

Phantora: Maximizing Code Reuse in Simulation-based Machine Learning System Performance Estimation

Jianxing Qin¹ Jingrong Chen^{3,†} Xinhao Kong^{4,†} Yongji Wu^{5,†} Tianjun Yuan¹ Liang Luo²
Zhaodong Wang² Ying Zhang² Tingjun Chen¹ Alvin R. Lebeck¹ Danyang Zhuo¹

¹Duke University ²Meta ³Uber ⁴NVIDIA ⁵University of California, Berkeley

Abstract

Modern machine learning (ML) training workloads place substantial demands on both computational and communication resources. Consequently, accurate performance estimation has become increasingly critical for guiding system design decisions, such as the selection of parallelization strategies, cluster configurations, and hardware provisioning. Existing simulation-based performance estimation requires reimplementing the ML framework in a simulator, which demands significant manual effort and is hard to maintain as ML frameworks evolve rapidly.

This paper introduces Phantora, a hybrid GPU cluster simulator designed for performance estimation of ML training workloads. Phantora executes unmodified ML frameworks as is within a distributed, containerized environment. Each container emulates the behavior of a GPU server in a large-scale cluster, while Phantora intercepts and simulates GPU- and communication-related operations to provide high-fidelity performance estimation. We call this approach hybrid simulation of ML systems, in contrast to traditional methods that simulate static workloads. The primary advantage of hybrid simulation is that it allows direct reuse of ML framework source code in simulation, avoiding the need for reimplementing. Our evaluation shows that Phantora provides accuracy comparable to static workload simulation while supporting three state-of-the-art LLM training frameworks out-of-the-box. In addition, Phantora operates on a single GPU, eliminating the need for the resource-intensive trace collection and workload extraction steps required by traditional trace-based simulators. Phantora is open-sourced at <https://github.com/QDelta/Phantora>.

1 Introduction

Large machine learning (ML) models have become a driving force behind advancements in natural language process-

ing [10, 30, 39], computer vision [13], computer graphics [29] and recommendation systems [25, 27, 47]. As models grow increasingly complex, high-performance model inference [19] and training [50] have become the main focus in the ML system community. Recent advancements in the field span from efficient GPU kernel optimizations [11, 12, 19], to parallelization strategies [50], and scheduling algorithms [31]. When deploying ML training jobs, it is often beneficial to estimate the system’s performance (*e.g.*, training time per iteration, model FLOPS utilization), which can help operators decide how many hardware resources to allocate for a particular job, and plan for future hardware needs.

There has been a growing interest in performance estimation methods for ML systems. Analytical models (*e.g.*, roofline [43]) provide rapid estimates but lack accuracy. More recently, both industry and academia have shifted towards static workload simulation. Figure 1 shows the two methods of static workload simulation and their problems. A common method is trace-based simulation [7, 14, 16, 17, 24, 33, 44, 51], where execution traces are collected from real runs of ML workloads on large clusters. These traces are then processed to extract workloads: a trace is lifted into higher levels of abstraction to make it suitable for configuration and event-driven simulation. Another method is SimAI [42], which implements a mocked version of ML frameworks in order to generate events that can be used in a simulator. However, both methods have to reimplement the scheduling logic of the simulated ML framework (*e.g.*, DeepSpeed [34], Megatron [37], TorchTitan [21]) and trace-based simulation may even need to implement a reversed scheduling logic in workload extraction. Reimplementation makes them difficult to maintain due to fast-evolving ML training frameworks.

In this paper, we explore the following question: *Can ML framework source code be directly reused in simulation-based performance estimation?* Modern ML frameworks already provide a rich set of configurable parallelization strategies (*e.g.*, pipeline parallelism, expert parallelism, data parallelism) and rematerialization strategies (*e.g.*, ZeRO [32], FSDP [48], activation recomputation [18]). By reusing these implementa-

[†]Jingrong Chen, Xinhao Kong, and Yongji Wu conducted this work while they were at Duke University.

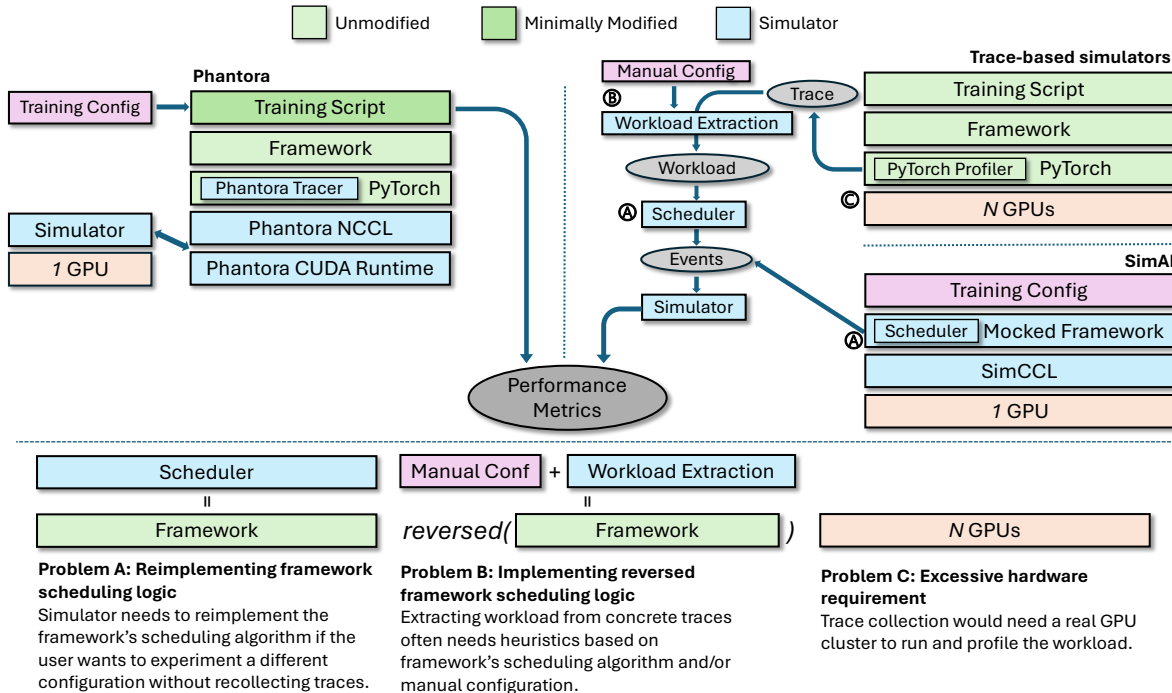


Figure 1: Comparison of simulators and problems of static workload simulation. Light green boxes show unmodified components; gree boxes show minimally modified components; blue boxes show simulator components and pink boxes show user input. Trace-based simulation requires workload extraction (reversing framework logic) and costly trace collection on large clusters. SimAI relies on mocked ML frameworks.

tions, we can evaluate different parallelization and rematerialization strategies without relying on the simulator’s reimplementations. In addition, the performance benchmarking code embedded in training scripts can be reused directly, allowing users to interact with the simulator in the same way they would when tuning ML system performance on a real GPU cluster. Most importantly, avoiding reimplementations of the framework greatly reduces the maintenance burden, making the simulator more sustainable as ML frameworks evolve rapidly.

We present a fundamentally different performance estimation approach for ML systems, called *hybrid simulation*. Our key idea is that we can *integrate real system execution directly with event-driven simulation, creating an illusion for ML frameworks that they are running on a real GPU cluster*. Based on this approach, we build Phantora, a GPU cluster simulator for ML system performance estimation. The core design is to use a containerized environment, equipped with a single GPU to directly run an ML system for simulation. Each running container simulates the execution of a multi-GPU server, where each rank operates one simulated GPU and is provided with one virtual clock. Phantora intercepts all the communications and GPU kernel invocations from the ML framework. CUDA kernel execution times are profiled on the single GPU, while communication execution times are cal-

culated using an event-driven network simulator. When a rank launches a CUDA kernel or initiates communication, Phantora adjusts the rank’s virtual clock accordingly to maintain accurate simulated time.

We still face three key research challenges. First, it is challenging to maintain the core abstractions (e.g., CUDA, NCCL) so that unmodified ML frameworks can directly run. Second, it is challenging to integrate a real, running system with an event-driven network simulator. To elaborate on the second challenge, an event-driven network simulator progresses in discrete time steps, and the timing for the next event is calculated by prior events in the system. However, in hybrid simulation, our distributed ML system (the set of containers) may inject a network flow at any time, causing the network simulator’s calculation to be incorrect. Finally, certain distributed ML workloads may quickly exhaust the memory capacity of the simulation environment.

Phantora maintains the core abstractions used by PyTorch and uses runtime patching to dynamically rewrite their dependencies (e.g., timer), so that the ML frameworks (e.g., Megatron [37], DeepSpeed [34], TorchTitan [21]) can run as is on top of Phantora. Phantora addresses the integration challenge by allowing an event-driven network simulator and a distributed ML system to run in a loosely synchronized manner. When an event that should occur earlier is injected by

the container to the network simulator, Phantora rolls back the simulator state and accommodates those past events triggered by the containers. To enhance scalability, Phantora allows containers to share memory, significantly reducing the simulator’s memory footprint. We evaluate Phantora using three state-of-the-art Large Language Model (LLM) training systems: Megatron [37], DeepSpeed [34], and TorchTitan [21]. All three systems run out-of-the-box without any modification to the source code, and *Phantora can support their feature sets without the need for corresponding reimplementations*. Their terminal outputs remain identical (except training losses) to those produced when running on a real GPU cluster. Our small-scale NVIDIA H200 testbed evaluation demonstrates that Phantora achieves simulation accuracy comparable to state-of-the-art workload simulation methods. Phantora’s simulation result on TorchTitan matches its reported performance on large-scale NVIDIA H100 and A100 clusters [4, 21].

This paper makes the following contributions:

- We are the first to propose *hybrid GPU cluster simulation for ML systems*, which eliminates the need for reimplementing ML frameworks in the simulator.
- We design and implement Phantora, which efficiently integrates real system execution with an event-driven simulation to enable hybrid simulation.
- We evaluate Phantora on three state-of-the-art LLM training systems, demonstrating Phantora’s generality and accuracy.

2 Background

Performance estimation is valuable for both ML system developers and infrastructure providers. For system developers, it enables rapid evaluation of the performance of the systems they are building. As model sizes grow, many system parameters require careful tuning, for instance, selecting an appropriate parallelization strategy [37, 50]. Being able to estimate the performance of different strategies makes it easier to identify the most efficient option. For infrastructure providers, performance estimation allows planning for future hardware deployments.

Due to the accuracy limitations of performance modeling (e.g., roofline [43]), developers have turned to static workload simulation for performance estimation. Today, two workload simulation methods exist. The first method is trace-based simulation [7, 14, 16, 17, 24, 33, 44, 51]. A trace is collected in a real execution, and a workload can be extracted from a collected trace. Another method is SimAI [42], where workloads are generated via implementing a mocked version of the ML frameworks. Workload simulation leads to accurate performance estimation and is increasingly adopted by industry [38].

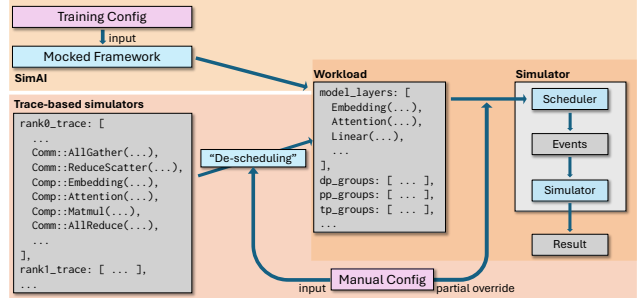


Figure 2: Scheduling logic needs to be reimplemented in current simulators for converting high level workload to detailed computation and communication events.

Why do static workload simulators need to reimplement ML frameworks? For trace-based simulators, there are two fundamental reasons why reimplementing of the ML frameworks is necessary. First, the key goal of simulation is to explore alternative configurations. Yet, a collected trace reflects only the specific configuration used during execution. To enable configuration exploration, trace-based simulation requires a workload extraction step that lifts the trace into abstract workload, revealing higher-level configurations from actual traces (Figure 2). This extraction step typically relies on human understanding of ML framework execution, effectively constructing a reversed version of the framework’s logic. Heuristics can be brittle and may fail to generalize across frameworks; therefore, trace-based simulation often relies on extra manual configurations to help with workload extraction, which increases manual effort. More importantly, after the user changes the configuration in abstract workload, the simulator needs to turn user’s configurations into a detailed execution plan (events) for actual simulation. This scheduling also needs to correctly reflect the framework’s scheduling being simulated, which is essentially a reimplementing of the framework’s logic.

Beyond trace-based simulation, SimAI [42] adopts an alternative approach. Rather than extracting workloads from collected traces, it uses mocked frameworks to directly produce low-level events. However, while this removes the burden of workload extraction, reimplementing scheduling remains necessary in the mocked frameworks. It tightly couples SimAI to specific framework and versions: whenever the underlying ML framework evolves or new framework appears, SimAI’s logic must be updated accordingly to ensure correctness.

Due to the need of reimplementing frameworks, both trace-based simulators and SimAI struggle to provide complete support of features in current frameworks and new frameworks. For example, none of the existing simulators support TorchTitan [21], and none of them can analyze throughput and memory usage of selective activation checkpointing [18, 21] at the same time. Simulators may even fail to generate the exact same model as the framework. For example, when given the same configuration matching Llama2 7B [39], the size

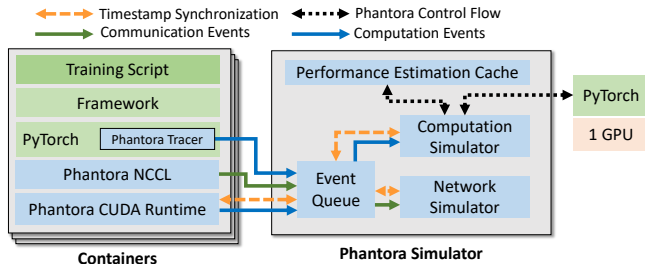


Figure 3: Phantora architecture. Components in light green are unmodified, components in green are minimally modified, and components in blue are constructed by Phantora.

of model generated by SimAI differs by 7.4% from native GPTModel in Megatron [37] model library.

Our approach: Hybrid simulation. Our objective is straightforward: to build a general simulator in which ML frameworks’ source code can be directly reused. Modern ML frameworks already provide well-maintained implementations and highly configurable parallelization strategies. Leveraging them directly makes the simulator both easier to maintain and more broadly applicable across different frameworks. To achieve this, our approach is to construct a GPU cluster execution environment that closely mirrors a real one, allowing ML frameworks to run on top of the simulated cluster with no modification. Next, we discuss the research challenges to realize this approach and the overview of our design.

3 Overview

We face three key research challenges in realizing hybrid simulation of ML frameworks: (1) supporting unmodified ML frameworks, (2) ensuring correct and efficient time synchronization, and (3) achieving scalability. First, modern ML frameworks have complex software and hardware dependencies. They assume the presence of GPUs, NVLinks, and RDMA networks, and they rely on a wide range of libraries. Constructing an execution environment in which an ML framework can run unmodified is therefore nontrivial. Second, integrating event-driven network simulation with ML framework execution presents a fundamental mismatch. Network simulators advance time in discrete time steps, while ML frameworks execute with continuous, real-time progression. Reconciling these two timing models is challenging. Finally, ML frameworks are resource-intensive, consuming substantial compute, memory, and communication bandwidth. While we would like to execute them as-is, we must avoid the prohibitive resource costs of training a full-scale model on a real cluster.

To support unmodified ML frameworks, Phantora runs an ML framework in a realistic containerized environment and

interacts with this real system for simulation.¹ Figure 3 shows Phantora’s architecture. Each container in the environment acts as a GPU server. Each container runs Python interpreters that execute ML framework code. There are two operations that require interaction between the containers and the Phantora simulator: (1) GPU computation and (2) communication. When a computation kernel is invoked, Phantora uses a single GPU to profile the kernel’s execution time. For communication, Phantora utilizes an event-driven flow-level network simulator to estimate completion time. Phantora only needs a single GPU because it profiles the performance once for each (computation kernel, tensor shapes) combination. This is sufficient because computation kernel performance is usually independent of the tensor values.² In this way, an application cannot distinguish whether it is running on Phantora or a physical GPU cluster as long as its control flow does not depend on tensor values (which would be junk values in Phantora). The time of each rank in every container will be maintained by Phantora using standard discrete event simulation, and the application can read this time to calculate its performance. A naive implementation of the above approach would still require modifying the ML framework source code. For example, one might need to change the performance timer implementation used in framework logging. To support ML frameworks out of the box, we instead leverage Python’s runtime patching to dynamically redirect underlying dependencies of an ML framework.

To appreciate the time synchronization challenge, let’s consider the following *past events* scenario. Traditional event-driven simulators require static workloads. For example, in a typical event-driven simulation workflow, all the events in the workload are pre-loaded into the event queue of the simulator. The simulator processes these events in chronological order, updating the simulation state and queuing new events as necessary. However, with real systems, events are generated dynamically and injected into the simulator, which can lead to past event scenarios: the simulator receives an event generated by the real system after the simulator has already advanced to the next event, which has a timestamp later than the real system’s event. One solution is to keep the event-driven network simulator tightly synchronized with the containers. In hardware simulators, for example, it is common to use a tiny time quantum and synchronize all components at every quantum [28, 35]. However, it introduces significant overhead and defeats the purpose of event-driven simulation in modern network simulators. Phantora uses loose time synchronization between the event-driven simulator and the real system execution to address this issue and enables fast simulation. To ensure simulation accuracy, we implement an event-driven network simulator capable of time traveling to rollback the

¹Using process-level isolation is sufficient to run Phantora. We prefer containers to seamlessly integrate with unmodified frameworks that assume a multi-host environment.

²There are some exceptions. We discuss this point in §6.

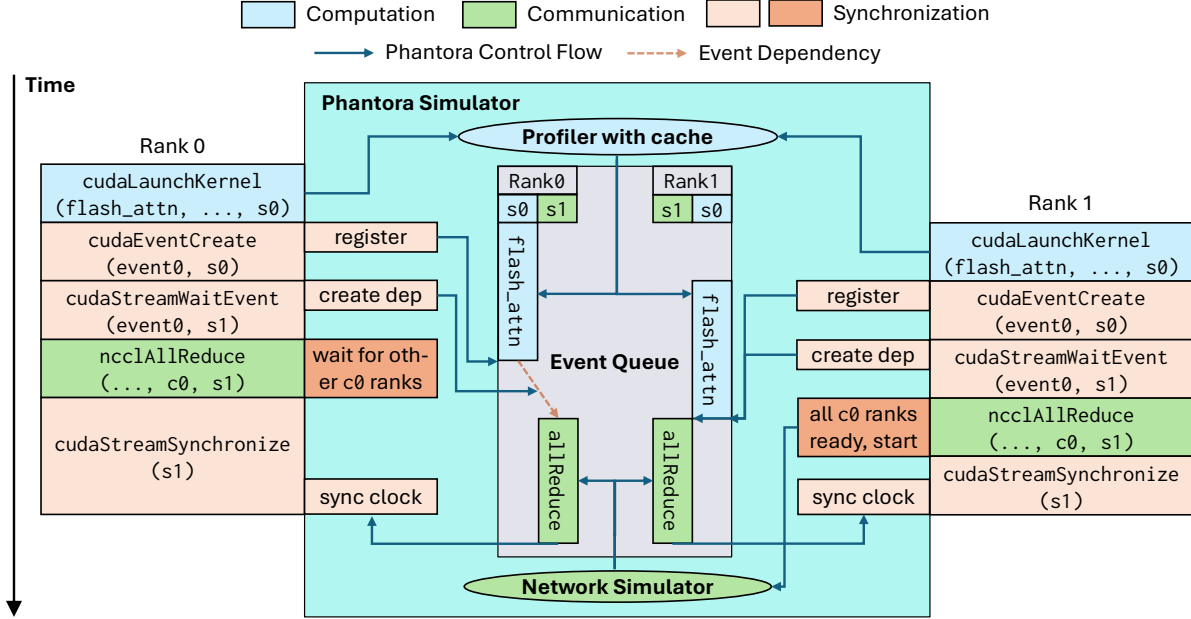


Figure 4: An example workflow of Phantora with two ranks. ML system places computation and communications on different CUDA streams for flexibility, and use CUDA events to manage the synchronization between them. Phantora needs to correctly handle these synchronizations to achieve accurate simulation.

states of all flows when events occur in the past.

We apply two techniques to scale Phantora. In LLM training, each GPU server may load training data and model weights, which consumes significant host memory. Our evaluation (§5.3) shows that a physical server with 256GB host memory can only simulate 9 GPUs for the training of Llama2 7B model without our technique. We introduce a novel memory sharing mechanism for different containers running on the same host, which significantly reduces the memory footprint. Second, Phantora measures only the actual CPU time consumed by each process, rather than the wall clock time. This choice preserves simulation accuracy in the presence of CPU core contention, where wall-clock measurements would otherwise overestimate execution time due to frequent context switches between the containers.

4 Details of Phantora

4.1 Supporting Unmodified ML Frameworks

Each rank executes unmodified ML framework code, using PyTorch and NCCL libraries, and interacts with the CUDA Runtime. We implement the Phantora Tracer in the PyTorch. This tracer collects all invoked PyTorch operators and the corresponding performance-related parameters, and pushes all information as computation events to the event queues in the Phantora simulator. It is worth noting that this tracer does not affect the execution of PyTorch, and the operators are still dispatched to the corresponding CUDA backend to initiate com-

putation in the CUDA Runtime. We replace the native CUDA Runtime with Phantora CUDA Runtime, which does not actually execute any CUDA calls. Instead, it only maintains necessary metadata to emulate actual CUDA Runtime behaviors. For example, `cudaMalloc/cudaFree` in Phantora does not actually allocate/deallocate GPU memory, but only tracks GPU memory usage and returns `cudaErrorMemoryAllocation` when an allocation will make usage exceed the configured memory capacity. In addition, it pushes CUDA calls as events to the event queue in Phantora simulator. Similarly, we replace the native NCCL library with Phantora NCCL library. Phantora NCCL library does not initiate communication, but forward all communication operations (e.g., `allreduce`) to the simulator by pushing communication events to the event queues. Furthermore, Phantora maintains a dependency graph of events to emulate CUDA’s asynchronous semantics, i.e. CPU launches computation and communication kernels and specify dependencies through streams/events.

Figure 4 shows an example workflow of Phantora and two ranks. These two ranks are running a distributed workload of attention and all-reduce on the attention result. The launch of FlashAttention [11, 12] is captured by Phantora CUDA Runtime and pushed to Phantora simulator. PyTorch operators traced by Phantora Tracer are also pushed to Phantora simulator as computation events. Phantora simulator will then invoke native computation libraries to profile these computation kernels. The profiler uses a performance estimation cache to store the performance results of operators that have been already faithfully executed. When invoking the same opera-

tors in the future, Phantora will directly use results stored in the cache. In this example, the FlashAttention call of Rank 1 will not be profiled again, instead, Phantora simulator will use the cached profiling result of Rank 0’s FlashAttention.

To accurately capture the dependencies and synchronizations of computation and communications and emulate the behavior of actual CUDA Runtime, Phantora needs to carefully handle related CUDA calls. Phantora event queue is designed to natively support dependencies and is used to emulate CUDA streams and events—two core constructs in CUDA asynchronous programming. Operations on the same stream will have implicit dependency in chronological order, and operations on different streams have no dependency unless explicitly specified via CUDA events. In Figure 4’s example, computation and communications are launched on different streams for flexibility, and the dependency between them is enforced via an additional CUDA event. Phantora correctly emulated that via dependencies in the event queue.

Phantora NCCL captures communication operations and pushes them to the Phantora simulator. For example in Figure 4, `ncclAllReduce` is first called by Rank 0. This is API is non-blocking so Phantora NCCL will return immediately after pushing the call to the simulator, but the simulator will not start network flows until all ranks in the same communicator are prepared (in this example, wait for Rank 1 to call `ncclAllReduce` with `c0`), which is in compliance with NCCL semantics. To accurately model the execution time of collective communication operations, Phantora adopts a standard flow-level network simulator adapted from NetHint [8], referred to as `netsim`. The `netsim` simulator takes a cluster topology configuration as input, where users can specify various properties of the cluster, including switch port bandwidth, cluster interconnection, and multipath routing and load balancing strategies. The throughput of flows at each time is computed based on the max-min fairness. Within `netsim`, we implement the communication patterns of different collective operations. For instance, we model `allreduce` using a ring-based approach, as configured in NCCL in our evaluation. When receiving communication operators, Phantora submits the corresponding data transfer flows to `netsim` with the appropriate timestamps. `netsim` then simulates network behaviors and computes the completion time of each flow based on network congestion and available bandwidth [8, 45].

Time synchronization is enforced through CUDA synchronization calls, which should block the host until certain GPU completion point (e.g., `cudaStreamSynchronize`). When these APIs are called, the Phantora CUDA Runtime pushes a synchronization event to the event queue and starts waiting for a response. After processing the preceding events in the event queue and completing this synchronization event, the simulator returns a response, including a completion time (a logical timestamp), to the rank’s Phantora CUDA Runtime. The rank’s virtual clock is then updated based on this completion time. In the example shown in Figure 4, The virtual clock

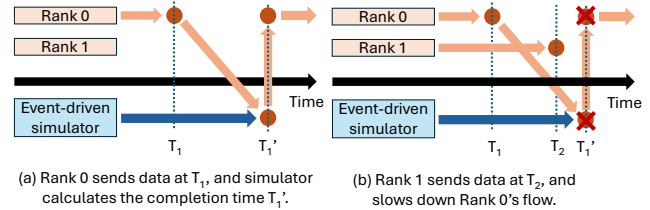


Figure 5: Challenges of synchronizing time between real execution and event-driven simulation.

of both Rank 0 and Rank 1 will be updated to the completion time of all-reduce after `cudaStreamSynchronize`.

Runtime patching for ML frameworks. For certain frameworks, there might be unconfigurable behaviors that do not satisfy Phantora’s assumption. To ensure no modification to the framework code and provide a close-to-natural user experience, we leverage the dynamic features of Python to runtime patch certain functions. For example, the built-in performance logging of TorchTitan utilizes `time.perf_counter` that needs to be replaced by Phantora timers. With runtime patching, Phantora can directly work with ML frameworks installed via “pip install”.

Intercepting CUDA kernel invocations and communication. One key design decision we have to make is how to intercept CUDA kernel invocations and communications. A strawman solution could intercept ML system execution only at the CUDA kernel level, which is widely used in profilers such as Nsight Systems [2]. But at this level only untyped pointers to arguments are provided, so the inspection of arguments would require extra configurations for each kernel.

Hence, we resort to a hybrid approach where most computation operations are intercepted at ML systems API level (e.g., PyTorch operators), while communication operators and some specific computation operations like FlashAttention [11, 12] are intercepted at runtime libraries level (e.g., NCCL, CUDA Runtime). This hybrid approach allows Phantora to maintain generalizability without introducing the development effort of inspecting every CUDA kernel involved. For computation operations, Phantora is aware of the type of operations and the shapes of the input tensors. Phantora implements a cache manager to reuse earlier profiling results of an operation with the same input shapes, eliminating redundant execution.

4.2 Loose Time Synchronization of Real Execution and Event-Driven Simulation

Phantora needs to correctly synchronize the real execution and the event-driven simulator for accurate end-to-end simulation, as events generated by the real execution may impact the event-driven simulator. Without proper synchronization, such impacts may be neglected or wrongly considered, leading to inaccurate simulation.

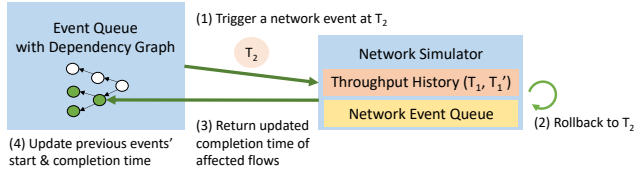


Figure 6: Handling past events in the event-driven simulator using time rollback. The network state at T_2 is a superposition of the states at T_1 and T_1' .

Figure 5 demonstrates a typical workload that requires correct synchronization. Rank 0 and 1 independently launch communications that may share network resources. At time T_1 , Rank 0 begins to wait for its communication and asks the simulator for the completion time. The simulator, however, cannot directly proceed to the completion point based on current information and respond T_1' , as a future communication from Rank 1 starts at T_2 may affect this completion time if $T_2 < T_1'$. Note that this is not an issue if the simulated workload is static (*e.g.*, the use cases of today’s static workload simulators). For static workload, the simulator already knows the existence of event at T_2 , so it can simply move the simulated time forward to T_2 instead of T_1' , and update the state of the system. However, in our case because Rank 1 is a real execution, the simulator does not know whether or when it will launch communication.

One approach to resolve this issue is to determine a small time quantum, and move time forward in both real execution and simulation. This is a common technique in hardware simulation. WWT [35] uses such an approach for simulating cache-coherent shared memory multiprocessors, and it determines the time quantum based on cross-core communication latency. However, using a fine-grained time quantum can significantly slow down the simulation speed, which is exactly why it is not used by most of today’s network simulators.

Our key observation is that the characteristics of ML systems provide an opportunity to change the running time of operations during hybrid simulation. Specifically, the time taken by a given operation (*e.g.*, a specific communication) does not affect the actual control flow of the real system. For example, changing the time consumed by a matrix multiplication in a rank’s runtime does not affect which is the next operator invoked by the rank. Therefore, we can *rollback* the simulator state and correct the real system state efficiently during simulation. We apply this insight by optimistically synchronizing clocks between the simulator and each rank’s runtime in Phantora. When past events occur, the simulator rolls back to a prior time, processes the past events, corrects time for a set of events in the past, updates the clock with each rank’s runtime, and continues the simulation process.

Time rollback. A traditional event-driven simulator keeps a priority queue of “events”, where the priority is the start time of the event. This allows the simulator to process the event in

the chronological order. Phantora augments this simulator by adding the ability to time travel to any particular time in the past. To realize this feature, the network simulator keeps the throughput history of all flows. Consider the same example in Figure 5. A new event arrives at a time T_2 earlier than the current time T_1' . For simplicity, let’s assume there are no other events between T_1 and T_1' and let $S(T)$ denote the state of all the network flows at time T . As Figure 6 shows, the network simulator can compute state $S(T_2)$ based on the stored throughput history between $S(T_1)$ and $S(T_1')$. This is because between neighboring events, network flows are assumed to have stable throughput, which is a common assumption made by existing event-driven network simulators. Such a time rollback can affect previously computed completion time of some network flows. The network simulator then sends these updated completion times to Phantora’s event queue to update other events’ start/completion time by traversing the dependency graph. For example, if another event previously started at T_1' for its dependency on this communication, then its start and completion time may be adjusted accordingly.

With this rollback mechanism, Phantora will still ensure forward progress if the system being simulated ensures forward progress. In Phantora, the simulated ML system only has finite past events, so there will only be finite rollbacks and the simulator will make forward progress after rollbacks are complete.

Garbage collection of historical states. The ability to time travel to the past comes with the cost of storing the simulation states at all the event timestamps. These states include the dependency graph stored in Phantora’s event queue and the historical flow states in the network simulator. As the simulation progresses, storing these historical states can occupy a lot of host memory. Phantora implemented garbage collection to address this issue. The key insight is that after all the ranks’ time has passed T , it is impossible to inject an event before T into the network simulator. Thus, all the simulator states before T (including both the dependency graph and the flow states) can be safely discarded.

Network simulator with time rollback. Our flow-level simulator enables time rollback with low additional runtime overhead. The simulator assumes per-flow fairness across the network and solves the max-min fair flow allocation problem using an iterative water-filling algorithm. At each iteration, the simulator identifies the bottleneck link and computes the necessary delta adjustments for flow rates.

To support time rollback, we provide two APIs: one for updating the start time of an existing flow, and another for advancing the simulation by one step or up to a specified time. The simulator records the throughput history for each flow, which is represented by a few floating point numbers. Since throughput changes are regular events to a network simulator, without garbage collection, the memory overhead is proportional to the number of discrete events the simu-

lator processes. Although a rollback can potentially update multiple flows, these computations are based solely on the throughput history and can be computed in an incremental manner, making them computationally efficient compared to solving the max-min fair flow allocation problem.

4.3 Improving Phantora’s Scalability

Phantora has the following two techniques to achieve scalable simulation.

Scalability Technique #1: Model parameter sharing on CPU. ML systems may initialize models in CPU memory, either randomly or using pre-trained weights stored on disk. Once initialized, the models are transferred to the GPU for subsequent tasks. In real GPU clusters, this initialization phase is generally not a bottleneck, as GPU memory size is usually smaller than the CPU memory size. However, when Phantora simulates a large cluster using limited hardware resource, this peak memory usage could become a scalability constraint. To address this limitation, Phantora implements parameter sharing, which allows model parameters on the same simulation server to be transparently mapped to the same region of shared memory. This ensures that at most one copy of the model is initialized per server. Phantora assumes that the control logic of ML systems does not depend on tensor values, allowing safe sharing of model parameters without impacting execution.

Scalability Technique #2: Use CPU time instead of system time. Similar to memory, CPU can also become a bottleneck for Phantora. Existing ML training systems are usually multi-process. If the number of CPU cores used for simulation is smaller than the number of processes launched, CPU oversubscription can slow down the execution of the ML systems and cause inaccuracies in simulation results. To address this problem, Phantora only counts the actual CPU time each process spent instead of the system time passed (wall clock). Thus, although the simulation process is still slowed down, the accuracy of the results will not be affected. Phantora can also be configured to ignore the CPU time completely, leaving only the GPU operation time and CUDA synchronization waiting time to be included in the results.

5 Evaluation

Our Phantora prototype consists of 9K lines of Rust, 1.8K lines of C, and 500 lines of C++: 1.8K lines of C and 1K lines of Rust for Phantora NCCL and CUDA Runtime, 3.6K lines of Rust for the flow-level network simulator, 3.4K lines of Rust for the event queue, 1K lines of Rust for the computation simulator, and 500 lines of C++ for Phantora Tracer.

We evaluate Phantora on three aspects. First, we test Phantora’s generality in supporting different ML frameworks, and their runtime behaviors. Second, we test a set of standard

metrics for simulators, such as accuracy, simulation speed and scalability. Finally, we do a case study on selective activation recomputation [18] to show Phantora’s capability on estimating ML system performance and GPU memory usage for features that existing static workload simulators have not fully reimplemented.

5.1 Generalizability

Effort for supporting ML frameworks. Phantora currently support three LLM training frameworks: Megatron [37], DeepSpeed [34] and TorchTitan [21]. Users can simply `pip install` these frameworks from official PyPI without building from source or using a different package index. All the runtime patches are applied when users `import` our helper library. Specifically, the size of runtime patches of these three frameworks are: *Megatron*: no patch needed. *DeepSpeed*: 4 lines of code where a NCCL setup validation is disabled. *TorchTitan*: 1 line of code where `time.perf_counter` is replaced with Phantora timer. In contrast, SimAI uses about 8K lines of Python for the mocked frameworks.

We believe supporting other PyTorch based frameworks in Phantora should also require minimal effort. Further, Phantora does not depend on model architecture. Our evaluation results focus on LLMs. See [Appendix A](#) for our evaluations using non-LLM workloads.

Modifications to the training script. For every training script, Phantora Tracer needs to be explicitly enabled at the beginning and disabled at the end. This, together with importing of our helper library, adds about 6 lines of extra code per training script.

In addition, when running Megatron [37] with Phantora, gradient clipping must be disabled. This feature performs fallible CPU operation (specifically, square root) of data copied from GPU, which could lead to math errors as GPU memory value is effectively random in Phantora.

User experience of Phantora. With Phantora, users are able to tune their ML system performance as if they are actually experimenting with a real GPU cluster. Phantora natively supports customized printing and logging mechanisms embedded in the ML systems.

The top part of [Figure 7](#) shows the performance measurement and logging code in TorchTitan [21], which represents how TorchTitan developers want to evaluate performance. In other existing simulators, printing these metrics would require a reimplement of this code in post-simulation analysis. In contrast, Phantora allows this code to run as is, and users can see results in exactly the same format as if they actually run the ML system on a real GPU cluster. The bottom part of [Figure 7](#) shows the console output of running TorchTitan on top of Phantora. To the best of our knowledge, Phantora is the only method that has this type of generalizability.

After developers update either the ML system code (*e.g.*,

```

time_delta = timer() - time_last_log
# tokens per second, abbr. as wps by convention
wps = ntokens_since_last_log / (
    time_delta * parallel_dims.model_parallel_size
)
# model FLOPS utilization
mfu = 100 * num_flop_per_token * wps / gpu_peak_flops
time_end_to_end = \
    time_delta / job_config.metrics.log_freq
time_data_loading = np.mean(data_loading_times)
metrics = {
    "wps": wps,
    "mfu(%)": mfu,
    "time_metrics/end_to_end(s)": time_end_to_end,
    "time_metrics/data_loading(s)": time_data_loading,
    ...
}
metric_logger.log(metrics, step=train_state.step)
logger.info(
    f"step: {train_state.step:2} "
    f"loss: {global_avg_loss:7.4f} "
    f"memory: {gpu_mem_stats.max_reserved_gib:5.2f}GiB"
    f"({gpu_mem_stats.max_reserved_pct:.2f}%) "
    f"wps: {round(wps):.1} "
    f"mfu: {mfu:.2f}%{color.reset}"
    ...
)

```

```

[rank0]:2024-09-19 14:46:00,990 - root - INFO - step: 5 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,876 mfu: 53.38%
[rank0]:2024-09-19 14:46:37,206 - root - INFO - step: 10 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,887 mfu: 53.60%
[rank0]:2024-09-19 14:47:13,114 - root - INFO - step: 15 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,902 mfu: 53.87%
[rank0]:2024-09-19 14:47:47,622 - root - INFO - step: 20 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,904 mfu: 53.90%
[rank0]:2024-09-19 14:48:22,964 - root - INFO - step: 25 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,905 mfu: 53.92%
[rank0]:2024-09-19 14:48:58,821 - root - INFO - step: 30 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,904 mfu: 53.90%
[rank0]:2024-09-19 14:49:34,490 - root - INFO - step: 35 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,886 mfu: 53.57%
[rank0]:2024-09-19 14:50:30,248 - root - INFO - step: 40 loss: 0.0000
memory: 41.56GiB(51.95%) wps: 2,894 mfu: 53.72%

```

Figure 7: Performance estimation code in TorchTitan [5] and the console output of running TorchTitan on top of Phantora. The console output is exactly the same as if TorchTitan runs on a real GPU cluster except losses.

changing parallelization strategies for LLM training), the model or the performance measurement and logging code, Phantora can immediately re-run and produce updated console output. For other existing simulators, the developer may have to craft extra configurations, collect additional traces or change post-simulation analysis—all of which significantly slow down the development cycle.

Phantora also supports feature-rich visualization via Perfetto UI [3] to help developers tune their system. Figure 8 shows a visualized simulation trace of TorchTitan exported by Phantora.

Computation/communication overlap and dynamic memory behaviors in ML systems. Phantora naturally captures computation/communication overlap and other runtime behaviors in ML systems. Figure 8 shows the timeline of Phantora executing TorchTitan, where x-axis is simulated time. As shown in the figure, the NCCL operations (communication)

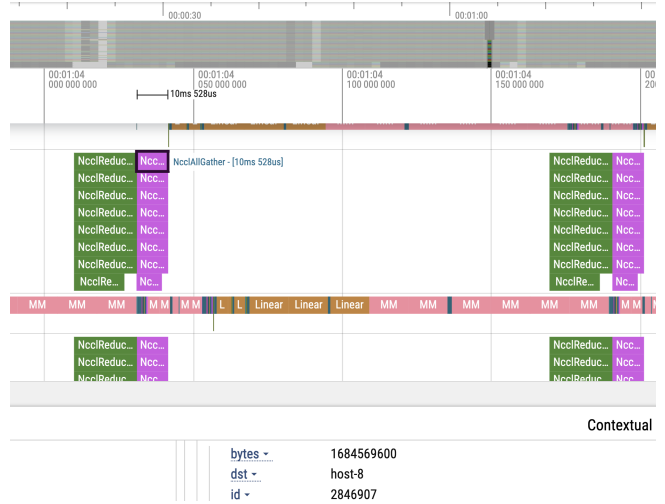


Figure 8: Perfetto [3] trace exported by Phantora.

overlap with matrix multiplication (computation).

Phantora can also naturally capture dynamic behaviors of the PyTorch caching allocator as it tracks memory management on CUDA Runtime level. Note that ML systems usually cannot utilize all of GPU memory due to memory fragmentation. Phantora can precisely reflect the fragmentation and dynamic behaviors of the PyTorch caching allocator, leaving the only imprecision under CUDA Runtime, i.e., the memory management in NVIDIA GPU driver.

5.2 Simulation Accuracy

Our hardware testbed. We run Phantora on two on-premise GPU servers, depending on which server has the closer hardware to the performance report we are comparing with. One server is equipped with 2 AMD EPYC 9355 CPUs and 4 NVIDIA H200 NVL GPUs connected via NVLink. We use all four GPUs to collect ground truth performance numbers. When we run Phantora, we restrict the GPU usage to a single GPU. We use *H200 testbed* to reference this testbed. Another server is equipped with 2 Intel Xeon Gold 6348 CPUs and a single NVIDIA A100 40G GPU. We use *A100 testbed* to reference this testbed.

Comparing with public performance report as ground truth. TorchTitan provides a comprehensive benchmark results [4, 21] utilizing up to 128 GPUs with a combination of FSDP2 and activation checkpointing. Figure 9 shows the accuracy of Phantora to simulate TorchTitan’s benchmark. The average error is 2.9% with the maximum error of 8.5% on Llama2 13B. Note that due to the limitation of accessible hardware, H100 reports [21] are evaluated on the H200 testbed, and A100-80G reports [4] are evaluated on the A100 testbed with a single A100-40G GPU. The difference of computing power (FLOPS) between the reported GPUs and testbed GPUs is minor. The main difference is memory capacity,

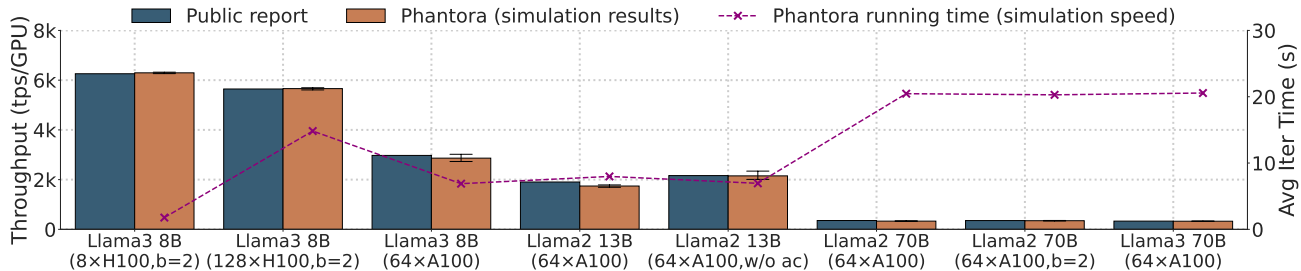


Figure 9: **Accuracy and speed of Phantora (large scale):** Training throughput reported by TorchTitan [21] using FSDP2, simulation results and simulation speed of Phantora on the testbeds. The error bars show 95% confidence interval. “ac” means activation checkpointing in TorchTitan [21].

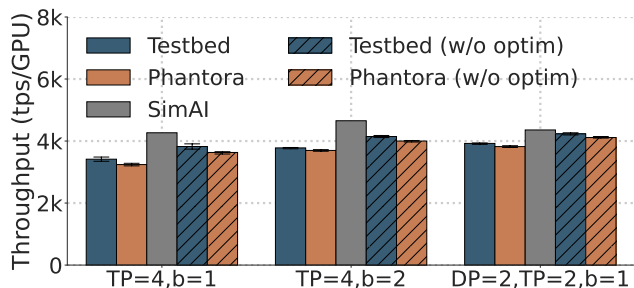


Figure 10: **Accuracy of Phantora (small scale):** Megatron training throughput of Llama2 7B on H200 testbed with or without optimizer, Phantora simulation results with or without optimizer, and SimAI simulation results. The error bars show 95% confidence interval. Note that SimAI currently does not include optimizer in its simulation.

which is configurable in Phantora and is set to the corresponding amount (80GB) in both experiments.

Comparing with testbed training performance as ground truth. Figure 10 shows the accuracy of Phantora using Llama2 7B training with different parallelization strategies and batch sizes. Compared with the ground truth, the average error of Phantora is 3.7% with the maximum error of 5.3% when tensor parallel size is 4 and micro batch size is 1. We hypothesize that SimAI’s error is larger than expected because a core component, SimCCL, though open-sourced, had not yet been integrated into SimAI’s open-sourced mocked frameworks at the time of the experiment.

5.3 Simulation Speed and Scalability

Simulation speed. Figure 9 also shows the simulation speed of Phantora on TorchTitan’s benchmark, which demonstrates Phantora’s ability to quickly evaluate a large scale workload. For example, Llama3 8B training with 128 GPUs takes around 15 seconds per iteration to simulate using Phantora, which means the user can easily estimate the training throughput of this workload in minutes.

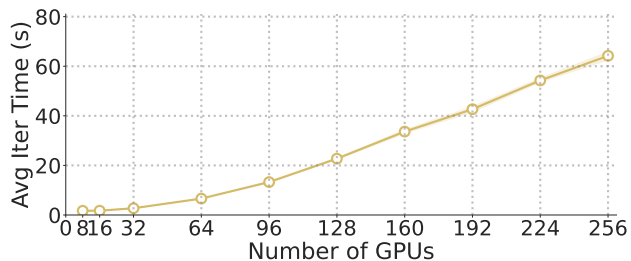


Figure 11: **Speed of Phantora:** Phantora simulation time of Llama2 7B training (Megatron, TP=8) using 32 CPU cores on H200 testbed.

Table 1 shows the simulation speed of Phantora in an actual training workload on the H200 testbed. The simulation time remains at the same level as the actual training time, and is significantly shorter than SimAI. This difference is mainly because Phantora uses a flow-level network simulator, while SimAI uses a packet-level network simulator.

Figure 11 shows the Phantora’s simulation speed on Llama2 7B training using Megatron with different numbers of GPUs. The training uses data parallelism over tensor parallelism, where tensor parallel size is fixed to 8. The batch size per GPU is fixed to 1. Figure 11 shows that the simulation time increases linearly with respect to the workload scale when the number of GPUs is greater than 100, which is expected given fixed 32 CPU cores. If we set a limit of 1 minute per iteration, Phantora can simulate approximately 240 GPUs under this setting. Note that Phantora simulator is still single-threaded with one dedicated CPU core, and Megatron containers are placed on the other 31 cores.

CPU memory usage. Sometimes users may choose to load or initialize a full model instead of a corresponding shard on each rank, especially when using a framework like DeepSpeed [34], which transparently and automatically shard all models. In this case, as discussed in §4.3, Phantora implemented model parameter sharing on CPU memory to overcome scalability limitation. Figure 12 shows the peak memory usage over different simulation scales: Without parameter sharing, 256GB

DP	TP	batch	Testbed	Phantora	SimAI
1	4	1	0.30s	0.91s	56.9s
1	4	2	0.54s	0.93s	63.4s
2	2	1	0.52s	0.96s	117.7s

Table 1: **Speed of Phantora (small scale):** Average time per iteration of Llama2 7B training using Megatron on H200 testbed; Average Phantora running time per iteration and SimAI running time for one iteration. Both Phantora and SimAI can use at most 16 cores. Note that SimAI uses packet-level network simulation while Phantora uses flow-level network simulation.

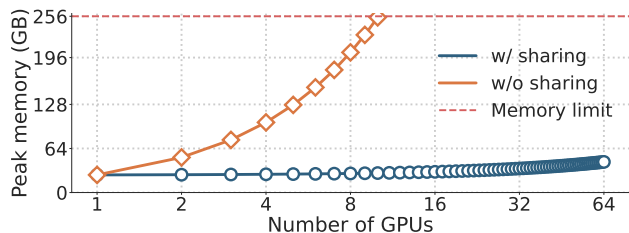


Figure 12: **CPU memory usage:** Peak CPU memory usage of Llama2 7B training simulation in DeepSpeed with or without model parameter sharing.

of CPU memory can only support 9 GPUs, while 64 GPUs only need less than 64GB of memory with parameter sharing. This greatly improves the scalability of Phantora without forcing users to change the training script.

5.4 Case Study: Activation Recomputation

Selective activation recomputation [18] is a technique to save GPU memory in large scale training by discarding certain intermediate activations in the forward pass and recomputing them in the backward pass. It is widely supported in many ML frameworks like Megatron [37], DeepSpeed [34] and TorchTitan [21]. However, to the best of our knowledge, no other existing simulator can simultaneously analyze its effect on throughput and memory usage, because the existing simulators have not yet fully reimplemented this feature. Meanwhile, Phantora can natively support this feature without implementing any specific logic related to selective activation recomputation.

Figure 13 shows the simulated peak memory usage and throughput of Llama2 training on 64 H100 GPUs, comparing activation recomputation with normal training with gradient accumulation—a more commonly used technique to achieve larger global batch size without increasing micro batch size. Our simulation results largely align with the results in the original selective activation recomputation paper [18]: *selective activation recomputation greatly reduces memory usage without introducing significant throughput overheads*. This experiment has shown that Phantora can be used to assess

the performance of new training optimizations without any redundant reimplementing effort for simulator developers.

6 Discussion

General support of custom kernels. As detailed in §4.1, Phantora can support custom kernel extensions in an ad-hoc manner but still requires extra engineering effort. JIT-compiled kernels, in principle, could be addressed by intercepting the JIT compilation process. Phantora could capture the compiled artifact for profiling, or capture the intermediate representation and utilize the simulator’s runtime to trigger the compilation. We leave this direction for future work.

Value-dependent performance in ML frameworks. Like other ML system simulators, Phantora currently cannot precisely reflect some value-dependent performance characteristics. One key example is expert parallelism [20], where performance depends on the distributions of activated experts. Phantora can simulate expert parallelism under the assumption of perfect load balance, but it does not model the performance overheads caused by expert imbalance. Another example is reinforcement learning (RL) for LLMs, where the performance depends on the generation length. We believe this limitation can be addressed through an annotation interface that allows users to specify distributions of certain values (e.g., activated expert indices, LLM generation lengths). With these distributions, Phantora can allow users to further estimate the performance of their system under different scenarios. We leave this direction to future work.

GPU itself can also have value-dependent performance of a single operation—for instance, the performance of sparse or quantized tensor operations may vary on different tensor values. This could also be addressed through a similar annotation interface where a desired sparsity or quantization level is provided by the user.

Improving Phantora’s accuracy. Our main goal is maximizing ML framework code reuse, not to improve simulation accuracy beyond existing static workload simulations. Similar to static workload simulations, Phantora uses standard discrete event simulation, so other techniques improving simulation accuracy should also apply to Phantora. For example, the SimCCL library in SimAI [42] could replace the current simple ring algorithm in Phantora for more accurate network simulation of NCCL. As another option, Phantora could integrate a packet-level network simulator to capture finer-grained behaviors—such as adaptive routing, packet spraying and congestion control dynamics—though this would likely increase simulation time. However, because communication in ML workloads is typically dominated by massive, long-lived bulk transfers, a flow-level approximation is often already very close to packet-level results.

Improving Phantora’s speed. Similarly, Phantora can adopt

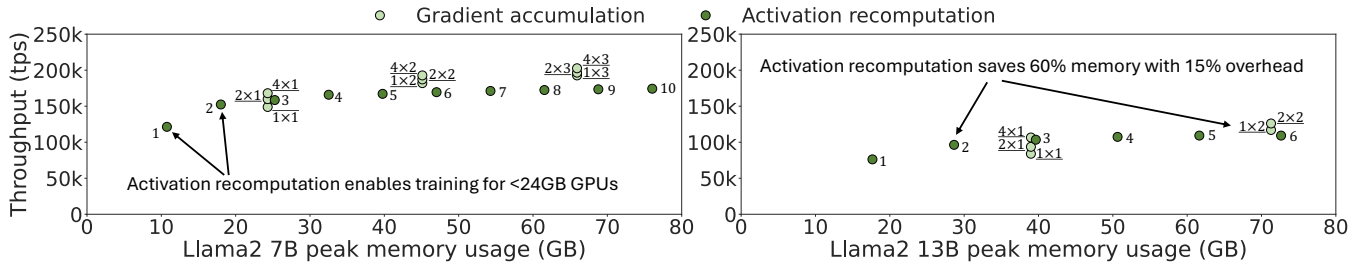


Figure 13: **Evaluating memory saving techniques:** Phantora estimated peak memory usage per GPU and throughput of Llama2 training with different memory saving techniques (64 H100 GPUs, Megatron, DP=8, TP=8). A single number n denotes the number of batches per GPU with activation recomputation. A pair in the form $m \times n$ denotes n batches per GPU with m gradient accumulation steps.

simulation performance enhancement techniques in existing static workload simulations. For example, Phantora simulator could parallelize profiling on multiple GPUs and adopt parallel discrete event simulation methods like UNISON [6] to leverage multiple CPU cores. Additionally, Phantora’s architecture could allow containers to be easily distributed on CPU-only clusters by changing the currently used inter-process communication to communication over network.

Support for heterogeneous environments. Phantora currently assumes a homogeneous cluster environment matching the hardware used for profiling. However, the architecture can be extended to support heterogeneous clusters (*e.g.*, clusters with mixed GPU models) by connecting the simulator to multiple distinct physical GPUs—one for each hardware model present in the simulated cluster. Alternatively, if a pre-populated performance estimation cache is available for the target devices, Phantora could simulate the cluster without requiring access to the corresponding hardware.

Non-independent computation/communication overlap performance. Overlapping communication with computation is a standard way to hide communication latency in distributed ML systems. However, this overlap could also slow down both operations as they share critical internal hardware resources [22]. Currently Phantora and other simulators do not consider this effect as it’s hard to simulate and impractical to profile for every possible overlap. An analytical model could help in the estimation, but we leave it for future work.

7 Related Works

Interception and Hybrid simulation. We are not the first to consider integrating an event-driven simulator with a real system. In the 90s, the Wisconsin Wind Tunnel (WWT) [28, 35] explored an execution-driven simulation approach to estimate performance of cache-coherent shared memory systems. WWT also encountered the problem of synchronizing simulated time with direct-execution. The approach adopted in WWT is to simulate every quantum, which is calculated from the minimum inter-core communication latency. In theory,

we could also build Phantora using this idea by carefully controlling the execution of containers (using interrupts to stop/resume container execution), so that containers’ times are synchronized with the simulator. However, this approach would make Phantora significantly slower. Another key difference between Phantora and execution-driven simulation from the computer architecture community [26] is that Phantora does not faithfully execute data operations. Phantora does not track the GPU buffer content and does not execute GPU computation (except profiling runtime) or communication.

Another effort is to enable event-driven network simulator with real systems. Early work [41] proposed a methodology for constructing high-fidelity TCP/IP simulators by re-entering the UNIX kernel, allowing the simulation to leverage the real-life TCP/IP protocol stack and existing network utilities. This concept evolved into modern network simulators like ns-3 [1], which could act as a special Linux network device. This allows Linux applications to run over an ns-3 simulated network. However, this approach introduces inaccuracies in performance estimation. ns-3 does not control the system time of the Linux environment in the same way Phantora controls the time of the ML system. As a result, if ns-3 takes 1 second to simulate 10 ms of network activity, the Linux application perceives the network as 100 times slower than it actually is.

The concept of intercepting calls to enable new functionality is well-established in systems research. For instance, Gandiva [46] intercepts GPU calls to enable fine-grained time-slicing and migration for deep learning scheduling. While Gandiva uses interception for cluster scheduling and fairness, Phantora employs similar interception mechanisms for the purpose of transparent performance estimation. By intercepting and emulating related library calls, Phantora enables users to estimate large-scale performance on a single GPU without modifying the underlying framework.

Predicting CUDA kernel execution time. Predicting a CUDA kernel’s execution time is a standard problem in ML compilers [9, 49]. The reason is that an ML compiler’s goal is to generate the most efficient CUDA kernel for ML operations. The approaches compilers take are usually ML-based

performance modeling, which eliminates the need to test every CUDA kernel’s performance on real hardware. For example, TVM [9] trains a gradient boosting decision tree by applying a set of manually-designed features on generated code. However, our problem of predicting CUDA kernel performance differs significantly from that of ML compilers. We focus on a limited set of kernels—those already selected by the ML systems (*e.g.*, from PyTorch, Megatron [37], DeepSpeed [34] and TorchTitan [21]). So, Phantora can use comprehensive runtime profiling of these CUDA kernels instead of other methods to estimate their performance.

Emulating the control plane and simulating the data plane. We draw inspirations from CrystalNet [23] for network testing. CrystalNet’s idea is to emulate a network environment for switch control programs without actually forwarding data. This allows CrystalNet to fully emulate large production networks on only tens of VMs/containers to find network bugs. Similarly, in Phantora, all the ML system code are running as is except the GPU operations and communication are simulated. Although in different contexts, both CrystalNet and Phantora rely on the assumption that the control flow does not have data dependency.

8 Conclusion

This paper introduces Phantora, a hybrid GPU cluster simulator for ML system performance estimation. Phantora runs unmodified ML models and frameworks, intercepting and simulating GPU- and network-related operations for high-fidelity performance estimation. It addresses key research challenges, including supporting unmodified ML frameworks, the integration of event-driven network simulators with real-time code execution, along with techniques to improve simulation scalability. Our evaluation shows that, Phantora achieves similar estimation accuracy to state-of-the-art static workload simulation methods. At the same time, Phantora’s design is general, supporting three state-of-art LLM training frameworks out-of-box. Further, Phantora only requires a single GPU, eliminating the need for the resource-intensive trace collection and workload extraction steps required by traditional trace-based simulators.

Acknowledgements

This work was supported in part by National Science Foundation grants CNS-2112562, CNS-2238665, CNS-2330333, CNS-2402696, and OAC-2503010, as well as by gifts from Amazon and Meta. This research was also supported by the National Science Foundation through the ACCESS program and AWS through the CloudBank project, which is supported by National Science Foundation grant 1925001. In addition, this work was supported in part by the Center for Ubiquitous Connectivity (CUbiC), sponsored by Semiconductor Research

Corporation (SRC) and Defense Advanced Research Projects Agency (DARPA) under the JUMP 2.0 program.

References

- [1] ns-3, A Discrete-event Network Simulator for Internet Systems. <https://www.nsnam.org/>, 2024. (Accessed on 09/19/2024).
- [2] Nsight Systems | NVIDIA Developer. <https://developer.nvidia.com/nsight-systems>, 2024. (Accessed on 09/19/2024).
- [3] Perfetto - System Profiling, App Tracing and Trace Analysis. <https://perfetto.dev/docs>, 2024. (Accessed on 01/29/2025).
- [4] TorchTitan A100 Performance Report. <https://github.com/pytorch/torchtitan/blob/217cc94e2abf8472db098c1c0e5e020e62dcfc7d/docs/performace.md>, 2024. (Accessed on 09/09/2025).
- [5] TorchTitan Training Script. <https://github.com/pytorch/torchtitan/blob/4b3f2e41a084bf79a8540068ed525539d1244edd/train.py>, 2024. (Accessed on 09/09/2025).
- [6] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. Unison: A Parallel-Efficient and User-Transparent Network Simulation Kernel. In *European Conference on Computer Systems (EuroSys)*, 2024.
- [7] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. vTrain: A Simulation Framework for Evaluating Cost-effective and Compute-optimal Large Language Model Training. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.
- [8] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey Chase, Ion Stoica, and Danyang Zhuo. NetHint: White-Box Networking for Multi-Tenant Data Centers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.
- [9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [10] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan

- Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. <https://lmsys.org/blog/2023-03-30-vicuna/>, 2023. (Accessed on 09/19/2024).
- [11] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.
- [12] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.
- [13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations (ICLR)*, 2021.
- [14] Fei Gui, Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Ran Zhang, Hongbing Yang, and Dian Xiong. Accelerating Design Space Exploration for {LLM} Training Systems with Multi-experiment Parallel Simulation. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2025.
- [15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE / CVF Computer Vision and Pattern Recognition Conference (CVPR)*, 2016.
- [16] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. In *Annual Conference on Machine Learning and Systems (MLSys)*, 2022.
- [17] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Annual Conference on Machine Learning and Systems (MLSys)*, 2019.
- [18] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing Activation Recomputation in Large Transformer Models. In *Annual Conference on Machine Learning and Systems (MLSys)*, 2023.
- [19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PageAttention. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2023.
- [20] Dmitry Lepikhin, Hyoungho Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations (ICLR)*, 2021.
- [21] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. TorchTitan: One-stop PyTorch Native Solution for Production Ready LLM Pre-training. In *International Conference on Learning Representations (ICLR)*, 2025.
- [22] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, 2024.
- [23] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully Emulating Large Production Networks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
- [24] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Li Li, Jingwen Leng, and Minyi Guo. DistSim: A Performance Model of Large-scale Hybrid Distributed DNN Training. In *ACM International Conference on Computing Frontiers (CF)*, 2023.
- [25] Liang Luo, Buyun Zhang, Michael Tsang, Yinbin Ma, Ching-Hsiang Chu, Yuxin Chen, Shen Li, Yuchen Hao, Yanli Zhao, Guna Lakshminarayanan, Ellie Wen, Jongsoo Park, Dheevatsa Mudigere, and Maxim Naumov. Disaggregated Multi-Tower: Topology-aware Modeling Technique for Efficient Large Scale Recommendation. In *Annual Conference on Machine Learning and Systems (MLSys)*, 2024.
- [26] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

- [27] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhat-tacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models. In *International Symposium on Computer Architecture (ISCA)*, 2022.
- [28] S.S. Mukherjee, S.K. Reinhardt, B. Falsafi, M. Litzkow, M.D. Hill, D.A. Wood, S. Huss-Lederman, and J.R. Larus. Wisconsin wind tunnel ii: a fast, portable parallel architecture simulator. *IEEE Concurrency*, 8, 2000.
- [29] OpenAI. DALL·E 3. <https://openai.com/index/dall-e-3>, 2024. (Accessed on 09/19/2024).
- [30] OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2024.
- [31] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [32] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2020.
- [33] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.
- [34] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2020.
- [35] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The wisconsin wind tunnel: virtual prototyping of parallel computers. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1993.
- [36] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE / CVF Computer Vision and Pattern Recognition Conference (CVPR)*, 2022.
- [37] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [38] Srinivas Sridharan, Taekyung Heo, Louis Feng, Zhaodong Wang, Matt Bergeron, Wenyin Fu, Shengbao Zheng, Brian Coutinho, Saeed Rashidi, Changhai Man, and Tushar Krishna. Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces. *arXiv preprint arXiv:2305.14516*, 2023.
- [39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shrutvi Bhosale, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.
- [40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *International Conference on Learning Representations (ICLR)*, 2018.
- [41] S.Y. Wang and H.T. Kung. A simple methodology for constructing extensible and high-fidelity TCP/IP network simulators. In *IEEE Conference on Computer Communications (INFOCOM)*, 1999.
- [42] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large Language Model Training with Scalability and Precision. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2025.
- [43] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 2009.

- [44] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.
- [45] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud. In *SIGCOMM Conference on Data Communication*, 2024.
- [46] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [47] Buyun Zhang, Liang Luo, Yuxin Chen, Jade Nie, Xi Liu, Daifeng Guo, Yanli Zhao, Shen Li, Yuchen Hao, Yantao Yao, Guna Lakshminarayanan, Ellie Dingqiao Wen, Jongsoo Park, Maxim Naumov, and Wenlin Chen. Wukong: Towards a Scaling Law for Large-Scale Recommendation. In *International Conference on Machine Learning (ICML)*, 2024.
- [48] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv preprint arXiv:2304.11277*, 2023.
- [49] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [50] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.
- [51] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *USENIX Annual Technical Conference (ATC)*, 2020.

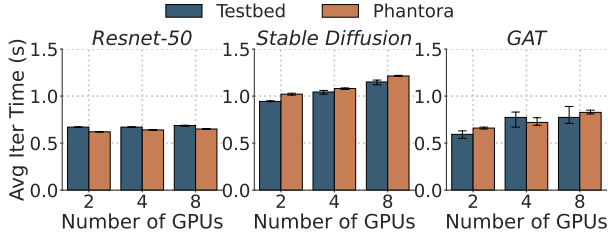


Figure 14: **Non-LLM models:** Testbed training time and Phantora’s simulation results with DeepSpeed. The error bars show 95% confidence interval.

A Evaluation for Non-LLM Workloads

Phantora’s design does not depend on any particular model architectures. Here we evaluate Phantora for non-LLM workloads.

We use a single NVIDIA RTX 3090 GPU for computation kernel profiling. The goal is to match the actual performance of our 4-server testbed where each server has 2 Intel Xeon Gold 5215 CPUs and 2 NVIDIA RTX 3090 GPUs. There are 8 NVIDIA RTX 3090 GPUs in total. The network topology and bandwidth of this testbed is the input to our network simulator. We evaluate a broad set of models on DeepSpeed [34] that are known to have different performance characteristics, including ResNet-50 [15], Stable Diffusion [36], and Graph Attention Network (GAT) [40].

Phantora achieves accurate simulation accuracy. Figure 14 compares Phantora’s predicted training time per iteration and growth truth performance numbers collected on the testbed. the average simulation error is 6.6% under these settings with the maximum error of 8.1% on diffusion model with 2 GPUs.