

LMTracer: Fine-Grained and Real-Time Performance Profiling for Production LLM Systems

Wei Liu^{†‡}, Yongchao He[‡], Bohan Zhao[‡], Hongyi Wang[‡], Zhenhua Li^{‡*}, Junping Zhao^{†*}

[†]Rank Computing [‡]Tsinghua University

Abstract

Training and serving large language models (LLMs) has become a core business for AI providers. To ensure a high-quality user experience while optimizing infrastructure costs, providers need to closely monitor the performance of LLM executions in production. However, existing performance profiling tools fall short in this context, as they are either too coarse-grained to capture performance bottlenecks, or too intrusive to avoid significant performance degradation.

This paper presents LMTracer, a fine-grained and real-time performance profiling framework for production LLM services. We find that the inefficiency of existing techniques mainly stems from disruptions in highly asynchronous GPU execution flows. To address this, our key idea is to embed profiling logic into the execution through *graph-embedded probing*. This is achieved by compiling lightweight *probe kernels* that realize profiling logic as special trace nodes in GPU execution graphs, and streaming buffered profiling data to CPUs on demand to keep the execution of user kernels uninterrupted. LMTracer has been deployed in production for eight months. It incurs only 0.9% average overhead across various LLM workloads (e.g., training, serving, and fine-tuning), and has proactively uncovered 15 performance issues. After fixing these issues, we observed 21.2% latency reduction and 24.5% throughput improvement for LLM services.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **General and reference** → **Performance**; • **Computer systems organization** → Parallel architectures; • **Hardware** → Hardware accelerators.

Keywords: Large Language Models; GPU Kernels; Real-time GPU Performance Profiling; Production Observability

ACM Reference Format:

Wei Liu, Yongchao He, Bohan Zhao, Hongyi Wang, Zhenhua Li, Junping Zhao. 2026. LMTracer: Fine-Grained and Real-Time Performance Profiling for Production LLM Systems. In *Symposium on Operating Systems Principles (SOSP '26)*, September 29–October 2,

*Corresponding authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SOSP '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2585-2/26/09

<https://doi.org/10.1145/3830418.3843871>

2026, Prague, Czech Republic. ACM, New York, NY, USA, 15 pages.
<https://doi.org/10.1145/3830418.3843871>

1 Introduction

Large language models (LLMs) have been emerging as a cornerstone of today's AI applications, powering a variety of services from chatbots to general problem solving and beyond [38, 51]. In the meantime, training and serving these models at scale becomes an important business and a primary source of revenue for mainstream AI providers [61, 68]. For example, OpenAI reported an 82% revenue increase from its serving business in 2025 [4], and AWS saw a 24% quarterly revenue increase in its training services [43].

However, training and serving LLMs is oftentimes resource-intensive and performance-sensitive. Training a single LLM can cost millions of dollars and require thousands of GPUs over weeks or months [50, 69], while serving these models to a large and concurrent user base demands substantial scheduling and optimizing capabilities on the infrastructure and computational resources [65]. Even a small performance degradation of a GPU (e.g., 20%) during distributed training can lead to significant training cost increases (e.g., 38%) [41], and a small latency increase (e.g., 1 s) in the serving phase can lead to a large number of user requests being dropped and thus cause significant revenue loss [3, 73].

In such a context, AI providers need to closely monitor the performance of their LLM training and serving systems in production, so as to ensure the optimal utilization of resources and a smooth user experience. To this end, LLM profiling tools [16, 21, 42] are a common practice that help identify inefficiencies and validate the optimizations. Unfortunately, our two-year experience in operating LLM systems at scale reveals that existing tools struggle to meet the needs of fine-grained and real-time profiling in practice. They are either too coarse-grained to capture performance insights, or too heavyweight to obtain accurate profiling data without significantly affecting the performance in production¹.

• **Telemetry-based monitoring.** Telemetry-based monitoring is achieved by aggregating high-level performance metrics such as token throughput, iteration time, and end-to-end latency of user requests. These metrics are often available in LLM frameworks like vLLM [46], SGLang [71],

¹Although offline profiling imposes less stringent overhead constraints, it may fail to accurately reflect real-world performance, as execution patterns can differ substantially between offline and production environments.

and Megatron-LM [60]. They can also be measured by tools like eBPF [30], which record data mostly over CPUs. While these methods are lightweight and easy to use, they typically provide limited performance insights, especially regarding the GPU execution time of LLM components.

- **Kernel-level profiling.** To overcome the limitations of telemetry-based monitoring, kernel-level profiling tools such as NVIDIA CUPTI [15], Nsight Systems [16], AMD ROCm Systems Profiler [19], and PyTorch Profiler [21] can be used to capture fine-grained performance traces at GPU kernels. However, these tools are too intrusive in practice, and sometimes hardware-dependent, incurring significant overheads and leading to inaccurate performance measurements [42]. Consequently, they are restricted to offline profiling or debugging, and are not suitable for monitoring real-time performance in production LLM systems.

To understand the essential challenges behind real-time profiling in production, we conduct a comprehensive study on the existing tools, finding that the highly *asynchronous* yet *streamlined* nature of GPU executions forms the root cause. Specifically, to minimize the idle time during LLM executions, almost all GPU architectures employ an asynchronous execution model [2], where CPUs issue tasks (e.g., computing and communication kernels) to GPUs in a non-blocking manner, and the GPUs execute these enqueued kernels concurrently without being blocked by the operations from CPUs. These kernels are oftentimes fused together in a streamlined manner as a single execution graph (e.g., through PyTorch Inductor [1] and CUDA/HIP Graph [8, 11]) to reduce the GPU idle time on launching different kernels.

When profiling tools collect profiling data, however, they often interrupt the streamlined GPU executions with intrusive barriers to make the timeline data accurate. For example, they may insert CUDA/HIP Events [7, 12] to record the wall time of GPU kernel executions. This process not only disrupts the streamlined execution (since only kernels rather than runtime events can be fused together [1]), but also introduces significant synchronization overheads.

This paper presents LMTracer, a fine-grained and real-time performance profiling framework for production LLM systems. Our key idea is to *embed* profiling logic into GPU execution through *graph-embedded probing*, so as to eliminate disruptions on asynchronous GPU executions. To this end, LMTracer employs special *probe kernels*, which are lightweight GPU kernels that enclose original LLM kernels, allowing users to specify profiling logic (e.g., record timestamps). They are compiled to GPU execution graphs and executed in order with LLM kernels, while asynchronously collecting and streaming profiling data without introducing barriers.

To realize the above idea, we need to address several challenges. Specifically, if we insert probe kernels at arbitrary locations in the execution graphs captured by GPUs, we may break kernel fusion and cause performance degradation. To

overcome this, LMTracer employs *stack-aware compilation*, which recursively tracks compile-time operator call hierarchies and their source-level code regions during kernel fusion. Using this information, LMTracer determines the optimal locations in execution graphs to insert probe kernels, and thus keeps a streamlined execution graph without interfering with the fused operators or affecting the performance.

At runtime, profiling data should be collected in real time without introducing barriers. Existing profilers fail to achieve this as they depend on barriers to ensure data alignments. To address this, we devise *doorbell-assisted* shadow buffering. It does not directly write the profiling data to the CPU memory (which is slower and introduces barriers), but instead writes the data together with the GPU clock value to two faster *shadow* GPU buffers, and proactively streams the buffer data to the CPU based on a “doorbell” signaling protocol when a shadow buffer is full. LMTracer then calibrates the collected data with GPU clock offsets (recorded once before the execution) on CPUs. In the meantime, probe kernels continue to write the data into the other shadow GPU buffer and thus avoid CPU-GPU synchronizations.

After collecting profiling data, LMTracer provides actionable insights for developers to optimize LLM systems through its profiling interfaces. In particular, LMTracer automatically analyzes the LLM execution graph by replaying the collected data, and quickly identifies potential performance bottlenecks or inefficiencies with the execution dependency information among different kernels and GPUs.

LMTracer has been deployed on a variety of LLM systems in our production environment for eight months, including vLLM, SGLang, and Megatron-LM, for both training and serving, with model sizes ranging from 7B to 671B parameters and GPUs of O(1k). On average, it only incurs 0.9% overhead while providing kernel-level profiling granularity, such as measuring the execution time of attention, multi-layer perceptron (MLP), and layer normalization, etc. Compared to the existing profilers like NVIDIA Nsight Systems [16] and PyTorch Profiler [21], LMTracer reduces profiling overheads by 11.6× with comparable accuracy.

LMTracer has proactively uncovered 15 latent performance issues, such as imbalanced GPU utilization and straggler GPUs. After fixing these issues, we improve the training throughput by 24.5% and reduce the serving latency by 21.2%.

This work makes the following contributions.

- We identify the challenges of performance profiling in production LLM systems, i.e., the disruptions of the asynchronous yet streamlined GPU executions.
- We propose LMTracer, a lightweight and real-time profiler that embeds probe kernels into GPU execution graphs to eliminate disruptions on asynchronous executions.
- We show our experience in applying LMTracer to production LLM systems. It identifies many performance issues, leading to significant resource utilization improvements.

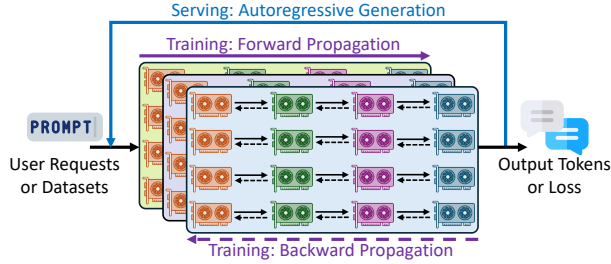


Figure 1. General workflow of LLM training and serving.

Code and data are publicly available at <https://llm-profiling.github.io>. This paper does not raise any ethical issues.

2 Background

Large language models (LLMs) are widely used in many real-world scenarios, such as chatbots [56], content generation [45], industrial control [34], and fundamental research [27, 53, 72]. Training and serving LLMs is currently the core business of AI providers like OpenAI [17], Google [10], and Meta [13]. However, such a business is extremely resource-intensive, requiring massive GPU infrastructure and computational power [35]. For example, training Llama3 requires thousands of GPUs running for weeks to months [31], and serving LLMs on the market demands hundreds of GPUs to handle user requests in real time [68].

LLM training and serving workflows. From the system perspective, the general workflows of LLM training and serving are illustrated in Figure 1. A single LLM training/serving task can be split across multiple GPUs by various parallelism techniques like data parallelism (DP), pipeline parallelism (PP), and tensor parallelism (TP) [50], so as to handle the increasing model size and massive data. DP replicates the entire model as shown in the different color planes in Figure 1, while PP partitions the model layers horizontally into different stages. In each PP stage, model layers are further partitioned vertically across the GPUs through TP.

With these parallelism techniques, when training starts, the forward and backward passes are performed iteratively until the training loss converges. Different GPUs are assigned different data batches, and different GPUs cooperate to complete the forward and backward passes. Any inefficiency in this workflow, such as suboptimal parallelism strategies [40], straggler GPUs [48, 67, 69], or network bottlenecks [49, 50], can lead to significant slowdowns and increased costs [58].

For LLM serving, the end-to-end serving process starts with user requests (i.e., user prompts) and ends with output tokens. Each user request is first processed by a scheduler that routes and enqueues it to the appropriate DP replica based on the service level agreement (SLA) and the current load of the LLM system. The DP replica then performs the prefill phase, which generates the initial KV cache [51, 59] for the prompt, and enters the decode phase to generate output

tokens in an autoregressive manner (i.e., generating one token at a time based on the previously generated tokens).

Kernel fusion. Almost all of today’s GPUs employ an asynchronous execution model to maximize hardware utilization, where the CPU enqueues kernels for computation or data transfer to the GPU, and the GPU launches and executes them in order [2] without mutual blocking on the CPU side. To fully take advantage of this asynchronous execution model, existing LLM frameworks typically use various kernel fusion techniques to combine multiple operators into a single large kernel, so as to reduce the kernel launch overhead and improve both CPU and GPU utilizations [1].

Figure 2 exemplifies the kernel fusion process of the layer normalization of a mainstream model, where the original implementation consists of four separate steps to compute the final output (i.e., the operators including power, addition, reciprocal square root, and multiplication), while the fused implementation combines these steps into a single kernel. In this way, LLM frameworks only launch one fused kernel instead of launching multiple kernels for each of the operators, thus significantly reducing the kernel launch overhead and improving the LLM performance.

Graph execution. Different fused kernels can be further organized together to form an execution graph on GPUs, so that these fused kernels are launched together without introducing idle time between two adjacent kernels. Such a graph execution scheme especially benefits the LLM training and serving workflows, since their computation often consists of a sequence of kernels that are executed iteratively. According to previous reports, graph execution schemes, such as CUDA graphs on NVIDIA GPUs, can improve LLM performance by up to $3\times$ [55]. Thus, using graph execution and kernel fusion is a common practice for LLM frameworks to optimize the performance of LLM training and serving.

3 Motivation

To ensure efficient and reliable execution of LLM training and serving workloads, AI providers need to closely monitor the real-time performance of their systems in production, so as to guide resource allocation and anomaly detection. Unfortunately, our two years of experience in operating LLM training and serving workloads in production has revealed that existing profiling solutions are inadequate for the challenges posed by LLM workloads. Table 1 summarizes the concrete limitations of the mainstream profiling tools.

Specifically, simple profiling solutions based on the aggregation of LLM framework-level metrics or telemetry (e.g., CPU, GPU, and memory usage) are insufficient for capturing the complex interactions between different components of LLM workloads, and often fail to provide actionable insights for performance optimization. They cannot even distinguish which GPU kernels or user code causes the inefficiency. For example, a low token throughput can be caused by many

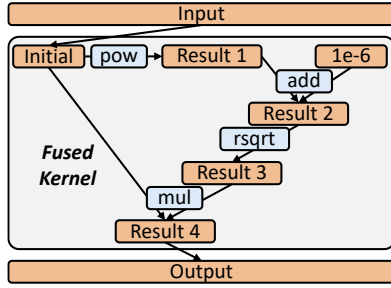


Figure 2. Example of kernel fusion on layer normalization.

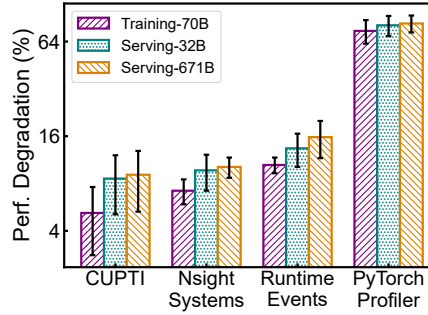


Figure 3. Performance degradation incurred by different profiling tools.

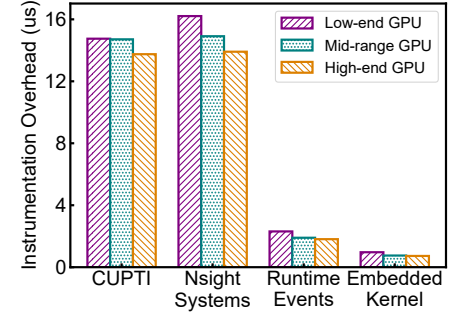


Figure 4. Instrumentation overhead incurred by different mechanisms.

Table 1. Comparing existing profiling tools for LLM systems, where Kernel Profiling stands for kernel-level profiling, Kernel Fusion represents the ability to distinguish fused kernels, Stack Trace is the ability to capture stack traces, Transparent is the ability to avoid user code modification, and Real-Time stands for the ability to capture performance data in real time.

Tool	Kernel Profiling	Kernel Fusion	Stack Trace	Transparent	Real-Time
Framework Metrics	✗	✗	✗	✓	✓
CUPTI	✓	✗	✗	✗	✗
Nsight Systems	✓	✗	✗	✓	✗
Runtime Events	✓	✗	✗	✗	✓
PyTorch Profiler	✓	✗	✓	✓	✗
LMTTracer	✓	✓	✓	✓	✓

factors like inefficient GPU kernel, suboptimal scheduling, or network bottlenecks, but the framework-level telemetry may not be able to pinpoint the root cause. As a result, they are often used for coarse-grained monitoring and alerting, rather than fine-grained performance tuning or optimization.

To overcome the above issue, there have been a number of fine-grained profilers on top of LLM frameworks, such as CUPTI [15], Nsight Systems [16], and PyTorch Profiler [21]. These tools expose fine-grained, kernel-level performance traces on GPUs. However, they are primarily designed for offline performance analysis. When applied to production workloads, we observed that they have significant limitations, especially in terms of their overhead and debugging capabilities. Note that although offline performance analysis can also be used for performance optimization, it is often not sufficient for production workloads, since the performance of LLM systems is highly sensitive to real-time input data.

Vendor-specific profilers. The official profiling tools provided by GPU vendors, such as CUPTI [15], Nsight Systems [16], and ROCm Systems Profiler [19], only capture kernel-level performance, and cannot capture user code. This makes it difficult to correlate and optimize the performance with the concrete implementation of LLM workloads. In addition, these tools are originally designed for offline performance analysis, and typically introduce significant overhead when applied to production. As shown in Figure 3, when training a 70B LLM, profilers like CUPTI and Nsight Systems

introduce 5.2% and 7.2% performance degradation in terms of the iteration time, respectively. When applied to LLM serving, the degradation rises to 9.1% and 10.2%.

Moreover, when we examine their trace data, we find that these tools cannot distinguish the fused kernels. This is because the fused kernels are generated before the graph execution on GPUs (cf. § 2), and thus the vendor-specific profiling tools have no visibility into their internal operations.

Runtime events. Commodity GPUs also provide a set of APIs for users to insert runtime events (e.g., CUDA/HIP Event [7, 12]), so as to record the corresponding event timestamp for performance tracing. However, these events require users to modify their code base, which is a barrier for adoption in production as well as for future maintenance. Runtime events also incur over 12% performance degradation when capturing crucial LLM execution phases like attention and MLP (as shown in Figure 3). Worse still, they are incompatible with kernel fusion and graph optimizations.

Framework-oriented profiler. Some LLM frameworks provide their own profilers (e.g., PyTorch Profiler [21]) to capture both kernel performance data and user stack traces. Unfortunately, they are not designed for real-time performance profiling either, and introduce significant overhead when applied to production workloads. As shown in Figure 3, when training a 70B LLM, PyTorch Profiler introduces 74.8% performance degradation on iteration time, and 83.4% decrease in token throughput when applied to LLM serving.

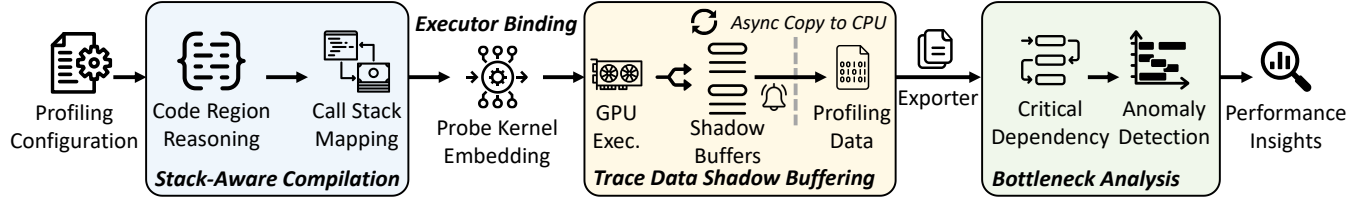


Figure 5. Architectural overview of LMTracer.

Limitations and implications. In summary, existing LLM profiling tools cannot meet the requirements of real-time performance profiling in our production environments. Vendor-specific and runtime event APIs cannot capture user-level stack traces, thus omitting the semantic mapping between the performance and the implementation. Profilers in LLM frameworks can capture a part of the user-level stack trace, but they cannot handle the kernel fusion and graph optimizations that are widely used in LLM workloads.

Runtime overheads of all the existing tools are significant, since they introduce additional barriers or instrumentation instructions. We performed a detailed analysis on the overheads of the barriers or instrumentation instructions introduced by the existing tools. As shown in Figure 4, vendor-specific profilers introduce at least 13 μ s for a single instrumentation point. On the other hand, if we directly embed GPU kernels to record crucial timestamps over GPUs for performance tracing, we can reduce the overhead to only 0.7–0.9 μ s for each trace point (as shown in Figure 4), which is significantly lower than the existing tools. This is easy to understand because GPU kernels are the basic execution units on GPUs. As a result, using kernels to record timestamps is naturally compatible with the other user kernels, and will not break the graph executions, thus mitigating the overheads of the profiling process. If we further design a set of lightweight GPU kernels to fully utilize the asynchronous execution of GPUs, we can reduce the overhead and enable real-time performance profiling.

4 System Overview

Building on the above insight, LMTracer enables fine-grained, real-time performance profiling of production LLM systems through *graph-embedded probing*. As shown in Figure 5, the core of LMTracer is to embed special probe kernels into the original GPU execution graph of LLM workload. These probe kernels capture in-situ performance traces, and stream them to CPUs asynchronously without affecting original execution flows. LMTracer also provides a set of interfaces that allows developers to profile at different granularity levels with a simple configuration (which is independent of the user’s code base), and a dependency-guided trace analysis engine that reconstructs the execution graph to attribute the performance bottlenecks to different system components.

Specifically, LMTracer takes the user configuration for profiling as the input, and performs the following steps to generate performance traces at the production scale.

- **Stack-Aware Compilation (§5.1).** Since the user can specify any modules, functions, or even code regions to be profiled, LMTracer needs to ensure that the probe kernels are correctly inserted into the execution graph without breaking the original execution and kernel fusion. To achieve this, LMTracer performs stack-aware compilation that recursively analyzes the static function call stacks of code regions on kernel fusion. During the compilation of the fused kernel, LMTracer inserts the probes that bound user operators in their corresponding call hierarchies based on the profiling configuration. In this way, LMTracer locates the probe kernels at the right positions for later graph capturing and execution of LLM frameworks.
- **Doorbell-Assisted Shadow Buffering (§5.2).** When LLM systems execute the model, the inserted probe kernels are launched with the original kernels to record the timestamps and other profiling data on GPUs. Existing profilers usually require the CPU to synchronize with the GPU to retrieve the performance data, causing significant performance perturbation to LLM execution. To address this, we devise a buffering mechanism that allows the probe kernels to write the data into one of the two *shadow* GPU buffers, while the CPU reads from the other buffer asynchronously based on a “doorbell” signal without waiting for the GPU. Thus, we eliminate the need for barriers.
- **Bottleneck Analysis (§5.3).** After collecting the performance traces from GPUs through the probe kernels, LMTracer automatically reconstructs the dependencies in the execution graph of the LLM system. It then performs bottleneck analysis to identify the straggler nodes, GPUs, or RDMA NICs (RNICs) that stagnate the execution flows. LMTracer also maps the bottlenecks back to the user-specified code lines if the bottlenecks are relevant to CPU or GPU execution, which allows developers to quickly locate the issues and optimize the implementation accordingly.

5 Design

5.1 Stack-Aware Compilation

Profiling configuration. LMTracer provides users with a set of profiling configuration interfaces. It enables users

to specify code regions or LLM modules to be profiled and the relevant profiling data to be collected or aggregated. As shown in Listing 1, users can define probes at module, function, and code region levels. The initial profiling configuration can be derived from the model structure, i.e., the LLM’s high-level components and execution stages such as attention, MLP, and communication. Such a coarse-grained execution view is sufficient for LMTracer to localize performance issues to specific components or stages. Without this execution view, instrumenting every detailed step of model execution would introduce unnecessary complexity and is impractical for production environments.

Users can specify different LLM frameworks, and LMTracer automatically adapts the instrumentation process to the corresponding framework (cf. §5.2). The configuration also supports multiple exporters to save collected trace data for offline analysis or export metrics to analytics systems such as Prometheus [18] and Flink [29] for real-time performance tracking. The configuration effort is a one-time setup, and the same configuration can be reused across subsequent runs without additional overhead.

Probe kernel. With user-defined profiling configurations, LMTracer instruments the LLM execution by inserting lightweight probe kernels at the specified code regions or module boundaries. These probe kernels are designed to collect critical runtime data on GPUs, in particular GPU kernel execution time. This is achieved by leveraging raw GPU assembly instructions in the probe kernel to read the GPU clock cycle register (e.g., read the `globaltimer` register for NVIDIA GPUs [36, 42] or invoke the `s_memtime` instruction for AMD GPUs [37]) before (or after) the execution of the target kernels and code regions defined in the profiling configuration (as exemplified in Listing 1), and calculating the execution time by differentiating the two clock values.

Unlike existing fine-grained profiling approaches that are intrusive to user kernels by altering kernel instructions [42], LMTracer does not modify the original user kernels so that the data collected by probe kernels preserve high fidelity and accuracy. Also, probe kernels do not modify the runtime states of the user kernels (e.g., registers, shared memory, and local memory [66]), which effectively avoids the state restoration overhead and performance degradation. Users can also customize probe kernels to collect additional traces such as the intermediate output between kernels [47].

Stack-aware compilation. To support flexible profiling over widely used GPU kernel fusion techniques [63], LMTracer needs to overcome a key challenge—the loss of visibility into the internal execution of the user kernels after the kernel fusion. Specifically, a single fused kernel can contain multiple logical operations from different code regions defined by the users (cf. §3). A single code region to be profiled can be fused into different kernels as well. After kernel fusion, the resulting kernels can be captured by CUDA/HIP

Listing 1. Example instrumentation configurations of LMTracer for SGLang runtime probing and data exporting.

```
llm_engine: sglang
probes:
  # LLM modules
  - phase: attention
    module: sglang.srt.models.gpt_oss.GptOssAttention
  # Functions or even code lines
  - phase: norm
    function: sglang.srt.layers.layer_norm.RMSNorm.
      forward_cuda
  # Optional: Flexibility for code-line profiling
  line_start: 109
  line_end: 126
exporters:
  # Exporter 1: Save binary trace data to files
  - module: lmtracer.core.exporter.
    FileByteDataExporter
    params:
      output_dir: /tmp/lmtracer_exports
  # Exporter 2: Export metrics to Prometheus
  - module: lmtracer.core.exporter.
    SGLangMetricPrometheusExporter
    params: {}
```

Graphs [8, 11], allowing the captured execution graph to be launched with a single CPU-side API call, thereby reducing kernel launch overhead. If we simply insert probe kernels at the boundaries of the user-defined code regions, we may break kernel fusion results and significantly degrade the performance. This is because probe kernels are not the basic operators (e.g., arithmetic operators such as addition and multiplication) that can be fused into one kernel.

To address this, we design stack-aware compilation that inserts probe kernels according to operator call stacks during compilation. This mechanism performs a compile-time static analysis on the user code. Specifically, when LLM frameworks compile user operators into kernels (e.g., through PyTorch Inductor [1]), it needs to *symbolically* execute the user code to collect tensor operations involved in kernel generation [9]. During this process, function calls into and out of different code regions of LLM execution can be tracked, forming a static call stack that records the source-code locations associated with each operator. LMTracer uses this call stack information to track the origin of each operator.

After recursively backtracking all call stacks that lead to the operator in each kernel, LMTracer checks whether the call stacks intersect with user-defined profiling regions. If so, LMTracer records the corresponding kernel, thereby establishing a mapping between compiled (potentially fused) kernels and user-defined code regions. Finally, probe kernels are inserted at the boundaries of compiled kernels (rather than between arbitrary user operators) that cover the target

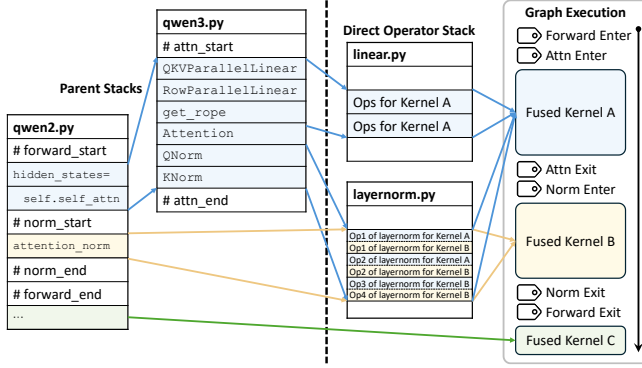


Figure 6. Stack-aware compilation to identify operator fusion and probe kernel insertion points in execution graphs, where colored arrows indicate the mappings from source code through operator call stacks to the fused kernels.

code regions, so that execution time is measured at runtime without disrupting kernel fusion.

As exemplified in Figure 6, suppose the user defines three code regions (i.e., forward, attn, and norm) in the configuration file for profiling, where the forward region contains the entire forward pass of the model in the `qwen2.py` source file, `attn` is the attention module in a single decoder layer in `qwen3.py`, and `norm` is the normalization process after all the decoder layers in `qwen2.py`. LMTracer hooks into the compilation process and first finds that the operators in Fused Kernel A directly originate from the code regions in both `linear.py` and `layernorm.py` (i.e., the code lines shown in blue in the two source files). Nevertheless, these two files have no user-defined code regions for profiling.

LMTracer then backtracks the call stacks of these operators, and discovers that the operators in the linear part of Fused Kernel A are derived from the user-defined `attn` code region in `qwen3.py`, whose stack frame is delimited by `attn_start` and `attn_end`. By further tracing upward, LMTracer confirms that this `attn` region is itself nested inside the outer forward region in `qwen2.py`, bounded by `forward_start` and `forward_end`, which is defined by the user in the profiling configurations. Therefore, Fused Kernel A is simultaneously associated with two user-defined regions, i.e., `attn` and forward. Since it is the first time that the call stack *activates* the `attn` region², LMTracer inserts an `Attn Enter` probe at the beginning of Fused Kernel A, so that the probe kernel can record the entry time of `attn`.

For the operators in Fused Kernel B, they are directly derived from `layernorm.py` when they are compiled (i.e., the code lines shown in yellow in `layernorm.py`), and eventually from the `norm` region in `qwen2.py`. Thus, they are inside the user-defined `norm_start` and `norm_end` regions, nested inside the outer forward region. However, they are

²The forward region is not the first time that the call stack enters, as we omit some other operations before attention regions for simplicity.

not intersected with the previous `attn` region in `qwen3.py`, which was activated when Fused Kernel A was compiled. We therefore determine that the attention region is *deactivated* when Fused Kernel B is sequentially compiled after Fused Kernel A³, so we insert an `Attn Exit` probe at the beginning of Fused Kernel B, and a `Norm Enter` probe at the same time to indicate that the call stack has left the `attn` region and the `norm` region is now activated. Note that the forward region is still active during the compilation of Fused Kernel B.

Finally, when the compiler fuses the operators in Fused Kernel C, LMTracer finds that the fused operators have no call stack intersection with the user-defined code regions, which means that we need to deactivate all the active regions (i.e., `norm` and forward) before Fused Kernel C is executed. Thus, we insert a `Norm Exit` and a `Forward Exit` probe before Fused Kernel C to record the corresponding timestamps.

Based on this stack-aware recursive analysis, LMTracer determines the kernel boundaries that correspond to user-defined profiling regions and inserts probe kernels accordingly. The profiling granularity is defined by user-specified regions rather than individual kernels, as a single region may span one or multiple compiled kernels due to kernel fusion. Therefore, LMTracer automatically identifies the boundary kernels covering the target regions and inserts probes without requiring any user intervention or modifications to the original codebase. Such a probe kernel incurs at most 0.9 μ s overhead as we have reported in Figure 4. Also, even if multiple operators from different source files are fused into GPU kernels (e.g., Fused Kernel A–C in the figure), LMTracer can still faithfully reconstruct the logical execution boundaries.

Kernel-level granularity. During the above stack-aware compilation, LMTracer can encounter the case where the user-defined trace point is inside a fused kernel, although this is rare (<2% of the trace points in production since in most cases profiling targets are aligned with operator or kernel boundaries). To address this, LMTracer supports two modes of probe kernel insertion. The first is inserting probes at the nearest fused kernel boundaries, and the second is splitting the kernel for exact probe placement. By default, LMTracer adopts the first mode to avoid perturbing execution behavior. If finer granularity is required, LMTracer can switch to the second mode to split the kernel and place probes at the exact location. Nevertheless, this mode is primarily intended for offline investigation because it modifies the generated code and may perturb the original performance characteristics.

Code line hashing and probe merging. The above probe insertion point information is used as the input parameter for the probe kernels, so that they can generate the trace data with the mapping information between the GPU kernel execution and the user-defined code regions.

³These fused kernels are also sequentially executed at runtime.

To parameterize this information, LMTracer hashes the source file and code lines to generate a unique integer identifier. This integer identifier will be translated back to the source file when LMTracer processes the trace data. To further reduce the number of inserted kernels and reduce the profiling overhead, LMTracer merges the probes that are inserted at the same kernel boundary. In the above example, the Attn Exit and Norm Enter probes are merged at the beginning of Fused Kernel B. The Norm Exit and Forward Exit probes are merged at the beginning of Fused Kernel C for the same reason. These merged probes are assigned a new hash value for later identification.

CUDA/HIP graph compatibility. Probe kernels are compatible with the CUDA/HIP graph execution, which is especially important for profiling LLM executions in production environments. As we have discussed above (cf. §2), to reduce the intermittent kernel launch overhead, today’s LLM frameworks widely take advantage of the graph execution mechanism provided by the GPU vendors (e.g., CUDA graph [8] and HIP graph [11]). Specifically, before the actual training or inference process, the LLM framework first captures a sequence of GPU kernel executions of the LLM workload into a CUDA/HIP Graph during a warm-up iteration [55]. The captured graph can then be replayed in subsequent iterations with a single CPU-side launch call, thereby reducing the overhead of launching individual kernels.

The probe kernels inserted by LMTracer are treated as normal GPU kernels by the GPU driver and can therefore be captured into the CUDA/HIP graph together with user kernels during the warm-up iteration. For non-linear CUDA/HIP graphs, LMTracer remains compatible because the kernel-to-region mapping established during compilation is independent of the graph topology, rather than relying on a linear kernel ordering. When the graph is replayed for subsequent iterations, the inserted probe kernels are seamlessly executed together with the user kernels in the graph. Such a process does not break graph-based execution, while still allowing LMTracer to collect trace data.

5.2 Doorbell-Assisted Shadow Buffering

The probe kernels inserted by the stack-aware compilation mechanism collect the trace data and export it to the exporters defined by the users. During this process, we need to efficiently manage the data transfer between the GPU and CPU to avoid barriers and minimize the performance overhead. Some approaches defer the data transfer until the end of LLM executions to avoid the barriers, but this leads to large memory usage, and is not suitable for our real-time performance monitoring. Some other approaches perform data transfer after each kernel execution, but this causes significant perturbation to the original execution flows.

To enable zero-barrier profiling in real time, LMTracer employs a doorbell-assisted shadow buffering mechanism. As

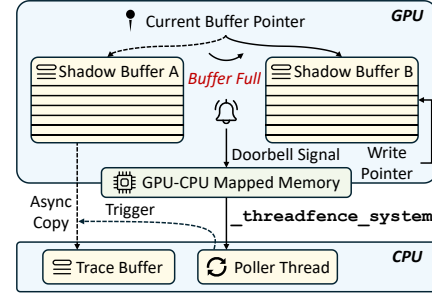


Figure 7. Doorbell-assisted shadow buffering mechanism to mitigate barriers on profiling LLM executions.

shown in Figure 7, LMTracer allocates two GPU-side shadow buffers (i.e., Buffer A and Buffer B) for storing profiling data (e.g., GPU clock values and trace point information) collected by probe kernels. The probe kernels write data into the current shadow buffer and switch to the other buffer when it becomes full. Meanwhile, LMTracer writes a doorbell signal to CPU-GPU mapped memory [22, 23, 44] to notify the CPU that the full buffer is ready for transfer. A background CPU poller monitors the doorbell signal and asynchronously transfers the full buffer in a separate GPU stream, while probe kernels continue writing to the other buffer. This design allows GPU execution and data transfer to proceed concurrently without synchronization overhead. LMTracer exports the collected trace data through user-defined exporters for further analysis (cf. §5.3).

Buffer management and consistency guarantee. To ensure correctness and robustness under high profiling rates, LMTracer uses a lightweight backpressure-free policy. When both shadow buffers become full, newly collected traces overwrite old traces rather than blocking LLM execution. Nevertheless, in long-term practice till now, we have not observed both buffers being full at the same time. A 1 MB shadow buffer is sufficient to store profiling data for over 200 iterations of a 671B-parameter LLM, and the occupancy of the inactive buffer remains below 5% in our production workloads. Moreover, the buffer size can be easily scaled with the number of probes, as the required memory grows approximately linearly with profiling granularity. To guarantee consistency between GPU writes and CPU reads, LMTracer enforces release-acquire ordering [25]. Probe kernels invoke `__threadfence_system()` calls [6] before updating the doorbell, ensuring that all prior buffer writes are visible to the CPU. The CPU poller checks this doorbell to ensure that buffer contents are consistent before transfer.

Process bindings. The runtime execution of probe kernels depends on sharing the user processes, so that they are executed with the user kernels for collecting corresponding trace data. This is because almost all GPU programming models are based on the runtime context of the user processes. GPU kernels can only be launched by the user process that owns the context and has initialized these kernels [26].

However, different LLM frameworks may have different process management and GPU context initialization mechanisms, which makes it challenging to insert the probe kernels into the user processes in a transparent way. Fortunately, by carefully inspecting the implementation of mainstream LLM frameworks, we find that they all follow a scheduler-executor architecture, where the scheduler schedules GPU executions, and the executor strictly binds to one GPU for executing the scheduled tasks. This is easy to understand since today's LLM systems are designed to run on distributed GPU clusters, and the scheduler-executor architecture effectively manages the GPU resources by scheduling the tasks across different GPUs and architectures.

Based on the above observation, we design an executor-binder abstraction layer to automatically find the executor component in the target LLM framework and bind the probe kernels to it. LMTracer locates the executor component of LLM frameworks by leveraging Python⁴ site-specific configuration hooks [20]. With the deployed hooks, LMTracer monitors Python package importing events at runtime, and matches the imported packages that involve GPU executors (e.g., the Worker class in vLLM and the ModelRunner class in SGLang). Once an executor component is detected, LMTracer patches the executor code before it is initialized, so that the probe kernels share the same process context after the executor is initialized. Users can also manually specify the concrete executor in the configuration if they know the implementation details of the target LLM framework.

Synchronizing clocks among GPUs and CPUs. As we mentioned in §5.1, probe kernels collect GPU kernel execution time by reading GPU clock registers. However, in real-world data centers, although CPU times on different hosts are mostly synchronized through the network time protocol [52] or the precision time protocol [57], GPU clocks are usually not synchronized with CPU clocks or the clocks of other GPUs. This is because GPUs are used for high-performance computing, and deviations in GPU clocks have limited impact on performance. However, this inconsistency will lead to inaccurate analysis in performance profiling.

To address this issue, LMTracer records the initial time offsets between GPU and CPU devices instead of continuously synchronizing their clocks, which is more efficient and practical in real-world data centers. After binding to the process of a GPU, LMTracer launches AllGather operations [28, 50] among all GPUs to exchange their initial clock values. For the master GPU of the AllGather operation, LMTracer records its CPU clock value at the same time. LMTracer then calculates the timestamps across all GPUs and CPUs based on their initial offsets to the master GPU. No further synchronization is performed during profiling,

⁴We focus on Python-based LLM frameworks since they are the most widely used. For frameworks implemented in other programming languages, similar techniques like eBPF [30] can be used to hook the executors.

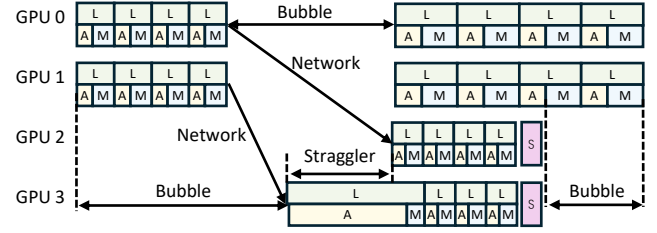


Figure 8. Example inference pipeline profiled by LMTracer, where **A** denotes attention, **M** denotes multi-layer perceptron, **L** denotes decode layers, and **S** denotes sampling.

since a one-time calibration provides sufficiently accurate relative timing for our performance analysis. This is because distributed LLM workloads mainly rely on relative timing across GPUs rather than precise CPU wall-clock time, and relative GPU clock drift is typically small in practice (e.g., at most 2 ms over a month); otherwise, significant clock drift itself may indicate an underlying GPU issue.

5.3 Bottleneck Analysis

LMTracer is designed to be an end-to-end performance profiling framework, which not only collects the trace data through probe kernels, but also provides a suite of interfaces for analyzing the collected data to identify performance issues. To achieve this goal, LMTracer enables users to define data exporters in the profiling configuration (as exemplified in Listing 1). For each exporter, LMTracer exports the collected trace data to the corresponding backend system (e.g., file system, Prometheus, Flink) for the analysis.

Training and inference pipelines. By analyzing trace data and their dependency collected by probe kernels, LMTracer reconstructs the training/serving pipelines, based on which it helps developers identify performance bottlenecks and optimize LLM executions. Specifically, LMTracer first reconstructs kernels' wait-for relationships [64] from their execution timelines and dependency information, and then groups counterpart operations across GPUs within each parallelism group. Bottlenecks are identified by analyzing stragglers and their dependencies. Figure 8 shows an example inference pipeline generated by LMTracer. The inference is configured with TP=2 and PP=2, where GPU 0 and GPU 1 are in the same TP group of the first PP stage. GPU 2 and GPU 3 are in another TP group of the second PP stage.

GPU bubble fraction. By default, LMTracer calculates the GPU bubble fraction to notify the user of resource under-utilization. We define the GPU bubble fraction as the time ratio where the GPU is idle and waiting for data or barriers from other GPUs (as shown in Figure 8). It is calculated as

$$\text{BubbleFraction} = 1 - \frac{\sum_{i=1}^N d_i}{t_N - t_1 + d_N}, \quad (1)$$

where N denotes the number of GPU execution records within the monitoring window, t_i represents the timestamp

when the i -th GPU kernel is launched, and d_i denotes its execution duration. The $\sum_{i=1}^N d_i$ term represents the total active GPU computation time, while $t_N - t_1 + d_N$ denotes the overall time span from the start of the first kernel to the completion of the last kernel. A higher bubble fraction indicates a larger fraction of idle GPU time caused by communication delays, suboptimal scheduling, or input data stalls.

GPU straggling fraction. LMTracer also quantifies the GPU straggling fraction to detect execution skew across distributed GPUs. In parallelized LLM executions, GPUs are expected to start kernels simultaneously. However, due to workload imbalance or hardware defects, some GPUs may finish later than others, resulting in straggling behavior. The GPU straggling fraction is defined as the normalized start-time skew across GPUs, which can be calculated as

$$\text{StragglingFraction} = \frac{\max_{g \in G} s_g - \min_{g \in G} s_g}{\max_{g \in G} e_g - \min_{g \in G} s_g}, \quad (2)$$

where G denotes the set of GPUs participating in the parallel execution (e.g., performing an AllReduce), s_g represents the start timestamp of the kernel executed on GPU g , and e_g denotes the end timestamp of the kernel executed on GPU g . The $\max_{g \in G} s_g - \min_{g \in G} s_g$ term captures the start-time skew among GPUs, while the $\max_{g \in G} e_g - \min_{g \in G} s_g$ term represents the overall execution span across all GPUs. A higher straggling fraction indicates more severe performance inconsistency among GPUs caused by abnormal devices.

Network observability. Since LMTracer allows users to insert probe kernels between adjacent kernels, it can also be leveraged to analyze network performance in distributed LLM executions. In modern LLM systems, communication operations are typically encapsulated into dedicated kernels and compiled together with computation kernels into a unified execution graph (e.g., the AllReduce and AllGather kernels in vLLM [46]). By placing probe kernels around these communication kernels, LMTracer accurately measures their execution time and helps developers identify potential network bottlenecks during distributed execution.

6 Implementation

We implement LMTracer in 4,915 lines of Python code and 230 lines of C++ code. We designed a suite of interfaces for different LLM frameworks to load LMTracer’s probe kernels, as well as different exporters to export the collected profiling data for various downstream analysis and visualization tools. Currently, LMTracer supports vLLM [46], SGLang [71], and Megatron-LM [60] for both training and serving. It also supports exporting the profiling data to different formats, including Prometheus [18], binary files, databases, real-time dashboards, and Perfetto trace [54].

Developers or users simply install LMTracer as a Python package, after which LMTracer detects the target framework and loads the corresponding hooks. LMTracer allows users

to configure the profiling settings through a configuration file specified by an environment variable. LMTracer also provides basic preset profiling configurations for different models like Qwen3, DeepSeek-V3, GPT-OSS, and Llama3.

7 Evaluation

7.1 Evaluation Setups

LMTracer has been deployed in production for eight months, covering training and serving of various LLMs (e.g., Qwen3, DeepSeek-V3, etc.) and frameworks (e.g., vLLM, SGLang, Megatron-LM, etc.) on different GPU models. In our experiments in §7.2 and §7.3, we evaluate LMTracer with the same hardware and software configurations as in production. We also compare LMTracer with other commonly used profilers (i.e., Nsight Systems, runtime events, and PyTorch Profiler) to show the advantages of LMTracer in overhead and fidelity.

For LLM serving, we measure the token throughput and time to first token (TTFT) of Qwen3-32B inference over eight low-end GPUs, DeepSeek-V3 inference over eight as well as 16 mid-range GPUs, and GPT-OSS-120B inference over eight mid-range GPUs. These models are representative of the LLMs we have in production, and we use real-world user requests during peak hours for evaluation. For LLM training, we measure the throughput of DeepSeek-V3 supervised fine-tuning (SFT) over 256 high-end GPUs, which is also a common training configuration in production. We use the preset profiling configurations (cf. §6) for evaluation. They track the duration of the attention and MLP operations in each of the LLM layers, as well as the total layer duration and model forward duration. We set the buffer flush interval to 1k records for all experiments.

7.2 Performance

We evaluate the profiling overhead of LMTracer by measuring the token throughput and latency of LLM training as well as LLM inference with and without LMTracer enabled.

Profiling overhead. Figure 9 shows the token throughput of Qwen3-32B inference on low-end GPUs with different profiling tools. LMTracer achieves negligible performance degradation of 0.9% (from 3,157 to 3,128 tokens/s) with TP=8, and 0.6% (from 3,743 to 3,721 tokens/s) with TP=4 and PP=2 on vLLM. On SGLang, LMTracer only causes 0.4% throughput degradation (from 3,103 to 3,090 tokens/s) with TP=8, and 0.3% throughput degradation (from 4,145 to 4,132 tokens/s) with TP=4 and PP=2.

In contrast, other profilers all cause greater throughput degradation. This is specifically for PyTorch Profiler, which causes 87.9%, 91.7%, 82.4%, and 86.5% degradation for the four configurations, respectively. This is because PyTorch Profiler breaks the graph executions on profiling, and introduces barriers when mapping GPU-side kernels to CPU-side call stacks with extensive Python-runtime instrumentation.

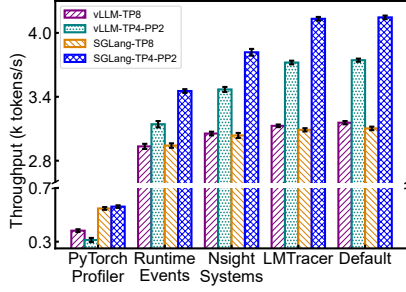


Figure 9. Qwen3-32B inference performance on low-end GPUs.

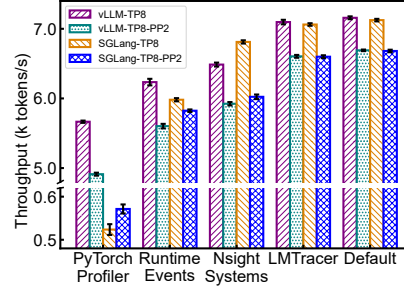


Figure 10. DeepSeek-V3 inference performance on mid-range GPUs.

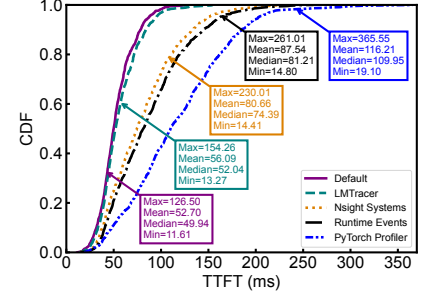


Figure 11. TTFT of GPT-OSS-120B on mid-range GPUs.

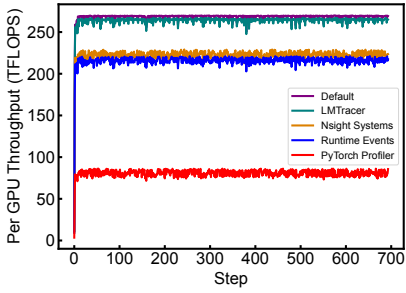


Figure 12. DeepSeek-V3 training performance on high-end GPUs.

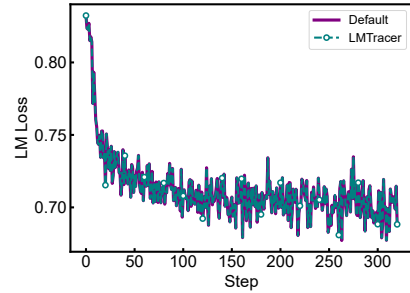


Figure 13. Language model loss with and without LMTracer.

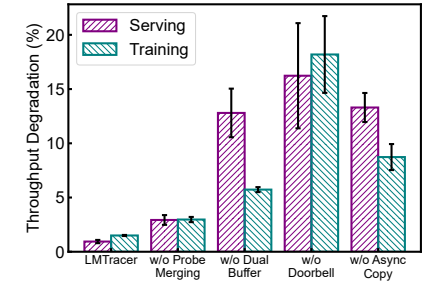


Figure 14. Ablation study of LMTracer.

NVIDIA Nsight Systems and runtime event tracing cause more than 7% throughput degradation on average, which is much higher than LMTracer’s overhead and limits their applicability in production environments. It should be noted that we enable graph-node profiling [24] for NVIDIA Nsight Systems so that it captures the execution time of the kernels inside the CUDA graph. Even if we disable this mode, it still causes more than 3.6% throughput degradation without visibility into the kernel execution time, which limits its applicability as well. Similar trends can be observed in Figure 10 for DeepSeek-V3 inference over mid-range GPUs.

LMTracer’s profiling overhead is oftentimes lower under inter-host configurations (TP=4 and PP=2) than under intra-host configurations (TP=8). This is because probe kernels inherently allow LMTracer to use inter-host pipelines to hide the profiling overhead. However, other profiling tools do not have such a capability because they rely on the synchronization among intra- and inter-host GPUs.

In terms of the TTFT metric, Figure 11 shows that LMTracer only incurs 3.4 ms average TTFT increase (from 52.7 to 56.1 ms) for GPT-OSS-120B inference over mid-range GPUs, which can hardly be noticed by users in the real world. In contrast, the other profiling tools cause up to 63.5 ms TTFT increase, which significantly affects user experience.

LMTracer also has negligible impacts on LLM training performance. As shown in Figure 12, LMTracer only causes an average of 1.5% throughput degradation for DeepSeek-V3

SFT over 256 high-end GPUs, while Nsight Systems and runtime event tracing cause more than 17% throughput degradation on average. For PyTorch Profiler, it causes 69.9% throughput degradation. For other LLM training and serving configurations, similar trends can be observed in production.

Correctness. The use of LMTracer does not affect the correctness of LLM training and inference. As shown in Figure 13, the language model loss in LLM training with LMTracer is almost the same as that without LMTracer, which indicates that LMTracer does not alter the data flow and the training dynamics of the LLM. Similarly, we also verify the correctness of LLM inference with LMTracer by comparing the output tokens (using the same set of input prompts and random seeds), and we find that they are exactly the same.

7.3 Ablation Study

We perform an ablation study to evaluate the effectiveness of LMTracer’s key design choices, including the design of probe merging, shadow buffers, doorbell signals, and the asynchronous copy of profiling data. When probe merging is disabled, we insert all probe kernels that the user has defined in profiling configurations, rather than merging adjacent ones. When shadow buffers are disabled, we only use one buffer to store the profiling data, and we flush the buffer after every profiling record is collected. When doorbell signals are disabled, LMTracer actively queries the GPU to check

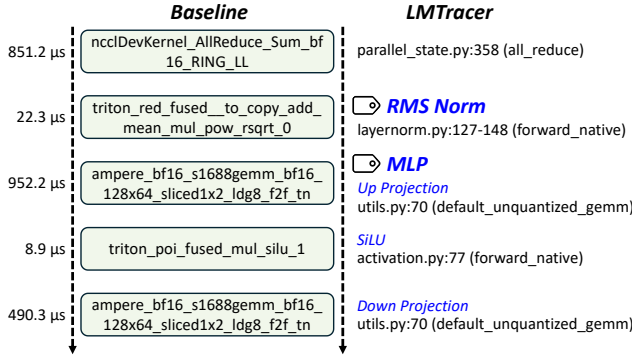


Figure 15. Comparing profiling fidelity of profiling tools. The baseline only provides kernel names, while LMTracer provides concrete mappings from the kernels to the user code based on user-defined profiling configurations.

whether one of the trace buffers is full after each LLM iteration. When the asynchronous copy of profiling data is disabled, LMTracer copies the profiling data from GPU to CPU in the same stream as the user kernels.

As shown in Figure 14, when any of the key design choices is disabled, the profiling overhead of LMTracer significantly increases. When probe merging is disabled, the throughput degradation increases to 2.9% and 3.0% for Qwen3-32B inference over low-end GPUs and DeepSeek-V3 SFT over high-end GPUs, respectively. This indicates that merging adjacent probe kernels like the boundary kernels between the attention and MLP operations in each LLM layer is effective in reducing the profiling overhead. When shadow buffers are disabled, the throughput degradation increases to 12.8% and 5.7%, indicating that buffers are crucial for hiding the data copy overhead. Similarly, when doorbell signals are disabled, the degradation reaches 16.2% and 18.2%. When the asynchronous copy of profiling data is disabled, the degradations are 13.3% and 8.7%. These all reveal that enabling asynchronous copy of profiling data is important.

7.4 Fidelity

LMTracer provides high-fidelity profiling data that accurately reflects the performance characteristics of even a single code region. Figure 15 shows a profiling result of LMTracer and the baseline profiling tool (i.e., NVIDIA Nsight Systems). The baseline profiling tool has shown the most detailed profiling outputs among all the other compared tools. Nevertheless, it can only capture the GPU-side execution time of the compiled kernels. These kernels have lost the high-level semantics of the original user code after compilation, which makes it difficult to understand which code lines or which LLM layers cause the performance bottlenecks. Similarly, the fidelity of runtime event tracing is worse than that of Nsight Systems, because it breaks kernel fusion and thus causes more kernels to be generated.

In contrast, LMTracer captures the GPU-side execution time of user kernels that are directly projected to the original code. For example, LMTracer identifies the kernels fused by Triton [62] in the root mean square (RMS) normalization, and also captures the regions in user code based on the profiling configuration, i.e., the phase labels (cf. Listing 1). Similarly, LMTracer helps distinguish the up projection, SiLU activation [70], and down projection in the MLP operation [39], while the baseline only captures the cuBLAS [5] kernels.

Note that directly using kernel names and tensor shapes does not provide comparable clues. Modern LLM compilers often generate generic kernels (e.g., cuBLAS kernels) that obscure the original model semantics. Also, different operators may have similar tensor shapes, while the same operator may map to different kernels in different workloads, making kernel names and tensor shapes unreliable indicators of operator-level behaviors. Instead, stack-aware compilation in LMTracer establishes this mapping reliably from compile-time call stacks and source-code locations.

7.5 Case Study

LMTracer has helped identify 15 performance bottlenecks and defects in our production LLM systems. Table 2 summarizes representative issues, covering configurations, frameworks, and hardware in training and serving. Nine of them would be missed by sampled NVIDIA Nsight Systems or PyTorch Profiler runs, as they arise from dynamic production workloads. Issues such as KV cache eviction, prefill blocking, transient RNIC flapping, and imbalanced PP scheduling depend on specific request patterns or runtime conditions, and may occur only intermittently. Consequently, they are difficult to capture within short profiling windows, whereas LMTracer can observe them reliably through always-on tracing. We also present some typical issues in the following.

Unbalanced scheduling. Figure 16 shows an example of the unbalanced scheduling issue in LLM serving. This is caused by the framework’s scheduling policy that does not consider the performance heterogeneity of user requests, which leads to different inference times of different forwarding passes, and causes bubbles in inference pipelines. With LMTracer, we can get notified of this issue based on the bubble fraction metric. For common profiling tools, it is hard to identify the root cause of this issue because they cannot capture the performance characteristics of each kernel execution at the granularity of user requests.

Hardware issues. LMTracer has also helped identify various hardware issues, such as NVLink down and RNIC flapping. Figure 17 is an example of the NVLink issue that LMTracer has detected in training. In the example, LMTracer captures each GPU’s execution time in each DP, PP, and TP group. We then notice that GPUs (that correspond to one PP instance) in DP 1 have longer execution time than the GPUs in DP 0 (i.e., the straggling fraction is abnormal). This makes

Table 2. Issues detected by LMTracer in production.

Component	Root Cause	Symptom
Configuration	Improper layer placement	PP stages are much longer.
Configuration	Improper parallelism	TP groups use slow paths.
Configuration	Not using CUDA graph	Separate kernel launching.
Framework	Unbalanced scheduling	Some GPUs are busier.
Framework	KV cache eviction	Repeated computation.
Framework	Prefill blocking	High TTFT for users.
Hardware	NVLink down	Training process is stalled.
Hardware	RNIC flapping	Intermittent unconnectivity.

the GPUs in DP 0 wait for the GPUs in DP 1 to finish each iteration, which significantly degrades the performance. Also, LMTracer’s profiling data show that the AllReduce time of the problematic GPUs in DP 1 is much longer. After further investigation, we find that the NVLink in one of the hosts in DP 1 reports errors. This link then straggles the entire PP instance, and further straggles DP 0 as well as the entire training job. Without LMTracer, it is hard to identify the root cause because the performance characteristics of each GPU and parallelism group can be affected by profiling overheads, which might be distorted by the profiling action itself.

In summary, after fixing all the issues identified by LMTracer, we have observed 21.2% latency reduction in LLM serving and 24.5% throughput improvement.

8 Related Work

LLM profiling tools. Profiling the performance of LLM systems is critical for understanding the performance bottlenecks. Among the existing LLM profiling tools, some of them focus on GPU-level profiling, such as NVIDIA Nsight Systems [16], NVIDIA Nsight Compute [14], and CUPTI [15]. Nevertheless, most of these tools are designed to be closed source given their low-level access to the GPU hardware. On the other side, open-source GPU-level profiling tools like NeutriNo [42] have inherent limitations that they can only capture the performance of the kernels with raw GPU instructions. Thus, for most kernels in LLM training and serving that are implemented with high-level libraries, these tools are unable to provide sufficient performance insights. Although some framework-level profiling tools, such as PyTorch Profiler [21], are open source, they cannot measure accurate kernel execution time due to the significant profiling overhead, and thus cannot be used in production.

LMTracer is designed to be a low-overhead profiler that captures performance data of GPU kernels. Our goal is not to replace existing profiling and diagnosis tools, but to fill an observability gap that is particularly important for production LLM systems, since none of the existing tools can realize fine-grained, real-time, and kernel-level performance profiling for online systems.

LLM failure diagnosis. In recent years, there have been several studies on diagnosing failures in LLM training and serving [48, 67]. Aegis [33] is a fault diagnosis system for

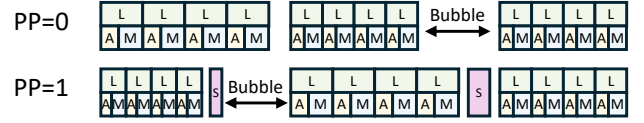


Figure 16. Example of unbalanced user request scheduling.

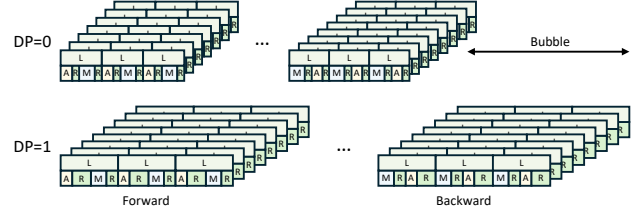


Figure 17. Example of straggling behaviors in LLM training caused by NVLink issues, where **R** denotes AllReduce.

LLM training in production, which achieves runtime failure localization by combining training-specific log analysis with a customized collective communication library. Mycroft [32] leverages flow- and chunk-level dependencies of NCCL to detect problematic GPUs in seconds, while Greyhound [67] hooks into NCCL function calls to identify slow worker groups. SkeletonHunter [50] is a system based on network probing to detect and localize network failures in LLM training. Nevertheless, these systems cannot capture detailed performance characteristics of GPU kernels, and thus cannot be used to diagnose the performance bottlenecks, despite the ability to localize the failure after the performance degradation has already happened.

9 Conclusion

Training and serving large language models (LLMs) is an important business of AI providers, while profiling LLM systems in production for performance and resource optimization is a critical yet challenging task. Existing profiling tools are either too coarse-grained to provide actionable insights, or too heavyweight to be used without significant overhead. This paper presents LMTracer, a high-fidelity and real-time profiling framework for production LLM systems. It seamlessly embeds profiling logic into the LLM workloads through *graph-embedded probing*, which inserts lightweight probe kernels into the execution while mitigating the CPU-GPU synchronizations. LMTracer has been deployed in production for eight months, and has identified 15 performance issues. After fixing these issues, we observed 21.2% latency reduction and 24.5% throughput improvement. In a broader sense, LMTracer demonstrates that always-on, fine-grained management on modern LLM infrastructure is practical with careful graph-level optimizations on GPU kernel executions.

Acknowledgments

We thank Hao Lin for the insightful discussions on extending the application scenarios of LMTracer. We thank our shepherd, John Wilkes, and the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by the National Natural Science Foundation of China under grants 62332012 and 62472245.

References

- [1] 2022. TorchInductor: A PyTorch-native Compiler with Define-by-Run IR and Symbolic Shapes. <https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747>.
- [2] 2025. Asynchronous Execution — CUDA Programming Guide. <https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html>.
- [3] 2025. LLM Inference Optimization Techniques: Speed & Cost Guide 2026. <https://hakia.com/tech-insights/llm-inference-optimization/>.
- [4] 2025. OpenAI's Annualized Revenue Hits \$10 Billion, up from \$5.5 Billion in December 2024. *Reuters* (2025).
- [5] 2026. cuBLAS 13.2 Documentation. <https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublas-api>.
- [6] 2026. CUDA C++ Programming Guide (Legacy) — CUDA C++ Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-fence-functions>.
- [7] 2026. CUDA Event Management. <https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>.
- [8] 2026. CUDA Graphs — CUDA Programming Guide. <https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html>.
- [9] 2026. Dynamo Deep-Dive — PyTorch 2.13 Documentation. https://docs.pytorch.org/docs/stable/user_guide/torch_compiler/torch_compiler_dynamo_deepdive.html.
- [10] 2026. Google Gemini. <https://gemini.google.com>.
- [11] 2026. HIP Graphs. https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/hip_runtime_api/hipgraph.html.
- [12] 2026. HIP Runtime API Reference: Event Management. https://rocm.docs.amd.com/projects/HIP/en/develop/doxygen/html/group__event.html.
- [13] 2026. Meta AI. <https://www.meta.ai/>.
- [14] 2026. Nsight Compute Documentation — NsightCompute 13.2 Documentation. <https://docs.nvidia.com/nsight-compute/>.
- [15] 2026. NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit. <https://developer.nvidia.com/cupti>.
- [16] 2026. NVIDIA Nsight Systems. <https://developer.nvidia.com/nsight-systems>.
- [17] 2026. OpenAI. <https://openai.com/>.
- [18] 2026. Prometheus - Monitoring System & Time Series Database. <https://prometheus.io/>.
- [19] 2026. ROCm/ROCm-Systems: Super Repo for ROCm Systems Projects. <https://github.com/ROCm/rocm-systems>.
- [20] 2026. Site — Site-specific Configuration Hook. <https://docs.python.org/3/library/site.html>.
- [21] 2026. Torch.Profiler — PyTorch 2.10 Documentation. <https://docs.pytorch.org/docs/stable/profiler.html>.
- [22] 2026. Unified and System Memory — CUDA Programming Guide. <https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/understanding-memory.html>.
- [23] 2026. Unified Memory — HIP 6.2.41133 Documentation. https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.0/how-to/unified_memory.html.
- [24] 2026. User Guide — Nsight Systems. <https://docs.nvidia.com/nsight-systems/UserGuide/index.html#cli-profile-command-switch-options>.
- [25] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking Under the Release-Acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA (2018).
- [26] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *Proc. of USENIX OSDI*. 499–514.
- [27] Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu, and Qi Tian. 2023. Accurate Medium-Range Global Weather Forecasting with 3D Neural Networks. *Nature* 619, 7970 (2023), 533–538.
- [28] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, Mingwei Xu, Tianshu Wang, Ning Wang, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2025. SyCCL: Exploiting Symmetry for Efficient Collective Communication Scheduling. In *Proc. of ACM SIGCOMM*. 645–662.
- [29] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. *Proc. of VLDB Endowment* 10, 12 (2017), 1718–1729.
- [30] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *Proc. of USENIX NSDI*. 407–425.
- [31] Weiwei Chu, Xinfeng Xie, Jiecao Yu, Jie Wang, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park. 2025. Scaling Llama 3 Training with Efficient Parallelism Strategies. In *Proc. of ACM ISCA*. 1703–1716.
- [32] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaozhong Liu, Yang Bai, Shuguang Wang, Wencong Xiao, Jianxi Ye, Minlan Yu, and Hong Xu. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. In *Proc. of ACM SOSP*. 254–269.
- [33] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, et al. 2025. Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production. In *Proc. of USENIX NSDI*. 865–881.
- [34] Mohamad Fakhri, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2024. LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems. In *Proc. of ACM ICSE-SEIP*. 192–203.
- [35] Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Lucioni, and Emma Strubell. 2025. Energy Considerations of Large Language Model Inference and Efficiency Optimizations. In *Proc. of ACL*. 32556–32569.
- [36] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. 2021. Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels. *ACM SIGMETRICS Performance Evaluation Review* 48, 3 (2021), 81–88.
- [37] Yue Guan, Yuanwei Fang, Keren Zhou, Corbin Robeck, Manman Ren, Zhongkai Yu, Yufei Ding, and Adnan Aziz. 2025. KPerfIR: Towards an Open and Compiler-Centric Ecosystem for GPU Kernel Performance Tooling on Modern AI Workloads. In *Proc. of USENIX OSDI*. 205–220.
- [38] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirog Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, et al. 2025. DeepSeek-R1 Incentivizes Reasoning in LLMs Through Reinforcement Learning. *Nature* 645, 8081 (2025), 633–638.
- [39] Muhammad Haris, Gregory Shakhnarovich, and Norimichi Ukita. 2018. Deep Back-Projection Networks for Super-Resolution. In *Proc. of IEEE CVPR*.

- [40] Jingkai He, Pengfei Chen, Yilun Wang, Haiyu Huang, Chuanfu Zhang, Haojia Huang, and Danwen Chen. 2025. LLMConf: Knowledge-Enhanced Configuration Optimization for Large Language Model Inference. In *Proc. of IEEE IWQoS*. 1–10.
- [41] Victor Holmin. 2025. Mastering LLM Inference: Cost-Efficiency and Performance. <https://medium.com/teconomics-innovation-and-cyber-contemplations/mastering-llm-inference-cost-efficiency-and-performance-7292ba759dc3>.
- [42] Songlin Huang and Chenshu Wu. 2025. NeutriNo: Fine-Grained GPU Kernel Profiling via Programmable Probing. In *Proc. of USENIX OSDI*.
- [43] Amber Jackson. 2025. AWS Cloud Revenue Hits US\$330bn as AI Hardware Drives Growth. <https://datacentremagazine.com/hyperscale/aws-remains-330bn-cloud-market-leader-driven-by-ai-growth>.
- [44] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hye-soon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proc. of ACM ASPLOS*. 1357–1370.
- [45] Dan Kondratyuk, Lijun Yu, Xiuye Gu, José Lezama, Jonathan Huang, Grant Schindler, Rachel Hornung, Vighnesh Birodkar, Jimmy Yan, et al. 2024. VideoPoet: A Large Language Model for Zero-Shot Video Generation. In *Proc. of ICML*.
- [46] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. of ACM SOSP*. 611–626.
- [47] Ignacio Laguna. 2019. FPChecker: Detecting Floating-Point Exceptions in GPU Applications. In *Proc. of IEEE/ACM ASE*. 1126–1129.
- [48] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, et al. 2025. Understanding Stragglers in Large Model Training Using What-If Analysis. In *Proc. of USENIX NSDI*.
- [49] Wei Liu, Kun Qian, Zhenhua Li, Feng Qian, Tianyin Xu, Yunhao Liu, Yu Guan, Shuhong Zhu, Hongfei Xu, Lanlan Xi, Chao Qin, and Ennan Zhai. 2025. Mitigating Scalability Walls of RDMA-based Container Networks. In *Proc. of USENIX NSDI*.
- [50] Wei Liu, Kun Qian, Zhenhua Li, Tianyin Xu, Yunhao Liu, Weicheng Wang, Yun Zhang, Jiankang Li, Shuhong Zhu, Xue Li, Hongfei Xu, Fei Feng, and Ennan Zhai. 2025. SkeletonHunter: Diagnosing and Localizing Network Failures in Containerized Large Model Training. In *Proc. of ACM SIGCOMM*. 527–540.
- [51] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proc. of ACM SIGCOMM*. 38–56.
- [52] David L Mills. 1989. On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System. *ACM SIGCOMM Computer Communication Review* 20, 1 (1989), 65–75.
- [53] Santiago Miret and N. M. Anoop Krishnan. 2025. Enabling Large Language Models for Real-World Materials Discovery. *Nature Machine Intelligence* 7, 7 (2025), 991–998.
- [54] José Miguel Moreno, Narseo Vallina-Rodriguez, and Juan Tapiador. 2023. Chrowned by An Extension: Abusing the Chrome DevTools Protocol Through the Debugger API. In *Proc. of IEEE EuroS&P*. 832–846.
- [55] Vinh Nguyen, Michael Carilli, Sukru Burc Eryilmaz, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, and Edward Yang. 2021. Accelerating PyTorch with CUDA Graphs – PyTorch. <https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/>.
- [56] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, et al. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Proc. of NeurIPS*. Article 2011.
- [57] Diana Andreea Popescu and Andrew W. Moore. 2021. Measuring Network Conditions in Data Centers Using the Precision Time Protocol. *IEEE Transactions on Network and Service Management* 18, 3 (2021), 3753–3770.
- [58] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proc. of ACM SIGCOMM*. 691–706.
- [59] Ruoyu Qin, Zheming Li, Weiran He, Jiale Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-Centric Architecture for Serving LLM Chatbot. In *Proc. of USENIX FAST*. 155–170.
- [60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. <http://arxiv.org/abs/1909.08053>. arXiv:1909.08053 [cs]
- [61] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumix: Dynamic Scheduling for Large Language Model Serving. In *Proc. of USENIX OSDI*.
- [62] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proc. of ACM MAPL*. 10–19.
- [63] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *Proc. of IEEE SC*. 191–202.
- [64] Weitao Wang, Xinyu Crystal Wu, Praveen Tammanna, Ang Chen, and T S Eugene Ng. 2022. Closed-Loop Network Performance Monitoring and Diagnosis with SpiderMon. In *Proc. of USENIX NSDI*. 267–285.
- [65] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proc. of ACM SOSP*. 640–654.
- [66] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: Detecting CUDA Synchronization Bugs Via Memory-Access Modeling. In *Proc. of ACM/IEEE ICSE*. 937–948.
- [67] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, and Wenchao Wu. 2025. Greyhound: Hunting Fail-Slows in Hybrid-Parallel Training at Scale. In *Proc. of USENIX ATC*.
- [68] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proc. of ACM SOSP*. ACM, 1030–1045.
- [69] Zhiyi Yao, Pengbo Hu, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, Zekun He, Yachen Wang, Xian-neng Zou, and Junchen Jiang. 2025. Holmes: Localizing Irregularities in LLM Training with Mega-Scale GPU Clusters. In *Proc. of USENIX NSDI*.
- [70] Xia Zhao, Limin Wang, Yufei Zhang, Xuming Han, Muhammet Deveci, and Milan Parmar. 2024. A Review of Convolutional Neural Networks in Computer Vision. *Artificial Intelligence Review* 57, 4 (2024), 99.
- [71] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proc. of NeurIPS*.
- [72] Yizhen Zheng, Huan Yee Koh, Jiaxin Ju, Anh T. N. Nguyen, Lauren T. May, Geoffrey I. Webb, and Shirui Pan. 2025. Large Language Models for Scientific Discovery in Molecular Property Prediction. *Nature Machine Intelligence* 7, 3 (2025), 437–447.
- [73] Yue Zhu. 2026. LLM Inference Optimization: Cut Latency 50% & Cost 60%. <https://seenos.ai/llm-optimization/llm-inference-optimization>.