LUMOS: EFFICIENT PERFORMANCE MODELING AND ESTIMATION FOR LARGE-SCALE LLM TRAINING

Mingyu Liang [1] **Hiwot Tadese Kassa** [2] **Wenyin Fu** [2] **Brian Coutinho** [2] **Louis Feng** [2] **Christina Delimitrou** [3]

ABSTRACT

Training LLMs in distributed environments presents significant challenges due to the complexity of model execution, deployment systems, and the vast space of configurable strategies. Although various optimization techniques exist, achieving high efficiency in practice remains difficult. Accurate performance models that effectively characterize and predict a model's behavior are essential for guiding optimization efforts and system-level studies. We propose Lumos, a trace-driven performance modeling and estimation toolkit for large-scale LLM training, designed to accurately capture and predict the execution behaviors of modern LLMs. We evaluate Lumos on a production ML cluster with up to 512 NVIDIA H100 GPUs using various GPT-3 variants, demonstrating that it can replay execution time with an average error of just 3.3%, along with other runtime details, across different models and configurations. Additionally, we validate its ability to estimate performance for new setups from existing traces, facilitating efficient exploration of model and deployment configurations.

1 INTRODUCTION

In recent years, large language models (LLMs) have transformed many aspects of daily life. The availability of vast datasets, along with advancements in computational resources, has enabled the development of increasingly complex models, such as ChatGPT (Ouyang et al., 2022), LLaMA (Touvron et al., 2023), and PaLM (Chowdhery et al., 2023). However, efficiently training these LLMs presents significant challenges, necessitating both hardware and software innovations across the system stack.

To meet these demands, efforts have focused on addressing various bottlenecks. Key areas of optimization include the development of AI-specific hardware (e.g., NVIDIA GPUs (NVIDIA, c) and SmartNICs (Ma et al., 2022)), improvements in memory systems (Kwon et al., 2023; Rajbhandari et al., 2020), the design of optimal parallelism strategies (Zheng et al., 2022; Isaev et al., 2023), overlapping communication with computation (Hashemi et al., 2019; Narayanan et al., 2021), and advancements in algorithms (Beltagy et al., 2020; You et al., 2019).

Despite these innovations, ensuring training efficiency remains a significant challenge. Diagnosing inefficiencies in LLMs is particularly difficult because runtime traces produced by machine learning (ML) frameworks (PyTorch; TensorFlow, a) are often dense and require deep expertise to interpret effectively. Moreover, runtime behavior can vary significantly across different model architectures, deployment configurations, accelerator types, network infrastructures, and other system components. These variations can cause performance bottlenecks to shift unpredictably, making them difficult to identify and address. An additional challenge lies in the vast search space of optimization possibilities. Finding the optimal solution within this space is time-consuming and resource-intensive, as it requires extensive experimentation on real hardware, demanding significant resources and incurring high costs.

A key step toward achieving efficiency is to accurately characterize and understand the behavior of these models. One common approach is to build performance models that capture model execution, which also provide a solid foundation for further optimization studies. While existing efforts (Moolchandani et al., 2023; Isaev et al., 2023) develop analytical models to predict high-level performance based on exposed model parameters, they often miss essential underlying execution details. To address this limitation, recent work (Hu et al., 2022; Zhu et al., 2020; Lin et al., 2022; Bang et al., 2023) has leveraged runtime traces to construct fine-grained execution graphs, providing deeper insights into the execution process.

However, current modeling methods struggle to address the new complexities in modern LLMs. Training LLMs at scale involves deploying across multiple machines, introducing substantial communication overhead. As one example of optimization, overlapping computation with communication

[1]Cornell University, Ithaca, USA [2]Meta, Menlo Park, USA [3]Massachusetts Institute of Technology, Cambridge, USA. Correspondence to: Mingyu Liang <ml2585@cornell.edu>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

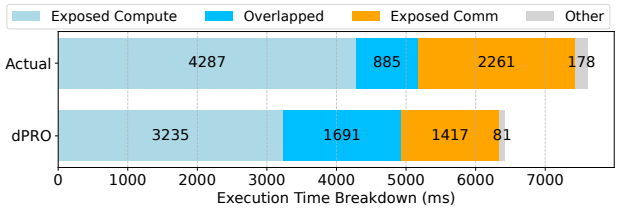


Figure 1. Execution breakdown for one training iteration of GPT-3 175B, configured with tensor parallelism = 8, pipeline parallelism = 4, and data parallelism = 8.

can reduce end-to-end training time, but it also introduces complex inter-stream dependencies that are challenging to model accurately. Figure 1 shows the execution time breakdown for a single training iteration of the GPT-3 (175B) model, along with replayed results from dPRO (Hu et al., 2022), a state-of-the-art trace-driven performance modeling tool. The comparison reveals significant gaps between the simulated and actual results, highlighting the challenges in capturing the full intricacies of LLM execution.

To overcome these difficulties, we propose Lumos, a trace-driven performance modeling and estimation toolkit for large-scale LLM training. To the best of our knowledge, Lumos is the first system to provide accurate performance models that effectively capture the execution behaviors of LLMs. It leverages built-in profiling tools from ML frameworks, such as PyTorch Kineto (Kineto), without requiring any custom instrumentation in models or frameworks, thereby minimizing the profiling overhead.

Furthermore, to streamline the exploration of optimization opportunities, Lumos also offers the flexibility to modify and generate new execution graphs from existing traces. This capability facilitates the exploration of optimal configurations, such as adjusting parallelism strategies (e.g., data and pipeline parallelism) and fine-tuning model architectures (e.g., number of layers, hidden size). By estimating performance through simulation rather than experimenting on real hardware, Lumos can significantly reduce cost and accelerate the optimization process.

The main contributions of our work are the following:

- By utilizing only built-in profiling traces from ML frameworks, Lumos constructs a comprehensive execution graph that identifies all dependencies between executed tasks, enabling accurate performance modeling of large-scale LLM training. Beyond estimating overall execution time, the fine granularity of Lumos allows it to reproduce detailed execution characteristics, facilitating deeper analysis and downstream optimization studies.
- With a detailed execution graph, Lumos offers users a convenient way to explore various model configurations, including adjustments to parallelism strategies and model architectures. By manipulating the existing graph to generate new ones for different configurations and by predicting performance through simulation, Lumos streamlines the optimization process and enables efficient and low-cost configuration exploration.
- We evaluate Lumos using various GPT-3 model variants on a production ML cluster with up to 512 NVIDIA H100 GPUs. Our results show that Lumos can accurately replay execution, achieving an average error of only 3.3% across different models and deployment configurations. Additionally, we demonstrate that Lumos accurately reproduces detailed execution statistics, such as execution time breakdown and SM utilization, showing significant improvements over existing approaches. Finally, we validate its ability to estimate performance for new configurations and deployments, achieving high accuracy when adjusting parallelism strategies and tuning various model architectures.

2 Related Work

2.1 Profiling Tools and Traces

As the ML system stack evolves rapidly, profiling tools play a crucial role in understanding model execution characteristics and identifying performance bottlenecks. As hardware accelerators like GPUs (NVIDIA, c) and TPUs (Jouppi et al., 2023) become increasingly essential, vendors offer specialized tools—such as NVProf (NVIDIA, d), CUPTI (NVIDIA, a), and Nsight (NVIDIA, e)—to expose hardware performance counters, providing developers with critical insights into performance metrics and enabling effective optimization.

To improve the interpretability of profiling results, ML frameworks also provide built-in tools for collecting execution statistics at the operator level. These tools often integrate hardware-level traces, offering a complete view of the entire stack—from host to device. For instance, PyTorch Kineto (Kineto) leverages CUPTI (NVIDIA, a) to capture runtime information for PyTorch operators, CUDA events, and GPU kernels, seamlessly linking them to provide a holistic perspective on model execution.

2.2 LLMs and Parallelism Strategies

Most modern LLMs are built on transformer architectures (Vaswani, 2017), which rely on self-attention mechanisms to capture long-range dependencies in sequential data. These models feature multiple stacked layers of attention and feedforward networks, with parameter sizes growing rapidly over the years. For example, GPT-2 (Radford et al., 2019) introduced in 2019 had 1.5 billion parameters, GPT-

3 (Brown, 2020) in 2020 expanded to 175 billion parameters, and PaLM (Chowdhery et al., 2023) reached 540 billion parameters by 2022.

Training LLMs presents significant computational and memory challenges, especially as model sizes grow beyond the capacity of individual GPUs. To address these limitations, 3D parallelism—a hybrid approach combining data, tensor, and pipeline parallelism—has become essential for efficient large-scale training (Narayanan et al., 2021; Shoeybi et al., 2019; Smith et al., 2022; Chowdhery et al., 2023). Each form of parallelism contributes uniquely: data parallelism (DP) distributes training batches across devices, synchronizing gradients during updates; tensor parallelism (TP) splits large tensors across multiple GPUs, allowing shared computation with frequent communication; and pipeline parallelism (PP) partitions the model into sequential stages, with each stage processed on different devices in a coordinated pipeline.

Despite the benefits, configuring 3D parallelism introduces significant complexity, requiring careful coordination across these strategies to balance workloads and minimize communication overhead. Recent research has focused on automating these configurations to reduce the burden on developers and ensure efficient distributed execution. For example, GSPMD (Xu et al., 2021) extends the XLA compiler (Sabne, 2020) to support various parallelism paradigms through user annotations. Alpa (Zheng et al., 2022) automates model parallelization by optimizing intra- and inter-operator parallelism for efficient distributed execution. Galvatron (Miao et al., 2022) introduces a decision tree to decompose the search space and designs a dynamic programming algorithm to generate the optimal plan.

Emerging techniques like sequence parallelism (Li et al., 2021; Jacobs et al., 2023; Liu et al., 2023) further address the challenges of training on long sequences by distributing computations along the sequence dimension, reducing memory overhead and communication bottlenecks.

2.3 Performance Modeling, Simulation, and Optimization

The complexity of LLMs poses challenges and opportunities in system design and optimization, with performance modeling serving as a critical foundation for diagnosing and optimizing overall efficiency.

There are two primary approaches to building performance models. The first relies on analytical models. AmPeD (Moolchandani et al., 2023) introduces an analytical model to estimate performance in distributed transformer training under various model parameters and parallelism strategies. Similarly, Calculon (Isaev et al., 2023) provides a parameterized analytical model that explores the co-design space of software and hardware configurations to identify optimal system designs for LLMs. However, these analytical models are often tailored to specific implementations and hardware configurations, limiting their ability to generalize in the face of rapid model and system evolution. Moreover, they typically provide high-level performance estimates, making them inadequate for optimizations like mixed precision training (Das et al., 2018; Zhu et al., 2020) and operator fusion (Zhao et al., 2022; Jia et al., 2019).

The second approach leverages trace-based models to simulate execution and derive optimization insights. For example, ASTRA-sim (Rashidi et al., 2020) and ASTRA-sim2.0 (Won et al., 2023) simulate distributed training with a cycle-level and analytical network backend, evaluating collective communication algorithms and network topologies. In (Lin et al., 2022), the authors analyze critical paths within profiled traces to predict per-batch training time for DLRM. Daydream (Zhu et al., 2020) uses kernel-level dependency graphs collected with CUPTI to predict runtime under specific optimizations, while dPRO (Hu et al., 2022) builds a global dataflow graph by tracking dependencies among operators to estimate DNN training performance. However, these trace-based approaches fail to fully capture the complexities inherent in LLM execution. To the best of our knowledge, this work is the first to leverage traces for accurately modeling the intricate behaviors of LLMs, accounting for detailed operator and kernel interactions essential for precise performance prediction.

3 DESIGN

3.1 Overview

Figure 2 presents the workflow of Lumos, our trace-driven performance modeling and estimation toolkit for distributed LLM training. The process begins with collecting runtime profiling traces from popular frameworks such as TensorFlow and PyTorch. These raw traces are then analyzed to extract key meta-information, which is used to construct a detailed task-level execution graph. The execution graph can be modified to adjust model configurations, such as parallelism strategies and architectures, generating new configuration-specific graphs. Finally, the simulator uses these graphs to either replay the original execution or predict performance under alternative configurations, providing insights into potential optimizations and enabling effective exploration of what-if scenarios.

We initially focus on PyTorch due to its widespread use in both academia and industry, along with its advanced profiling capabilities. However, our approach is flexible by design and can be extended to support other ML frameworks. We will discuss the adaptability of Lumos in Section 5.

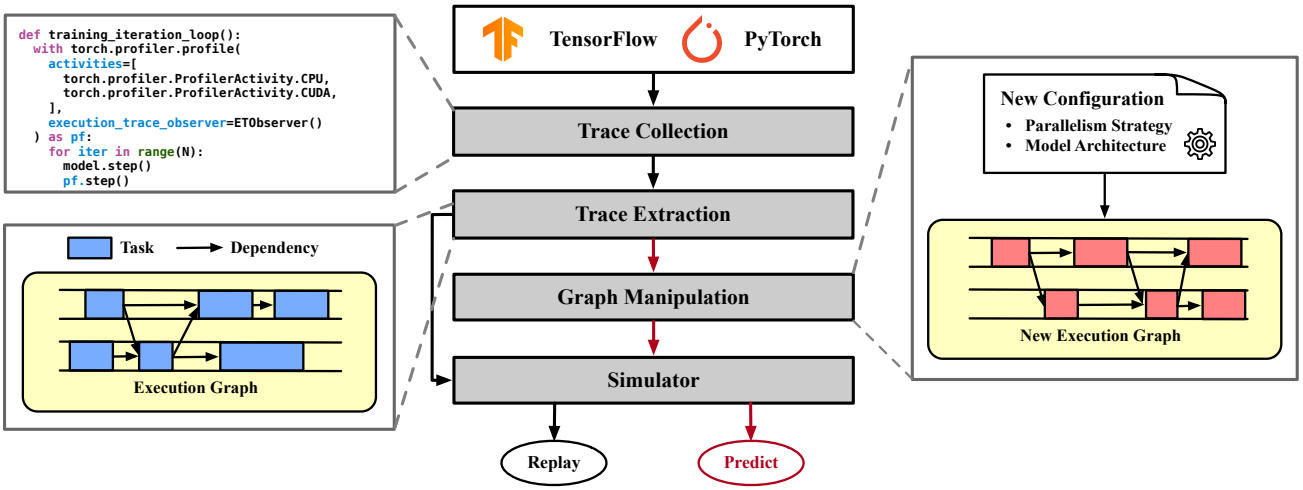


Figure 2. Overview of Lumos's workflow.

3.2 Traces Collection

We collect profiling traces using PyTorch Kineto (Kineto), which captures comprehensive runtime information about PyTorch operators, CUDA runtime events, and GPU kernels, including name, start time, duration, CUDA stream ID, thread ID, correlation ID, and more. Unlike previous methods such as Daydream and dPRO, which necessitate extensive framework and model instrumentation, our profiling involves adding only a few lines of code into the model, as shown in the code snippet at the top left of Figure 2, significantly improving usability with minimal effort.

3.3 Execution Graph

The essence of a model's execution lies in its execution graph, which maps out the tasks being performed and the dependencies between them. Motivated by prior approaches (Zhu et al., 2020; Hu et al., 2022; Bang et al., 2023), we construct a low-level execution graph to accurately represent model execution. However, we have incorporated several enhancements to capture the complex execution characteristics of LLMs, ensuring more accurate modeling and offering the flexibility to estimate performance for new model configurations.

3.3.1 Tasks

To streamline the design, our execution graph includes only the following two types of tasks:

CPU tasks: These include all executions happened on the CPU, including PyTorch operators and CUDA runtime events. For each CPU task, we record its metadata along with the specific CPU thread on which it runs.

GPU tasks: These include all executions happened on the GPU, which primarily consist of GPU kernels. For each GPU task, we log its metadata along with the corresponding CUDA stream responsible for its execution.

3.3.2 Dependency

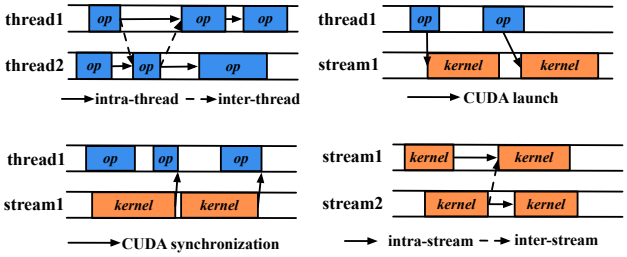


Figure 3. Four types of dependencies between the tasks.

Next, we identify four types of dependencies that capture all possible relationships between tasks:

CPU to CPU: This dependency includes both intra-thread and inter-thread relationships between CPU tasks. Tasks within the same thread naturally execute sequentially, forming intra-thread dependencies between consecutive tasks. Inter-thread dependencies occur when tasks on one thread block tasks on another. For example, in PyTorch, the backward pass runs on a separate thread, requiring the first backward operator to wait until the last forward operator completes. We detect these dependencies by identifying significant execution gaps within threads and establishing cross-thread dependencies accordingly.

CPU to GPU: GPU tasks are typically launched by corresponding CPU-side CUDA events, such as `cudaLaunchKernel` and `cudaMemsetAsync`. In Kineto traces, both CUDA runtime events and GPU ker-

nels are tagged with a correlation ID, which we use to link CPU tasks with their corresponding GPU tasks.

GPU to CPU: CUDA synchronization events, such as `cudaDeviceSync` or `cudaStreamSync`, are common during model execution. When invoked on the CPU, these events block execution until the relevant GPU kernels complete. As a result, they create dependencies from one or more GPU tasks to the initiating CPU task.

GPU to GPU: Similar to CPU-to-CPU dependencies, this includes both intra-stream and inter-stream dependencies between GPU tasks. GPU kernels within the same CUDA stream execute sequentially, meaning consecutive tasks in the same stream have direct dependencies. To identify inter-stream dependencies, we leverage a specialized event-based synchronization mechanism captured in the Kineto trace. Specifically, we focus on a pair of CUDA runtime events: `cudaEventRecord` and `cudaStreamWaitEvent`. The `cudaEventRecord` marks a synchronization point in one stream, recording an event after all preceding kernels on that stream have completed. The corresponding `cudaStreamWaitEvent` ensures that a different stream waits until the recorded event is reached, creating a dependency between the two streams. This mechanism allows us to accurately capture inter-stream dependencies, providing a precise representation of the execution order across streams.

Training LLMs at scale typically spans a large number of machines, resulting in significant communication overhead. To mitigate this, overlapping the execution of computation and communication kernels is a common practice to reduce end-to-end iteration time. However, this overlap introduces complex inter-stream dependencies, which are overlooked by existing modeling approaches. Lumos is the first to target LLMs and capture their intricate dependencies, a critical step toward accurate performance modeling and reliable downstream optimization studies.

3.4 Graph Manipulation

To improve and optimize LLM training performance, researchers and engineers can have many configurable options and optimization strategies. Commonly, they will ask what-if questions, such as:

How will the performance scale with additional GPUs? Which parallelism configuration will deliver the best results? How will changes to the model architecture impact performance? Will a specific optimization improve performance, and by how much?

While current distributed training frameworks make it easier to change configurations, deploying models with new settings on real hardware requires substantial resources and incurs high costs, leading to long iteration cycles. The process becomes even more challenging if the desired changes, such as introducing a new operator fusion pattern, are not supported by the framework, forcing developers to hack the underlying code, which can be both time-consuming and prone to errors.

To address these challenges, the fine granularity and flexibility of execution graphs, combined with simulation, provide an effective solution. By modifying the existing graph to reflect different model configurations, we can estimate performance and explore what-if scenarios without requiring large-scale physical deployments, accelerating iteration speed and significantly reducing costs.

Lumos offers an interface that allows users to specify new model configurations, after which it manipulates the existing execution graph to generate a new one reflecting the changes for performance estimation. It currently supports modifications to both model architectures—such as adjusting the number of transformer layers and hidden size—and parallelism strategies, including data parallelism and pipeline parallelism.

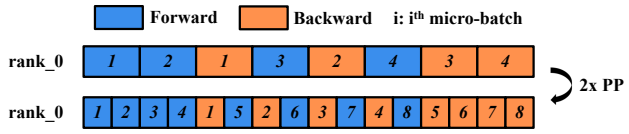


Figure 4. Updated pipeline schedule for rank_0 with 2x PP, assuming the number of micro-batches is equal to TP × PP and 1F1B scheduling policy (Narayanan et al., 2021).

For changes in data parallelism, only the communication needs adjustment by assigning new execution time to the communication tasks, as the local computation for each worker remain unchanged. For pipeline parallelism adjustments, we first update the pipeline schedule to align with the new configuration based on the scheduling policy, determining the execution order of the forward and backward passes, as illustrated in Figure 4. Next, we group the tasks by layers and partition the original layers and their underlying tasks into new stages. For example, assuming layers are evenly distributed, we calculate how many layers belong to each stage. The corresponding tasks are reassigned to their new stages, and communication tasks are inserted at appropriate points, to ensure correct synchronization and execution. We currently do not support modifications to tensor parallelism, as it is typically fixed in practice (e.g., within a single node) due to its high communication overhead. We leave the support for it as our future work.

For changes to model architecture, such as adjusting the hidden size, we modify the input tensor dimensions for the relevant operators and kernels and update their execution times during simulation. When changing the number of layers, we follow a process similar to that used for pipeline

parallelism, dividing tasks into layers and applying the adjustments accordingly.

Throughout this manipulation process, we ensure that the dependency patterns from the original trace are preserved in the new graph to maintain correct execution, as we will demonstrate in the evaluation.

3.5 Simulation

Algorithm 1 Lumos's Simulation Algorithm

Input: Execution graph: G = (V, E)
Output: Trace with runtime details of all tasks
$\mathcal{R} \leftarrow \emptyset$ {Initialize the ready task set}
$P \leftarrow \{0\}$ {Initialize task processors}
for each task $t \in G.tasks$ **do**
  $t.dep \leftarrow |\{t's \text{ fixed dependencies}\}|$
  if $t.dep = 0$ **then**
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{t\}$
  end if
end for
while $\mathcal{R} \neq \emptyset$ **do**
  $t \leftarrow pick(\mathcal{R})$ {Select a ready task to execute}
  $p \leftarrow t.Processor$
  $\mathcal{R} \leftarrow \mathcal{R} \setminus \{t\}$
  for each $r \in get_runtime_dependencies(t)$ **do**
    $r.dependents \leftarrow r.dependents \cup \{t\}$
    $t.dep \leftarrow t.dep + 1$
  end for
  if $t.dep = 0$ **then**
    $t.start \leftarrow \max(P[p], t.start)$
    $P[p] \leftarrow t.start + t.duration$
    for each $c \in t.dependents$ **do**
      $c.dep \leftarrow c.dep - 1$
      $c.start \leftarrow \max(c.start, t.start + t.duration)$
      if $c.dep = 0$ **then**
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{c\}$
      end if
    end for
  end if
end while

Both the original and modified execution graphs will be fed into the simulator to simulate execution and estimate performance. During the simulation, the four types of dependencies outlined in Section 3.3.2 are maintained through two mechanisms. Fixed dependencies are determined at initialization and remain unchanged throughout execution, such as the sequential order of CPU tasks on the same thread. Runtime dependencies, on the other hand, are assigned dynamically during runtime. For example, a `cudaStreamSync` task must wait for the last kernel on a specific stream to complete, but which kernel will be last cannot be known prior to execution.

Algorithm 1 outlines the simulation process, beginning with the assignment of fixed dependencies. In each iteration, a ready task is selected and allocated to its respective processor. The algorithm then checks for any runtime dependencies. If all dependencies are met, the task is executed, updating the processor's progress and the status of its dependent tasks. Otherwise, the task is deferred until all dependencies are resolved. The simulation generates a trace similar to the input trace initially profiled from the real run, recording all runtime information of the tasks. This output trace can be used not only to estimate the overall execution time but also to analyze fine-grained execution characteristics, as we show in Section 4.2.

4 Evaluation

We implement Lumos in Python with approximately 5,200 LoC. To leverage it, users need access to the source code to insert profiler hooks into their PyTorch models for collecting traces, typically requiring around 10 lines of code. Lumos then offers a fully automated workflow: it begins by constructing the execution graph from the raw traces, manipulates the graph based on new configurations, and concludes with performance estimation through simulation. Depending on the complexity of the original traces, the entire process can range from a few seconds to several minutes.

4.1 Methodology

Models. We evaluate Lumos using NVIDIA's open-source GPT-3 implementation (NVIDIA, b) from the MLPerf Training Benchmarks. Our experiments involve training four model variants, adjusting the number of layers, hidden size, feedforward network size, and attention heads, with model parameters ranging from 15 billion to the full 175 billion, as summarized in Table 1. We collect traces with PyTorch Kineto and evaluate performance across various parallelism strategies, exploring different combinations of tensor, pipeline, and data parallelism.

Infrastructure. Our evaluation is conducted on a production ML cluster, using up to 512 NVIDIA H100 GPUs (on 32 servers) interconnected with 8x 400Gbps per host in a RoCE DC-scale network. Our testing environment is based on CUDA 12.4, PyTorch 2.5, Transformer Engine 0.12.0, and PyTorch Lightning 1.9.4.

We select dPRO (Hu et al., 2022) as the state-of-the-art baseline for comparison. In our evaluation, we first validate the replay accuracy by comparing both iteration time and execution breakdown against the ground truth and the baseline. Next, we evaluate the accuracy of our approach in estimating performance for new configurations, including changes in both parallelism strategies and model architectures.

Table 1. Model sizes and architectures used in the evaluation. All other parameters follow the default values from the open-source GPT-3 implementation (NVIDIA, b).

MODEL NAME	n_{params}	n_{layers}	d_{model}	d_{ffn}	n_{heads}	d_{head}
GPT-3 15B	15B	48	6144	12288	48	128
GPT-3 44B	44B	48	12288	24576	48	128
GPT-3 117B	117B	96	12288	24576	96	128
GPT-3 175B	175B	96	12288	49152	96	128

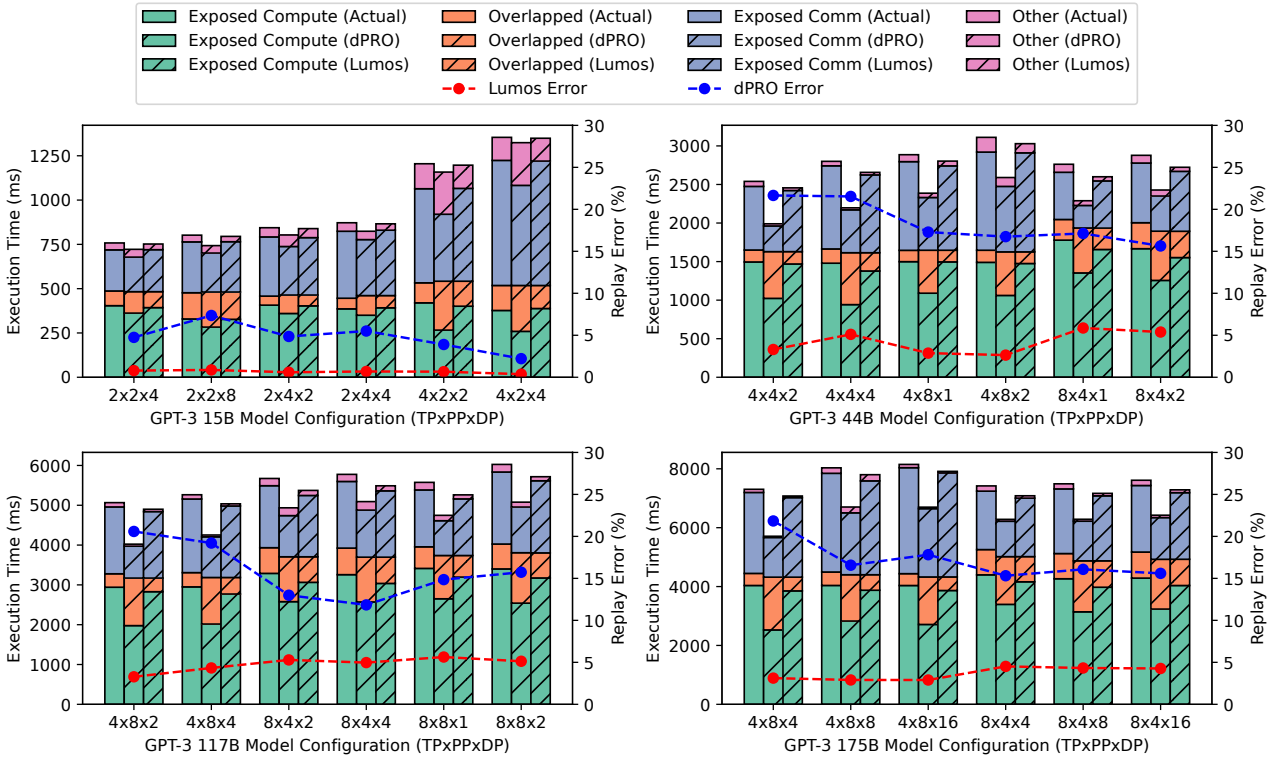


Figure 5. Per-iteration training time with its breakdown across various model sizes and parallelism strategies: comparison of actual execution, dPRO, and Lumos.

4.2 Replay

4.2.1 Overall Iteration Time

In Figure 5, we firstly compare the per-iteration execution time replayed by Lumos and dPRO against the real execution time measured from actual training across various GPT-3 model sizes (15B, 44B, 117B, and 175B) and parallelism strategies. Across all configurations, Lumos maintains a replay error mostly under 5%, with an average error of 3.3%. In contrast, dPRO's error reaches up to 21.8%, with an average of 14%. While smaller models and simpler setups allow dPRO to predict the overall time relatively well, its accuracy deteriorates as model size and complexity grow. The discrepancy highlights Lumos's robustness in accurately capturing execution behaviors and modeling performance, even for larger models and complex deployment setups.

4.2.2 Execution Breakdown

Figure 5 also provides a detailed breakdown of the execution into key components: exposed compute (computation that does not overlap with communication), exposed communication (communication that does not overlap with computation), overlapped execution (where computation and communication run concurrently), and other (primarily idle periods). This breakdown offers deeper insights into the differences in iteration times across configurations.

The analysis reveals that dPRO consistently overestimates overlapped execution and underestimates total iteration time, primarily due to its inability to accurately model inter-

stream dependencies, leading to overly optimistic predictions of parallel execution. In contrast, Lumos effectively captures the complex dependencies within the model and faithfully replays the execution. It accurately reflects the dynamic interactions between computation and communication, adapting to changes in model size and deployment configuration, and closely aligning with the real measurement.

In large-scale distributed training, particularly for LLMs, a substantial portion of execution time is spent on communication and synchronization between GPUs. To optimize performance, engineers aim to maximize the overlap between computation and communication kernels. Therefore, an accurate performance model that not only replays overall execution time but also captures fine-grained details, such as the degree of overlap, is essential. Such a model can provide valuable insights for identifying performance bottlenecks and guiding optimization efforts.

4.2.3 *SM Utilization*

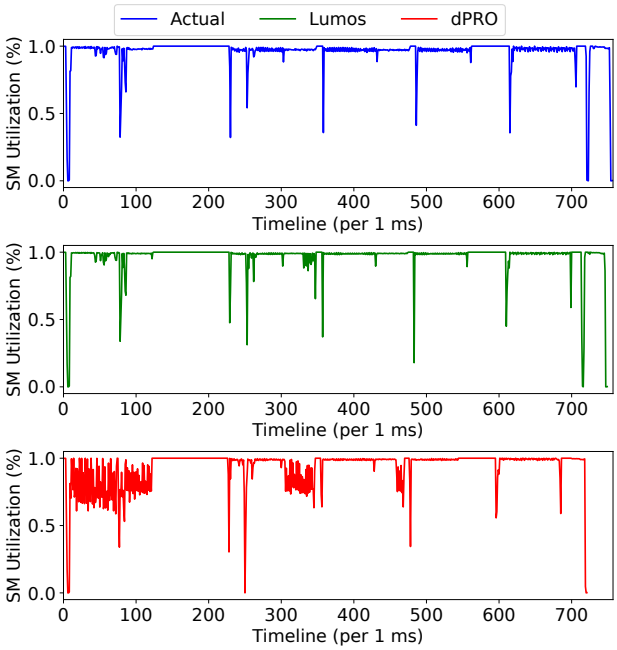


Figure 6. SM utilization of one iteration when training GPT-3 15B under TP = 2, PP = 2 and DP = 4.

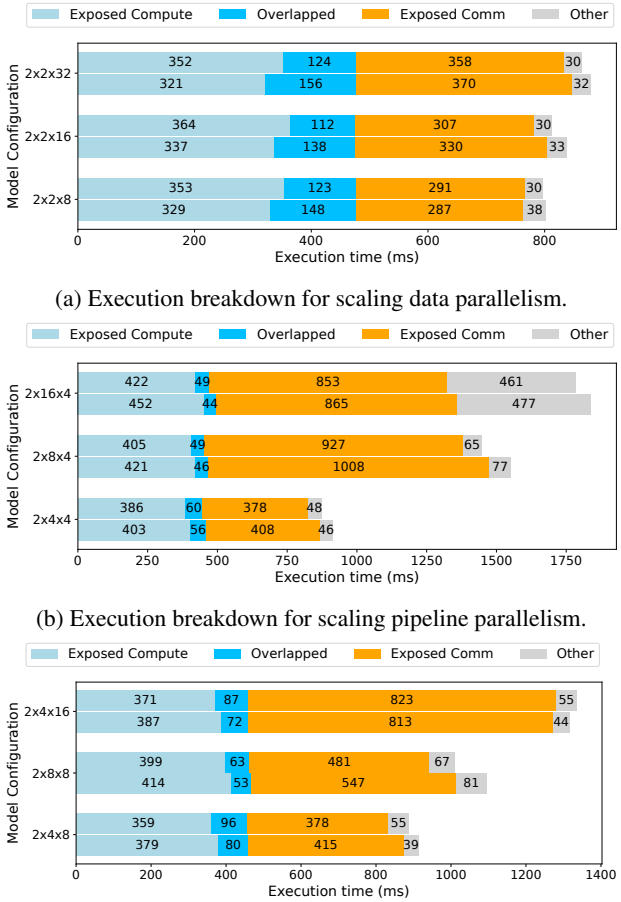
Analyzing SM (Streaming Multiprocessor) utilization is essential for identifying performance bottlenecks, such as idle periods or imbalanced workloads, to enhance GPU efficiency.

In this section, we examine the SM utilization over one iteration of training the GPT-3 15B model, configured with tensor parallelism = 2, pipeline parallelism = 2, and data parallelism = 4. Utilization is defined as the fraction of time, over 1ms intervals, during which at least one CUDA stream is actively executing tasks. This data is derived from profiled and simulated traces by analyzing kernel activities throughout the execution.

As shown in Figure 6, the SM utilization replayed by Lumos closely match the actual measured utilization. In contrast, dPRO exhibits more fluctuations and significant discrepancies. This comparison, again, highlights Lumos's ability to capture fine-grained execution details, validating its effectiveness in accurately modeling execution behavior.

4.3 Graph Manipulation

4.3.1 *Parallelism Strategy*



(a) Execution breakdown for scaling data parallelism.

(b) Execution breakdown for scaling pipeline parallelism.

(c) Execution breakdown for simultaneous scaling of data and pipeline parallelism.

Figure 7. Runtime predictions for scale-out configurations. Each configuration (TPxPPxDP) is represented by two horizontal bars: the upper bar shows the predicted value by Lumos, and the lower bar shows the actual value.

Next, we evaluate Lumos's flexibility in generating new execution graphs from existing ones to estimate performance under new configurations. Specifically, we demonstrate its

ability to predict scale-out performance by adjusting parallelism strategies. Our experiments focus on GPT-3 15B, using traces collected from a baseline configuration with tensor parallelism = 2, pipeline parallelism = 2, and data parallelism = 4.

We begin by exploring changes to data parallelism, where only the execution time of communication tasks needs to be updated. We currently estimate new communication time using an in-house performance model built from fleet traces, as it is both readily available and accurate. While network simulators like ASTRA-sim (Won et al., 2023) or analytical models (Moolchandani et al., 2023; Rashidi et al., 2022) could also be used, predicting the runtime of individual kernels is beyond the scope of this work. We will explore potential integrations with these tools in the discussion section. To validate these predictions, we compare them against actual traces collected at larger scales. As shown in Figure 7a , Lumos accurately predicts both the total runtime and detailed performance breakdowns when scaling from 16 GPUs to 32, 64, and 128 GPUs.

Similarly, Figure 7b demonstrates that Lumos can also accurately estimate execution time and breakdown when scaling pipeline parallelism. We modify the baseline traces by splitting layers and underlying tasks into new stages, adding communication tasks, and reordering task execution according to the new pipeline schedule. Finally, Figure 7c shows that Lumos maintains high accuracy, with an average error of just 4.2% when scaling both data and pipeline parallelism simultaneously. These results prove that Lumos can effectively generate correct new executions through graph manipulation for new parallelism strategies.

4.3.2 Model Architecture

Table 2. Sizes and architectures for model variations.

MODEL NAME	n_{params}	n_{layers}	d_{model}	d_{ffn}
GPT-3 15B	15B	48	6144	12288
GPT-3 v1	20B	64	6144	12288
GPT-3 v2	30B	96	6144	12288
GPT-3 v3	28B	48	9216	18432
GPT-3 v4	44B	48	12288	24576

We now validate Lumos's accuracy in estimating performance for different model architectures. Our evaluation continues with GPT-3 15B as the base model. To generate several variants, we modify the number of layers, hidden sizes, and feedforward network sizes. Table 2 summarizes the sizes and architectures of the models used in this evaluation. All models are trained using the configuration of tensor parallelism = 2, pipeline parallelism = 2, and data parallelism = 4.

When increasing the number of layers, we duplicate the layers and corresponding tasks from the existing trace, insert them into the graph at appropriate places, and reconstruct dependencies with neighboring tasks according to the original dependency pattern. For changes in hidden size or feedforward network size, we adjust the input tensor dimensions for all relevant operators and kernels. Ideally, new execution times should be assigned to all affected tasks to reflect the input changes. However, we observe that only a few key kernels, such as GEMM and communication-related ones, exhibit significant runtime changes under different configurations. We similarly update the execution times for these kernels using the in-house performance model described in Section 4.3.1.

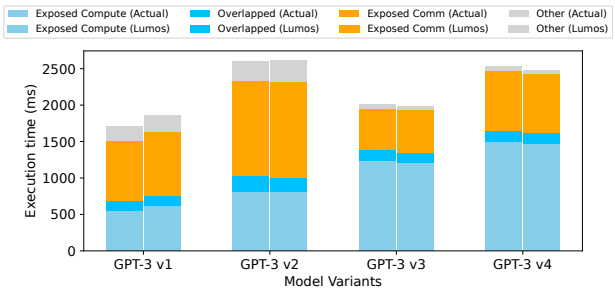


Figure 8. Iteration time breakdown of model variations. The left bars show the actual values, and the right hatched bars show the predicted values.

Figure 8 presents the iteration time breakdown across these model variations, showing both the actual and predicted performance. The predicted values, represented with hatched patterns, align closely with the actual measurements, demonstrating that Lumos accurately reproduces the execution and estimates the performance under different model architecture changes.

Overall, the results shown in Section 4.3.1 and Section 4.3.2 demonstrate Lumos's ability to leverage existing traces to generate new execution graphs for both parallelism configurations and model architecture variations, to provide accurate performance estimates through simulation. This predictive capability significantly reduces the need for costly hardware resources, positioning Lumos as a practical tool for efficient model configurations exploration.

5 DISCUSSION

In this section, we discuss the profiling overhead, adaptability, scope, and limitations of Lumos.

Profiling Overhead and Cost. Lumos leverages PyTorch Kineto (Kineto) to collect traces and construct the execution graph. Profiling requires only a few lines of hook code, which makes it much more user-friendly than ex-

isting approaches that require additional instrumentation. Although profiling can impact execution, capturing a single iteration—or just a few—is sufficient, as the model's execution pattern remains consistent across iterations. This ensures that the overall profiling overhead is negligible in the context of the entire training process.

Adaptability of Lumos. Lumos requires profiling traces that capture both CPU and GPU activities, including framework operators, CUDA runtime events, and GPU kernels. Similar profiling capabilities are available in other ML frameworks, such as TensorFlow Profiler (TensorFlow, b). Lumos's post-processing stages for constructing and manipulating execution graphs are framework-agnostic, making it easy to adapt to other frameworks.

Our methodology also extends well to other LLMs and ML models in other domains, as it does not rely on model-specific information to construct execution graphs for performance modeling. While we make certain assumptions during graph manipulation, such as where to insert layers and which tasks would be affected, our method remains broadly applicable, given the shared transformer-based architecture of most modern LLMs.

Similarly, although this paper focuses on LLM training, where communication overhead is higher and model behavior more complex, Lumos is also applicable to the inference. We anticipate even broader use cases as LLM inference grows more complicated, such as distributed inference (Wei et al., 2022; Li et al., 2023) and SSD-based inference (Wilkening et al., 2021; Sun et al., 2022).

Kernel Execution Time Prediction. Changing configurations can introduce new GPU kernels not present in the original trace, such as new communication kernels when adjusting parallelism strategies or new computation kernels when modifying model architectures. Accurately predicting performance for new configurations requires estimating the runtime of these new or altered kernels. Currently, we estimate the runtime of unseen kernels using an in-house GPU kernel performance model, built by analyzing fleet GPU traces, as it is available and accurate. However, alternative methods are also available. For communication kernels, we can use metadata like message size, collective algorithm, and networking environment to estimate performance with network simulators like ASTRA-sim (Won et al., 2023) and HeterSim (Tang et al., 2024), or analytical models (Moolchandani et al., 2023; Rashidi et al., 2022). For computation kernels, we can simply measure runtime through individual microbenchmarks. However, predicting kernel runtimes is beyond the scope of this work.

The primary goal of Lumos is to deliver a fine-grained execution graph and an accurate performance model to capture the complexities of LLM execution and provide reliable performance estimates. Developers can implement optimized individual kernels, profile their runtime, and integrate the results into Lumos to predict the overall runtime, saving the engineering efforts of porting them into the frameworks. More importantly, it can offer invaluable insights for optimization even before implementation by answering what-if questions, such as how much the overall runtime would be reduced if a kernel ran twice as fast, and identifying which optimization would yield the greatest performance improvement. By modifying existing traces and estimating performance through simulation, Lumos makes performance evaluation and optimization more efficient and cost-effective.

Limitations. Lumos currently focuses on modeling and simulating the timing of model execution. In predicting performance for modified configurations, we assume the model will function as expected under the new settings, without unforeseen issues such as out-of-memory errors. Estimating system-level metrics, such as FLOPS utilization, memory consumption, bandwidth usage, or energy efficiency, lies beyond Lumos's current scope. These metrics, essential for optimizing LLM efficiency, are part of our future plans to provide more comprehensive performance insights.

6 Conclusion

In this paper, we introduced Lumos, a trace-driven performance modeling and estimation toolkit for large-scale LLM training. Lumos captures complex behaviors through detailed execution graphs built from profiling traces, enabling accurate performance modeling and estimation. Our evaluation on a production ML cluster with up to 512 NVIDIA H100 GPUs shows an average replay error of only 3.3% across diverse model architectures and parallelism configurations. By manipulating existing graphs to generate new ones for different configurations and predicting performance through simulation, Lumos supports efficient optimization exploration.

Acknowledgements

We sincerely thank the anonymous reviewers for their suggestions on earlier versions of this manuscript. This work was in part sponsored by Meta through a student researcher appointment. This work was also in part supported by NSF CAREER Award CCF-2326182, an Intel Research Award, a Sloan Research Fellowship, a Microsoft Research Fellowship, and a Facebook Research Faculty Award.

References

Bang, J., Choi, Y., Kim, M., Kim, Y., and Rhu, M. vtrain: A simulation framework for evaluating cost-effective and compute-optimal large language model training. *arXiv*

preprint arXiv:2312.12391, 2023.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Brown, T. B. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Das, D., Mellempudi, N., Mudigere, D., Kalamkar, D., Avancha, S., Banerjee, K., Sridharan, S., Vaidyanathan, K., Kaul, B., Georganas, E., et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

Hashemi, S. H., Abdu Jyothi, S., and Campbell, R. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1:418–430, 2019.

Hu, H., Jiang, C., Zhong, Y., Peng, Y., Wu, C., Zhu, Y., Lin, H., and Guo, C. dpro: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training. *Proceedings of Machine Learning and Systems*, 4:623–637, 2022.

Isaev, M., McDonald, N., Dennison, L., and Vuduc, R. Calculon: a methodology and tool for high-level co-design of systems and large language models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2023.

Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–14, 2023.

Kineto. Pytorch kineto. https://github.com/pytorch/kineto.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Li, J., Jiang, Y., Zhu, Y., Wang, C., and Xu, H. Accelerating distributed {MoE} training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 945–959, 2023.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.

Lin, Z., Feng, L., Ardestani, E. K., Lee, J., Lundell, J., Kim, C., Kejariwal, A., and Owens, J. D. Building a performance model for deep learning recommendation model training on gpus. *arXiv preprint arXiv:2201.07821*, 2022.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Ma, R., Georganas, E., Heinecke, A., Gribok, S., Boutros, A., and Nurvitadhi, E. Fpga-based ai smart nics for scalable distributed ai training systems. *IEEE Computer Architecture Letters*, 21(2):49–52, 2022.

Miao, X., Wang, Y., Jiang, Y., Shi, C., Nie, X., Zhang, H., and Cui, B. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878*, 2022.

Moolchandani, D., Kundu, J., Ruelens, F., Vrancx, P., Evenblij, T., and Perumkunnil, M. Amped: An analytical model for performance in distributed training of transformers. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 306–315. IEEE, 2023.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. 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*, pp. 1–15, 2021.

NVIDIA. Cupti. https://docs.nvidia.com/cuda/cupti/, a.

NVIDIA. Gpt-3 implementation. https://github.com/mlcommons/training/tree/master/large_language_model/megatron-lm, b.

NVIDIA. Nvidia blackwell architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/, c.

NVIDIA. Nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html, d.

NVIDIA. Nsight. https://developer.nvidia.com/nsight-systems, e.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

PyTorch. Pytorch. https://pytorch.org/.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Rashidi, S., Sridharan, S., Srinivasan, S., and Krishna, T. 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)*, pp. 81–92. IEEE, 2020.

Rashidi, S., Won, W., Srinivasan, S., Sridharan, S., and Krishna, T. Themis: A network bandwidth-aware collective scheduling policy for distributed training of dl models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 581–596, 2022.

Sabne, A. Xla: Compiling machine learning for peak performance. *Google Res*, 2020.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

Sun, X., Wan, H., Li, Q., Yang, C.-L., Kuo, T.-W., and Xue, C. J. Rm-ssd: In-storage computing for large-scale recommendation inference. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1056–1070. IEEE, 2022.

Tang, Y., Yuan, T., Cao, F., Wang, L., Guo, Z., Zhao, Y., and Li, R. Simulating llm training in cxl-based heterogeneous computing cluster. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1–6. IEEE, 2024.

TensorFlow. Tensorflow. https://www.tensorflow.org/, a.

TensorFlow. Tensorflow profiler. https://www.tensorflow.org/guide/profiler, b.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Wei, Y., Langer, M., Yu, F., Lee, M., Liu, J., Shi, J., and Wang, Z. A gpu-specialized inference parameter server for large-scale deep recommendation models. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pp. 408–419, 2022.

Wilkening, M., Gupta, U., Hsia, S., Trippel, C., Wu, C.-J., Brooks, D., and Wei, G.-Y. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 717–729, 2021.

Won, W., Heo, T., Rashidi, S., Sridharan, S., Srinivasan, S., and Krishna, T. 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)*, pp. 283–294. IEEE, 2023.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

Zhao, J., Gao, X., Xia, R., Zhang, Z., Chen, D., Chen, L., Zhang, R., Geng, Z., Cheng, B., and Jin, X. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems*, 4:1–19, 2022.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Zhu, H., Phanishayee, A., and Pekhimenko, G. Daydream: Accurately estimating the efficacy of optimizations for {DNN} training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 337–352, 2020.