TrioSim: A Lightweight Simulator for Large-Scale DNN Workloads on Multi-GPU Systems

Ying Li
William & Mary
Williamsburg, VA, USA
yli81@wm.edu

Yuhui Bao*
Northeastern University
Boston, MA, USA
bao.yu@northeastern.edu

Gongyu Wang
Lightmatter
Boston, MA, USA
wgongyu@gmail.com

Xinxin Mei
Jefferson Lab
Newport News, VA, USA
xmei@jlab.org

Pranav Vaid*
Stanford University
Palo Alto, CA, USA
pvaid@cs.stanford.edu

Anandaroop Ghosh
Lightmatter
Boston, MA, USA
anandaroopg08@gmail.com

Adwait Jog
University of Virginia
Charlottesville, VA, USA
ajog@virginia.edu

Darius Bunandar
Lightmatter
Boston, MA, USA
darius@lightmatter.co

Ajay Joshi
Lightmatter/Boston University
Boston, MA, USA
ajay@lightmatter.co

Yifan Sun
William & Mary
Williamsburg, VA, USA
ysun25@wm.edu

Abstract

Deep Neural Networks (DNNs) have become increasingly capable of performing tasks ranging from image recognition to content generation. The training and inference of DNNs heavily rely on GPUs, as GPUs' massively parallel architecture delivers extremely high computing capability. With the growing complexity of DNNs and the size of training datasets, training DNNs with a large number of GPUs is becoming a prevalent strategy. Researchers have been exploring how to design software and hardware systems for GPU farms to achieve the best utilization, efficiency, and DNN accuracy during training or inference. However, when designing and deploying such systems, designers usually rely on testing on physical hardware platforms equipped with many GPUs, incurring high costs that are almost prohibitive for system designers to test different configurations and designs, even for highly resourceful companies. While an alternative solution is to test on GPU simulators, they are often too slow for these large-scale systems and depend on profiling details collected from real distributed systems to initiate the simulation. To address these challenges, we present TrioSim, a novel lightweight simulator for DNNs on multi-GPU systems. TrioSim combines performance modeling techniques and simulation methods to achieve high flexibility, high simulation speed, and high accuracy. TrioSim minimizes the required input from users by only relying on operator-level execution traces collected on a single GPU and can simulate new software and hardware designs that involve multiple GPUs connected with complex, asymmetrical interconnects. Completing within seconds, TrioSim predicts the execution time of DNN training on multi-GPU systems with average errors of 2.91%, 4.54%, and 6.82% for data parallelism, tensor parallelism, and pipeline parallelism, respectively.

CCS Concepts

• **Computing methodologies** → **Simulation tools**.

Keywords

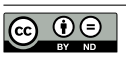
Deep Neural Networks; Multi-GPU Systems; Simulation

ACM Reference Format:
Ying Li, Yuhui Bao, Gongyu Wang, Xinxin Mei, Pranav Vaid, Anandaroop Ghosh, Adwait Jog, Darius Bunandar, Ajay Joshi, and Yifan Sun. 2025. TrioSim: A Lightweight Simulator for Large-Scale DNN Workloads on Multi-GPU Systems. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731082

1 Introduction

Today, Deep Neural Networks (DNNs) are one of the most common classes of workloads running on high-performance computing platforms that equip GPUs [6, 18, 22, 27]. GPUs can provide extremely high computing capabilities for DNN training and inference, with their massively parallel architecture [47, 53]. Multi-GPU systems are commonly used today for DNN training and inference due to the large training dataset and model sizes [11, 46, 69]. However, multi-GPU systems introduce new challenges caused by data

*Part of this work was done while Yuhui Bao and Pranav Vaid were interns at Lightmatter.



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731082

management, communication, and synchronization. The overhead caused by the slow network between GPUs can diminish the performance improvement gained from adding more GPUs.

Over the years, a variety of novel hardware techniques [10, 37, 42, 54, 61, 76] have been developed to address challenges in effectively using multi-GPU systems for DNN workloads. Yet, researchers often rely on real GPU hardware testbeds for validating design techniques, incurring high costs. In many cases, due to a hardware modification associated with a research idea, faithfully validating the research idea can be almost impossible. These drawbacks urge the development of methods that evaluate machine learning systems, memory management solutions, and scheduling schemes with a low cost, minimal entry barrier, and reasonable fidelity.

A possible solution is to evaluate DNN systems with GPU simulators (e.g., Accel-Sim [28], MGPUSim [4, 67], Multi2Sim [72], gem5 AMD GPU Models [62]). However, these cycle-level simulators are too slow for large-scale DNN workloads and multi-GPU platforms. As estimated in the work, Principal Kernel Analysis [1], simulating such workloads running on GPUs may take centuries, rendering them unusable. Recent research [1, 40, 45] has also been developing sampling-based methods to skip part of the simulation, enabling cycle-based simulators to simulate DNN workloads. Yet, they are still resource-demanding when identifying the execution phases and sampling opportunities.

Recent developments in data-driven performance models (e.g., AstraSim [60, 74], DistSim [41], vTrain [3]) are fast and reasonably accurate. However, a major problem with these existing solutions is that they heavily rely on the profiling traces that include GPU-GPU communication tasks. This requirement causes two problems: 1) users must acquire multi-GPU platforms to gather the traces, which may increase the barrier, and 2) modeling different communication patterns and network configurations is constrained.

To deliver a practical solution for evaluating DNN systems, we introduce TrioSim.[1] TrioSim takes the operator-level traces collected from the PyTorch (extending to other frameworks is possible) executing on a single GPU and extrapolates execution details for more complex multi-GPU execution.

Central to TrioSim is a *trace extrapolator*, an *operator-level performance model* [34], and a *high-level network simulator*. The trace extrapolator determines which GPU should perform the operation of a layer and what data is necessary. In case the data is not available on the GPU, the trace extrapolator adds communication tasks, including direct memory movement tasks and collective communication tasks analogous to the communication primitives of the NVIDIA Collective Communications Library (NCCL) [50]. For the operator-level performance model, we use Li's Model [34], to predict the execution time of DNN operators and layers. Finally, to estimate the data movement time, we built a high-level flow-based network model, which ignores protocol details but focuses on the latency and bandwidth of network links and bandwidth sharing between streams of data (similar to flow-based network simulator [12, 16]).

The novelty of TrioSim is twofold. First, TrioSim offers superior capability over existing simulators by requiring only a single GPU trace to faithfully simulate a multi-GPU system. This minimal trace requirement allows users to dynamically adjust key parameters, including batch size, network topology (e.g., NVSwitch [49], mesh, fat tree, etc.), parallelism strategy (e.g., data parallel [33, 75], tensor parallel [65], pipeline parallel [25, 65]), and collective communication schemes (e.g., NCCL-style reduction [50]), all without requiring additional trace collection. Second, this capability is enabled by TrioSim's unique trace extrapolation method, which balances performance modeling and simulation. At a high level, TrioSim functions as a simulator, reconstructing future events based on the system state. At a low level, it integrates Li's Model [34] for computation prediction and a flow-based network model for communication, ensuring accurate estimation of execution time.

The main goal of this paper is to deliver a blazingly fast simulator that can model the performance of large-scale DNN training on massive multi-GPU platforms (see subsection 8.4 for its limitations). In particular, this paper contributes:

1) **A tracer.** The tracer collects the GPU execution data at layer or operator levels. We highlight the capability of also tracing tensor information and recording which layer or operator accesses which part of data. Connecting computing trace with tensor-access trace provides critical information for jointly modeling computation and data movement.

2) **A trace extrapolator.** The trace extrapolator converts single GPU traces into multi-GPU traces according to the parallelism scheme (e.g., data parallelism, tensor parallelism, pipeline parallelism). Trace extrapolation enables the simulation of multi-GPU execution without the hardware.

3) **The simulator, TrioSim.** We combine the trace extrapolator, a linear regression-based performance model, and a lightweight network model to build TrioSim. TrioSim can simulate the GPU execution for DNNs on multi-GPU systems with high performance and reasonable accuracy. TrioSim can finish the simulation of multiple batches of DNN training within seconds. The average errors when predicting DNNs training time on a system with 2 GPUs are 2.91%, 4.54%, and 6.82% for data parallelism, tensor parallelism, and pipeline parallelism, respectively.

2 Background

2.1 DNN Training on Multi-GPU Platforms

Multi-GPU platforms are widely utilized for accelerating DNN training as the GPUs jointly provide higher computing throughput and memory capacity than single GPU platforms [46, 69]. To fully utilize multiple GPUs in DNN training, schemes that properly place the data, schedule computing tasks, communicate across GPUs, synchronize progress, and aggregate results are required. Below we introduce popular parallel DNN training, GPU-GPU communication schemes, and performance modeling approaches.

Parallel DNN training. Central to parallel DNN training schemes (see Figure 1 for an overview) is the decision on how to partition workloads across GPUs, with strategies tailored to address specific challenges. Data parallelism (DP) distribute batches of input data across GPUs, allowing each GPU to process the batches simultaneously. Each GPU performs forward and backward propagation on their data batches individually, calculating the gradients of the parameters. Then, inter-GPU communication synchronizes gradients to update weights. Tensor parallelism (TP) splits each tensor (e.g.,

[1] https://github.com/sarchlab/triosim

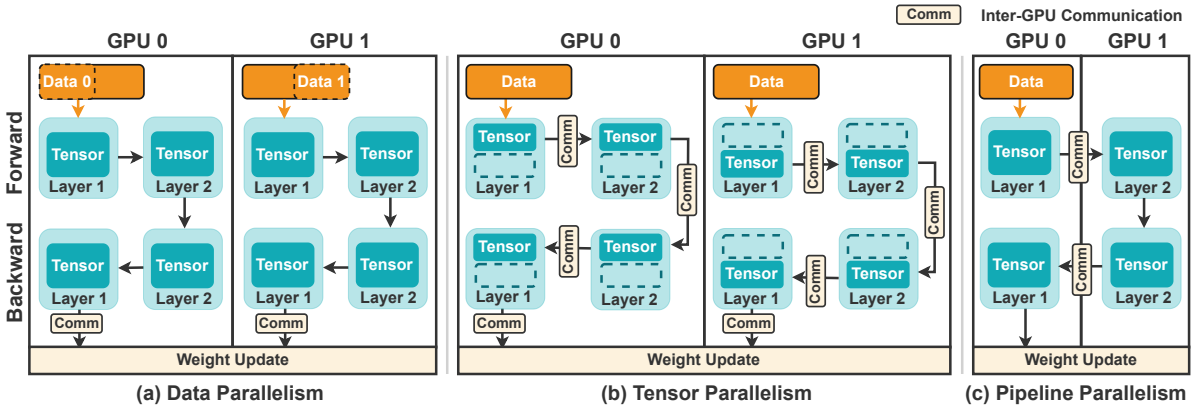


Figure 1: Parallelism strategies include: (a) Data parallelism, splitting the dataset into parts and processing them in parallel. Then GPUs communicate and synchronize to update model weights. (b) Tensor parallelism, dividing tensor into parts and assigning each part to be processed by a GPU. Afterward, GPUs communicate to gather the output. (c) Pipeline parallelism, assigning layers to GPUs. Each layer calculates and sends the output to the next GPU.

parameters of a layer, layer input) to be processed across multiple GPUs and performs computing on each part of the tensors in parallel. When the layer applied tensor parallelism finishes, each GPU communicates with others to collect the partial outputs of the layer from all devices, forming the whole results. Pipeline parallelism (PP) distributes DNN layers across GPUs and can further divide each mini-batch into micro-batches. Each GPU handles its assigned micro-batch and forwards the processed micro-batch to the next GPU. While the next GPU processes this micro-batch, the current GPU begins the next, forming an overlapping pipeline that runs until all micro-batches are done, enabling continuous data flow and parallel execution across GPUs. These methods, often combined as hybrid parallelism (HP), optimize scalability and resource utilization in multi-GPU systems.

The AllReduce algorithms. The AllReduce algorithm is a fundamental operation used for communication among devices while training DNNs. It is especially prevalent in training DNNs across multiple GPUs or nodes. AllReduce is designed to aggregate data across all participating processors and distribute the results back to them, ensuring each processor has the complete set of results. This operation is critical in scenarios such as averaging gradients during DNN training, where synchronization and aggregation of data from all nodes are necessary for accurate model updates.

A popular implementation of the AllReduce operation is the ring-based algorithm, which offers an efficient and scalable solution for data aggregation in distributed systems. In the ring-based AllReduce, each node in a ring topology sequentially sends and receives data to and from its neighbors. Specifically, each node passes on its data to the next node and simultaneously receives data from the previous node in the ring. This process continues until all nodes have a complete set of aggregated data. The ring-based approach minimizes bandwidth usage by evenly distributing the load among all nodes. This method is distinguished by its predictable communication pattern and the simplicity of its implementation, which contributes to its widespread adoption in various high-performance computing environments. TrioSim's trace extrapolator can generate a sequence of tasks of ring-based AllReduce algorithm, allowing TrioSim to simulate the collective communication.

Multi-GPU architectures. While multi-GPU setups offer substantial advantages of higher memory and compute capacity, effective coordination of tasks and communication across GPUs remains a critical challenge. Especially, the choice of network topology and interconnect technology between GPUs significantly impacts communication overhead and overall performance. Common topologies include ring, tree, mesh, and bus. Modern implementations that facilitate connections among multiple GPUs include PCI Express (PCIe) [56], NVLink [52], and NVSwitch [49]. PCIe, the standard interconnect, is widely employed in many systems and is typically arranged in a hierarchical tree-like structure to optimize both spatial and temporal data flow. NVLink provides a high-speed alternative to PCIe, significantly boosting data transfer rates and supporting flexible topologies such as ring and mesh, which are essential for complex computational tasks requiring rapid data sharing across GPUs. In systems like the NVIDIA DGX-2, NVLink utilizes a hypercube mesh topology, connecting 8 GPUs within each machine. This setup includes links with double bandwidth, strengthening connections to form a loop that optimally supports the ring-based AllReduce algorithm. Extending NVLink's capabilities further, NVSwitch enables any-to-any communication, eliminating bottlenecks found in traditional interconnects by allowing a fully connected network topology within the system.

Given the pivotal role of interconnects in system performance, TrioSim needs to model the network behaviors carefully. However, since our primary objective is high-performance simulation, employing cycle-by-cycle or flit-by-flit simulation may not be practical. Consequently, TrioSim develops and integrates a lightweight network model that balances detail and performance.

3 Related Work

GPU performance analysis and prediction are crucial in optimizing the efficiency of DNN models. Traditional profiling techniques are

Table 1: Comparison of TrioSim with Similar Performance Modeling Tools across Different Features

Feature	Li's Model [34]	AstraSim [60, 74]	DistSim [41]	vTrain [3]	TrioSim (this work)
Target Workload	DNN inference	DNN training	DNN training	Transformer training	DNN training
Parallelism	Not supported	DP, TP, PP	DP, TP, PP, HP	DP, TP, PP, HP	DP, TP, PP
Network	Not supported	Symmetrical (e.g., ring, switch)	Profile-based	Profile-based	Flexible
Trace Requirement	Single-GPU	Multi-GPU	Multi-node	Multi-node	Single-GPU
Performance Model	Analytical	Mainly cycle-level simulation	Analytical	Analytical	Hybrid analytical & simulation
Support New GPU	Yes	No	No	No	Supported using Li's Model
Claimed Error	7% (single GPU)	N/A	<4% (multi-GPU)	8.37% (single node)	2.91% (DP), 4.54% (TP)
	15.2% (new GPU)		<5% (single GPU)	14.73% (multi-node)	6.82% (PP)

often insufficient to capture inter-GPU interactions at scale [2, 8, 19, 29, 51, 77, 78], prompting the development of approaches for modeling DNN performance on multi-GPU systems. We analyze the potential solutions, highlighting their strengths and limitations.

GPU simulators. The most straightforward method to evaluate DNN systems running on GPU systems is using cycle-level GPU simulators (e.g., Accel-Sim [28], Multi2Sim [72], MGPUSim [67], gem5 AMD GPU Models [62]). However, these simulators are inefficient for large-scale DNN workloads and multi-GPU platforms due to their slow speeds. Simulating a full-scale DNN workload (e.g., full-scale ResNet on a single GPU) may take years [1].

Sampling-based simulation. To accelerate GPU system simulation, sampling-based methods [1, 40, 45] have been developed. These methods selectively bypass portions of the simulation where performance can be accurately predicted based on detailed simulations of representative execution segments. For example, a GPU simulator can simulate some of the warps in a kernel in detail and skip the rest of the warps that execute the same set of instructions [40]. Despite these efforts, sampling-based methods remain relatively slow and resource-intensive, mainly due to the need for detailed, full-scale simulation during non-sampling periods.

GPU scale-model. GPU scale-model simulation [64] predicts large-scale GPU performance by extrapolating from smaller-scale GPU performance numbers. Scale-model simulation relies on pre-profiled small-scale models (e.g., GPUs with 4 Streaming Multi-Processors) and assumes predictable scaling. However, the scale-model lacks explicit communication modeling, reducing its adaptability to new architectures and multi-GPU parallelism strategies. TrioSim, in contrast, requires only a single GPU trace, explicitly simulates inter-GPU communication, and supports configurable network topologies and parallelism schemes. While scale-model simulation is useful for high-level GPU performance estimation and architecture-level GPU research, TrioSim is better suited for system-level GPU research (e.g., exploring the best parallelism strategy for GPUs connected with complex networks).

Trace-based DNN performance models/simulators. Several trace-based performance models have been developed to evaluate DNN workloads on multi-GPU systems, including AstraSim [60, 74], DistSim [41], and vTrain [3]. These models aim to estimate system performance efficiently by leveraging execution traces rather than detailed cycle-level simulation or extensive multi-GPU profiling. We compare these tools and TrioSim in Table 1 in detail.

AstraSim [60, 74] is designed for large-scale deep learning training and provides detailed communication modeling for distributed systems. DistSim [41], on the other hand, is tailored for hybrid parallelism exploration and supports multi-GPU execution modeling. While both tools are valuable, they require multi-GPU or multi-node profiling traces, which raises their barrier to use. TrioSim, in contrast, requires only a single GPU trace, significantly reducing the profiling burden and making it more accessible for users who don't have large-scale hardware. Additionally, as TrioSim adopts a hybrid simulation and performance modeling method, TrioSim provides more flexible configuration capability. For example, TrioSim can easily support non-symmetrical network configuration (e.g., one link is slower than other links), which can be challenging to model and evaluate in AstraSim and DistSim.

Li's Model [34, 35] provides a kernel-level or operator-level analytical model to predict GPU execution times efficiently. However, it is a per-kernel or per-operator model rather than a full simulator, meaning it cannot capture system-wide execution behaviors or inter-GPU communication dynamics. By leveraging Li's Model, TrioSim extends its capability to support full-scale DNN training simulations for better evaluating multi-GPU workloads.

A recent notable tool is vTrain [3], which is specifically designed for transformer-based large language models (LLMs) and optimizes profiling by leveraging the repetitive nature of transformer layers. However, vTrain has two key limitations that prevent it from fully meeting the broader needs of the community: (1) vTrain is specialized for LLMs, and while it has been validated on a 512-GPU system, it is tested on only a single workload, limiting its generalizability. In contrast, TrioSim provides a more general DNN simulation platform with validation across multiple workloads, demonstrating broader applicability. (2) vTrain has strict trace requirements, as it not only requires multi-GPU hardware for trace collection but also mandates that batch size and parallelism settings remain identical between profiling and simulation. This rigid requirement hinders comprehensive design space exploration. By comparison, TrioSim requires only a single GPU trace, allowing users to flexibly configure network topology, parallel training strategies, and batch sizes, making it a more versatile platform.

4 TrioSim

The design of TrioSim (see Figure 2) encompasses the following main functions: trace capturing, multi-GPU trace extrapolation, linear-regression-based time modeling, and lightweight network modeling. TrioSim is implemented using the Go language and on top of the Akita Simulator Engine [67]. The event-driven simulation engine allows us to fast-forward unnecessary details and accelerate

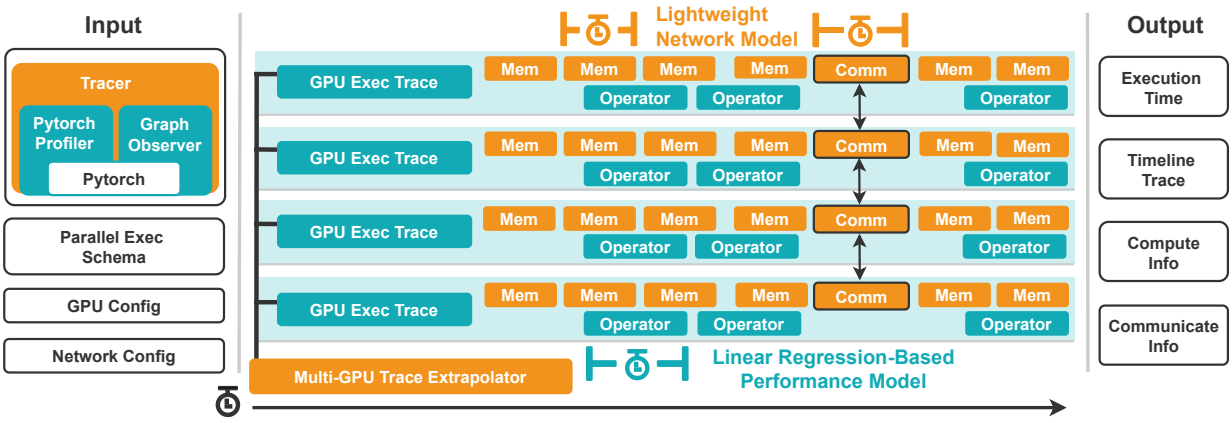


Figure 2: The design of TrioSim comprises trace capturing, multi-GPU trace extrapolation, linear regression-based time modeling, and lightweight network modeling.

simulation. The Akita Simulator Engine also allows TrioSim to natively support real-time monitoring with AkitaRTM [44] and visualization with Daisen [68].

4.1 Overview

The primary input for TrioSim is the trace generated by the tracer tool. The additional inputs to TrioSim include the network topology, the GPU parameters, and the parallelism schemes. We support different types of parallelism, such as data parallelism, tensor parallelism, and pipeline parallelism.

With the input configurations, the multi-GPU trace extrapolator will distribute the single GPU trace across multiple devices during the simulation process. The linear regression-based performance model will predict the computation time of operators, while the lightweight network model will estimate the memory fetching time. During synchronization among GPUs, the communication process will be initiated by TrioSim and the lightweight network model measures the communication time.

For the output of TrioSim, it can return the total predicted execution time for DNNs on multiple GPUs with different parallelism applied. It can also return the communication time and computation time of each layer or stage for workloads. At the same time, it shows the timeline of the communication process among GPUs or the computation process on each GPU. The other information provided by TrioSim includes details on the input and output tensors of each layer and the amount of data involved in gradient calculations during AllReduce operations.

4.2 Tracer

Trace format. TrioSim traces include information about the GPU operations performed in DNN training processes. We require each entry to include the operator name, measured execution time, and input/output as a list of tensor IDs. Additionally, we maintain a second table that records all the tensors used in the DNN training and inference process. Other than the tensor ID, we mainly record tensor dimensions to estimate the number of bytes that need to be moved if the tensor is not on the GPU that uses it.

Tracer design. TrioSim's tracer is built on PyTorch [55], one of the most popular DNN frameworks. The tracer mainly relies on the output generated by the PyTorch Profiler and the Execution Graph Observer to form a holistic view of the execution. The trace format is standard, so supporting other frameworks is possible. We leave supporting other frameworks as future work.

We rely on PyTorch Profiler [55] to extract the operators executed by the GPU and the operators' execution times. We first use the PyTorch Profiler recorded CPU traces, which include the DNN layers and the underlying operators (e.g., matrix multiplication, matrix transpose) that are executed during training. PyTorch Profiler can also record the GPU kernels executed, with their start and end times. We use the PyTorch Profiler recorded information to reconstruct the mapping between the kernels, operators, and layers. We consider the operator start and end time as the time that its first kernel starts and its last kernel ends, respectively.

Additionally, we use the Execution Graph Observer [38] to extract both the operators for DNN layers executed and the associated inputs and outputs data dependencies during the DNN training process. For each operator, Execution Graph Observer provides rich tensor-related information, including 1) the input/output tensors as a list, 2) tensor category (input/weight/gradient/output), and 3) tensor format (element data type, dimension).

Using the layer information as a bridge, we blend the data provided by the Pytorch Profiler and the Execution Graph Observer to reconstruct both the operator execution and memory access information. Overall, the TrioSim's tracer captures details of operator specifications, operator execution time, and tensors accessed (see Figure 3).

4.3 Multi-GPU Trace Extrapolator

Simulating multi-GPU DNN workloads requires multi-GPU execution traces. However, users may not physically have the multi-GPU systems that they want to evaluate. Therefore, we decide to only require users to provide traces collected from single GPU execution and we provide a trace extrapolator to automatically convert single-GPU traces to multi-GPU traces. The whole extrapolation

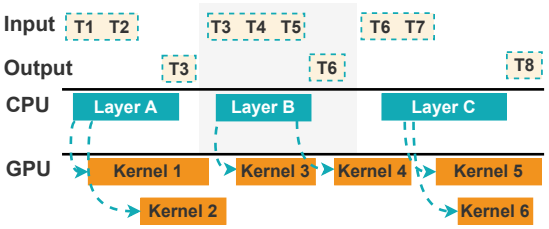


Figure 3: Trace capturing in tracer. Trace generated by PyTorch Profiler shows layers in CPU and kernels in GPU, while Execution Graph Observer extracts the operators on the CPU and identifies inputs and outputs data dependencies.

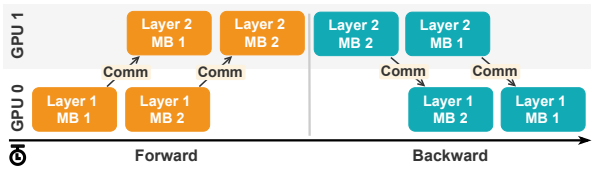


Figure 4: GPipe in TrioSim. Assuming the simple model has two layers, with each GPU assigned one layer, the GPipe implementation in TrioSim divides the one mini-batch into two micro-batches in this example.

process happens while the simulation unfolds. Extra calculation and communication operators are added on the fly, throughout the whole simulation process.

TrioSim initiates the extrapolation process by processing the single GPU trace. Upon reading the first operator, the trace extrapolator decides which GPU (can be multiple GPUs) needs to perform the operator. Assuming 2 GPUs need to perform the operator, TrioSim then checks if these 2 GPUs have the required data for the operator in their local memory. If not, TrioSim inserts data movement operators before the duplicated calculation operators.

After the insertion process, TrioSim processes the inserted data movement and calculation operators, moving the time forward. The virtual time movement (virtual time is the time within the simulated world) is determined by the computing performance model (see subsection 4.4) and the network model (see subsection 4.5). After finishing the first element in the trace, TrioSim will move on to the next element until there is no more element left in the trace.

Users can specify the parallel execution and communication schemes and the hardware connection topology. We natively support data parallelism, tensor parallelism, and pipeline parallelism training. For inter-GPU communication, we support NCCL-style collective communication, including reduce, scatter, and gather processes. TrioSim supports extending network topology, parallelism, and collective communication schemes, allowing flexible and straightforward extension.

Data parallelism training. TrioSim supports simulating the data parallelism strategy on multiple GPUs. For the forward and backward propagation process, the trace extrapolator simply duplicates all the computing operators. After receiving gradient tensors from the forward pass, the trace extrapolator adds the necessary operators for the AllReduce operation either parallel with the backward pass to save execution time or after the backward pass.

TrioSim allows changing the batch sizes different from what is recorded in the trace, which is not easy for prior simulators (e.g., AstraSim, vTrain). The operator calculation time will need to scale, estimated using the linear regression-based performance model (see subsection 4.4). The memory movement time will also need to change accordingly. This feature allows users to explore the impact of batch size on training speed and efficiency.

Tensor parallelism training. TrioSim also supports the simulation of the tensor parallelism strategy on multiple GPUs. Tensor parallelism divides the weight matrix across different devices. Each device computes its partition of the weight matrix and then collects the partial outputs from all devices. We simulate tensor parallelism for layers, such as convolution, linear, and embedding, as they are also parallelly executed by PyTorch. When simulating a layer executed with tensor parallelism, the trace extrapolator distributes divided operators into each GPU's queue and appends the necessary communication operators at the layer's end. TrioSim then processes all operators according to the queue.

Pipeline parallelism training. In the implementation of pipeline parallelism, different layers or stages of DNNs are assigned to different devices by TrioSim according to the user's specification. We implement the GPipe (see Figure 4) in TrioSim because it is widely adopted by popular frameworks [46, 57].

Based on GPipe, the trace extrapolator generates micro-batch computation operators by partitioning mini-batches into equal-sized micro-batches. It initially assigns the computation operators for the first micro-batch layers to the first GPU. Next, it initiates communication operators, allowing the first GPU to send its output data to the next GPU. Simultaneously, the trace extrapolator begins the computation operators for the next micro-batch layers on the first GPU. This cycle repeats, enabling continuous entry of micro-batches into the pipeline. TrioSim then executes all the computation and communication operators, establishing pipeline parallelism across multiple GPUs and measuring the execution time.

Ring-based collective communication. TrioSim natively supports NCCL-style collective communication for ring-based networks. In a ring topology, each device connects to two adjacent devices: a left and a right neighbor. TrioSim supports ring-based collective communication as part of the trace extrapolation process. Memory transfer tasks are added to the extrapolated trace when GPUs communicate with each other. Then, the tasks will be picked up by the simulator, and the transfer time will be estimated by the network model (to be introduced).

4.4 Operator Performance Model

TrioSim replays the extrapolated trace, one operator after another. To start the execution of one operator, TrioSim needs to make sure that the data to be used by the operator is available on the device. If not, input data will be fetched (requires the network model to estimate time), and output data will be allocated. Here, we make assumptions that 1) a tensor is always stored on a single remote location and 2) the execution cannot start until the data is available. The assumption may not be true for architecture-level research where data can be loaded during kernel execution but is generally

acceptable for system-level research [14, 17, 63]. Once the data to be accessed by an operator is on the local GPU, the operator execution starts. TrioSim does not simulate any detail of operators but simply predicts the execution time of the operator execution and directly fast forwards to the operator completion time.

TrioSim provides two different solutions to predict operator execution time. A simpler solution supported by TrioSim uses the trace provided time, which is measured on the single GPU platform, to serve as the execution time. This method is accurate but not flexible as we cannot change the batch size (in data parallelism) or tensor size (as in tensor parallelism). TrioSim always uses the trace-provided operator execution time when the environment and parameters of multiple GPU simulations are the same with the single GPU trace.

A more sophisticated solution is to use Li's Model [34], which has demonstrated high time-prediction accuracy without extensive calculation. TrioSim passes the operator input/output dimensions to the performance model, allowing the performance model to predict the new operator's (different input/output size) execution time. Thus, TrioSim can use single-GPU operator time to predict the time for multi-GPU operators by comparing the difference in floating-point operations (FLOPs) and using the prediction results as the new operator execution time on the extrapolated trace. While we mainly use Li's Model, TrioSim add a significant extension to Li's Model by supporting DNN training.

4.5 Network Modeling

The goal of TrioSim is to support the design of large-scale multi-GPU distributed DNN training systems. A major design factor is the network that communicates the GPUs and computing nodes. Therefore, developing or integrating a network performance model in TrioSim is necessary.

A major consideration with integrating an existing architectural network simulator [26] is performance. Most existing network simulators [26] in the computer architecture domain model networks in cycle-level details, which ensures great accuracy at the cost of slow simulation. To allow fast network simulation, we must discard the transmission details and focus on key factors that impact the performance. In packet-switch networks, packets are divided into flits (the units that can be sent in one cycle), and the flits are independently transferred to the destination. The links can be generally considered as pipelines (stages and cycles per stage jointly control throughput and latency), allowing for easy modeling and performance estimation.

TrioSim consider the transfer of a packet as a 4-step process, including 1) routing, 2) bandwidth allocation, 3) progress update, and 4) delivery. When a packet starts to send (see Figure 5), the first step is to establish a packet-switching network route. We use the shortest path algorithm to determine the route. Then, once the route is determined, we allocate bandwidth for the links on the route. Given that we use an event-driven simulation engine, we schedule a potential delivery event at the time calculated using Little's law [39], assuming the bandwidth allocation will not change afterward. Scheduling events allow us to skip the transfer details and fast-forward the simulation. In case if there is a new packet starts or completes the transfer, we recalculate the bandwidth and re-schedule the delivery events of all the packets that are in transit currently. Finally, at the time of the delivery event, we push the packet to the destination buffer, notify the receiver, and recalculate the bandwidth allocation for other packets.

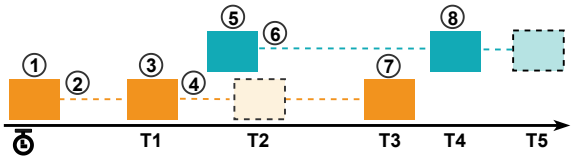


Figure 5: Message sending process in packet-switching network in TrioSim. Case A: only one message in the link. 1) A new message comes. 2) Routing, bandwidth allocation, and scheduling an event at the time, T1. 3). No new message comes, the destination device receives the message at the time, T1. Case B: two messages in the link. 4) Send the message to the destination. Routing, bandwidth allocation, and scheduling events at the time, T2. 5) A new message starts to transfer, and the current two messages share at least one link. 6) Recalculate the bandwidth and reschedule the delivery events of all the packets that are currently in transit at the time, T3 and T5 separately. 7) The destination receives the orange message at the time, T3. Because the orange message is received, it recalculates the bandwidth and reschedules the delivery event for the blue message at the earlier time, T4. 8) The destination receives the message at the time, T4.

While TrioSim supports packet-switching networks by default, the network model can be easily swapped. In subsection 7.1, we will demonstrate TrioSim's capability of modeling a circuit-switching photonic network.

5 Experiment Methodology

We use a series of experiments to validate the accuracy of TrioSim. The details of the experiments are provided below.

Hardware environment. We validate against three platforms (P1, P2, P3). P1 has 2 NVIDIA A40 GPUs, P2 has 4 NVIDIA A100 GPUs, while P3 has 8 NVIDIA H100 GPUs. GPUs in P1 are connected with PCIe, while GPUs in P2 and P3 are connected with NVLinks. Both the GPU and connection diversity demonstrate that TrioSim is capable of modeling the performance of a wide range of configurations.

Users can set up any bandwidth value of the links in TrioSim. For validation purposes, we need to provide a reasonable bandwidth that matches the hardware being validated against. We find that the theoretical bandwidth of the links is not usually useful as the achieved bandwidth is usually only a fraction of the theoretical one. To measure the achieved bandwidth, we use nccltest [48]. We use the measured bandwidth as the input parameter for communication in our simulator. Throughout all the experiments, we use a single set of throughput numbers for the platforms.

Software environment. We use the same system configuration for all the platforms. Python and CUDA versions are 3.10.12 and 12.1, respectively. For PyTorch and its related libraries, we use torch 2.1.0+cu121, torchvision 0.16.0+cu121, and torchaudio 2.1.0+cu121

correspondingly. For transformer models, since the pipeline parallelism of transformer training is under active development, we use the nightly versions, torch 2.5.0.dev20240623+cu121, torchvision 0.20.0.dev20240623+cu121, and torchaudio 2.4.0.dev20240623+cu121.

Workloads. We choose two types of representative DNNs. The first type is image classification DNNs, including DenseNet [24] (DenseNet-121, DenseNet-161, DenseNet-169, DenseNet-201, represented by DN-121, DN-161, DN-169, DN-201 in figures) and ResNet[23] (ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-152, represented by RN-18, RN-34, RN-50, RN-101, RN-152 in figures), and VGG [66] (VGG-11, VGG-13, VGG-16, VGG-19), all from the torchvision library. The second type includes transformer DNNs, mainly used for natural language processing (NLP), including GPT-2 [58], BERT [9] (BERT-Base-Uncased), T5 [59] (T5-Small), FLAN-T5 [7] (FLAN-T5-Small), and Llama [13, 70, 71] (Llama-3.2-1B) sourced from Hugging Face [73].

Parallelism strategy. We validate TrioSim against real hardware with standard data parallelism, distributed data parallelism (DDP), tensor parallelism, and pipeline parallelism training of DNN models in Pytorch library.

Standard data parallelism and distributed data parallelism are modules, `DataParallel` and `DistributedDataParallel` provided by PyTorch. The key difference is that distributed data parallelism is multi-processing while standard data parallelism is multi-threading, allowing distributed data parallelism to avoid performance overhead caused by the Global Interpreter Lock (GIL) of the Python interpreter. A more simulator-related difference is that distributed data parallelism overlaps NCCL communication with layer backward propagation calculation, while standard data parallelism waits until all the backward propagation is finished before triggering the NCCL communication process.

For tensor parallelism, we use the open-source library Black-Samorez/tensor_parallel [5]. For pipeline parallelism, we use the PyTorch module, `torch.distributed.pipeline.sync.pipe` for image classification DNNs. And for transformer DNNs, we use `torch.distributed.pipelining`. We need another module for transformers because `torch.distributed.pipeline.sync.pipe` requires converting the model layers in a sequential manner to define the desired order of execution before the parallelism. However, it does not support transformer models in the experiment set. Both of the libraries are built based on the GPipe algorithm, which maps the pipeline parallelism implementation method in TrioSim.

Time measurement. We measure end-to-end execution time using `torch.cuda.Event` API provided by PyTorch. It can take time stamps before and after the execution of each batch. We run each batch for 41 iterations, with the first 30 batches serving as a warm-up phase. For the single GPU case, we calculate the average time for batches 31 to 40 after the warm-up. We use the PyTorch Profiler to gather layer or kernel time information for batch 41. The data gathering happens on different batches to prevent profiling overhead impact on measured performance. Additionally, we use the Execution Graph Observer tool to collect detailed input, output, and other tensor or data information from batch 41. For each DNN, we also run multi-GPU cases that apply different parallelisms. We average the execution time from batch 31 to 40, after the 30 batch warm-up, as the ground truth to be validated against. Note that, for the real-hardware time when using tensor parallelism in our experiment, we only count the time that at least one GPU is busy or at least one data movement task is taking place. We do not consider CPU overhead, because the main goal of TrioSim is to model large-scale DNN training where GPUs are likely to be heavily utilized.

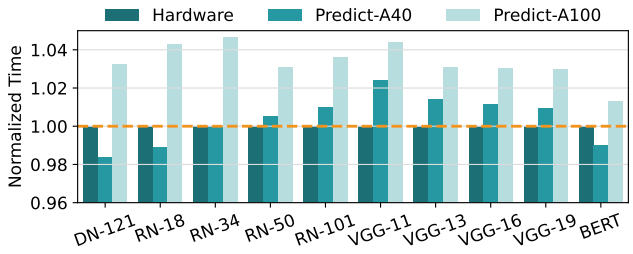


Figure 6: Comparing the TrioSim-predicted and real-hardware time when the batch size is 256, while only providing TrioSim with traces collected for batch size 128. Other models are out of memory when the batch size is 256 on real hardware. The average error is 1.10% and 3.25% for A40 and A100, respectively.

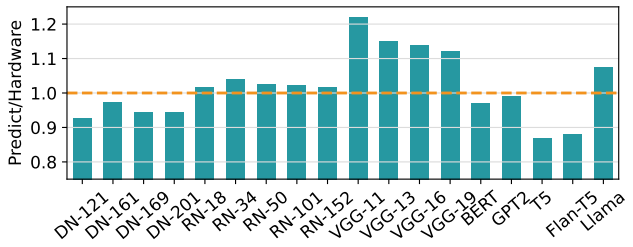


Figure 7: Comparing the TrioSim-predicted and real-hardware time when applying standard data parallelism on P1. The average error of the experiment set is 7.39%.

6 Validation

We present a series of experiment results to demonstrate the validity of TrioSim, focusing on comparing the TrioSim-predicted and real-hardware execution time.

Single GPU validation. First, we validate the accuracy of TrioSim for single GPU training. We use the single GPU trace when the batch size is 128 to predict the single GPU time when the batch size is 256. Our results in Figure 6 show that the normalized times (predicted time/real hardware time) of all the models are close to 1, which indicates that TrioSim can accurately predict the performance for both A40 and A100. The average error across all tested models is 1.10% for A40 GPU and 3.25% for A100 GPU.

Data parallelism validation. Next, we validate if TrioSim can correct model standard data parallelism in Pytorch on real hardware. We use TrioSim to predict the performance of data parallelism training and compare the results with the real hardware execution time on P1. The results (see Figure 7), demonstrate that TrioSim can accurately predict the performance of models training with

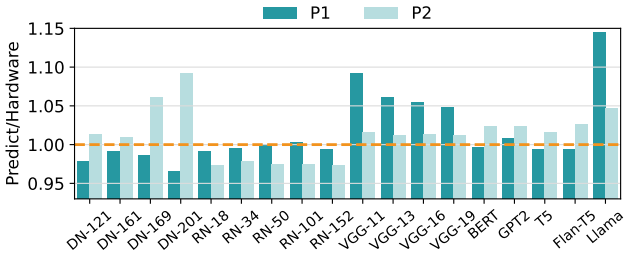


Figure 8: Comparing the TrioSim-predicted and real-hardware time when applying distributed data parallelism on P1 and P2. The average errors of the experiment set for P1 and P2 are 2.91% and 2.73%, respectively.

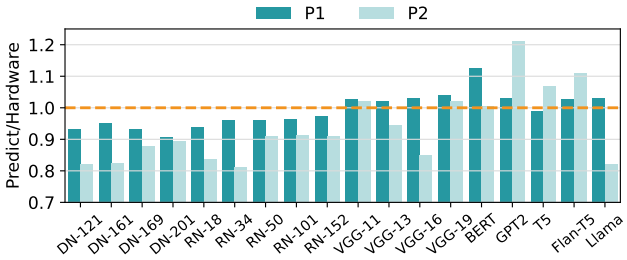


Figure 9: Comparing the TrioSim-predicted and real-hardware time when applying tensor parallelism on P1 and P2. The average errors of the experiment set for P1 and P2 are 4.54% and 11.24%.

standard data parallelism. The average error across all models in the experiment set is 7.39%.

Second, TrioSim simulates distributed data parallelism using `DistributedDataParallel` from PyTorch on real hardware. We simulate the training performance with distributed data parallelism on P1 and P2. The predictions are more accurate than the standard data parallelism, with errors (see Figure 8) as low as 2.91% and 2.73% on P1 and P2, respectively. TrioSim is better at predicting distributed data parallelism, which is recommended over standard data parallelism because distributed data parallelism overlaps NCCL AllReduce operation with backpropagation. However, the prediction of both standard data parallelism and distributed data parallelism are sufficiently accurate to satisfy academic research purposes.

Tensor parallelism validation. TrioSim also simulates the DNN training with tensor parallelism. We simulate the time for tensor parallelism on P1 and P2, getting average errors (see Figure 9) of 4.54% and 11.24%, respectively. The results demonstrate TrioSim's effectiveness in modeling the performance of workloads that apply tensor parallelism.

Pipeline parallelism validation. We simulate pipeline parallelism for different DNNs on P2 with 2 A100 GPUs used and 4 A100 GPUs used, separately. During the simulation, we divide each mini-batch into micro-batches with the number of micro-batches (chunks) equal to 1, 2, and 4. Micro-batches are processed in parallel in different pipeline stages.

We feed in TrioSim with a single GPU trace with a batch size of 128 (for Llama, we drop this to 16 to avoid out-of-memory issues during real-hardware tracing). As the number of chunks increases, the micro-batch size on the GPUs becomes smaller. This reduction in micro-batch size can cause CPU scheduling overhead to become significant, leading to longer total end-to-end execution times, contrary to the theoretical expectation that execution times should decrease with more chunks. For instance, for DenseNet-169, the times are roughly 1.4x for 4 chunks compared with 2 chunks on P2. Therefore, these cases are not the main targets of TrioSim, as indicated by bars with an orange triangle on the top in Figure 10 for DenseNet-169 and DenseNet-201 on the 2 A100 GPUs simulation and for ResNet-18, ResNet-101, ResNet-152, DenseNet-169, and DenseNet-201 on the 4 A100 GPUs simulation.

The results shown in Figure 10 suggest that TrioSim can also handle the simulation for pipeline parallelism with reasonable accuracy. The average error for simulations with 1 chunk, 2 chunks, and 4 chunks on 2 A100 GPUs is 6.82%, 6.58%, and 15.10%, respectively, while the average error for simulations with 1 chunk, 2 chunks, and 4 chunks on P2 simulation is 5.14%, 8.96%, and 8.18%, respectively.

New GPUs validation. We validate various parallelism strategies on P3 to demonstrate TrioSim's effectiveness on an 8-GPU configuration. At the same time, to show that TrioSim can predict performance for new GPUs and varying batch sizes, we first use traces from single A40 and A100 GPUs with a batch size of 128—following Li's Model—to predict the performance on the 8 H100 system at batch size 256. For comparison, we also collect traces for a batch size of 256 on a single H100 GPU and use it as input of TrioSim to predict performance for the 8 H100 system.

Figure 11 shows TrioSim's prediction errors of real hardware time for models using distributed data parallelism, tensor parallelism, and pipeline parallelism (with different chunk sizes) on P3. Transformer models are excluded from the figures because collecting traces on real hardware leads to out-of-memory errors and incomplete executions, necessitating non-trivial code modifications and debugging. For Case 1 (cross-GPU prediction), the average error is 9.09% for data parallelism, 9.07% for tensor parallelism, and 5.65%, 16.28% for pipeline parallelism with 1, and 2 chunks, respectively. For Case 2 (same-GPU prediction), the average error is 6.69% for data parallelism, 9.09% for tensor parallelism, and 4.20%, 13.76% for pipeline parallelism with 1, and 2 chunks, respectively. Although cross-GPU modeling adds errors, the error remains relatively low.

Parallelism comparison. To demonstrate that TrioSim provides accurate relative comparisons between different configurations, we experiment on P2 to compare the performance of data, tensor, and pipeline parallelism across models, using a fixed total batch size of 128 on 4 GPUs and a pipeline micro-batch size of 64. The results (see Figure 12) demonstrate that TrioSim can always predict the relative performance across parallelism. Both the actual hardware results and TrioSim support that data parallelism is the most efficient option when the total workload is constant. Tensor parallelism generally does not perform well except on transformers. Moreover, TrioSim accurately predicts for each model whether tensor parallelism is more efficient than pipeline parallelism.

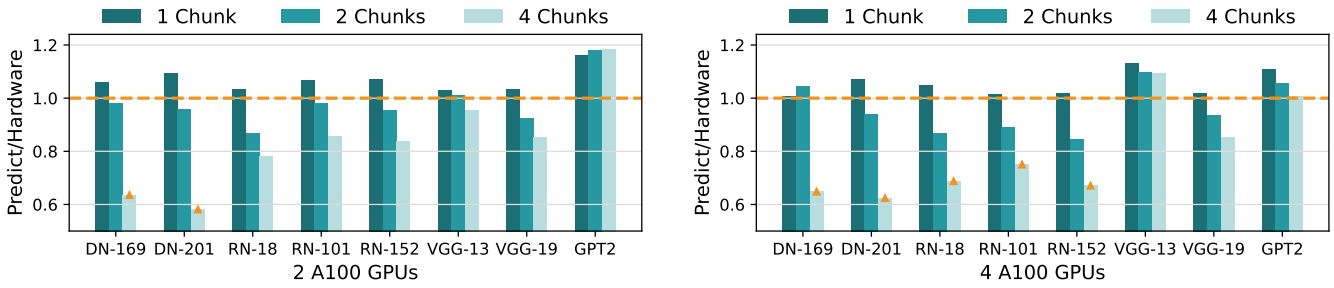


Figure 10: Comparing the TrioSim-predicted and real-hardware time when applying pipeline parallelism on 2 A100 GPUs and 4 A100 GPUs with different chunks. Other models are not shown because they require extension code modification to support pipeline parallelism. The bar with an orange triangle on the top means an abnormal value contrary to the theoretical expectation that execution times should decrease with more chunks. We do not address this particular case as they are not the main target of TrioSim. The average errors of the experiment set for 2 A100 GPUs are 6.82%, 6.58%, and 15.10% for 1 chunk, 2 chunks, and 4 chunks, respectively. The average errors of the experiment set for 4 A100 GPUs are 5.14%, 8.96%, and 8.18% for 1 chunk, 2 chunks, and 4 chunks, respectively.

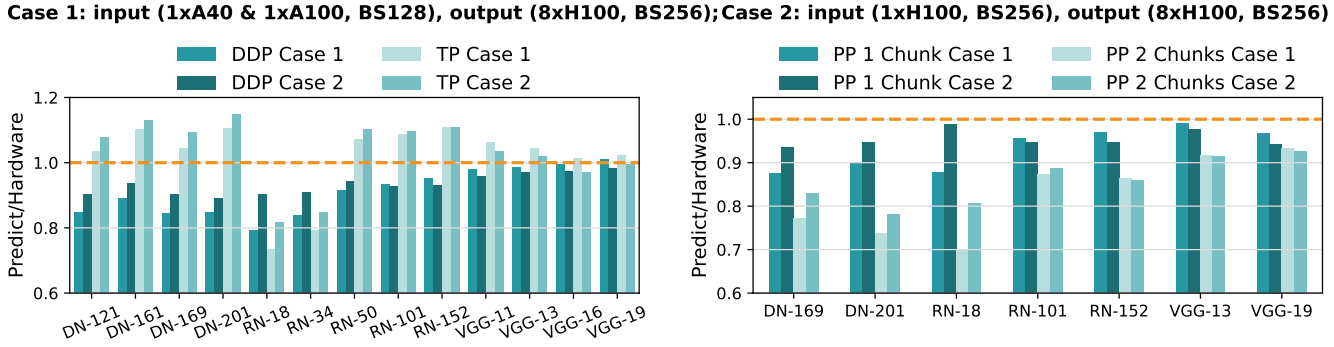


Figure 11: Comparing the TrioSim-predicted and real-hardware time when applying distributed data (left), tensor (left), and pipeline (right) parallelism with different chunk sizes on P3. Each group shows four bars: two bars for Case 1—use input traces collected on single A40 and single A100 to predict the 8 H100 system; two bars for Case 2—use input traces collected on single H100 to predict the 8 H100 system.

Communication and computation time offered by TrioSim. TrioSim can provide a detailed timeline trace of the multi-GPU trace extrapolator. Similarly, TrioSim can extract and output the communication and computation time for each DNN model (see Figure 13 for the breakdown of communication and computation times on P1). The breakdown of times allows users to analyze the impact of communication on total GPU time. The results indicate that the communication time ratio in tensor parallel is higher than in data parallel on P1. By evaluating the total computation and communication times, users can make informed decisions about which parallelism strategy to use and determine the optimal number of GPUs for their multi-GPU systems.

TrioSim execution time. The simulation of TrioSim can be completed within seconds. Figure 14 illustrates the simulator's execution time when modeling distributed data parallelism for P2. The execution time is primarily influenced by the total number of GPUs in the system, and the size of the trace files.

7 Case Studies

In this section, we perform two case studies to demonstrate how TrioSim can easily evaluate novel designs.

7.1 Photonic-Connected Wafer-Scale GPUs

In prior experiments, we have utilized a packet-switching network to connect all the GPUs. However, other innovative hardware networks can enhance transmission efficiency. We present a case study that employs a photonic network to connect the GPUs in the system to reduce communication overhead. We develop the photonic network model for TrioSim to replace the standard packet-switching network model. This case study demonstrates the capability and the low difficulty of modeling a new, user-defined network in TrioSim.

Photonic interconnects. Photonic interconnects [15, 30, 36, 43] utilize light as the medium for data transmission. The high frequency and broad usable frequency band of light signals enable these optical links to offer high bandwidth and low latency while maintaining energy-efficient information exchange channels.

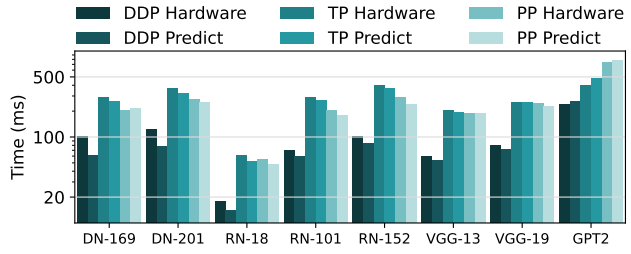


Figure 12: Comparing data, tensor, and pipeline parallelism on P2, with a fixed total batch size of 128 for 4 GPUs and a pipeline micro-batch size of 64.

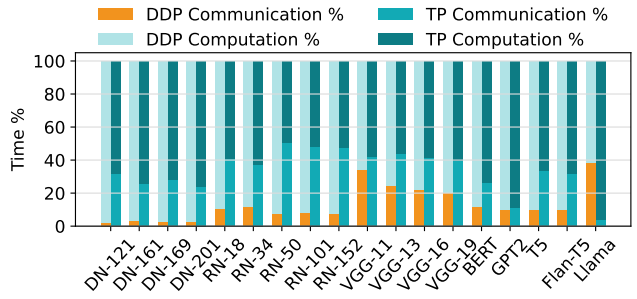


Figure 13: Communication and computation time ratio for models training with tensor parallelism and distributed data parallelism simulations on P1.

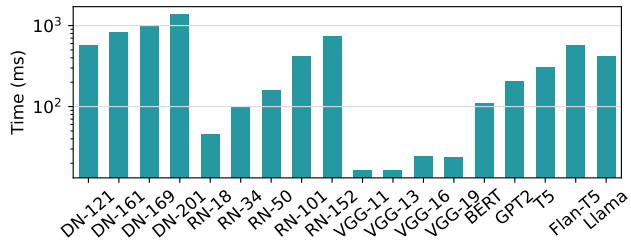


Figure 14: Simulator's execution time in log scale when modeling distributed data parallelism on P2.

A notable application of photonic interconnects on multi-GPU systems is Lightmatter's Passage [21]. Passage provides a photonic interposer that connects multiple chiplets (e.g., GPUs, accelerators, memory pool) on a wafer. Being a circuit-switching network, Passage requires establishing logical links that occupy frequency bands of the underlying fiber. Once the logical links are established, each Passage connection can deliver high bandwidth and ultra-low latency. Consequently, packets can be delivered to any node on the wafer with nearly the same latency, regardless of physical distance.

Photonic network model implementation. TrioSim only requires a network model to implement the `Send` and `Deliver` functions that mark the start and end of a transfer, respectively. While the `Deliver` function is straightforward and only involves passing the data to the receiver by reference, the `Send` function is a 3-step process, including 1) establishing the link, 2) reserving buffer space, and 3) moving data.

Upon sending data, we first check if there is already a link established between the sender and receiver. If no link is available, we spend some time (the latency is configurable) establishing the link by scheduling a simulator event. Since the photonic port of a GPU is limited, we destroy the unused links that have been idle for the longest time if there is no port available. After establishing the link, the sender waits until the destination has buffer space available. Note that this process does not require polling, but the Akita Simulator Engine used in TrioSim can support notifying the sender when the buffer space is available. Finally, once we reserve the buffer space, we can calculate the delivery time as the data size divided by bandwidth, plus the link's latency. We then schedule the delivery event according to the calculated transfer time.

Configuring the hardware platform to use the new interconnect is straightforward. A user would only need to instantiate the photonic interconnect class and call the `PlugIn` method to associate the device port with the connection. No need to modify the device code since the data transfer is independent of device logic.

Hardware configuration. Since Passage is designed for wafer-scale systems, we model a wafer with $12 \times 7 = 84$ GPUs, with each GPU equivalent to an NVIDIA A100 GPU. The number is consistent with the reticles on Cerebras Wafer-Scale Engine design [31] and is easily configurable. In our experiment, we configure the Passage to provide a bandwidth of 484 GB/s across 8 links and set the latency for establishing a link to 20 ms. For electrical and optical networks, we use the mesh network on the wafer to form a ring to perform AllReduce operations.

Results. The results (see Figure 15) indicate that at such a scale, communication significantly impacts the execution time in the electrical network system. For instance, in networks like VGG-19, communication accounts for 92.21% of the total execution time, demonstrating that relying solely on data parallelism training is suboptimal for electrical network-based wafer-scale GPUs. In contrast, the optical network reduces communication time by nearly half. However, this case study also reveals that merely adopting a photonic network does not completely resolve scalability issues. Further design of communication schemes is necessary to fully leverage the capabilities of fiber-connected wafer-scale GPUs.

7.2 Heterogeneous Training with Hop

Heterogeneity among multiple worker nodes is a critical challenge in distributed training tasks, which leads to performance bottlenecks as faster workers must wait for slower ones during synchronization phases. The impact of this issue is particularly profound in decentralized training systems that rely on synchronous communication methods like AllReduce, which are not designed with consideration of heterogeneity. TrioSim's flexibility makes it a good fit for studying heterogeneous multi-GPU systems. Simulating such systems with other similar tools [3, 41, 60] can be challenging due to their limited capability of deviating from the traces collected.

In this case study, we repeat the experiments conducted in the Hop [42] design. The Hop [42] protocol introduces a heterogeneity-aware approach to decentralized training. Hop's main idea is to

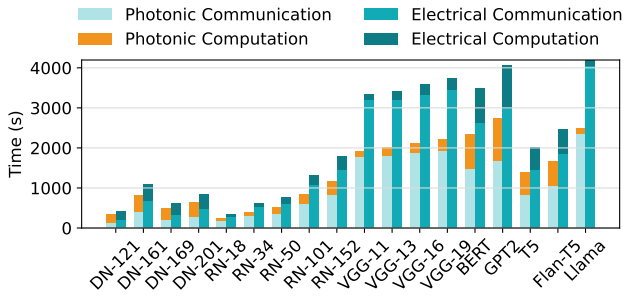


Figure 15: The performance of a wafer-scale GPU with $12\times7 = 84$ chiplets connected with electrical and photonic interconnects when training DNNs with data parallelism. Optical networks can significantly reduce communication time. The time for Llama in this figure is scaled down by a factor of 56.

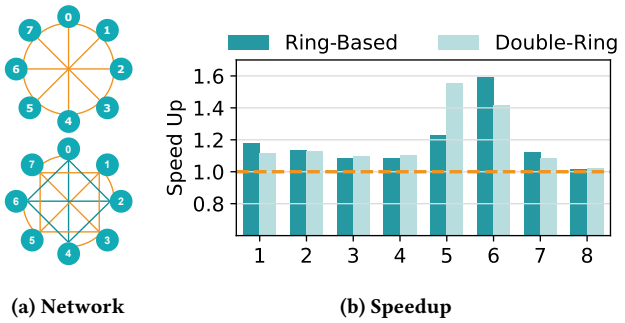


Figure 16: (a) Ring-based graph (top): GPUs are connected in a circular topology with bidirectional edges, and an additional connection is established between each node and its most distant node. Double-ring graph (bottom): two ring-based graphs are interconnected node-to-node. (b) Each data group (8 groups in total) on the x-axis represents a random slowdown scenario in an 8 A100 system. The y-axis represents the speedup achieved with one backup worker in heterogeneous multi-GPU systems connected with ring-based and double-ring networks.

use a queue-based synchronization mechanism, which intelligently manages the iteration time differences between worker nodes. This mechanism is implemented by update queues and token queues. Update queues are dynamically sized based on the observed iteration gaps, enabling efficient coordination without waiting for the slowest node. Meanwhile, token queues regulate the iteration gaps more strictly to prevent excessive divergence among worker states. Hop enhances decentralized training with several strategic features, including backup workers, bounded staleness, and skipping iterations. Backup workers are introduced to maintain productivity when primary workers become bottlenecks. Bounded staleness leverages the flexibility of queue-based synchronization. Hop allows for a controlled staleness in updates, exploiting the known benefits of stale synchronous parallel models without compromising convergence.

Configuration. We implement the Hop protocol in TrioSim and conduct extensive experiments to evaluate the impact of backup workers on different network configurations. Our experimental setup includes 8 A100 GPUs, with each simulation of VGG-11 running a batch size of 128, matching the workload size in Hop paper. We model a heterogeneous environment by introducing a mechanism that slows down GPU communication bandwidth by a factor of random number between 1 and 10. The experiments are conducted across two different types of communication graphs (see Figure 16: ring-based and double-ring). To explore the effectiveness of backup workers, we introduce one backup worker in both the ring-based and double-ring configurations, meaning that each node could miss one update without impacting the overall progression. This configuration demonstrates TrioSim's capability of simulating non-standard networks and asymmetrical GPU configurations.

Results. Our dataset consists of 8 groups, representing scenarios with random slowdowns across 8 A100 GPUs. Each group corresponds to the speedup achieved under different configurations: the ring-based network with one backup worker versus without a backup worker, and the double-ring network with one backup worker versus without a backup worker. As shown in Figure 16, the backup worker's effect on performance greatly varies with different slow-down ratios. The results demonstrate significant variations in performance, which proves that backup workers enhance the robustness and efficiency of decentralized training systems across diverse network environments as in the paper [42].

Our implementation of Hop within TrioSim showcases the simulator's robust flexibility and capability to adapt to different network conditions and training configurations. TrioSim's architecture allows users to integrate various synchronization strategies, including the innovative approaches proposed by Hop. This adaptability makes it an ideal platform for exploring and validating cutting-edge distributed training protocols like Hop.

8 Discussion

8.1 Is TrioSim Sufficiently Accurate

TrioSim is a tool designed to quickly model and predict execution times of large-scale DNN training, which may vary by several orders of magnitude. Yet, TrioSim provides highly useful estimations as errors fall within a reasonable range. TrioSim generally demonstrates an error of less than 20%, with many instances showing errors limited to less than 10% depending on the complexity of the workload and system. Such error rates are generally considered adequate for future academic research.

8.2 Error Sources

TrioSim is designed to provide efficient and accurate predictions for large-scale DNN training, but its accuracy can be influenced by certain error sources and limitations in its modeling assumptions. These issues primarily stem from the linear regression-based performance model, network model, and multi-GPU trace extrapolator.

Linear regression-based performance model. The linear regression-based performance model predicts GPU execution times by extrapolating single-GPU traces and scaling batch sizes based on the parallelism strategy (e.g., data, tensor, or pipeline parallelism). This approach works well for large-scale workloads where GPUs

are fully utilized, but it encounters challenges with smaller batch sizes or lightweight models. Under these conditions, the GPUs may not exert enough pressure on computational or memory resources, leading to inaccuracies in predicted performance. To address this limitation, TrioSim allows the integration of alternative compute models, such as NeuSight [32], which can better handle underutilized workloads, offering users the flexibility to refine predictions based on specific scenarios.

Network model. The network model simulates inter-GPU communication and memory movement using Little's law [39], estimating transfer times based on bandwidth and latency. However, TrioSim does not model communication protocols or account for nuanced differences caused by varying data transfer unit sizes or handling multiple simultaneous data streams, which could introduce minor inaccuracies. These limitations make accurately simulating fine-grained network behaviors challenging. Despite this, TrioSim's high prediction accuracy demonstrates that it effectively captures the most critical factors, such as latency and bandwidth while excluding less impactful details.

Trace extrapolation. The multi-GPU trace extrapolator expands single-GPU traces to simulate multi-GPU execution, introducing potential errors when replicating the process for parallelism strategies. For example, in pipeline parallelism, the simulator automatically assigns layers to GPUs to balance workloads, which may not align with the actual behavior of real-world systems. Discrepancies can arise if interdependencies between layers or communication bottlenecks are not accurately represented, potentially leading to deviations between simulated and actual performance.

CPU overhead. TrioSim only considers GPU compute and data movement (including CPU to GPU data movement). We observe that, when training small workloads, the CPU can be the performance bottleneck [20] (as seen in Figure 10). However, small workloads are not the main goal of TrioSim. When training large workloads, the data movement and layer computing dominate the execution time. Estimating CPU overhead requires more sophisticated CPU performance models, which requires dedicated future research.

8.3 How TrioSim Should Be Used

TrioSim is primarily designed for system-level research, focusing on exploring the performance of large-scale DNN workloads on multi-GPU systems. Rather than modeling fine-grained architectural changes, TrioSim enables researchers to study high-level system behaviors, such as the impact of parallelization strategies, network topologies, and scaling configurations. For example, given an LLM and a specific GPU interconnect topology, users can evaluate different parallelism strategies (data, tensor, or pipeline parallelism) to determine the most efficient configuration. Because TrioSim can accurately model multi-GPU execution using just one GPU trace, this single-trace capability—allows for unlimited parameter changes—is well-suited for this purpose.

Although TrioSim does not support value-based simulation or fine-grained architectural modeling, it can be combined with low-level simulators for architecture-level investigations. If a hardware modification (e.g., a new TensorCore design or atomic operation optimization) impacts specific DNN layers, a viable approach is to first simulate those layers using a cycle-accurate simulator (such as gem5 [62], Accel-Sim [28], or MGPUSim [67]) to obtain updated execution time estimates. These adjusted execution times can then be fed into TrioSim's trace-driven extrapolation framework, allowing users to observe the impact of architectural changes at full scale across multi-GPU configurations and end-to-end workloads. This hybrid workflow enables large-scale performance evaluation, which would be infeasible using cycle-accurate simulators alone due to their high computational cost.

8.4 Limitations

The design of TrioSim makes key trade-offs to enable efficient evaluation of regular DNN workloads with limited computing resources. As a result, it has several limitations: (1) TrioSim does not support functional emulation and therefore cannot model value-dependent or highly irregular workloads. (2) Fine-grained hardware modifications cannot be directly evaluated because TrioSim is not an architecture or cycle-level simulator. (3) TrioSim does not collect kernel traces related to communication. Instead, TrioSim recreates the behavior of the open-sourced NCCL implementation as part of the extrapolation process. If NCCL changes in the future, a user may need to update TrioSim to adapt. (4) Limited by Li's Model [34], TrioSim assumes high GPU utilization, making it less accurate for the cases where the CPU is the bottleneck or the kernels are small. (5) The highly abstract network model ignores protocol-level effects and low-level communication behaviors. And (6) TrioSim has only been validated against CNNs and Transformers. We leave supporting other types of workload (e.g., reinforcement learning, Graph Neural Network) as future work. Despite these limitations, the high accuracy observed across diverse workloads demonstrates that TrioSim can provide trustworthy insights and reliable guidance for designing large-scale DNN systems.

9 Conclusions

This paper introduces TrioSim, a trace-driven simulator that enables simulating large-scale DNN workloads on multi-GPU systems with high accuracy and short simulation time. The capability of quickly modeling the performance of such large-scale systems is becoming increasingly critical amid the extreme popularity of LLMs. TrioSim only requires single GPU traces to serve as input, significantly reducing the hardware requirement to perform research in the domain. Combining high-level performance models and network models allows TrioSim to simulate the performance of a wide range of DNNs, parallelisms, and inter-GPU networks. TrioSim represents our continuous effort in finding a mid-ground between detailed simulations and low-level simulations, balancing simulation accuracy and performance.

Acknowledgments

We thank the anonymous reviewers for their detailed and constructive feedback. This material is based upon work supported by the US National Science Foundation (NSF) awards (#2402804, #2402805). Part of this work was conducted under the Laboratory Directed Research and Development Program at Thomas Jefferson National Accelerator Facility for the U.S. Department of Energy.

References

[1] Cesar Avalos Baddouh, Mahmoud Khairy, Roland N Green, Mathias Payer, and Timothy G Rogers. 2021. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 724–737.

[2] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. 2017. An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures. In *Proceedings of the Machine Learning on HPC Environments*. 1–8.

[3] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2023. vtrain: A simulation framework for evaluating cost-effective and compute-optimal large language model training. *arXiv preprint arXiv:2312.12391* (2023).

[4] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2022. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *The 31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[5] BlackSamorez. 2024. *Tensor Parallel*. Retrieved July 20, 2024 from https://github.com/BlackSamorez/tensor_parallel

[6] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro* 38, 2 (2018), 8–20.

[7] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, 70 (2024), 1–53.

[8] Jinku Cui, Qidong Zhao, Yueming Hao, and Xu Liu. 2024. DrPy: Pinpointing Inefficient Memory Usage in Multi-Layer Python Applications. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 245–257.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2024. D-STACK: High Throughput DNN Inference by Effective Multiplexing and Spatio-Temporal Scheduling of GPUs. *IEEE Transactions on Cloud Computing* (2024).

[11] Weiguang Ding, Ruoyan Wang, Fei Mao, and Graham Taylor. 2014. Theano-based large-scale visual recognition with multiple gpus. *arXiv preprint arXiv:1412.2302* (2014).

[12] Lukasz Drzewiecki and Monika Antoniak-Lewandowska. 2007. Flow Simulator-a flow-based network simulator. In *EUROCON 2007-The International Conference on" Computer as a Tool"*. IEEE, 2132–2136.

[13] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[14] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. 2013. Data transfer matters for GPU computing. In *2013 International Conference on Parallel and Distributed Systems*. IEEE, 275–282.

[15] Michael Georgas, Jonathan Leu, Benjamin Moss, Chen Sun, and Vladimir Stojanović. 2011. Addressing link-level design tradeoffs for integrated photonic interconnects. In *2011 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 1–8.

[16] Thomas J Giuli and Mary Baker. 2002. Narses: A scalable flow-based network simulator. *arXiv preprint cs/0211024* (2002).

[17] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 134–144.

[18] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501.

[19] Yueming Hao, Nikhil Jain, Rob Van der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. 2023. DrGPU: A Top-Down Profiler for GPU Applications. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 43–53.

[20] Yueming Hao, Xu Zhao, Bin Bao, David Berard, Will Constable, Adnan Aziz, and Xu Liu. 2023. TorchBench: Benchmarking PyTorch with High API Surface Coverage. arXiv:2304.14226 [cs.LG]

[21] Nicholas C Harris, Darius Bunandar, Ajay Joshi, Ayon Basumallik, and Robert Turner. 2022. Passage: A wafer-scale programmable photonic communication substrate. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 1–26.

[22] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[24] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[26] Nan Jiang, George Michelogiannakis, Daniel Becker, Brian Towles, and William J Dally. 2010. Booksim 2.0 user's guide. *Standford University* (2010), q1.

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

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

[29] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. 2017. Performance analysis of CNN frameworks for GPUs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 55–64.

[30] Ashok V Krishnamoorthy, Ron Ho, Xuezhe Zheng, Herb Schwetman, Jon Lexau, Pranay Koka, GuoLiang Li, Ivan Shubin, and John E Cunningham. 2009. Computer systems based on silicon photonic interconnects. *Proc. IEEE* 97, 7 (2009), 1337–1361.

[31] Mandy La and Andrew Chien. 2020. Cerebras Systems: Journey to the Wafer-Scale Engine. *University of Chicago, Tech. Rep* (2020).

[32] Seonho Lee, Amar Phanishayee, and Divya Mahajan. 2024. Data-driven Forecasting of Deep Learning Performance on GPUs. *arXiv preprint arXiv:2407.13853* (2024).

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

[34] Ying Li, Yifan Sun, and Adwait Jog. 2023. Path Forward Beyond Simulators: Fast and Accurate GPU Execution Time Prediction for DNN Workloads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 380–394.

[35] Ying Li, Yifan Sun, and Adwait Jog. 2023. A Regression-based Model for End-to-End Latency Prediction for DNN Execution on GPUs. In *the Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), Raleigh, NC*. 343–345.

[36] Yuan Li, Ke Wang, Hao Zheng, Ahmed Louri, and Avinash Karanth. 2022. Ascend: A scalable and energy-efficient deep neural network accelerator with photonic interconnects. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 7 (2022), 2730–2741.

[37] Shao-Fu Lin, Yi-Jung Chen, Hsiang-Yun Cheng, and Chia-Lin Yang. 2023. Tensor Movement Orchestration in Multi-GPU Training Systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1140–1152.

[38] Zhongyi Lin, Louis Feng, Ehsan K Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Arun Kejariwal, and John D Owens. 2022. Building a performance model for deep learning recommendation model training on gpus. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 48–58.

[39] John DC Little and Stephen C Graves. 2008. Little's law. *Building intuition: insights from basic operations management models and principles* (2008), 81–100.

[40] Changxi Liu, Yifan Sun, and Trevor E Carlson. 2023. Photon: A fine-grained sampled simulation methodology for GPU workloads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1227–1241.

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

[42] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware decentralized training. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 893–907.

[43] Christopher Monroe, Robert Raussendorf, Alex Ruthven, Kenneth R Brown, Peter Maunz, L-M Duan, and Jungsang Kim. 2014. Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects. *Physical Review A* 89, 2 (2014), 022317.

[44] Ali Mosallaei, Katherine Isaacs, and Yifan Sun. 2024. Looking into the Black Box: Monitoring Computer Architecture Simulations in Real-Time with AkitaRTM. In *The 57nd IEEE/ACM International Symposium on Microarchitecture*.

[45] Mahmood Naderan-Tahan, Hossein SeyyedAghaei, and Lieven Eeckhout. 2023. Sieve: Stratified GPU-Compute Workload Sampling. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 224–234.

[46] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[47] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. 2014. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics* 15, 2 (2014), 285–329.

[48] NVIDIA. 2024. *NCCL Tests*. Retrieved July 20, 2024 from https://github.com/NVIDIA/nccl-tests

[49] NVIDIA Corporation. 2018. *NVSwitch Technical Overview*. Retrieved July 25, 2024 from https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf

[50] NVIDIA Corporation. 2024. *NVIDIA Collective Communications Library (NCCL)*. Retrieved November 15, 2024 from https://developer.nvidia.com/nccl

[51] NVIDIA Corporation. 2024. *NVIDIA Nsight Compute*. Retrieved November 15, 2024 from https://developer.nvidia.com/nsight-compute

[52] NVIDIA Corporation. 2024. *NVIDIA NVLink*. Retrieved July 18, 2024 from https://www.nvidia.com/en-us/data-center/nvlink/

[53] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.

[54] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. 2019. Optimizing multi-GPU parallelization strategies for deep learning training. *Ieee Micro* 39, 5 (2019), 91–101.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[56] PCI-SIG. 2024. *PCI Express Base Specification*. Retrieved July 18, 2024 from https://pcisig.com/specifications

[57] PyTorch Contributors. 2024. *Distributed Pipelining*. Retrieved July 27, 2024 from https://pytorch.org/docs/main/distributed.pipelining.html

[58] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[59] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[60] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 81–92.

[61] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[62] Kyle Roarty and Matthew D Sinclair. 2020. Modeling modern gpu applications in gem5. In *gem5 Users Workshop*.

[63] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 49–68.

[64] Hossein SeyyedAghaei, Mahmood Naderan-Tahan, and Lieven Eeckhout. 2024. GPU Scale-Model Simulation. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1125–1140.

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

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

[67] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *46th International Symposium on Computer Architecture*.

[68] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D Shah, Cody Dunne, and David Kaeli. 2021. Daisen: A Framework for Visualizing Detailed GPU Execution. In *Computer Graphics Forum*, Vol. 40. 239–250.

[69] Alexey Svyatkovskiy, Julian Kates-Harbeck, and William Tang. 2017. Training distributed deep recurrent neural networks with mixed precision on GPU clusters. In *Proceedings of the Machine Learning on HPC Environments*. 1–8.

[70] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[71] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[72] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 335–344.

[73] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.

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

[75] Eric P Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1335–1344.

[76] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yang Li, Xiaowen Chu, and Huaicheng Li. 2024. Missile: Fine-Grained, Hardware-Level GPU Resource Isolation for Multi-Tenant DNN Inference. *arXiv preprint arXiv:2407.13996* (2024).

[77] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2020. GVProf: A value profiler for GPU-based clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[78] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: Exploring value patterns in GPU-Accelerated applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 171–185.