Phantora: Live GPU Cluster Simulation for Machine Learning System Performance Estimation

Jianxing Qin Jingrong Chen Xinhao Kong Yongji Wu[‡] Liang Luo[†]
Zhaodong Wang[†] Ying Zhang[†] Tingjun Chen Alvin R. Lebeck Danyang Zhuo
Duke University [†]*Meta* [‡]*University of California - Berkeley*

ABSTRACT

To accommodate ever-increasing model complexity, modern machine learning (ML) systems have to scale to large GPU clusters. Changes in ML model architecture, ML system implementation, and cluster configuration can significantly affect overall ML system performance. However, quantifying the performance impact before deployment is challenging. Existing performance estimation methods use performance modeling or static workload simulation. These techniques are not general: they requires significant human effort and computation capacity to generate training data or a workload. It is also difficult to adapt ML systems to use these techniques. This paper introduces, Phantora, a live GPU cluster simulator for performance estimation. Phantora runs minimally modified ML models and frameworks, intercepting and simulating GPU-related operations to enable high-fidelity performance estimation. Phantora overcomes several research challenges in integrating an event-driven network simulator with live system execution, and introduces a set of techniques to improve simulation speed, scalability, and accuracy. Our evaluation results show that Phantora can deliver similar estimation accuracy to the state-of-the-art workload simulation approach with only one GPU, while reducing human effort and increasing generalizability.

1 INTRODUCTION

Large machine learning (ML) models have become a driving force behind advancements in natural language processing [12, 32, 40], computer vision [15], computer graphics [31] and recommendation systems [28, 29, 47]. As models grow increasingly complex, high-performance model inference [21] and training [51] have become the main focus in the ML system community. Recent advancements in the field span from efficient GPU kernel optimizations [13, 14, 22], to parallelization strategies [51], and scheduling algorithms [34]. 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.

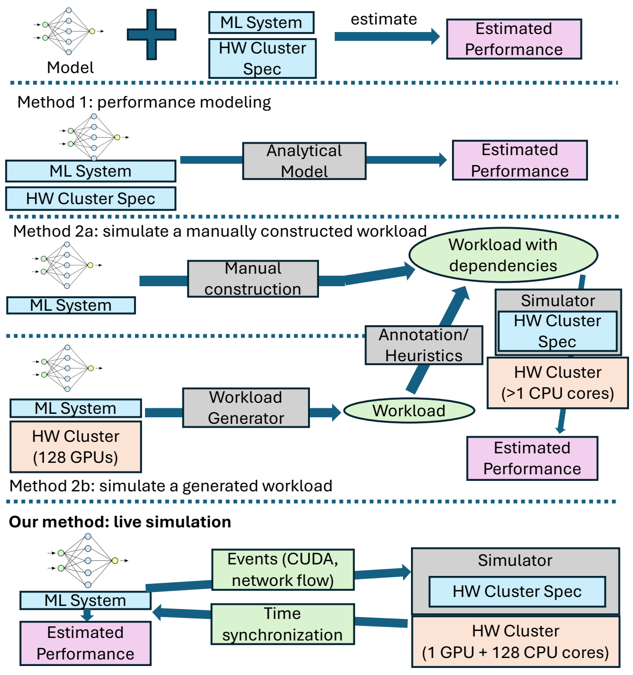


Figure 1: Comparing performance estimation methods for a 128-GPU training job. Grey boxes represent system components that require manual effort to build. Orange boxes represent components requiring substantial computation capacity.

There have been a growing number of performance estimation methods for ML systems. Figure 1 shows the existing methods. One method is to use an analytical performance model, such as a roofline [33, 43, 52] or a learned model [7, 27, 46]. Once such a model is built or trained, it is very fast to perform performance estimation. However, static models could require significant computational resources to train and can easily overfit to specific hardware or software environments, limiting their generalizability.

The second method is to estimate performance by simulating a static workload. The workload can either be manually constructed by an expert familiar with the ML system, or extracted with a workload generator or a tracer. The workload (or execution trace) is usually a directed acyclic graph (DAG) that captures the computation and communication task dependencies. This method has demonstrated high estimation accuracy [17, 19, 26, 35, 44, 52]. However, this approach has several limitations regarding generalizability: (1) Manually written workloads are susceptible to errors and are difficult to maintain over time. (2) Generating workloads often requires substantial resources, including access to an actual GPU cluster for tracing. (3) Creating an accurate global timeline through distributed tracing is difficult [17]. (4) Capturing the dependencies between operations in tracing is challenging and typically requires manual annotations or heuristics [17, 26, 52].

In this paper, we explore this question: Can performance estimation be made general? Our goal is to provide developers with the illusion of access to a large GPU cluster. From the user experience perspective, when running an ML system, developers should observe identical terminal output—aside from training loss—regardless of whether they are using real or simulated hardware. They should also be able to insert print statements anywhere in the code to debug performance without restrictions. This makes it broadly applicable across various ML systems without the need for manually writing workloads, annotating dependencies, designing heuristics or analyzing performance explicitly. Furthermore, when modifying the ML system, such as applying new parallelization strategies or updating models, the system should run immediately without requiring the developer to re-collect traces from an actual GPU cluster.

We present a new performance estimation approach for ML systems, called *live simulation*. Our key idea is that we can *integrate real system execution with event-driven simulation, creating an illusion for applications (e.g., PyTorch) as if they are running on a real GPU cluster.* We draw inspiration from CrystalNet [25]—Microsoft's network testing system that emulates the network control plane using containers.

We design and implement Phantora, a live GPU cluster simulator for ML systems. The core design is to use a containerized cluster, which is equipped with a single GPU and a few CPU cores to directly run an ML system for simulation. Each container running on the cluster simulates the execution of a GPU server. Phantora intercepts all the communication and GPU kernel execution from the ML framework, and it provides one clock for each container. CUDA kernel execution times are profiled on the single GPU, while communication execution times are calculated using an event-driven network simulator. When a container launches a CUDA kernel or initiates communication, Phantora adjusts the container's clock accordingly to maintain accurate simulation.

We still face two key research challenges. First, it is challenging to integrate a real, running system with an event-driven network simulator. 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 live simulation, our ML system (the set of containers) may inject a network flow at any time, causing the network simulator's calculation to be incorrect. Hardware simulators [30, 37] address this issue by enforcing tight time synchronization across system components, but this results in high simulation performance overhead. Second, running a large number of containers on a small set of physical servers quickly exhausts the CPU and memory capacity of the physical servers, limiting Phantora's scalability.

Phantora addresses the integration challenge by allowing an event-driven network simulator and a set of containers to run in a loosely synchronized manner. The time in the containers and the network simulator is synchronized only when applications explicitly synchronize or read time from Phantora. When an event that should occur earlier is injected by the container to the network simulator, Phantora rollbacks the simulator state and accommodates those *occurred events* triggered by the containers. This design reduces the overhead and enables faster simulation while ensuring simulation accuracy. To enhance scalability, Phantora allows containers to *share memory*, significantly reducing the simulator's memory footprint. Additionally, Phantora can *scale across distributed servers*, enabling the simulation of larger GPU clusters.

We evaluate Phantora using two state-of-the-art Large Language Model (LLM) training systems: DeepSpeed [36] and torchtitan [6]. Porting these systems to run on top of Phantora requires minimal manual effort, described in §6.1. The terminal output of each system remains identical to that produced when running on a real GPU cluster, enabling us to directly capture their reported performance metrics without the need of writing performance analysis code for each ML system. Our evaluation also demonstrates that Phantora achieves simulation accuracy comparable to static workload simulation for both LLM training systems.

This paper makes the following contributions:

- We are the first to propose *live GPU cluster simulation for ML systems*, which enables accurate and general ML system performance estimation while eliminating the need for workload generation and manual annotation/heuristics for dependency tracking.
- We design and implement Phantora, which efficiently integrates real system execution with an event-driven simulation to enable live simulation.

- We evaluate Phantora on two state-of-the-art LLM training systems, DeepSpeed and torchtitan, demonstrating Phantora's accuracy and generality.

2 BACKGROUND

Performance estimation is useful for ML system developers and infrastructure providers. For an ML system developer, it allows for quick estimation of the performance of the ML system that they are building. Many parameters in ML systems require tuning as model sizes grow. For example, it is important to choose an appropriate parallelization strategy [38, 51]. Thus, it is appealing to estimate the performance of different parallelization strategies in order to pick the most efficient one. For an infrastructure provider, performance estimation is critical for projecting future hardware deployment as their focus is identifying fundamental performance bottlenecks instead of performance on existing hardware.

Existing method #1: performance modeling. One straightforward approach is to build an analytical performance model. The roofline model [43] is widely used, which estimates execution time by $T_{\text{roofline}} = \max\left(\frac{\text{num\_flops}}{\text{flops}}, \frac{\text{mem\_access}}{\text{mem\_bw}}, \frac{\text{comm\_size}}{\text{comm\_bw}}\right)$. Here, flops and mem_bw are determined by the GPU, and comm_bw is determined by the GPU interconnect (i.e., Ethernet/Infiniband, NVLink). The assumption is that the latency of a training iteration or model inference is capped by one of the computation, memory bandwidth, or the communication capabilities of the GPU cluster. The roofline approach applies to any type of job that uses the GPU cluster. Its calculation is very simple when num_flops, mem_access, and comm_size are known for the job. However, roofline models are not accurate, because they ignore important factors such as CUDA kernel design, collective communication strategies, and CPU overheads. For example, the roofline model assumes at least one of the resources is always fully utilized, which is rarely the case for model training or inference.

A more advanced performance modeling approach is to use ML to estimate ML system performance [20, 27, 34, 49]. However, even assuming an accurate performance estimation model can be trained, this ML-based approach still has two key problems. First, it requires the collection of representative training data, which can be difficult to obtain. For example, training a model to estimate performance on a 10,000-GPU system is nearly impossible without training data from such a large-scale cluster. Second, even a highly accurate ML model lacks interpretability. This makes it difficult to identify performance bottlenecks in the system or generalize the model to estimate performance on future hardware.

Existing method #2: static workload simulation. Due to the limitations of performance modeling, developers have turned to static workload simulation for performance estimation. Workload simulators [17, 19, 26, 35, 44, 52] take a description of workload, and outputs estimated performance given a GPU cluster specification. Workloads are generated via manual construction, or tracing an actual execution on a real GPU cluster. Standardization of workload representation is an ongoing effort among several large companies. This approach leads to accurate performance estimation and is increasingly adopted by industry [39].

Limitations of static workload simulation. While static workload simulation can provide accurate performance estimations, it faces significant limitations in terms of generality. The primary challenge lies in workload generation. Manually crafting workloads based on an understanding of the ML system is error-prone and difficult to maintain over time. Alternatively, directly running the ML system and using tracing to produce a distributed execution trace demands substantial hardware resources. Collecting precise traces with a global timeline in a distributed system is inherently challenging due to the need for accurate timelines of distributed events, where time synchronization becomes problematic [17]. Additionally, creating an distributed workload with accurate global dependencies often requires manual annotations or heuristics [17, 26, 52]. These heuristics are usually designed by experts based on characteristics of specific workloads, which lacks generality if new parallelization strategies are introduced. Regardless of the trace generation method—be it manual or direct tracing on a real GPU cluster—static workload simulation requires substantial modifications or customizations to ML systems to support performance estimation. This includes generating traces and identifying which operations belong to the main training loop and operations' dependencies, etc. Consequently, it is difficult to generalize this approach across different ML systems.

Our method: Live simulation. Our objective is straightforward: we would like to enable an ML system operates as if it is running on a large GPU cluster on a simulator. The ML system should output to terminal exactly as it would on real GPU clusters except for numbers related to tensor values. We thus directly read their logs to get ML system performance. This approach offers the most general user experience, mirroring the process of tuning performance on a real GPU cluster. Once the user changes the ML system or adds more logging, our system can simply rerun without the need to regenerate the workload or modifying the simulator. With this vision, we inherently eliminate all the limitations of static workload simulation discussed above. Additionally, this method naturally captures dynamic behaviors within ML systems, such as PyTorch's caching allocator. However, implementing this approach poses significant challenges, which we will address in the next section.

3 OVERVIEW

Phantora runs an ML system in a realistic containerized environment and interacts with this real system for accurate simulation (§4.1). Each container in the environment acts as a GPU server. For example, for PyTorch, each container runs Python interpreters that execute PyTorch code. There are two operations that require interaction between the containers and the Phantora simulator: (1) GPU computation and (2) communication. For GPU computation invocation, Phantora uses a single GPU to profile the kernel's execution time and advances *the clock of the container.* For communication, Phantora utilizes an event-driven flow-level network simulator to estimate completion time. Our approach can be conceptually viewed from the container as replacing GPU computation and communication calls with `sleep(exec_time)`, where `exec_time` is the estimated execution time of corresponding operation.

Applications running on Phantora only require a few lines of code modification (§6.1) to ensure that when the application reads the system clock, it retrieves the time provided by Phantora instead. Phantora only needs a single GPU because it only needs to profile the performance (i.e., execution time) once for each encountered (CUDA kernel, tensor shape) combination, which is limited within an ML system. CUDA kernel performance usually does not depend on the actual tensor values.[1]

In this way, an application cannot distinguish whether it is running on Phantora or a physical GPU cluster as long as it does not check the tensor values (which would be junk values on Phantora). Containers' time will be maintained by Phantora, and the application can read the time for calculating its performance.

There are two main technical challenges. The first challenge is *how can we enable the real system (i.e., the containers) to effectively interact with the event-driven simulator?* 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 [30, 37]. 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 (§4.2). To ensure simulation accuracy, we implement an event-driven network simulator capable of time traveling to rollback the states of all flows when events occur in the past. Phantora enforces strict synchronization only when applications explicitly synchronize or read time from Phantora, ensuring that the simulation's output is final and never subject to rollback.

The second challenge is *how to achieve scalable simulation?* While a physical server can easily support tens of containers running concurrently for general tasks, this is not the case for containers involved in ML training or inference. For example, in large language model (LLM) training, each GPU server must load training data, which consumes significant host memory. Our evaluation (§6.3) shows that a physical server with 256G host memory can only host 9 containers for simulating the training of Llama 7B models.

We apply two techniques to scale Phantora (described in §4.3). First, we introduce a novel memory sharing mechanism for different containers running on the same host. which significantly reduces the memory footprint. Second, Phantora supports distributed execution to scale out the simulation across many physical servers.

4 PHANTORA

We now present how Phantora addresses the aforementioned challenges and enables live simulation for ML systems. We first introduce the system architecture of Phantora and provide an overview of the workflow. We then dive into three key designs that tackle the corresponding challenges.

4.1 Phantora Architecture and Workflow

Figure 2 shows Phantora's software architecture. Phantora consists of two core components, the Phantora simulator and a set of containers. Phantora has a dedicated event queue (distinct from the network simulator's event queue) that synchronizes time across all containers. This queue orchestrates execution time calculations within both the network simulator and the computation simulator, which profiles CUDA kernel executions on the GPU.

Each container runs real ML applications, using PyTorch and NCCL libraries, and interacts with the CUDA Runtime. We implement Phantora Tracer in the PyTorch. This tracer collects all invoked PyTorch operators and the corresponding performance-related parameters, and pushes all information

[1]There are some exceptions, which we discuss in §7. One such example is that multiplication with an all-zero tensor is faster than multiplication with a non-zero tensor.

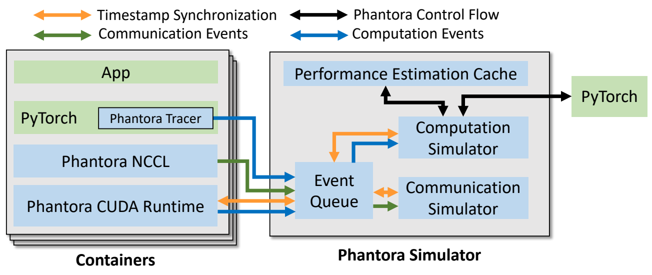


Figure 2: Phantora architecture. Components in green are minimally unmodified native components, and components in blue are constructed or modified by Phantora.

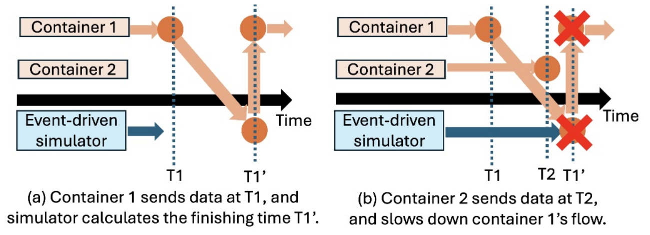

Container 1
Container 2
Event-driven simulator
Time
T1
T1'
(a) Container 1 sends data at T1, and simulator calculates the finishing time T1'.
Container 1
Container 2
Event-driven simulator
Time
T1
T2
T1'
(b) Container 2 sends data at T2, and slows down container 1's flow.

Figure 3: Challenges of synchronizing time between real execution and event-driven simulation.

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 computation 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, it records current GPU memory usage and reports an error when OOM occurs. In addition, it pushes CUDA calls as another type of computation events to the event queue in Phantora simulator. Similarly, we replace the native NCCL libraries with Phantora NCCL libraries. Phantora NCCL libraries do not initiate communication, and forward all communication operations (*e.g.*, allreduce) to the simulator by pushing communication events to the event queues. Furthermore, Phantora has to maintain a dependency graph of events because of CUDA's asynchronous nature, i.e. CPU launches computation and communication kernels and specify dependencies through streams/events.

Phantora computation simulator processes these computation events, including the PyTorch operators and CUDA calls. It invokes native PyTorch libraries and interacts with the GPU to profile these computational operators. Phantora uses a performance estimation cache to store the performance results of operators that have been already faithfully executed. When invoking the same operators in the future, Phantora directly use results stored in the cache.

To accurately model the execution time of collective communication operations, Phantora adopts a standard flow-level network simulator used in NetHint [10], referred to as `netsim`. `netsim` 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 received communication operators, Phantora submits the corresponding data transfer flows to `netsim` with the appropriate timestamps. `netsim` then simulates network behavior and computes the completion time of each flow based on network congestion and available bandwidth [10, 45].

All computation and communication operations in Phantora are represented as event spans in the event queue with explicit dependencies to enforce correct execution order. By integrating `netsim` into Phantora, we can model the interplay between computation and communication, accurately capturing their overlap.

Time synchronization is enforced through the the interaction between a container and Phantora simulator. There are specific CUDA calls that require synchronization (*e.g.*, `cudaStreamSynchronize`). When such a event happens, the Phantora CUDA runtime pushes a synchronization event to the event queue and starts to wait for a response. After processing the preceding messages in the event queue and completing this synchronization event, the simulator returns a response, including a completion time (a logical timestamp), to the the container's CUDA runtime. The container's local clock is then updated based on this completion time. When a *past event* situation occurs, the container's CUDA runtime corrects its local clock using the rollback mechanism we describe next, ensuring accurate simulation.

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 3 demonstrates a typical workload that requires correct synchronization. Container 1 and 2 independently launch communications that may share network resources. At time T_1, container 1 begins to wait for its communication

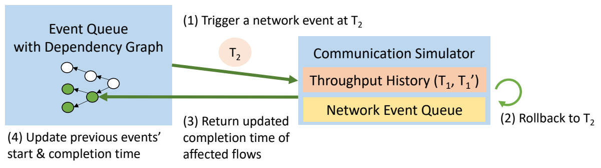

Event Queue with Dependency Graph
(1) Trigger a network event at T_2
T_2
Communication Simulator
Throughput History (T_1, T_1')
Network Event Queue
(2) Rollback to T_2
(3) Return updated completion time of affected flows
(4) Update previous events' start & completion time

Figure 4: Handing 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'.

and asks the simulator for the completion time. The simulator, however, cannot directly proceed to the completion point based on current information and response T_1', as a future communication from container 2 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 network simulators, workload/trace analysis). For static workload, the simulator can directly compare T_2 and T_1', move the simulated time forward to whichever is less, and update the state of the system. However, in our cases because container 2 is a real execution, the simulator does not know whether or when it will launch communication.

One approach in 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 [37] uses such an approach for simulating multi-core hardware, 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 most of today's network simulators use an event-driven approach.

Our key observation is that the characteristics of ML systems provide an opportunity to change the running time of operations during live simulation. We assume that the time consumption of a given operation (*e.g.*, a specific CUDA kernel invocation) does not affect the actual control flow of the real system. For example, changing the time consumed by a matrix multiplication in a container's runtime does not affect which instruction is the next to run for the container. 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 container's runtime in Phantora. When past events occur, the Phantora simulator rollbacks to a prior time, processes the past events, corrects time for a set of events in the past, updates the clock with each container'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 3. A new event comes at a time T_2 earlier than the current time T_1'. For simplicity, let's assume there are no events other than T_2 between T_1 and T_1' and let $S(T)$ denote the state of all the network flows at time T. The network simulator can compute the state of the flows $S(T_2)$ 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 a compute event previously started at T_1' because it depends on this communication, its start and completion time will be adjusted accordingly.

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. Garbage collection is thus a necessary component. The key insight is that after all the containers' 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 garbage collected.

When we have to force synchronization between the event-driven simulator and the containers? At some point, the application running on Phantora has to output. For example, the performance measurement code shown in Figure 6. Once an output of Phantora is generated to the application (i.e., PyTorch), there is no way we can rollback, because application may have behavior that depends on Phantora's output. Hence Phantora will force a synchronization before the application fetch time from Phantora. This means, if Phantora outputs time T to the application, then Phantora has made sure that all the containers' time and the simulator time has passed T. This ensures that no past events can be generated to affect the output anymore.

4.3 Improving Phantora's Scalability

Model parameter sharing on CPU. ML systems typically 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.

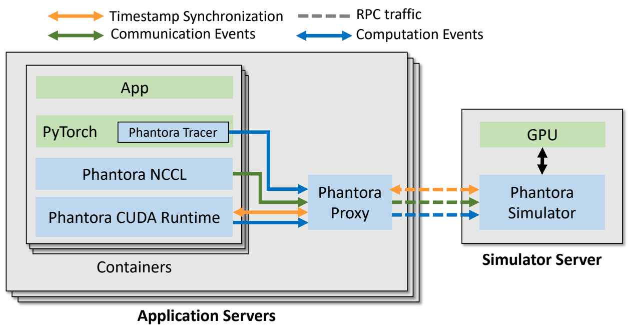


Figure 5: Distributed runtime in Phantora. Solid lines are communication through IPC on a single server; Dashed lines are communication through RPC between application servers and simulator server.

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 only a few hardware servers, this peak memory usage suddenly become a scalability constraint. As shown in §6.4, a server with 256GB of memory can initialize only 8-10 instances of Llama 7B simultaneously. To address this limitation, Phantora implements parameter sharing, which allows model parameters on the same server to be transparently mapped to the 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.

Distributed runtime. Similar to memory, CPU can also become a bottleneck for Phantora. Existing ML training systems usually launch one or more processes per GPU. To faithfully execute these processes on CPU, we need at least as many CPU cores as the number of the processes launched. In practice, we observe that if the application is launched on a server with limited number of CPU cores, the CPU oversubscription occurs. This can slow down the execution of the ML systems and cause inaccuracies in simulation results, limiting the size of GPU cluster we can simulate with a single GPU. To scale Phantora, we introduce a distributed runtime as shown in Figure 5. Instead of collocating all application containers on the same GPU server, we enable application containers to run on remote CPU-only servers. Each container will push events to and receive responses from the remote simulator (running on the GPU server) through Phantora Proxy. This ensures that each container receives sufficient CPU cycles, mirroring the environment of a real GPU cluster where the CPU is usually not a limiting factor.

4.4 Improving Phantora's Accuracy for CUDA Kernel Performance Estimation

As we build out Phantora, an important challenge is how to accurately estimate CUDA kernel performance. One naive approach is to profile standalone kernels, and reuse the profiled results for future invocation of the same CUDA kernel. Unfortunately, we found this to be insufficient. The data dependencies between operators can affect whether the tensor arguments are in the cache, which then affects the performance of the kernel execution, especially for memory bounded kernels like matrix additions [1, 42].

However, reconstructing global contexts for replaying each kernel is costly and inefficient. We empirically found that the following simple strategies are highly effective:

- **Delimiting the context.** Phantora profiles each standalone kernel once, combining these running times with their launch time to determine which kernels are effectively queued on a same CUDA stream (previous launch time + previous running time ≤ next launch time). This continuous sequence of kernels is profiled as a whole, and the profiling results are reused later for the same sequence.
- **Approximating data dependencies within a context.** During simulation, Phantora maintains a shape-indexed LRU cache of output tensors. This cache is queried first for allocation of tensors as kernel arguments to try reusing previous output tensors. In that way Phantora can approximate the data dependencies between operators.

5 IMPLEMENTATION

Our implementation consists of 9K lines of Rust, 8K lines of C, and 500 lines of C++: 8K lines of C 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 communication between the simulator and containers, 1K lines of Rust for the computation simulator, and 500 lines of C++ for Phantora Tracer in PyTorch.

Intercepting CUDA kernel invocations and communication. One key design decision we have to make is how to intercept CUDA kernel invocations and communications. Consider a strawman solution that intercepts ML system execution at the CUDA kernel level, which is widely used in GPU profiling systems such as Nsight Systems [3]. We could intercept `cudaLaunchKernel` API, where only the pointers to the parameters are provided. But this is not enough for us to inspect the shape of the tensors involved.

Hence, we resort to a hybrid approach where computation operations are intercepted at ML systems API level (*e.g.*, PyTorch operators), while the communication operators are intercepted at collective communication libraries level (*e.g.*, NCCL calls). This hybrid approach allows Phantora to disentangle computation and communication, delegating communication to the network simulator. For computation operations, Phantora is aware of the type of the operations and the shapes of the tensors involved. Phantora implements a cache manager to reuse earlier profiling results of an operation with the same input and output shapes, eliminating redundant execution.

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 simulator 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 inexpensive compared to solving the max-min fair flow allocation problem.

Communication between containers and simulator. When the simulator is on the same hardware server of the containers, Phantora Tracer and Phantora libraries connect to Phantora simulator via Unix domain sockets. In distributed runtime, Phantora Tracer and Phantora libraries still use Unix domain sockets to connect to Phantora Proxy on each CPU server. This proxy then forward events to the remote simulator using RPC. Phantora uses tonic [5] as its RPC client and server implementation. We choose this proxy-based architecture, rather then having each client directly connect to the simulator server, to minimize modifications to PyTorch and Phantora libraries.

6 EVALUATION

We evaluate Phantora on two aspects. First, we test Phantora's generality in supporting different ML systems, and their dynamic behaviors. Second, we test a set of standard metrics for simulators, such as accuracy, simulation speed and scalability. For all the experiments in this section, we run each experiment 10 times and show 95% confidence intervals.

Software/Hardware	Model	# params	AC	batch/GPU
torchtitan/A100	Llama 2	13B	partial	2
	Llama 2	13B	no	2
	Llama 2	70B	partial	1
	Llama 2	70B	full	2
	Llama 3	8B	partial	1
	Llama 3	70B	full	1
DeepSpeed/RTX 3090	Llama	1B	no	1
	ResNet-50	25M	no	128
	Diffusion	866M	no	1
	GAT	106M	no	1

Table 1: Setup used in evaluation of Phantora. AC represents activation checkpointing policies in torchtitan [6], "partial" stands for selective op.

Our hardware testbed for simulator. We run Phantora on one and up to four servers in an on-premise GPU cluster, depending on the the scale of simulation. One server is equipped with 2 Intel Xeon Gold 6348 CPUs and a single NVIDIA A100 40G GPU, to mimic the documented test environment of torchtitan [6] with 64 NVIDIA A100 80G GPUs, and we use the corresponding network configuration in Phantora's network simulator. We use *A100 testbed* to reference this testbed. We also use a single NVIDIA RTX 3090 GPU for CUDA kernel profiling. The goal is to match the actual performance of our 4-server testbed with 2 Intel Xeon Gold 5215 CPUs and 2 NVIDIA RTX 3090 GPUs on each server. The network topology and bandwidth of this testbed is the input to our network simulator.

Models. On the A100 testbed, we evaluate Llama-series models for 8B, 13B, and 70B. Two key configurations we vary here are activation checkpointing policy and the batch size. These configurations are identical to the ones evaluated in torchtitan [6]. Table 1 shows the detailed model and configuration for the workload. On the RTX 3090 testbed, we evaluate a broad set of models on DeepSpeed [36] that are known to have different performance characteristics, including Llama, ResNet-50, Diffusion-based text-to-image models, and Graph Attention Network (GAT) [41].

6.1 Generalizability

Porting effort for ML systems. We manually port two LLM training systems torchtitan [6] and DeepSpeed [36] to Phantora. Porting other PyTorch based systems requires minimal manual effort: (1) NCCL setup validations (*e.g.*, running broadcast and checking if tensor values match on all ranks) should be disabled as communication in Phantora is not actually executed. (2) Timers in training script (*e.g.*, `time.time()` in Python) should be replaced with corresponding functions provided by Phantora to reflect the simulation results. In total, this requires no change in torchtitan, 4 lines of changes in

DeepSpeed and an average of 6 lines of changes per training script in our evaluation.

User experience of Phantora. With Phantora, we want users to be able to tune their ML system performance as if they are actually experimenting with a real GPU cluster. This means we need to support customized printing or logging mechanisms embedded in the ML systems.

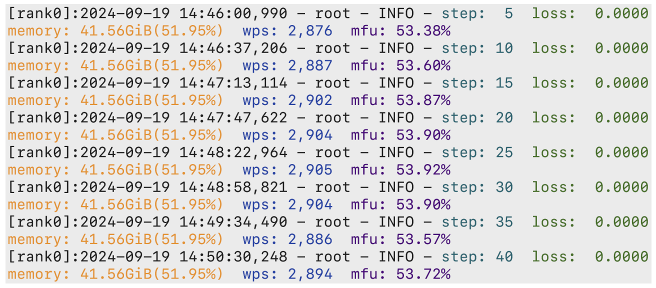
The top part of Figure 6 shows the performance measurement and logging code in torchtitan [6], which represents how torchtitan developers want to evaluate performance. With static workload simulation, porting such an ML system has to rewrite this code in the analyzer. In contrast, Phantora allows this 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 6 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 all, Phantora is just a simulator; it does not understand whether the application is running ML training or inference, nor does it understand what a training iteration even means.

After a developer updates either the ML system code (*e.g.*, changing parallelization strategies for LLM training) or the performance measurement and logging code, Phantora can immediately re-run and produce updated console output. For static workload simulation, the developer may have to collect additional traces, manually annotate operator dependencies, and perform other time-consuming tasks—all of which significantly slow down the development cycle.

Computation/communication overlap and other dynamic behaviors in ML systems. Phantora naturally captures computation/communication overlap and dynamic behaviors in ML systems. Figure 7 shows the timline of Phantora executing torchtitan, where x-axis is simulated time. As shown in the figure, the NCCL operations (communication) overlap with matrix multiplication (computation).

Phantora can also naturally capture dynamic behavior of PyTorch caching allocator. Figure 8 shows Llama 1B training for a parallelization strategy that will cause RTX 3090 to trigger an OOM. The simulated execution exhibits almost identical behavior to the real execution. Both trigger the OOM when batch size is 4 and GPU memory utilization is around 22-23G. 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 PyTorch caching allocator. Phantora still has slight imprecision, as it does not know the fragmentation below CUDA APIs, i.e., the implementation of `cudaMalloc` and `cudaFree` in NVIDIA CUDA Toolkit and driver.

```
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):,}  "
    f"mfu: {mfu:.2f}%{color.reset}"
    ...
)
```



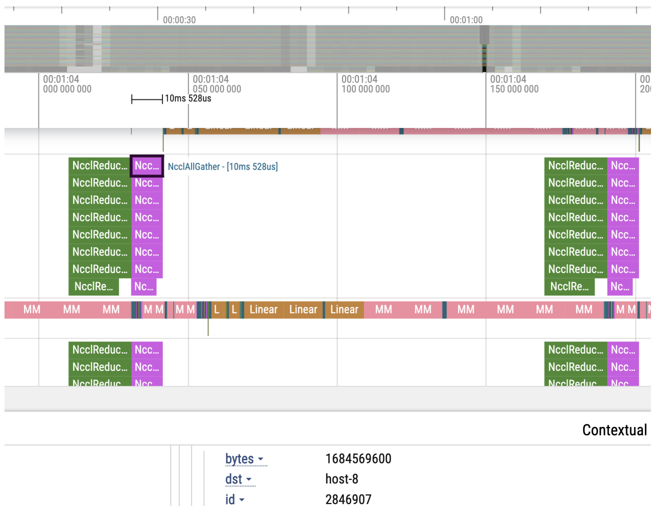
```
[rank0]:2024-09-19 14:46:00,990 - root - INFO - step:  5  loss:  0.0000
memory: 41.56GiB(51.95%)  wps: 2,876  mfu: 53.38%
[rank0]:2024-09-19 14:46:37,206 - root - INFO - step: 10  loss:  0.0000
memory: 41.56GiB(51.95%)  wps: 2,887  mfu: 53.60%
[rank0]:2024-09-19 14:47:13,114 - root - INFO - step: 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 6: Performance estimation code in torchtitan 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.

6.2 Simulation Accuracy

Ground truth. We have two sources of ground truth performance data for comparison. First, we use the RTX 3090 testbed to collect ground truth. Second, we use torchtitan GitHub repository [6], which documents training performance on its physical testbed consisting of 64 NVIDIA A100 80G GPUs. Torchtitan provides a comprehensive test coverage of Phantora as it uses a combination of FSDP [48], tensor-parallelism and activation checkpointing.

Baselines. We compare our results with ground truth data. In addition, we consider a baseline using our simulator to generate a static workload (i.e., execution trace) and then execute the static workload with our simulator. We slightly



bytes 1684569600
dst host-8
id 2846907

Figure 7: Perfetto [4] trace exported by Phantora.

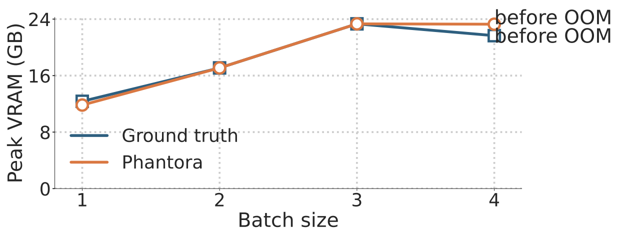


Figure 8: Peak VRAM reserved by PyTorch in Llama 1B BF16 training.

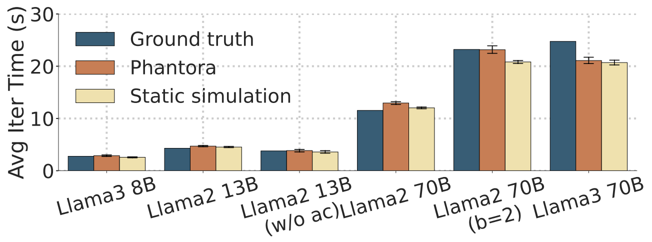


Figure 9: Simulation accuracy: Training time reported by torchtitan and simulation results of Phantora and static workload simulation on the A100 testbed. The error bars show 95% confidence interval.

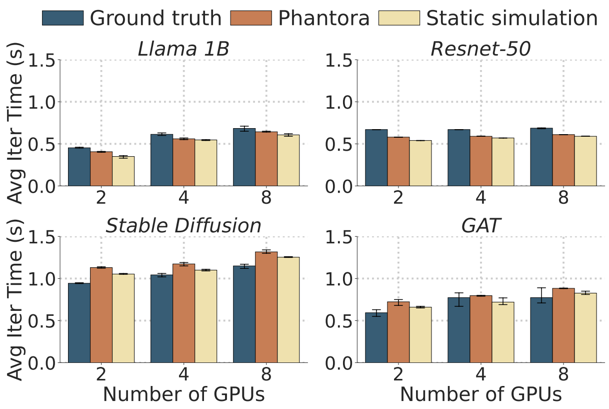


Figure 10: Simulation accuracy: Training time comparison between ground truth and Phantora's simulation results on the RTX 3090 testbed with DeepSpeed. The error bars show 95% confidence interval.

modified Phantora to support operating in this mode. This mirrors the increasingly popular approach used in industry. We do not use existing open-source workload generator as they require a real GPU cluster to generate the workload through PyTorch tracing [39]. We do not have access to 64 A100s to generate a workload. Additionally, we found that existing open-source workload representations lack important details in low-level CUDA events which are important for accurate global dependencies. Due to the above two reasons, we implement the static workload simulation baseline ourselves by changing Phantora into a workload generator and analyzer. It first captures PyTorch operations, CUDA events, and communication into a trace, then simulates the training for performance estimation.

Results. On our testbed, we show that both Phantora and static workload simulation can provide accurate performance estimation. We use per-iteration training latency as the performance metric. Figure 9 and Figure 10 show the performance estimation results on the A100 testbed and the RTX 3090 testbed. On the A100 testbed, we run 40 iterations for 8B and 13B models, and 15 iterations for 70B models. The estimation error of Phantora is 4%, 9%, 2%, 12%, 1%, and 15% for Llama 3 8B, Llama 2 13B, Llama 2 13B (no AC), Llama 2 70B, Llama 2 70B (batch size = 2), and Llama 3 70B respectively. In comparison, static workload simulation's estimation errors are 7%, 5%, 5%, 4%, 10%, and 15%. On the RTX 3090 testbed, we run 16 training iterations for each model, and Phantora's estimation errors are 7%, 10%, 11%, and 10% for Llama 1B, ResNet-50, diffusion model, and GAT, respectively. In comparison, estimation errors of static workload simulation are 14%, 13%, 8%, and 4%. Our simulator's estimation is slightly larger than the estimation of static workload simulation, because our simulator considers the latency of CPU processing. Both Phantora and static workload simulation are accurate, except for one anomaly in the simulation of Llama3 70B, where our estimation error was 15% (rightmost bars in Figure 9). We suspect this is because the original torchtitan benchmark is conducted on AWS, a shared environment that may introduce unpredictable overheads.

6.3 Simulation Speed and Scalability

We evaluate Phantora's simulation speed by comparing the simulator's runtime with the real training time reported by torchtitan [6] on Llama series models. Figure 11 shows that

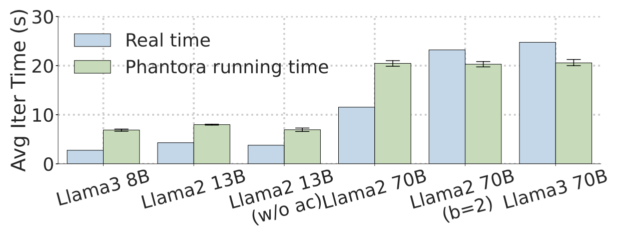


Figure 11: Simulation speed: Training time per iteration reported by torchtitan and running time of Phantora. The error bars show 95% confidence interval.

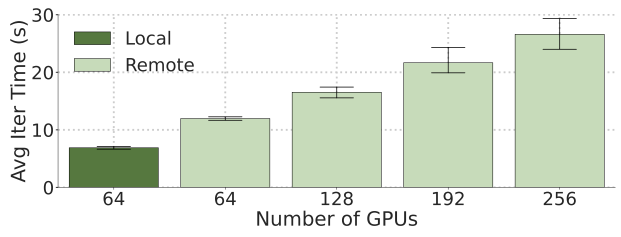


Figure 12: Simulation speed: Simulation running time of torchtitan on Llama 3 8B with distributed runtime. The error bars show 95% confidence interval.

the simulator's runtime is at the same level of the real training time per iteration. One might think that the simulator should run faster, because it skips the execution of all the CUDA kernels and communication. However, we have a centralized simulator process that needs to act on every event in the distributed system. We believe the speed is reasonable, as the developer only needs to wait tens of iterations to accurately assess training performance.

To evaluate the scalability of Phantora, we simulate Llama 3 8B training of increasingly larger scales with torchtitan. For this experiment, we use the A100 testbed to profile CUDA kernels and the RTX 3090 testbed to run containers. To ensure sufficient CPU resources, we also incorporate two additional CPU machines, each equipped with 2 AMD EPYC 7542 32-Core Processors (128 hyperthreads). Figure 12 shows the simulator runtime for one iteration of training. The leftmost bar, representing 64 GPUs, is simulated on a single physical server, while the other bars represent distributed execution. The simulator runtime scales linearly with the number of GPUs from 64 to 256. We also observe increasing variance in runtime as the simulation scale grows, which we attribute to performance fluctuations in the physical network connecting our servers, as the A100 testbed is located in a separate server room within the institution.

6.4 Ablation Study

We conduct ablation studies of our technique to improve scalability and accuracy. We first study the effects of our

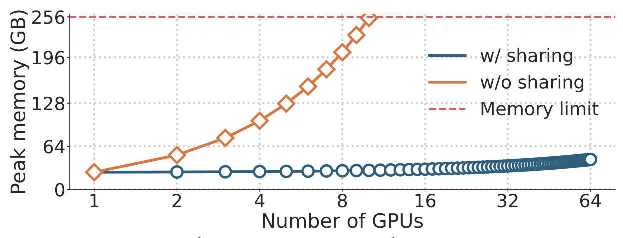


Figure 13: Simulation cost: Peak memory usage of Llama 7B training simulation in DeepSpeed with or without model parameter sharing.

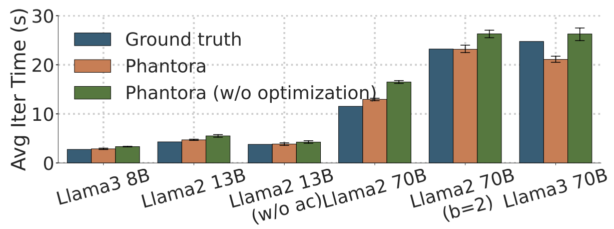


Figure 14: Simulation accuracy: Training time per iteration reported by torchtitan and simulation results of Phantora with or without accuracy optimizations. The error bars show 95% confidence interval.

memory sharing technique on our RTX 3090 testbed. Note that the choice of testbed does not affect the results, as we are evaluating host memory usage. Figure 13 shows the peak memory usage of Phantora on our server during simulation. When we turn off our memory sharing capability, one hardware server can only simulate 10 GPUs. This is because the PyTorch runtime itself can consume significant memory due to loading model weights and other userspace processing. After we turn on our memory sharing, one hardware server can simulate 64 GPUs using only 44.5GB memory.

To quantify the effect of our optimization on simulation accuracy, Figure 14 shows simulation accuracy of torchtitan workloads with and without the optimizations described in §4.4. Without the optimizations, Phantora simply uses the standalone CUDA kernel profiling results. Our technique improves the accuracy by 12% on average.

7 DISCUSSION

What performance factors are not considered in Phantora? There are several performance factors that are not considered in Phantora. The first one is the congestion control behavior in network. Our flow-level simulator assumes a network flow immediately reaches its fair share. We believe this is enough to provide accurate simulation due to large flow sizes generally in ML systems. The second source of inaccuracy comes from the CPU cache/TLB effects as we are running many containers on the same host. Although we pin different container to non-overlapping physical cores, the containers still share the last-level cache. Finally, GPU kernels may exhibit value-dependent performance variations. One such example is that GPU is faster at computing multiple-by-zero or multiple-by-one, compared to standard matrix multiplication. We ignore these effects, because they don't appear frequently in ML systems.

Can we improve the accuracy of Phantora's performance estimation? The primary goal of this paper is to show that live simulation can achieve comparable accuracy to today's static workload simulation methods while offering better generalizability. Improving accuracy beyond static workload simulation is out of the scope of this work. However, as industry continues to adopt and refine static workload simulation techniques, we believe that these advancements can also be integrated into Phantora.

ML systems for which control logic depends on GPU computation results. A key limitation of Phantora is that it cannot support ML systems where control logic depends on tensor values of GPU computation or communication. For example, in mixture-of-experts (MoE) models, computation and communication patterns depend on the dynamic routing results of a gate network [18, 24]. Although we have not explored this approach, we believe that extending Phantora to support MoE is feasible by replacing the gate network's decision mechanism with a probability distribution. Another example is the set of inference methods based on guess-and-check [8, 9, 16, 23]. These methods guess the inference results first and validate it against an LLM, speculatively avoiding the autoregressive decoding procedure. Phantora shares this limitation with a large body of existing estimation methods and ML system works [51], where performance is assumed not to change when tensor values change.

Further scaling Phantora. Another key limitation of Phantora is that all containers must run simultaneously, meaning that simulating a 512-process workload requires 512 CPU cores. One could trade off simulation scalability with execution time by scheduling containers on a limited number of CPU cores and carefully tracking their execution time. We didn't explore this trade-off, because our testbed has enough CPU cores and we can directly scale out to distributed execution (§4.3). However, it is an interesting research direction to explore CPU-constrained setups.

8 RELATED WORKS

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) [30, 37] explored an approach to estimate performance of cache-coherent shared memory systems. WWT also encountered the problem of synchronizing simulated time with direct-execution and the simulator. 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 effort is to enable today's event-driven network simulator with real systems. For instance, ns-3 [2] supports a feature called TapBridge, which implements 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 containers. 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.

Predicting CUDA kernel execution time. Predicting a CUDA kernel's execution time is a standard problem in ML compilers [11, 50]. 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 trains a gradient boosting decision tree by applying a set of manually-designed features on the CUDA kernel source 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 DeepSpeed and torchtitan). In fact, we can comprehensively profile each CUDA kernel and tensor shape used in these ML systems. The goal of our technique (§4.4) is to enhance accuracy even after exhaustive profiling of individual CUDA kernels.

Emulating the control plane and simulating the data plane. We draw inspirations from CrystalNet [25] 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 skipped. Although in different contexts, both CrystalNet and Phantora rely on the assumption that the control flow does not have data dependency.

9 CONCLUSION

This paper introduces Phantora, a live GPU cluster simulator for performance estimation. Phantora runs minimally modified ML models and frameworks, intercepting and simulating GPU- and network-related operations for high-fidelity performance estimation. It addresses key research challenges, including the integration of event-driven network simulators with real-time code execution, along with techniques to improve simulation accuracy, speed, and scalability. Our evaluation shows that, with only a single GPU, Phantora achieves similar estimation accuracy to state-of-the-art static workload simulation methods, while increasing generalizability. Phantora's source code will be publicly available.

ACKNOWLEDGMENT

This work was supported in part by NSF grants 2112562, 2238665, and 2402696, as well as by gifts from Amazon and Meta. This work was also 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] CUDA C Best Practices Guide. https://https://docs.nvidia.com/cuda/cuda-c-best-practices-guide, 2024.

[2] ns-3, a discrete-event network simulator for Internet systems. https://www.nsnam.org/, 2024.

[3] Nsight Systems | NVIDIA Developer. https://developer.nvidia.com/nsight-systems, 2024. (Accessed on 09/19/2024).

[4] Perfetto - system profiling, app tracing and trace analysis. https://perfetto.dev/docs, 2024. (Accessed on 01/29/2025).

[5] tonic: A native grpc client & server implementation with async/await support. https://https://github.com/hyperium/tonic, 2024.

[6] torchtitan/docs/performance.md at main · pytorch/torchtitan. https://github.com/pytorch/torchtitan/blob/217cc94e2abf8472db098c1c0e5e020e62dcfc7d/docs/performance.md, 2024. (Accessed on 09/19/2024).

[7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[8] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. *arXiv preprint arXiv: 2401.10774*, 2024.

[9] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating Large Language Model Decoding with Speculative Sampling, 2023.

[10] 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 *NSDI*, 2022.

[11] 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 *OSDI*, 2018.

[12] 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, March 2023.

[13] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, 2023.

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *NIPS*, 2022.

[15] 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. *CoRR*, abs/2010.11929, 2020.

[16] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the Sequential Dependency of LLM Inference Using Lookahead Decoding, 2024.

[17] 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 *MLSys*, 2022.

[18] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *MLSys*, 2023.

[19] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*, 2019.

[20] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the Computational Cost of Deep Learning Models. In *2018 IEEE International Conference on Big Data (Big Data)*, 2018.

[21] 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 PagedAttention. In *SOSP*, 2023.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*, 2023.

[23] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *ICML*, 2023.

[24] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed {MoE} training and inference with lina. In *USENIX ATC*, 2023.

[25] 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 *SOSP*, 2017.

[26] 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 *CF*, 2023.

[27] Liang Luo, Peter West, Arvind Krishnamurthy, and Luis Ceze. Srift: Swift and Thrift Cloud-Based Distributed Training. *CoRR*, abs/2011.14243, 2020.

[28] 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 *MLSys*, 2024.

[29] 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 Bhattacharya, 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 *ISCA*, 2022.

[30] 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.

[31] OpenAI. DALL·E 3. https://openai.com/index/dall-e-3, 2024.

[32] OpenAI. GPT-4 Technical Report, 2024.

[33] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR*, 2022.

[34] 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 *OSDI*, 2021.

[35] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *ISPASS*, 2020.

[36] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *KDD*, 2020.

[37] 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 *SIGMETRICS*, 1993.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, abs/1909.08053, 2019.

[39] 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, 2023.

[40] Hugo Touvron et al. Llama 2: Open Foundation and Fine-Tuned Chat Models, 2023.

[41] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks, 2018.

[42] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *SC*, 2014.

[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 *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*, 2024.

[46] Jun Yi, Chengliang Zhang, Wei Wang, Cheng Li, and Feng Yan. Not all explorations are equal: Harnessing heterogeneous profiling cost for efficient mlaas training. In *IPDPS*, 2020.

[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, 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, 2023.

[49] Haoyue Zheng, Fei Xu, Li Chen, Zhi Zhou, and Fangming Liu. Cynthia: Cost-Efficient Cloud Resource Provisioning for Predictable Distributed Deep Neural Network Training. In *ICPP*, 2019.

[50] 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 *OSDI*, 2020.

[51] 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 *OSDI*, 2022.

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