XPUTIMER: Anomaly Diagnostics for Divergent LLM Training in GPU Clusters of Thousand-Plus Scale

Weihao Cui[1]*, Ji Zhang[2]*, Han Zhao[1], Chao Liu[2], Wenhao Zhang[1], Jian Sha[2]†
Quan Chen[1]†, Bingsheng He[3], Minyi Guo[1]
[1]*Shanghai Jiao Tong University,*[2]*Ant Group,*[3]*National University of Singapore*

Abstract

The rapid proliferation of large language models has driven the need for efficient GPU training clusters. However, ensuring high-performance training in these clusters is challenging due to the complexity of software-hardware interactions and the frequent occurrence of training anomalies. Since existing diagnostic tools are narrowly tailored to specific issues, there are gaps in their ability to address anomalies spanning the entire training stack. In response, we introduce XPUTIMER, a real-time diagnostic framework designed for distributed LLM training at scale. XPUTIMER first integrates a lightweight tracing daemon to monitor key code segments with minimal overhead. Additionally, it features a diagnostic engine that employs novel intra-kernel tracing and holistic aggregated metrics to efficiently identify and resolve anomalies. Deployment of XPUTIMER across 6,000 GPUs over eight months demonstrated significant improvements across the training stack, validating its effectiveness in real-world scenarios.

1 Introduction

The advent of large language models (LLMs) has revolutionized the deep learning training community, driving substantial advancements in artificial intelligence-generated content (AIGC). Recognizing their transformative potential to enhance user experiences, leading corporations are proactively leveraging LLMs to enhance a wide array of user-oriented services [1–3]. To meet the computational demands of LLM training, they construct large-scale training clusters comprising the latest GPUs interconnected via high-bandwidth links.

Figure 1 depicts the general training stack of the large-scale training cluster in modern corporations. As shown, the operations team manages low-level resources, and the infrastructure team delivers training optimizations [4–7], with particular emphasis on parallel backbones [8–10]. Supported by these two teams, multiple algorithm teams focus on adapting LLMs for user-facing applications through various training methods [11, 12]. Notably, since LLMs do not solve all problems, the training cluster also supports other deep learning jobs, such as recommendation models and their specific parallel backbone, TorchRec [13].

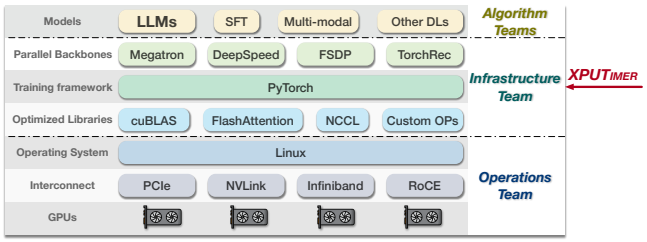

Figure 1: The summarized training stack of large-scale training cluster in Ant Group, highlighting XPUTIMER's position.

Efficient distributed LLM training of large scale requires delicate collaboration among the teams across the stack. However, due to the involvement of thousands of software and hardware components, training anomalies frequently arise from various layers. These anomalies include obvious job failures and non-obvious training slowdowns, which should be resolved by different teams.

Specifically, the algorithm teams may unintentionally use incorrect training configurations, leading to training slowdowns. Meanwhile, the slowdowns may also come from unoptimized operators, which fall under the responsibility of the infrastructure team. Moreover, a significant number of job failures are caused by hardware errors. However, the current training stack lacks a dedicated diagnostic framework to address these anomalies. In its absence, resolving such issues requires cross-team investigations, resulting in high communication overhead and low resolution efficiency.

In order to design a deployable diagnostic framework in the training cluster, we identify three main challenges based on real-world operational experiences. ***C-1****: Designing a long-running and lightweight diagnostic framework is challenging.* Addressing non-obvious training slowdowns and low-level communication errors requires long-term and in-depth mon-

*Weihao Cui and Ji Zhang contributed equally to this work.
†Jian Sha and Quan Chen are the corresponding authors

itoring. However, exhaustive tracing mechanisms, such as PyTorch's built-in profiler, impose a high runtime memory overhead, making it unsuitable for meeting long-running requirements. ***C-2**: Detecting and diagnosing the underlying root cause is challenging.* Errors or slowdowns in LLM training often manifest with similar symptoms, such as process hangs or decreased training speed, which obscure the actual problematic machine or code. ***C-3**: Providing a backbone-extensible diagnostic mechanism is challenging.* As shown in Figure 1, at least three backbones are commonly used across the LLM community, not to mention those supporting other deep learning models. Making diagnostic mechanisms extensible to multiple backbones is a significant challenge.

Faced with these challenges, previous efforts [2, 3, 14] targeting distributed LLM training fail to provide comprehensive solutions. This is because these approaches are narrowly designed for specific problems or scenarios, making them inadequate for addressing issues across the software-hardware stack. For example, FALCON [14] and C4D [3] are capable of diagnosing low-level network issues. However, our observations suggest that many slowdowns are caused by upper-level teams, as shown in Table 1. Likewise, tools such as MegaScale [2] are optimized for pre-training scenarios where teams collaborate closely and use only the Megatron backbone, limiting their adaptability to the advancing training stack.

To this end, we present XPUTIMER, a real-time and holistic anomaly diagnostic framework designed for efficient distributed LLM training in GPU clusters of thousand-plus scale. XPUTIMER consists of two components: a per-training-process tracing daemon and a diagnostic engine. To collect runtime data with low overhead (**C-1**), the tracing daemon instruments only key code segments rather than blindly tracing all runtime data. Furthermore, as the code segments are carefully selected and intercepted at the level of both Python and C++ runtime, the instrumentation provides sufficient critical information for diagnosis (**C-2**) while remaining backbone-agnostic (**C-3**). Notably, XPUTIMER is also hardware-extensible, which could provide seamless support for other NPUs.

With the real-time data collected by the tracing daemon, the diagnostic engine detects and diagnoses anomalies, including errors and slowdowns(**C-2**). For error diagnostics, XPUTIMER introduces a novel intra-kernel tracing mechanism, providing fine-grained diagnostics specifically targeting communication-related hang errors. Compared to a binary search using NCCL tests [2, 5], it reduces the complexity of faulty machine diagnostics from $O(\log N)$ to $O(1)$. For slowdown diagnostics, XPUTIMER proposes holistic aggregated metrics that encompass not only commonly used macro metrics like training throughput but also novel micro metrics, such as issue latency distribution. With these metrics, XPUTIMER can diagnose non-obvious slowdowns (e.g., 2.66%) in real-world workloads.

We extensively evaluated XPUTIMER in terms of its runtime overhead. XPUTIMER incurs an average latency overhead of only 0.43% across various LLMs and backbones on 1024 H800 GPUs. Meantime, XPUTIMER only generates just 1.5MB of tracing logs per GPU in a real-world model trained on 1536 H800 GPUs. In addition, XPUTIMER has been deployed in our training cluster in Ant Group [1], utilizing more than 6,000 GPUs over the span of 8 months. XPUTIMER has helped to optimize the whole stack of LLM training, in terms of model designing, infrastructure optimization, and cluster operations. We present detailed case studies of diagnosed anomalies in §7 and practical insights in §8 derived from XPUTIMER's daily deployment.

XPUTIMER has been open-sourced at https://github.com/intelligent-machine-learning/dlrover/tree/master/xpu_timer. XPUTIMER serves as a core component of DLRover [15], an automated distributed DL system deployed within Ant Group and supported by the LF AI & Data Foundation [16].

Our contributions are as follows.

- We highlight the urgent need for a real-time, holistic diagnostic framework capable of identifying LLM training anomalies across the entire training stack.
- We introduce XPUTIMER, a diagnostic framework specifically designed to tackle the critical challenges of long-term monitoring, root cause diagnosis, and backbone extensibility in LLM training diagnostics.
- We deploy XPUTIMER across more than 6,000 GPUs over an 8-month period, deriving typical case studies and practical insights from its daily operations.

2 Background and Motivation

In this section, we introduce the training of LLMs within corporations like Ant Group, and motivate the design of XPUTIMER for anomaly diagnostics of LLM training.

2.1 Large-Scale LLM Training Stack

The pursuit of more powerful large language models [17–26] has sparked an intense competition between leading corporations. Referring to the software-hardware stack for training LLMs in Figure 1, we delve into how LLMs are reshaping corresponding teams.

Advancing of LLM applications. With the advancing of LLM algorithms [18], LLMs excel not just in natural language understanding, but also in tackling advanced tasks such as multimodal tasks [27, 28], reasoning tasks [29, 30]. For instance, customer service teams fine-tune LLMs to develop chatbots that generate accurate responses to complicated questions. Similarly, product development teams leverage multimodal LLMs to generate comprehensive product descriptions by integrating diverse inputs, such as product images and textual data. Consequently, different algorithm teams continue

Table 1: A comprehensive analysis of common anomalies encountered in Ant Group, annotated with XPUTIMER's primary target.

\	Anomalies										
Type	Error					Slowdown					
Taxonomy	Algorithm bugs	Infrastructure bugs	OS errors	GPU errors	Network errors	New algorithms	Unnecessary synchronization	Un-optimized kernels	Memory management	GPU underlocking	Network jitter
Symptom	Startup crash or hang error		Runtime hang or crash error			Slowdown compared to historical jobs and prior training steps			Slowdown compared to prior training steps		
Team	Algorithm	Intrastructure	Operations			Algorithm		Intrastructure		Operations	
Attribution	Obvious		Obscure							Obvious	
Comment	No need to diagnose		**XPUTIMER's main target**								

to innovate LLM models [11, 18, 23–26, 31–36] tailored for diverse application scenarios.

Advancing of training cluster. Training these LLMs is computationally intensive, requiring large-scale GPU clusters of thousand-plus scale [26]. Therefore, leading corporations are continuously investing in large training clusters powered by state-of-the-art hardware [37, 38]. These clusters not only expand in scale but also incorporate the latest GPUs [39, 40] with higher computational performance and advanced interconnect [41, 42]. Efficient operation of these large-scale clusters is critical to the corporations, which requires the operations team to ensure uninterrupted training performance. Its responsibilities include job scheduling, driver updates, hardware maintenance, etc.

Advancing of training infrastructure. To enable easy, efficient, and scalable LLM training within large-scale clusters, the infrastructure teams build the dedicated software stack. This stack bridges the gap between algorithm teams and operations team. Specifically, the infrastructure team focuses on optimizing the software stack by integrating advanced operator libraries [4, 5, 43], training framework [7], and state-of-the-art model parallel backbones [8, 9, 13, 44]. By leveraging this highly optimized training infrastructure, we unlock the full potential of LLMs, enabling innovation and scalability across our various services.

2.2 Anomalies of Large-scale LLM Training

Due to the complexity of the entire training stack, various issues can easily occur. These issues affect both the training speed of individual training jobs and the overall utilization of training clusters, collectively termed as anomalies. Table 1 presents a distilled analysis of the common anomalies in our real-world cluster, broadly categorized into two primary types: errors and slowdowns. These anomalies often span responsibilities across multiple teams [2, 3, 14], making them notoriously difficult to diagnose. Thus, resolving these anomalies demands a diagnostic framework that can effectively identify their attributions. However, designing such a framework is far from trivial, posing three key challenges.

The pain of triggering condition. Anomalies in large-scale LLM training can arise at any stage and time, as shown in the fourth row of Table 1. First, low-level OS or hardware issues, often attributed to the operations team, may occur randomly, causing runtime hangs or slowdowns. For example, Figure 2 illustrates a significant MFU decline resulting from single-GPU underclocking. Second, training slowdowns caused by codebase modifications may also occur randomly. Specifically, codebase updates on the LLM algorithm typically cause slowdowns at the start, while modifications related to the training infrastructure can result in mid-training slowdowns. A notable example is the slowdown caused by on-demand garbage collection (GC) [2] of Python runtime, as detailed in §5.2.2. In such cases, a long-term background diagnostic mechanism is required to identify these anomalies. However, given the variety of anomalies across the stack, it is challenging to design a lightweight diagnostic framework capable of efficiently collecting key metrics.

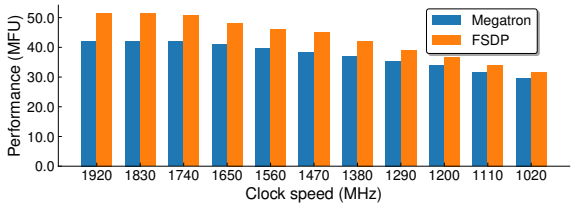

Figure 2: Slowdown of a 1024-GPU training job with Llama2-70B under various single-GPU underclocking configurations. Results are evaluated using both Megatron [9] and FSDP [8].

The pain of obscured attribution. The attribution of encountered anomalies is often obscured by similar symptoms. For example, many large-scale training jobs suffer from hang errors, in which the failure of a single training process results in all participating processes becoming unresponsive. These errors can arise from specific faulty machines due to computation operators, GPU communication operators, or issues within the operating system [2]. Besides, identifying the root causes of training slowdowns is inherently challenging. This is because the slowdowns cannot be identified using a single metric. Specifically, previous researches often rely on identifying computation operators with low FLOPS (floating point operations per second) for optimization [4, 43]. However, when matrix multiplication overlaps with communication operators, it is expected to naturally exhibit lower FLOPS. For example, in a model trained on an A100 GPU, the FLOPS of the same matrix multiplication kernel can de-

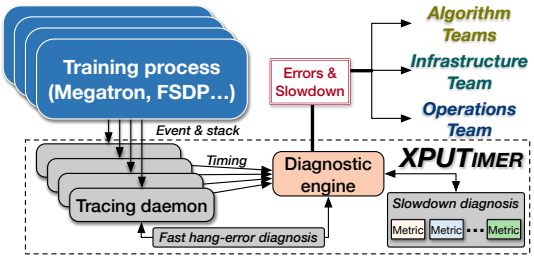

Figure 3: Architecture overview of XPUTIMER.

crease from 283.9TFLOPS to 155.5TFLOPS. Consequently, it is challenging for diagnostic frameworks to precisely attribute anomalies and assign them to the appropriate resolution teams.

The pain of backbone extensibility. With decades of advancements in model parallelism for large-scale training, a variety of divergent backbones have been developed to parallelize LLMs. Most LLMs are pre-trained using Megatron [9], a specialized parallel backbone designed for transformer-based LLMs. However, researchers continue to investigate alternative backbones [8, 10, 44] for fine-tuning pre-trained LLMs on downstream tasks or for training more complex large models, such as multimodal models. For instance, fully sharded data parallelism (FSDP) [8] and DeepSpeed [10] are widely utilized within our cluster for training multimodal LLMs. These alternatives are often selected to better align with specific training requirements. However, integrating the diagnostic framework into divergent parallel backbones remains challenging, as their runtime environments and programming interfaces vary significantly.

Existing works have made efforts to build a more robust training stack for large-scale LLM training [2, 3]. They generally operate under several assumptions: (1) training tasks rely on limited backbones like Megatron, (2) all teams work in close collaboration to pre-train a single LLM, or (3) The focus is on specific anomalies, such as communication-related issues. These assumptions fail to reflect reality and these works cannot solve the problems across the full stack. To this end, we propose XPUTIMER, a cluster-wide solution deployed in Ant Group's training cluster. It monitors the real-time progress of LLM training jobs and diagnoses various anomalies precisely.

3 XPUTIMER Design

In this paper, we propose XPUTIMER, a holistic and real-time diagnostic framework that identifies anomalies in large-scale training with low overhead and pinpoints their corresponding root causes. XPUTIMER is designed and implemented following three principles.

- XPUTIMER should be lightweight enough. In this case, it could run as a long-term diagnostic mechanism without compromising the training performance at any scale.
- XPUTIMER should cover enough runtime information. In this case, it could identify the attribution precisely with detailed analysis when encountering anomalies.
- XPUTIMER should be backbone-agnostic. In this case, it is sufficiently extensible to function as a cluster-wide solution for anomaly diagnostics, regardless of the backbones employed.

Architecture overview. Figure 3 illustrates the architecture of XPUTIMER, which is deployed in Ant Group's large-scale training cluster. XPUTIMER is composed of two components: the tracing daemon and the diagnostic engine.

By automatically attaching a tracing daemon to each training process in LLM training jobs, XPUTIMER enables a framework-agnostic and lightweight tracing mechanism. Specifically, the tracing daemon instruments only key code segments at the level of Python and C++ runtime, focusing on carefully selected Python APIs and GPU kernels. During instrumentation, events are injected to measure the latencies of these code segments. This approach draws on our experience hosting various large-scale LLM training jobs, targeting critical paths that commonly impact training efficiency.

Then, the timing data collected by the daemons is transmitted to the diagnostic engine for holistic anomaly diagnostics. The engine employs a fast hang-error diagnostic method and leverages novel aggregated metrics to effectively identify slowdowns. The aggregated metrics encompass both macro-level metrics, such as training throughput, and micro-level metrics, including issue latency distribution of GPU kernels. Once the diagnostic engine identifies errors and slowdowns, the issues are routed to attribution teams, enabling swift resolution.

4 Lightweight Selective Tracing

To collect sufficient real-time data for anomaly diagnosis in the LLM training cluster, XPUTIMER's tracing daemon offers backbone-agnostic and lightweight full-spectrum tracing. Its design focuses on two key aspects: determining what information to collect and establishing how to collect it efficiently.

4.1 Key Segment Instrumentation

Since profiling APIs like CUPTI [45] can operate in a background thread, the runtime data collection overhead primarily stems from high memory usage rather than interference with computing resources. For instance, profiling a Llama-70B model trained on 512 H800 GPUs using PyTorch's built-in profiler produces a log file of 5.5GB (in JSON format, compressed to 451MB) for each training step. This substantial memory overhead renders such arbitrary profiling methods impractical for continuously collecting real-time data to support anomaly diagnostics.

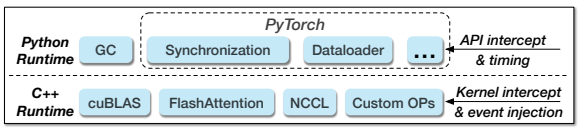

Figure 4: Instrumented key code segments in XPUTIMER.

Therefore, XPUTIMER selectively instruments code segments of key APIs and kernels to collect real-time information. This design is based on an insight into LLM training on large-scale GPUs: LLM training is predominantly dominated by a limited set of deep learning operators. These operators mainly include matrix multiplication and cross-GPU communication operators. Figure 4 presents the specific code segments instrumented by XPUTIMER for efficient anomaly diagnostics.

As shown in the figure, the instrumented code segments can be broadly categorized into two groups. The first category involves intercepting key API calls, including those related to Python's garbage collection (GC), PyTorch's dataloader, and GPU synchronization. These APIs are carefully selected based on empirical insights into performance issues and optimization opportunities. These insights are detailed in §5.2, with corresponding cases discussed in §7.

The second category focuses on intercepting critical GPU computation and communication kernels executed at the C++ runtime level. These kernels, primarily provided by optimized libraries [4–6, 46], account for the majority of the workload during large-scale training. Additionally, there are customized kernels developed by the infrastructure team.

Notably, the above design enables XPUTIMER to support backbone-agnostic and extensible tracing capabilities. Extending tracing capabilities for Python-related APIs is straightforward, requiring only the configuration of the specific environment variables in the training scripts, as shown below.

```
export TRACED_PYTHON_API="torch.cuda@synchronize"
```

Meanwhile, intercepting C++ kernels necessitates explicit registration through a C++ interface. This requirement is feasible, as the infrastructure team takes charge of the development of both these customized operators and XPUTIMER, ensuring seamless integration and functionality.

4.2 Timing in the Background

With intercepted Python APIs and GPU kernels, FLARE measures their elapsed latencies, as shown in Figure 5. Specifically, a dedicated tracing thread runs in the background to efficiently manage timing data. It employs different timing mechanisms for Python APIs and GPU kernels.

For synchronous Python API calls, FLARE directly records their start and end timestamps and forwards them to the timing manager. For GPU kernels, which execute asynchronously, FLARE injects CUDA events [47] after an interception to record execution status. These events are enqueued for further processing. The timing manager queries the status of the

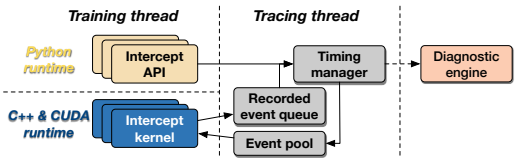

Figure 5: Intercepting and timing the training in the background.

queued events in the background, avoiding any disruption to the training thread. Additionally, during GPU kernel interception, FLARE extracts input specifications, such as memory layout, to support subsequent anomaly diagnostics in §5.2.1.

As training progresses, the timing manager proactively transmits all real-time data to XPUTIMER's diagnosis engine. By employing key segment instrumentation and running timing tasks in the background, XPUTIMER minimizes both computing resource and memory overhead, ensuring efficient data collection for real-time anomaly diagnostics. A detailed evaluation of XPUTIMER's real-time overhead is provided in §6.1.

5 Anomaly Detection and Diagnosis

Using the real-time data collected by the tracing daemon, XPUTIMER's diagnostic engine identifies and analyzes anomalies encountered during distributed LLM training. In this section, we present XPUTIMER's holistic diagnostic workflow for addressing two common anomaly symptoms: errors and slowdowns.

5.1 Fast Runtime Error Diagnosis

As shown in the left of Table 1, errors encountered at the beginning of a training job are typically caused by bugs in the training scripts, which can often be addressed easily by the algorithm teams and infrastructure team. However, diagnosing errors that occur during the training progress is more challenging and critical. Such errors often stem from issues like operating system crashes, GPU failures, or network disruptions, which can generally be resolved by isolating the problematic machines and restarting the training job.

A typical symptom associated with these errors is the hanging of the training job. Training LLMs across numerous GPUs in a distributed manner inherently relies on the coordination of training processes. When the aforementioned errors occur, they rarely affect all training processes simultaneously. In this context, XPUTIMER focuses on rapidly diagnosing hang errors by identifying faulty machines. Then, XPUTIMER routes this information to the operations team, enabling the training job to restart with healthy machines.

Specifically, XPUTIMER's diagnostic engine first detects hang errors by examining the status of tracing daemons. The tracing daemon operates in the background of the training

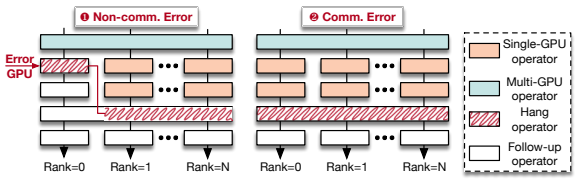

Figure 6: Diagnosing hang errors via call stack analysis.

thread and continuously queries events recorded during job execution. If it fails to confirm the completion of an event within a predefined timeout interval, it proactively reports a potential hang error to the diagnostic engine. Similarly, if a tracing daemon does not transmit any real-time data within the specified timeout interval, the diagnostic engine also interprets this as an indication of a hang error.

After hang errors are reported, they are classified as either communication or non-communication errors. XPUTIMER diagnoses these errors in two steps: first, a coarse-grained diagnosis through call stack analysis; and second, a fine-grained diagnosis using intra-kernel tracing

Diagnosis using call stack analysis. This diagnosis is used to identify problematic machines encountering non-communication errors. Figure 6 illustrates an example of hang-error diagnosis via call stack analysis. As shown in the left of Figure 6, when the training process of rank-0 crashes or is suspended due to a non-communication error, it halts at a call stack corresponding to a non-communication function. In contrast, the training processes of other ranks continue executing correctly and eventually stop at a call stack associated with a communication-related function that depends on coordination with rank-0. In this scenario, the machine associated with rank-0 is identified as the source of the error. It should be noted that, although these non-communication errors may cause direct crashes, the call stack analysis could also locate the faulty machine.

However, communication hang errors cannot be identified through call stack analysis. As shown in the right of Figure 6, the training processes of all ranks terminate at the same call stack corresponding to a communication function, such as allreduce or allgather. In this scenario, there are no distinct differences between ranks based on call stack analysis.

We further investigate the symptoms of communication hang errors and obtain two observations. Firstly, some communication hang errors generate error logs. For instance, if the link between RDMA NICs breaks, an error code of `12` is produced. Secondly, more hang errors result in an endless loop within the launched communication kernels, ultimately leading to job termination after a predefined timeout. To identify the unhealthy machine responsible for such errors, a straightforward solution is to perform a binary search by executing communication tests across all involved GPUs. This approach has a complexity of $O(\log N)$ and requires hours to pinpoint the faulty machine among thousands of GPUs [14].

Diagnosis using intra-kernel tracing. Faced with this problem, XPUTIMER introduces a minute-level diagnostic approach using intra-kernel tracing. This intra-kernel tracing leverages CUDA-GDB, the debugging tool for CUDA programming.

Specifically, XPUTIMER's diagnostic engine instructs the tracing daemon to attach the halted training processes with CUDA-GDB before terminating them. Once attached, the tracing daemon executes a script capable of automatically extracting detailed communication statuses to identify unhealthy machines. Figure 7 depicts an example of diagnosing communication hang errors in a hanging ring-allreduce kernel.

In the ring-allreduce kernel, each thread block is responsible for transmitting data between linked adjacent ranks within the kernel's constructed ring. The data are split into chunks and thread blocks of adjacent ranks work together to transmit the chunks step by step. Thus, XPUTIMER could retrieve the register values corresponding to the loop steps used for data transmission between linked ranks. Theoretically, the connection with the minimum step reveals the related GPUs experiencing errors. This intra-kernel tracing process is performed in parallel across all involved GPUs. As a result, its complexity is $O(1)$, enabling completion within a few minutes.

XPUTIMER then routes the diagnostic information for detected errors to the operations team, assisting with tasks such as isolating faulty machines and restarting the training job.

5.2 Aggregation for Slowdown Diagnosis

As demonstrated in §2.2, slowdowns can be attributed to changes across the entire training stack. Meantime, slowdowns caused by software changes introduced by the algorithm and infrastructure teams are often subtle and challenging to detect. Identifying these changes typically requires comparisons across historical training jobs and prior training steps. In contrast, hardware changes, such as GPU underclocking or network jitter, are more apparent and can be detected solely through comparisons across training steps.

To holistically identify these anomalies, XPUTIMER aggregates real-time data collected from the tracing daemon into five primary metrics, shown in Figure 8. *These metrics are based on the consensus that a "healthy" training pipeline should exhibit a timeline saturated with GPU kernels dedicated to either computation or communication.* Computation kernels should achieve high FLOPS, while communication kernels are expected to utilize high bandwidth. Any deviations from these characteristics point to idle GPU resources, signaling potential slowdowns in training jobs. Of the five metrics, three are commonly used in existing works [2, 14], while the other two are newly introduced by XPUTIMER.

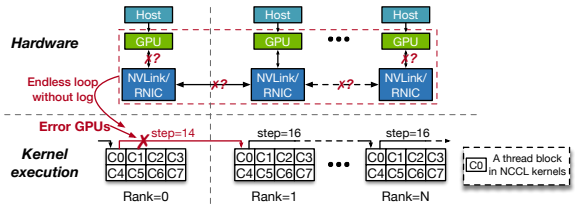

Figure 7: An example of diagnosing communication hang errors in a ring-allreduce kernel using intra-kernel tracing.

5.2.1 Diagnosing Obvious Slowdown

❶ **Training throughput for detecting slowdown.** Training throughput is the most straightforward metric for detecting slowdowns. XPUTIMER measures training throughput by timing the rate at which input data is consumed by the training pipeline. This is achieved by instrumenting the dataloader API of Pytorch. As a macro performance metric, training throughput directly reflects slowdowns in training efficiency through comparison to historical training jobs and between training steps of the same job. However, XPUTIMER cannot diagnose the specific factors contributing to the slowdown. To address this, XPUTIMER relies on the following four micro metrics to further investigate the underlying causes.

❷ **FLOPS for slow critical kernel.** XPUTIMER monitors the FLOPS of instrumented critical computation kernels, leveraging timing data and input layout. By comparing the FLOPS of identical kernels across different ranks, XPUTIMER diagnoses GPUs that exhibit poor computational performance, often caused by issues like GPU underclocking. Machines affected by GPU underclocking are then routed to the operations team for isolation. Additionally, by analyzing FLOPS, XPUTIMER identifies un-optimized kernels in training jobs, particularly those with large input sizes but low FLOPS. These anomalies are detected without interrupting training jobs and are subsequently routed to the infrastructure team for further investigation. Notably, when analyzing FLOPS data, XPUTIMER accounts for the impact of communication kernels that overlap with computation kernels.This ensures that computation kernels with falsely low FLOPS are not mistakenly flagged.

❸ **Bandwidth for slow connection.** XPUTIMER monitors the bandwidth of communication kernels. A communication operator requires launching the communication kernels on all ranks. Since variations in kernel-issue timestamps exist across different ranks, XPUTIMER calculates the communication bandwidth by utilizing the start and end timestamps of the final communication kernels issued across all participating ranks. The captured communication bandwidth is compared with offline profiled data. If low-bandwidth communication is detected, XPUTIMER conducts a communication test using binary search to pinpoint machines experiencing issues such as network congestion. These slowdowns are then identified and routed to the operations team for resolution.

5.2.2 Diagnosing Obscured Slowdown

While the above three metrics ensure that both critical computation and communication GPU kernels operate at high performance, they do not cover the less critical operations, such as various CPU operations and element-wise activation GPU kernels. Meantime, XPUTIMER's selective key segment instrumentation also omits the monitoring of these operations.

To diagnose their potential contributions to slowdowns, we further classify these not-instrumented operations into three categories: intra-step CPU operations, inter-step CPU operations, and minority GPU kernels. Intra-step CPU operations and inter-step CPU operations differ due to their occurrences within the timeline of training steps. Minority GPU kernels refer to those GPU kernels that often occupy little GPU computation resources. Specifically, two metrics are introduced for the diagnostics: issue latency distribution for intra-step CPU operations and void percentage for inter-step CPU operations and minority GPU kernels.

❹ **Issue latency distribution for kernel-issue stall.** In a well-optimized parallel backbone, only the necessary intra-step CPU operations for launching GPU kernels or coordinating the training processes are expected. However, algorithm teams may inadvertently introduce unnecessary GPU synchronizations when modifying the LLM model. Meantime, certain function calls, such as GC [2, 9], may be implicitly triggered by the Python runtime. These intra-step CPU operations can occur repeatedly during the model's forward pass, bringing considerable overhead. In such cases, these operations cause an anomaly known as a kernel-issue stall, leading to GPU idle time within the training step.

❹–1 in Figure 8 shows the example of Python runtime GC. In the figure, the Python runtime GC stalls the CPU thread and causes the lagging of GPU kernels on rank-1. Although the communication kernel on rank-0 is issued without stalling, it simply waits for the one on rank-1, ultimately causing the overall training speed to decline. ❹–2 in Figure 8 shows an example of unnecessary GPU synchronization introduced by the developers from the algorithm teams. As all ranks wait for the completion of communication kernels, the kernel issue of follow-up kernels is stalled and not overlapped with GPU computation. When such unnecessary synchronization occurs repeatedly across the model's forward pass, it ultimately results in a slowdown of the training speed.

Originally, detecting these anomalies of kernel-issue stall requires investigating the aggregated timeline with much human effort. Faced with this issue, XPUTIMER proposes a new metric, named issue latency distribution, for diagnosing this issue without human intervention. Kernel-issue latency is defined as the time elapsed between the kernel's issue timestamp and the start timestamp of its execution on the GPU. Based on our observation of anomalies of kernel-issue stall, the kernel-issue latencies of unhealthy training jobs should be much shorter than those of a healthy training job.

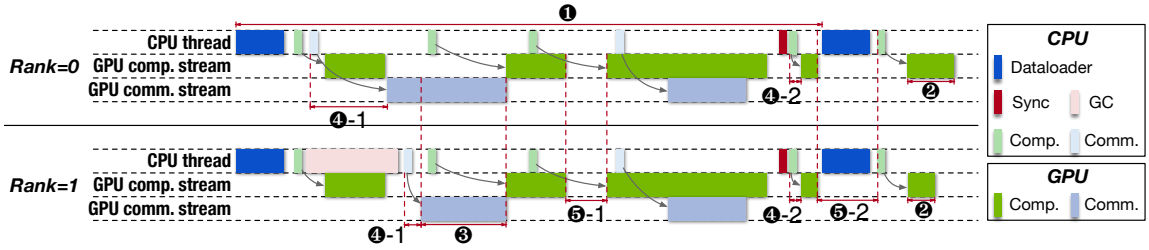

Figure 8: A timeline of a distributed training job annotated with aggregated metrics used for diagnosing slowdowns in XPUTIMER.

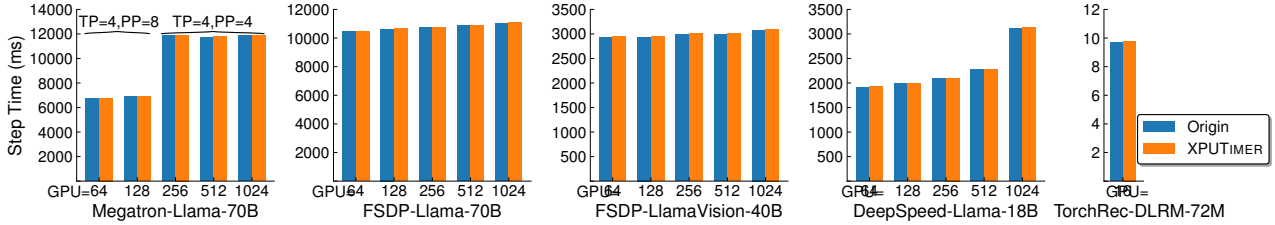

Figure 9: Runtime overhead in terms of latency with various models, backbones, and number of GPUs.

By monitoring runtime issue latency distribution, and comparing it with the historical data, XPUTIMER could identify the anomaly of kernel-issue stall. Then, XPUTIMER routes them to algorithm and infrastructure teams for resolution, as they are commonly software issues. §6.3 and §7.2 demonstrate the effectiveness of issue latency distribution in diagnosing kernel-issue stalls, even when the slowdown is minimal.

❺ **Void percentage for other un-covered operations.** While the tracing daemon only instruments the critical operators, inter-step CPU operations and minority GPU kernels both manifest as empty time slots in the visualized timeline, as shown in Figure 8. Consequently, XPUTIMER introduces a metric, termed the void percentage, to identify slowdowns caused by these factors.

As for inter-step CPU operations, as depicted by ❺–2 in Figure 8, XPUTIMER measures the latency between the last kernel preceding the dataloader and the first kernel following the same dataloader. XPUTIMER then computes the void percentage for inter-step CPU operations using the following equation:

$$V_{inter} = T_{inter} \;/\; T_{step} \tag{1}$$

where T_{inter} represents the latency associated with inter-step CPU operations, and T_{step} denotes the total latency of the training step.

As for minority GPU kernels, as shown by ❺–1 in Figure 8, XPUTIMER first automatically detects empty slots where GPU kernels are launched but remain un-executed. These empty slots signify that the GPUs are occupied by kernels outside the scope of XPUTIMER's tracing mechanism. XPUTIMER subsequently accumulates these slots for each training step and computes the void percentage using the following equation:

$$V_{minority} = T_{minority} \;/\; (T_{step} - T_{inter}) \tag{2}$$

where $T_{minority}$ is the latency of all minority GPU kernels.

When the void percentages (V_{inter} and $V_{minority}$) surpass the predefined thresholds for a specific parallel backbone, XPUTIMER annotates the training job with potential slowdowns attributed to inter-step CPU operations or minority GPU kernels. Then, XPUTIMER notifies the algorithm and infrastructure team for further investigation.

6 Evaluation

In this section, we present experiments to demonstrate XPUTIMER's effectiveness from various perspectives. Specifically, we first evaluate XPUTIMER's runtime overhead in terms of latency and memory, showcasing its lightweight nature for runtime anomaly diagnosis. Then, we assess the effectiveness of XPUTIMER's novel diagnostic mechanisms: intra-kernel tracing for communication hang error diagnosis and issue distribution metric for kernel-issue stall diagnosis.

6.1 Runtime Overhead

We evaluate XPUTIMER on four parallel backbones: Megatron [9], FSDP [8], DeepSpeed [10], and TorchRec [13]. Among these, Megatron, FSDP, and DeepSpeed are widely used for LLM training, while TorchRec is employed for training large recommendation models within Ant Group. Four models are benchmarked, spanning language, vision, and recommendation tasks: two large language models (Llama 18B and 70B), one large vision model (Llama Vision 40B), and one recommendation model (DLRM 72M).

The latency overhead experiment is conducted on 1,024 H800 GPUs deployed across 128 servers with RoCE connectivity. Figure 9 presents the latency overhead introduced by XPUTIMER with various models, backbones, and number of GPUs. As shown, XPUTIMER incurs a latency overhead

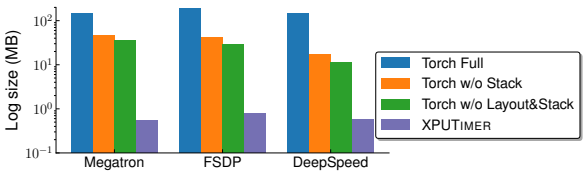

Figure 10: Memory consumption of dumped logs per GPU per step using the PyTorch profiler and XPUTIMER while training a Llama-70B model on 16 A100 GPUs.

of 0.43% for three LLM training backbones and 1.02% for TorchRec.

The memory overhead experiment is conducted on two setups, which are 16 A100 GPUs on 2 nodes and 1536 H800 GPUs on 192 nodes. We compare XPUTIMER with `Torch Full`, `Torch w/o Stack`, `Torch w/o Layout&Stack`, and XPUTIMER. `Torch w/o Stack` refers to using the PyTorch builtin profiler with stack tracing disabled, while `Torch w/o Layout&Stack` further disables matrix layout tracing.

Figure 10 shows the memory overhead results on 16 A100 GPUs. XPUTIMER consumes only 0.39%, 1.76%, and 2.48% of memory overhead for the respective configurations of PyTorch profiler. Specifically, XPUTIMER generates a maximum of 0.78MB of tracing logs per GPU. Besides, in a real-world Llama-20B training job on 1536 H800 GPUs, XPUTIMER generated only a 1.5MB tracing log per GPU. TorchRec is omitted from this experiment, as monitoring a recommendation model generates minimal logs.

From the above results, we can conclude that, regardless of the GPU scale, parallel backbone, parallel strategy, or model type used, XPUTIMER consistently maintains an extremely low runtime overhead in terms of both latency and memory. The lightweight selective tracing facilitates XPUTIMER's deployment within our large training cluster, serving as a diagnostic framework for diverse training jobs.

6.2 Effectiveness of Intra-kernel Tracing

We evaluate the intra-kernel tracing mechanism on 16 A100 GPUs across two servers with RoCE connectivity. Given that most communication kernels are ring-based, this experiment focuses on evaluating ring-allreduce. Specifically, we customize the training script composed solely of communication kernels, with one GPU intentionally suspended to simulate a hang error caused by communication issues.

Figure 11 illustrates the pinpointing latencies for intra- and inter-server communication. The figure presents the latency results for three communication protocols [48] and cross-node configurations. As shown, XPUTIMER requires 29.4~309.2s to detect erroneous GPUs across different scenarios. Among the protocols, XPUTIMER performs best when the SIMPLE protocol is used for communication. This is because, with the SIMPLE protocol, XPUTIMER only needs to scan the first thread of each thread block to check the steps, whereas the

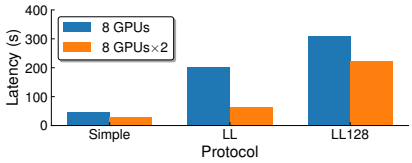

Figure 11: Latency for pinpointing the erroneous GPUs causing a hang error in ring-allreduce with different protocols.

other two protocols require scanning the entire thread block.

When comparing intra-server and inter-server results, XPUTIMER performs better when the ring-allreduce operation spans multiple servers. This is because intra-server GPUs are connected via NVLink, whereas inter-server GPUs communicate through NICs. Communication kernels launched over NICs involve fewer thread blocks, as NICs have fewer internal links compared to NVLink. As a result, XPUTIMER scans fewer thread blocks for error diagnosis in inter-server scenarios.

In summary, the intra-kernel tracing mechanism can detect erroneous GPUs in a maximum of 309.2s. Notably, as the complexity of intra-kernel tracing is $O(1)$, these results remain consistent regardless of scale.

6.3 Effectiveness of Issue Latency Distribution

In this experiment, we evaluate the issue latency distribution using Llama-20B running on 256 H800 GPUs across 32 servers connected via RoCE. Figure 12 illustrates the issue latency distribution for all communication kernels in the `Unhealthy-GC`, `Unhealthy-Sync`, and `Healthy` scenarios. In the `Unhealthy-GC` scenario, GC is implicitly triggered by the Python runtime. In the `Unhealthy-Sync` scenario, an unintended GPU synchronization call is added within the transformer block, leading to repetitive GPU synchronizations during the model's forward pass. In the `Healthy` scenario, GC is efficiently managed by the parallel backbone, and no unnecessary synchronizations are introduced.

As shown in the figure, the issue latency distribution patterns align with our claim in §5.2.2. The issue latency CDF of a healthy LLM training job increases linearly, whereas the issue latency CDFs for `Unhealthy-GC` and `Unhealthy-Sync` exhibit a much steeper rise. This is because the issue latencies of different ranks in the healthy scenario are solely influenced by the collective communication operator, resulting in a uniform distribution. In contrast, in the cases of `Unhealthy-GC` and `Unhealthy-Sync`, while some ranks are affected, their latencies become very short due to the delayed start of the issue time.

Since both GC and GPU synchronizations span the entire model forward pass, all communication kernels are affected, as illustrated in Figure 12. Furthermore, each training process triggers GC independently, and the GC operation for a single process is more time-consuming than GPU synchronization. Consequently, the issue latency distribution for

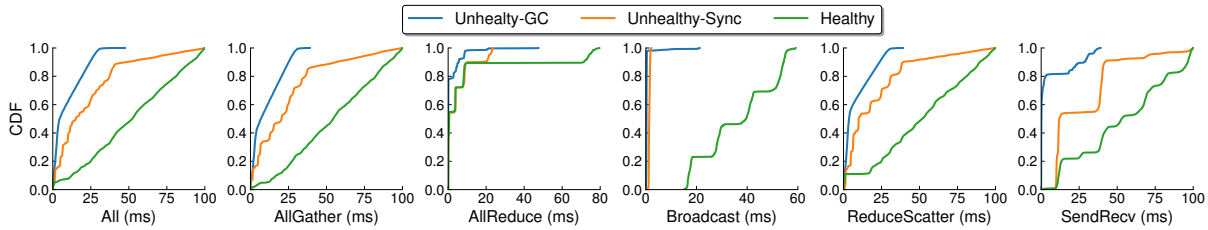

Figure 12: Issue distribution across all communication kernels for a Llama-20B model trained with Megatron and 256 GPUs, including the overall CDF and the CDFs for each type of communication kernel, respectively.

Table 2: Typical errors detected by XPUTIMER.

Taxonomy	Details	Numbers	Mechanism
OS errors	Checkpoint storage	10	Stack analysis
	OS crash	1	
GPU errors	GPU Driver	26	
	Faulty GPU (Unknown)	37	
Network errors	NCCL hang	36	Intra-kernel tracing
	RoCE issue	17	

`Unhealthy-GC` is worse than that of `Unhealthy-Sync`.

7 Deployment & Case Studies

In this section, we further demonstrate XPUTIMER's effectiveness in diagnosing LLM training anomalies through statistical analysis of its cluster-wide deployment and detailed case studies of various anomalies.

7.1 Cluster-wide Deployment

XPUTIMER has been deployed within a training cluster with 6,000 GPUs for over 8 months. During this period, it is responsible for monitoring, detecting, and diagnosing training jobs for various deep learning models, especially large-scale distributed LLM training. Its diagnostic capabilities enable the algorithm teams, infrastructure team, and operations team to seamlessly enhance model training efficiency and improve training cluster utilization.

Table 2 presents a subset of error anomalies detected by XPUTIMER. As shown in the table, XPUTIMER effectively identifies OS- and hardware-related anomalies, including crashes, hangs, and slowdowns caused by such issues. While these issues are typically conspicuous and could be detected by existing methods based on noticeable training interruptions [9, 14], XPUTIMER's novelty lies in providing richer error information through techniques like intra-kernel tracing. Such runtime information helps ease and accelerate the attribution process of low-level issues for operations teams.

Moreover, the true value of XPUTIMER lies in its ability to detect anomalies caused by both algorithm teams and the infrastructure team. Table 3 summarizes slowdowns diagnosed by XPUTIMER using its aggregated metrics. In the table, anomalies newly identified by XPUTIMER, as compared to prior works [2], are highlighted in bold. The following subsections provide a detailed discussion of typical cases.

Table 3: Slowdowns diagnosed by XPUTIMER, with "**Details**" showing training job specifics and associated MFU decline.

Metric	Attribution	Details
FLOPS	GPU underclocking	480 GPUs, Llama-65B, 14% ↓
	Backbone migration	**1856 GPUs, Llama-80B,** 33.3% ↓
Bandwidth	Network jitter with increased CRC	928GPUs, Llama-65B, 10~20% ↓
	Down of GDR module	32GPUs, Llama-10B, 80% ↓ 128GPUs, Llama-10B, 62.5% ↓ ...
	Host-side hugepage caused high sysload	**128GPUs, LlamaVision-11B,** 20% ↓
Issue latency distribution	Python GC	2048GPUs, Llama-80B, 10% ↓ 280GPUs, LlamaVision-11B, 60% ↓ ...
	Unnecessary GPU Sync	**256GPUs, Llama-20B,** 2.66% ↓ **...**
	Package chcecking	**280GPUs, LlamaVision-20B,** 30% ↓
Void percentage	**Frequent GPU mem. management**	**1344GPUs, Llama-176B,** 19% ↓
	Dataloader	**512GPUs, Llama-80B,** 41% ↓

7.2 Case-1: Towards Stall-free Kernel Issuing

Kernel-issue stalls are among the most frequent causes of slowdowns encountered in a training cluster not dedicated exclusively to a single pre-training task. While Python runtime GC is well-known and now carefully managed by the parallel backbone in most cases, most encountered kernel-issue stalls arise from code introduced by algorithm teams to enhance LLM performance in downstream tasks.

A typical case encountered by XPUTIMER is a training job of Llama20B running on 256 H800 GPUs. The developer from the algorithm team mistakenly enables the timer provided by Megatron for performance profiling of several key code segments. This profiling incurs kernel-issue stalls because it requires GPU synchronizations to obtain accurate timestamps.

Although no significant slowdown was observed in training throughput, XPUTIMER successfully detects the abnormal issue latency distribution. After removing these unnecessary synchronizations by disabling the timer, the MFU of the training job improves from 41.4% to 42.5%, representing a 2.66% increase. While this slowdown is much smaller than that

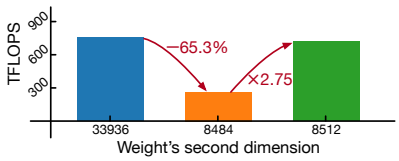

Figure 13: The change in computation TFLOPS when migrating Llama-80B from FSDP to Megatron.

caused by Python runtime GC (typically exceeding 10%), XPUTIMER could still uncover it, ensuring the training job's performance.

In addition to the above two cases, XPUTIMER also detects other kernel-issue stalls, such as unnecessary package version checking, frequent CUDA memory management within the PyTorch runtime, and others. These cases are listed in Table 3.

7.3 Case-2: Migration between Backbones

Different parallel backbones are suited to varying hardware conditions. While FSDP typically delivers ease of use and good efficiency on fewer than 1000 GPUs or relatively short sequences for LLMs (e.g., 4k), Megatron demonstrates superior scalability as the scale increases beyond 1000 GPUs or when handling extremely large sequences (e.g., 64k). In this context, algorithm teams may migrate an LLM between backbones to meet their specific demands. However, this migration process can potentially introduce anomalies.

A typical scenario encountered by XPUTIMER is an anomalous MFU decline when migrating a Llama-like 80B model from FSDP (1888 H800 GPUs) to Megatron (1586 H800 GPUs with a data-parallel degree of 58, pipeline-parallel degree of 8, and tensor-parallel degree of 4). This anomaly specifically stems from a matrix layout change. The weight dimension of the LLM's FFN layer, initially configured as $[8192 \times 33936]$ during training on FSDP, changes to $[8192 \times 8484]$ after migration to Megatron with a tensor parallelism degree of 4.

After migration to Megatron, this operator exhibits significantly lower FLOPS due to smaller batch size and the unfavorable 8484 layout for Tensor Cores, which require alignment to 128 bytes. In contrast, the dimension 33936 and larger batch size on FSDP meet this alignment requirement. Following XPUTIMER's diagnosis, our infrastructure team customizes a kernel that pads 8484 to 8512. Figure 13 further illustrates the FLOPS of the same operator before migration, after migration, and post-optimization guided by XPUTIMER. As shown, the operator experiences a 65.3% decline in FLOPS after migration, and XPUTIMER successfully facilitates the performance diagnosis. From the perspective of the training job, the overall MFU increases from 27% to 36%, reflecting a 33.3% improvement.

Table 4: Changes in detected $V_{minority}$ and corresponding TFLOPS when different minority kernels are not optimized.

	Healthy	-PE	-PE-ACT	-PE-ACT-NORM
$\mathbf{V_{minority}}$	9%	14%	15%	28%
N. TFLOPS	1	0.95	0.93	0.83

7.4 Case-3: New Algorithms and Data

Algorithm teams continually strive to enhance model performance by modifying the LLM architecture and incorporating new training data. However, this process often introduces anomalies.

Firstly, algorithm teams generally modify position embeddings (PE), activation functions (ACT), and normalization operators (NORM), while preserving the core structure of the transformer. Table 4 illustrates the detected changes in $V_{minority}$ caused by these operator modifications during daily deployments. In this table, the parallel backbone is Megatron. The "Healthy" column represents a fully optimized training job, whereas the "-PE" column reflects changes in modifying the position embeddings. Similarly, the other columns correspond to modifications of the respective operators.

Since these modified operators are less critical and not instrumented by XPUTIMER, $V_{minority}$ increases proportionally with their computational complexity. Our infrastructure team leverages XPUTIMER's detection of high $V_{minority}$ to develop targeted kernel implementations. Once optimized through techniques like kernel fusion, the job's $V_{minority}$ returns to a normal level. In this process, XPUTIMER eliminates the need for manual identification, thereby accelerating anomaly diagnosis.

Secondly, newly filtered data is continuously incorporated into the LLM for training. XPUTIMER has successfully diagnosed anomalies arising from variance in the training data. In one specific case, the algorithm team attempts to train a Llama-80B model with data containing a sequence length of 64k, while the original training script is designed for sequence lengths of 4k. XPUTIMER identifies a significant anomalous decline in MFU (41%) on 512 H800 GPUs, accompanied by an increase in V_{inter}.

After routing this anomaly to the infrastructure team, the root cause was identified in the dataloader, specifically in the attention mask generation process. When the sequence length is short, the latency incurred by mask generation is minimal. However, the complexity of mask generation scales as $O(L^2)$, where L represents the sequence length. As a result, the dataloader experiences extremely poor performance when the sequence length increases to 64k.

8 Lessons Learned and Future Work

In this section, we share the practical lessons learned during the development and deployment of XPUTIMER, along with directions for future work.

8.1 Practical Usages

Algorithm teams. Algorithm teams are dedicated to continuously integrating new innovations into the existing LLM training pipeline. They employ XPUTIMER to verify that submitted training jobs meet the expected training throughput. By leveraging XPUTIMER, algorithm teams can identify and resolve slowdown anomalies caused by elementary inefficient code, without requiring intervention from the infrastructure team.

Infrastructure team. The infrastructure team is committed to optimizing GPU kernel libraries, training frameworks, and parallel backbones. With XPUTIMER's lightweight logging system, the infrastructure team can gather sufficient runtime data to analyze submitted jobs and identify new optimization opportunities. Anomalies caused by codebase modifications that cannot be resolved by the algorithm team are ultimately routed to the infrastructure team.

Operations team. The operations team is responsible for maintaining the stability of all low-level resources. When a training job is terminated due to low-level issues, the operations team queries intra-kernel tracing information provided by XPUTIMER. This allows them to identify the faulty machine without requiring a fully comprehensive test.

8.2 Holistic Diagnostics

The training stack for advancing AIGC is inherently complex. Previously, in the absence of a holistic diagnostic framework like XPUTIMER, significant effort was wasted on communication among algorithm, infrastructure, and operations teams. Algorithm teams often required collaboration with the infrastructure team to address slowdowns caused by minor but inefficient codebase modifications. When low-level issues arose, the operations team had to conduct extensive low-level benchmarking tests to identify the problem, resulting in delays to the algorithm teams' training jobs. The holistic diagnostics provided by XPUTIMER streamline this process by pinpointing elementary inefficiencies in the codebase for algorithm teams, only routing complex anomalies with aggregated metrics to the infrastructure team, and supplying the operations team with valuable runtime low-level data.

8.3 Using Historical Data

Historical traces are essential for enhancing the effectiveness of anomaly detection. While runtime data is crucial, it is insufficient on its own for accurately identifying anomalies. XPUTIMER detects various issues by comparing real-time data against historical data. For instance, when using issue latency distributions to detect kernel-issue stalls, XPUTIMER relies on historical data from specific backbones operating on specific hardware. By collecting historical data from healthy jobs, XPUTIMER can efficiently identify anomalies in newly submitted LLM training jobs. In the future, as deployment data grows, we aim to release relevant datasets to further streamline the diagnosis of LLM training.

8.4 Hardware Extensibility

Currently, XPUTIMER does support the extensibility of additional hardware, particularly NPUs dedicated to DL training. Its tracing mechanism is not only backbone-agnostic but also designed for seamless extension to other NPUs. Since XPUTIMER directly instruments key code segments at the Python and C++ runtime levels, extending it is straightforward, provided the relevant computation kernel is supplied by the NPU vendor.

8.5 Diagnosing Communication Slowdown

Diagnosing communication slowdowns caused by issues such as network jitter is challenging, as runtime tracing can introduce significant overhead or lack accuracy. Currently, XPUTIMER utilizes a method similar to MegaScale [2], leveraging NCCL tests to identify faulty machines or switches. We are in the process of developing an eBPF-based [49] tracing tool to accurately monitor bandwidth across RDMA NICs with minimal overhead. In the future, XPUTIMER will be further enhanced with these fine-grained tracing capabilities.

9 Related Work

Anomaly Diagnosis in Distributed Training. Anomaly diagnosis in large-scale distributed deep learning training tasks has consistently been a hot research focus [2, 3, 14, 50–52]. Megascale [2] only focuses on LLM training tasks based on Megatron-LM. It identifies network-related hardware and software issues in the training process by conducting intra-host network tests and NCCL tests. Falcon [14] detects prolonged iterations using the Bayesian Online Change-Point Detection algorithm. C4D [3] modifies the Collective Communication Library to collect message statistics, such as sizes and durations of transfers, to identify the performance bottlenecks. However, these approaches primarily address communication-related performance issues and fail to encompass the anomalies across the LLM training stack. Additionally, they depend heavily on a single parallel backbone, Megatron, which limits their generality. Meanwhile, many researches [50–52] focus on task recovery. These approaches are orthogonal to XPUTIMER, which could seamlessly integrate with XPUTIMER.

Anomaly Diagnosis in Large-scale Datacenter. Many research efforts [53–57] focus on anomaly diagnosis at different levels of the datacenter, including runtime, network, and storage. SCALENE [53] introduces an algorithm to assist Python programmers in optimizing their code by distinguishing between inefficient Python execution and efficient native

execution. AND [54] is a unified application-network diagnosing system that leverages a single metric, TCP retransmissions (TCP retx), to identify network anomalies in cloud-native scenarios. AAsclepius [55] proposes a PathDebugging technique to trace fault linkages between the middle network and autonomous systems. Researchers [56] from Alibaba analyze four key factors that impact SSD failure correlations: drive models, lithography, age, and capacity. As these works address anomaly problems in specific scenarios, they are unable to resolve the challenges targeted by XPUTIMER in large-scale distributed LLM training.

10 Conclusion

In this paper, we introduce XPUTIMER, a real-time diagnostic framework for LLM training. By addressing critical challenges such as lightweight long-term monitoring, root cause detection, and backbone extensibility, XPUTIMER provides a comprehensive solution for anomaly diagnostics across large-scale GPU clusters. With its novel intra-kernel tracing mechanism and holistic aggregated metrics, XPUTIMER not only reduces diagnostic complexity but also improves the detection and resolution of non-obvious slowdowns and errors. Its deployment in real-world training clusters, spanning over 6,000 GPUs, demonstrates its efficacy in diagnosing distributed LLM training. Furthermore, XPUTIMER is open-sourced to encourage broader adoption.

References

[1] Ant group. https://www.antgroup.com/en. Accessed: 2025-01-13.

[2] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, and et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. (arXiv:2402.15627), February 2024.

[3] Jianbo Dong, Bin Luo, Jun Zhang, Pengcheng Zhang, Fei Feng, Yikai Zhu, Ang Liu, Zian Chen, Yi Shi, Hairong Jiao, Gang Lu, Yu Guan, Ennan Zhai, Wencong Xiao, Hanyu Zhao, and et al. Boosting Large-scale Parallel Training Efficiency with C4: A Communication-Driven Approach. (arXiv:2406.04594), June 2024.

[4] NVIDIA Corporation. cublas library user guide. https://docs.nvidia.com/cuda/archive/12.6.2/cublas/, 2024.

[5] NVIDIA Corporation. Nccl: Nvidia collective communication library. https://developer.nvidia.com/nccl, 2024.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. (arXiv:2205.14135), June 2022.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, and et al. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 721, pages 8026–8037. Curran Associates Inc., Red Hook, NY, USA, December 2019.

[8] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, and et al. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.

[9] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. (arXiv:1909.08053), March 2020.

[10] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, Atlanta, GA, USA, November 2020. IEEE.

[11] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. (arXiv:2409.19256), October 2024.

[12] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. (arXiv:2403.14608), April 2024.

[13] Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. TorchRec: A PyTorch domain library for recommendation systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, pages 482–483, New York, NY, USA, September 2022. Association for Computing Machinery.

[14] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang,

Lin Qu, and Liping Zhang. FALCON: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training. (arXiv:2410.12588), October 2024.

[15] Qinlong Wang et al. Dlrover: An automatic distributed deep learning system. https://github.com/intelligent-machine-learning/dlrover, 2023. Accessed: 2025-01-13.

[16] Lf ai & data foundation. https://lfaidata.foundation/. Accessed: 2025-01-13.

[17] OpenAI. Gpt-4 is openai's most advanced system, producing safer and more useful responses. https://openai.com/index/gpt-4/, 2024. Accessed: 2024-10-23.

[18] Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y Wu, et al. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066*, 2024.

[19] Anthropic. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family, 2024. Accessed: 2024-10-23.

[20] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[21] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[22] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. (arXiv:2302.13971), February 2023.

[24] Hugo Touvron, Louis Martin, and Kevin Stone. Llama 2: Open foundation and fine-tuned chat models.

[25] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, and et al. The llama 3 herd of models. (arXiv:2407.21783), August 2024.

[26] Hugging Face. Llama 3.1: A new milestone for open large language models. https://huggingface.co/blog/llama31, 2024. Accessed: 2024-10-20.

[27] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[28] OpenAI. Creating video from text. https://openai.com/index/sora/, 2024. Accessed: 2024-10-20.

[29] OpenAI. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/, 2024. Accessed: 2024-10-20.

[30] Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*, 2024.

[31] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with condi- tional computation and automatic sharding. 2021.

[32] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[33] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[34] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[35] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity.

[36] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. RLHFuse:

Efficient RLHF training for large language models with inter- and intra-stage fusion. (arXiv:2409.13221), September 2024.

[37] xAI. Colossus training cluster. https://www.techradar.com/pro/xai-cluster-is-now-the-most-powerful-ai-\training-system-in-the-world-but-questions\-remain-over-storage-capacity-power-usage\-and-why-it-s-actually-called-colossus, 2024.

[38] Meta Engineering Team. Maintaining large-scale ai capacity at meta. https://engineering.fb.com/2024/06/12/production-engineering/maintaining-large-scale-ai-capacity-meta, 2024.

[39] NVIDIA Corporation. Nvidia ampere architecture. https://www.nvidia.com/en-us/data-center/ampere-architecture/, 2020.

[40] NVIDIA Corporation. Nvidia hopper architecture. https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/, 2022.

[41] NVIDIA Corporation. Nvidia nvswitch: The world's highest-bandwidth on-node switch. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, 2018.

[42] NVIDIA Corporation. Nvidia gb200 nvl72: Hpc & ai gpu for data centers. https://www.nvidia.com/en-us/data-center/gb200-nvl72/, 2024.

[43] NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2024.

[44] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. (arXiv:2410.06511), November 2024.

[45] NVIDIA Corporation. Cuda profiling tools interface (cupti) user guide. https://docs.nvidia.com/cupti/index.html, 2024.

[46] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. (arXiv:2307.08691), July 2023.

[47] NVIDIA Corporation. Cuda runtime api - event management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html, 2024.

[48] NVIDIA Corporation. Nvidia nccl user guide: Environment variables. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-proto, 2025. Accessed: 2025-01-11.

[49] ebpf: Introduction, tutorials & community resources. https://ebpf.io/. Accessed: 2025-01-13.

[50] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[51] Sajjad Haider, Naveed Riaz Ansari, Muhammad Akbar, Mohammad Raza Perwez, and KM Ghori. Fault tolerance in distributed paradigms. In *In2011 International Conference on Computer Communication and Management, Proc. of CSIT*, volume 5, 2011.

[52] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. Elastic parameter server load distribution in deep learning clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 507–521, 2020.

[53] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with SCALENE. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 51–64, 2023.

[54] Zhe Wang, Huanwu Hu, Linghe Kong, Xinlei Kang, Qiao Xiang, Jingxuan Li, Yang Lu, Zhuo Song, Peihao Yang, Jiejian Wu, Yong Yang, Tao Ma, Zheng Liu, Xianlong Zeng, Dennis Cai, and et al. Diagnosing application-network anomalies for millions of IPs in production clouds. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 885–899, 2024.

[55] Kaicheng Yang, Yuanpeng Li, Sheng Long, Tong Yang, Ruijie Miao, Yikai Zhao, Chaoyang Ji, Penghui Mi, Guodong Yang, Qiong Xie, Hao Wang, Yinhua Wang, Bo Deng, Zhiqiang Liao, Chengqiang Huang, and et al. AAsclepius: Monitoring, diagnosing, and detouring at the internet peering edge. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 655–671, 2023.

[56] Shujie Han, Patrick PC Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. An in-depth study of correlated failures in production ssd-based data centers. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 417–429, 2021.

[57] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao.

Zero overhead monitoring for cloud-native infrastructure using RDMA. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 639–654, 2022.