation fidelity.

C.2 Results of CrystalLLM vs. SimAI

Figure 15 presents the comparison between CrystalLLM and SimAI across all model and strategy configurations used in Section 7.4. SimAI is excluded from the memory evaluation as it does not support memory allocation simulation. Consistent with the analysis in Section 7.4, SimAI underestimates iteration time across all configurations, with an average error of 77.2%, mainly because its workload files omit the communication bubbles of PP and the computation overheads of MoE-specific operations.