



# USENIX

THE ADVANCED COMPUTING  
SYSTEMS ASSOCIATION

## **FLARE: Anomaly Diagnostics for Divergent LLM Training in GPU Clusters of Thousand-Plus Scale**

*Weihaio Cui, Shanghai Jiao Tong University and National University of Singapore;  
Ji Zhang, Independent Researcher; Han Zhao, Shanghai Jiao Tong University;  
Chao Liu, Independent Researcher; Jian Sha, Tsinghua University;  
Bo Sang, Ant Group; Bingsheng He, National University of Singapore;  
Minyi Guo and Quan Chen, Shanghai Jiao Tong University*

<https://www.usenix.org/conference/nsdi26/presentation/cui>

This paper is included in the Proceedings of the 23rd USENIX Symposium  
on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium  
on Networked Systems Design and Implementation is sponsored by



جامعة الملك عبد الله  
للعلوم والتقنية  
King Abdullah University of  
Science and Technology

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

Weihaio Cui<sup>1,2,\*</sup>, Ji Zhang<sup>3,\*</sup>, Han Zhao<sup>1,†</sup>, Chao Liu<sup>3</sup>, Jian Sha<sup>4,†</sup>, Bo Sang<sup>5</sup>  
Bingsheng He<sup>2</sup>, Minyi Guo<sup>1</sup>, Quan Chen<sup>1</sup>  
<sup>1</sup>Shanghai Jiao Tong University, <sup>2</sup>National University of Singapore  
<sup>3</sup>Independent Researcher, <sup>4</sup>Tsinghua University, <sup>5</sup>Ant Group

## Abstract

The rapid proliferation of large language models has driven the need for efficient GPU training clusters. However, it is challenging due to 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 FLARE, a diagnostic framework designed for distributed LLM training at scale. FLARE first integrates a lightweight tracing daemon for full-stack and backend-extensible tracing. Additionally, it features a diagnostic engine that automatically diagnoses anomalies, with a focus on performance regressions. The deployment of FLARE across 6,000 GPUs has demonstrated significant improvements in pinpointing deficiencies in real-world scenarios, with continuous operation for over eight months.

## 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 backends [8–10]. Supported by these two teams, various algorithm teams focus on adapting LLMs for user-facing applications through various training methods [11, 12]. Notably, the training cluster also supports other deep learning jobs, such as recommendation models and their specific parallel backend, TorchRec [13].

\*Equal contribution

†Corresponding author

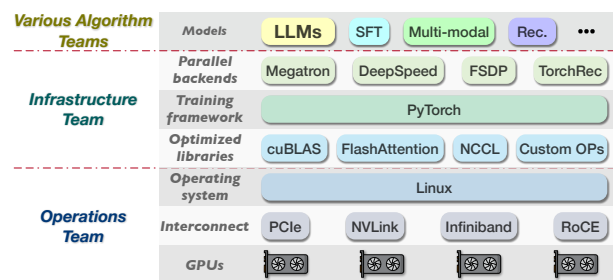


Figure 1: The training stack of large-scale training cluster.

Efficient large-scale distributed LLM training requires effective coordination of software and hardware across the stack. However, training anomalies frequently arise from various layers. These anomalies include obvious job failures and non-obvious training slowdowns, which can be categorized into three types: performance regressions, job failures, and fail-slows. Performance regressions may result from incorrect training settings (addressed by algorithm teams) or unoptimized operators (addressed by infrastructure teams). In contrast, most job failures and fail-slows arise from hardware issues, which are handled by the operations team. **One key insight in this paper is that fail-slows are sudden and easy-to-detect slowdowns due to transient issues in components, whereas performance regressions are persistent and hard-to-detect slowdowns caused by code updates or configuration drift.**

Drawing on our daily operational experiences, teams have spent significant time in cross-team collaborations to address these anomalies, often repeating efforts on similar cases. For instance, performance regressions caused by unnecessary synchronizations introduced by algorithm teams often require collaboration with the infrastructure team. Yet, eliminating these synchronizations usually only involves deleting a few lines of code in the training scripts. Therefore, a dedicated diagnostic framework is urgently needed to mitigate this.

The framework first should collect sufficient runtime data for anomaly detection and potential causes identification, enabling teams to resolve them with minimal collaboration. To

this end, runtime data must be continuously gathered from all layers of the training stack, especially from the software in the algorithm and infrastructure teams. Secondly, root causes should be narrowed as much as possible; otherwise, even if recurrent anomalies are routed to the right teams, they may still be unable to resolve them independently.

To design such a diagnostic framework, we identify two main challenges. **C-1: Designing a full-stack tracing mechanism with backend extensibility is challenging.** The tracing mechanism collects runtime data for anomaly diagnosis. While full-stack tracing is needed to capture sufficient runtime data, diverse parallel backends calls for backend-extensibility. However, these requirements often conflict, as full-stack tracing usually intrudes into parallel backend codebases, reducing backend-extensibility. **C-2: Detecting and diagnosing root causes is challenging.** Errors or fail-slows in LLM training often show similar symptoms, such as process hangs or sudden drops in training speed. Worse, regressions cannot be directly confirmed by comparing macro metrics, such as current training throughput, with historical runs, since throughput declines may be expected. These factors make it difficult to detect anomalies and narrow down their root causes.

Faced with these challenges, previous efforts [2, 3, 14] targeting distributed LLM training fail to provide comprehensive solutions. This is because they are narrowly designed for specific problems or scenarios, making them inadequate for addressing issues across the software-hardware stack. For example, Greyhound [14], C4D [3], and Holmes [15] can only trace and diagnose errors or fail-slows under the responsibility of operations team. **However, they could not solve the performance regressions originated from upper-level teams.** Besides, tools such as MegaScale [2] are optimized for pre-training scenarios involving a single algorithm team and a single backend. It intrudes into the backend codebase to achieve full-stack tracing, but **this tight coupling makes it difficult to plug into other parallel backends.** Moreover, MegaScale only provides visualization of distributed training and relies on cross-team collaboration and manual effort to pinpoint root causes for detected anomalies.

To this end, we present FLARE, an anomaly diagnostic framework for divergent LLM training in GPU clusters at thousand-plus scale. It places particular emphasis on diagnosing performance regression anomalies from algorithm and infrastructure teams. FLARE consists of two components: a per-training-process tracing daemon and a diagnostic engine. To collect sufficient runtime data with backend-extensibility (**C-1**), the tracing daemon selectively instruments key code segments in a plug-and-play manner across both Python and C++ runtimes. Most importantly, it leverages CPython tracing mechanisms to trace Python code segments without intruding into backbone codebases. It also provides an easy-to-play interface—simply adding shell environments—for extending to new backends and allowing all teams to selectively trace Python code segments.

With real-time data collected by the tracing daemon, the diagnostic engine detects and diagnoses anomalies while narrowing their root causes (**C-2**). As for error diagnostics, FLARE introduces a novel intra-kernel inspecting mechanism, providing fine-grained diagnostics specifically targeting communication-related hang errors. It enables  $O(1)$ -complexity faulty machine diagnostics without exhaustive or blind searches based on NCCL tests. As for slowdown diagnostics, FLARE 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. Importantly, the new proposed micro metrics enable FLARE to detect and diagnose regressions. Overall, with these metrics, FLARE enables teams to independently address those routed slowdowns, including subtle ones such as a 2.66% regressions in real-world workloads.

We extensively evaluated FLARE in terms of its runtime overhead. FLARE incurs an average latency overhead of only 0.43% across various LLMs and backends on 1024 H800 GPUs. Meantime, FLARE only generates just 1.5MB of tracing logs per GPU in a real-world model trained on 1536 H800 GPUs. FLARE has been deployed in our training cluster at Ant Group [1], utilizing over 6,000 GPUs to date. The greatest benefit of FLARE's deployment is that it eases the handling of recurrent anomalies. FLARE has been open-sourced at [DLRover](#). It serves as a core component of DLRover [16], an automated distributed DL system deployed within Ant Group and supported by the LF AI & Data Foundation [17]. 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 FLARE, a diagnostic framework specifically designed to tackle the critical challenges of full-stack tracing, root cause diagnosis, and backend extensibility in LLM training diagnostics.
- We deploy FLARE 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

### 2.1 Large-Scale LLM Training Stack

Referring to the software-hardware stack in [Figure 1](#), we delve into how LLMs are reshaping teams in leading corporations. **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 [19, 20], reasoning tasks [21, 22]. 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 inputs, such as product images and textual data.

Table 1: A comprehensive analysis of anomalies encountered in Ant Group, with FLARE’s primary target highlighted in yellow.

Anomalies										
Type	Error			Slowdown						
Taxonomy	OS errors	GPU errors	Network errors	New algorithms	Unnecessary synchronization	Un-optimized kernels	Memory management	GPU underlocking	Network jitter	
Symptom	Runtime hang or crash error			Regressions compared to historical jobs				Fail-slows compared to prior training steps		
Team	Operations			Algorithm			Infrastructure		Operations	

Consequently, different algorithm teams continue to innovate LLM models tailored for diverse application scenarios.

**Advancing of training cluster.** Training these LLMs is computationally intensive, requiring large-scale GPU clusters of thousand-plus scale [23]. Therefore, leading corporations are continuously investing in large training clusters powered by state-of-the-art hardware [24, 25]. These clusters not only expand in scale but also incorporate the latest GPUs [26, 27] with higher computational performance and advanced interconnect [28, 29]. Efficient operation of these large-scale clusters is critical to the corporations, which requires the operations team to ensure uninterrupted training performance.

**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, 30], training framework [7], and state-of-the-art model parallel backends [8, 9, 13, 31]. Parallel backends are fundamental for efficiently training LLMs, enabling tensor, data, and pipeline parallelism across thousands of GPUs.

## 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: runtime errors and slowdowns. Based on symptom differences, slowdowns can be further divided into two types: performance regressions (persistent slowdowns) and fail-slows (sudden and acute slowdowns).

Based on three months of traces from a cluster with 6000+ GPUs, there are 127 errors and 135 slowdowns observed across 3047 jobs. The slowdowns include 78 performance regressions and 57 fail-slows. When such anomalies occur, they are first reported by algorithm teams, and then multiple teams coordinate and spend significant effort to resolve them.

Specifically, for runtime errors and fail-slows, the operations team intervenes to resolve these acute and easy-to-detect job failures. Without timely intervention, training jobs may fail or run at very low throughput, significantly reducing training efficiency. For performance regressions, algorithm teams are often unfamiliar with why these regressions occur with

their own code changes, so collaboration with infrastructure teams is required to resolve them. In a training stack like Figure 1, the situation worsens: the infrastructure team supports multiple algorithm teams, each potentially using different parallel backends for LLM training. As recurrent regressions arise across different algorithm teams, the infrastructure team must repeatedly expend effort to address them.

This dilemma motivates a dedicated diagnostic framework to streamline anomaly resolution, especially performance regressions. To achieve this, we must collect sufficient runtime data across the training stack. For example, overall training throughput helps detect fail-slows, while per-operator performance reveals regressions introduced by minor software changes. We must also seamlessly support multiple parallel backends used by different algorithm teams. Finally, for detected anomalies, we aim to narrow down their root causes as much as possible, enabling them to be routed to the appropriate team and resolved independently without unnecessary cross-team collaboration.

However, designing such a framework is far from trivial, posing two key challenges.

**The difficulty of full-stack, backend-extensible tracing.** Diagnosing performance regressions across diverse parallel backends requires full-stack and backend-extensible tracing to collect sufficient runtime data. These goals often conflict. Full-stack tracing tends to be intrusive to each backend’s codebase, whereas extensibility demands minimal changes to the backends. A typical example is MegaScale [2], which enables low-overhead tracing for FSDP—the native PyTorch backend—by patching the FSDP codebase. Extending support to other backends would require similar patches. A tracing design for runtime data collection in training jobs of multiple algorithm teams should avoid such backend-intrusive modifications.

**The difficulty of narrowing down attribution.** Errors and fail-slows in Table 1 often show obvious symptoms, as the job either fail or show sudden training speed degradation. There are various works like C4D [3], Greyhound [14], and Holmes [15] that are proposed to diagnose them. However, automatic performance regressions are rarely explored in prior work. Attributing them is more challenging than diagnosing fail-slows. First, they are harder to detect. Regressions are persistent slowdowns caused by software changes, and monitoring training throughput alone is insufficient. Algorithm teams may not notice them and may assume jobs are running normally. Second, linking a detected regression to the corresponding software change is difficult. For example, whether

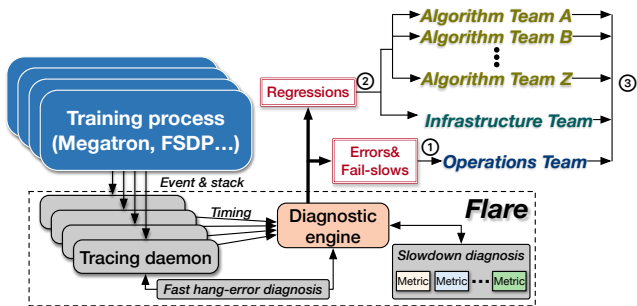


Figure 2: Architecture overview of FLARE.

Python garbage collection causes a regression depends on how it is invoked. Thus, new metrics must be designed based on operational expertise, especially from infrastructure team, to detect and diagnose regressions.

### 3 FLARE Design

In this paper, we propose FLARE, a diagnostic framework for diverse large-scale LLM training. It places particular emphasis on diagnosing performance regression automatically from algorithm and infrastructure teams. Figure 2 illustrates the architecture of FLARE, deployed in Ant Group’s large-scale training cluster. FLARE is composed of two components: the tracing daemon and the diagnostic engine.

By automatically attaching a tracing daemon to each training process in LLM jobs, FLARE provides a lightweight tracing mechanism. The daemon is backend-extensible but also enables the collection of sufficient runtime data across the entire training stack. For each job, runtime data such as function timing events and call stacks are intercepted for monitoring and diagnostics. The diagnostic engine employs a fast hang-error diagnostic method and leverages novel aggregated metrics to effectively identify slowdowns, especially regressions. The aggregated metrics encompass both macro-level metrics, such as training throughput, and micro-level metrics, including issue latency distribution of GPU kernels.

The overall diagnostic pipeline is as follows:

① Fast hang-error diagnosis enables error detection and analysis. Fail-slows are detected through macro metrics and validated by micro metrics. Errors and fail-slows are then routed to the operations team for independent handling.

② Regressions are detected by monitoring the proposed micro metrics and diagnosed via Python API analysis. FLARE routes potential regressions with narrowed root causes to the algorithm or infrastructure team.

③ The algorithm, infrastructure, and operations teams collaborate only when anomalies cannot be resolved independently by the routed team.

### 4 Lightweight Selective Tracing

The runtime data collection overhead primarily stems from high memory usage rather than interference with computing

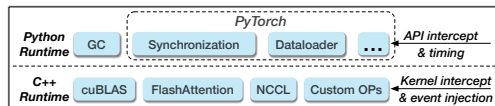


Figure 3: Instrumented key code segments in FLARE.

resources, since profiling APIs like CUPTI [32] can operate in a background thread. 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.

Therefore, FLARE 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. In the rest of this section, we introduce the way FLARE achieves full-stack and backend-extensible tracing through employing two techniques: plug-and-play instrumentation and timing with stack reconstruction.

#### 4.1 Plug-and-Play Instrumentation

Full-stack, backend-extensible tracing requires plug-and-play instrumentation that captures the required runtime data without intrusive backend changes. Figure 3 shows the code segments instrumented by FLARE, which are widely used in backend codebases and broadly fall into two groups. The first intercepts key API calls, including Python’s garbage collection (GC), PyTorch’s dataloader, and GPU synchronization. The second targets critical GPU computation and communication kernels executed at the C++ runtime level. These kernels dominate the workload in large-scale training.

Intercepting C++ functions is straightforward with backend extensibility, as mechanisms like LD\_PRELOAD can hook any dynamically linked C++ functions. In contrast, intercepting Python APIs is more challenging when enforcing backend extensibility. A common method for tracing is to provide interfaces such as Python decorators to wrap APIs in backend. E.g., MegaScale supports FSDP by patching PyTorch, which requires intruding into each backend for full-stack support.

To avoid such intrusiveness for backend extensibility, FLARE adopts a CPython-based intercepting mechanism. It maintains a list of tracing-required APIs for each backend. When a training job starts, FLARE imports these APIs and retrieves their bytecode. During long training runs, it intercepts them directly using CPython’s profiling API PyEval\_SetProfile based on the bytecode.

The above design allows FLARE to instrument either Python or C++ functions in a plug-and-play manner. For easy plug-in, FLARE does not modify any backend codebase before tracing. For easy play, FLARE

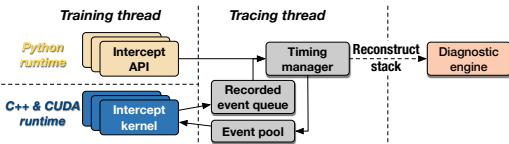


Figure 4: Timing the training in the background.

exposes user-friendly interfaces for tracing new functions. Specifically, users only need to configure environment variables before launching jobs, such as `export TRACED_PYTHON_API="torch.cuda@synchronize"`.

Intercepting C++ functions requires explicit registration through a C++ interface. This is practical because the infrastructure team develops both these C++ functions and FLARE, ensuring seamless integration and functionality.

## 4.2 Timing with Stack Reconstruction

With intercepted Python APIs and GPU kernels, FLARE measures their elapsed latencies, as shown in Figure 4. 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 [33] after an interception to record execution status. These events are enqueued for further processing. The timing manager queries the status of the 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.

As training progresses, the timing manager proactively streams real-time data to FLARE’s diagnostic engine. However, since plug-and-play instrumentation uses separate methods for Python APIs and C++ functions, it omits the call stack linking them, which is essential for anomaly diagnostics. Fortunately, we record not only their durations but also their start and end timestamps. Using this information, the tracing thread reconstructs the call-stack relationships before sending the data to the diagnostic engine.

## 5 Anomaly Detection and Diagnosis

In this section, we present FLARE’s automated diagnosis of anomalies: runtime errors, fail-slows, and performance regressions. Among these, FLARE pays particular attention to regressions, which have not been explored in prior work.

### 5.1 Fast Runtime Error Diagnosis

Runtime 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

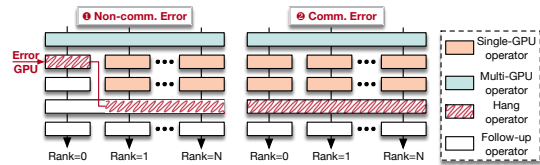


Figure 5: Diagnosing hang errors via call stack analysis.

relies on the coordination of training processes. When the aforementioned errors occur, they rarely affect all training processes simultaneously. The training process hangs if a GPU fails. In this context, FLARE focuses on rapidly diagnosing hang errors by identifying faulty machines. Then, FLARE routes this information to the operations team, enabling the training job to restart with healthy machines.

Specifically, the diagnostic engine detects hang errors by examining the status of tracing daemons. The daemon operates in the background of the training 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. If a tracing daemon transmits none 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. FLARE diagnoses these errors in two steps: first, a coarse-grained diagnosis through call stack analysis; and second, a fine-grained diagnosis using intra-kernel inspecting.

**Diagnosis using call stack analysis.** This diagnosis is used to identify problematic machines encountering non-communication errors. Figure 5 illustrates an example of hang-error diagnosis via call stack analysis. As shown in the left of Figure 5, 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.

However, communication hang cannot be identified through call stack analysis. As shown in the right of Figure 5, the training processes of all ranks terminate at the same call stack of a communication function, such as `allreduce` or `allgather`. 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 faulty machine responsible for such errors, a common approach is to terminate the training process and run NCCL tests across all GPUs. However, in large-scale

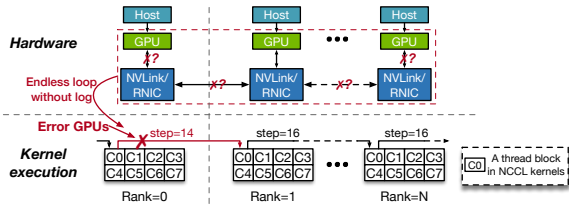


Figure 6: Diagnosing hang errors via intra-kernel inspecting.

training where multiple model parallelisms—such as tensor [9], pipeline [34], and expert parallelism [35]—are combined, the NCCL tests must span all configured communication groups. In our experience, this exhaustive and blind search can take over half an hour at the thousand-GPU scale.

**Diagnosis using intra-kernel inspecting.** Faced with this problem, FLARE introduces a minute-level diagnostic approach using intra-kernel inspecting with  $O(1)$  complexity. Instead of stopping the hanged training processes and blindly probing with NCCL tests, FLARE fully utilizes real-time data at the error point for diagnosis. Specifically, the intra-kernel inspecting leverages CUDA-GDB, the debugging tool for CUDA programming. It instructs the tracing daemon to attach to the halted training processes using CUDA-GDB. Once attached, the tracing daemon executes a script capable of automatically extracting detailed communication statuses to identify unhealthy machines.

Figure 6 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, FLARE retrieves 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 inspecting process is performed in parallel across all involved GPUs. As a result, its complexity is  $O(1)$ , enabling completion within a few minutes.

FLARE 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. Notably, this mechanism does not degrade the performance of the training job. The tracing daemon directly inspects the SASS code to retrieve register values. Thus, FLARE does not require recompiling the NCCL library with debugging information enabled, which would disable several compiler optimizations. Moreover, FLARE attaches to the training processes only after the job hangs, without interfering with normal execution.

## 5.2 Aggregation for Slowdown Diagnosis

To holistically identify fail-slow and regression slowdowns, FLARE aggregates real-time data from the tracing daemon into five primary metrics, shown in Figure 7. These metrics reflect the consensus that a “health” training pipeline should

exhibit a timeline saturated with GPU kernels dedicated to computation or communication. Computation kernels should deliver high FLOPS, while communication kernels should sustain high bandwidth. Deviations from these characteristics indicate idle GPU resources, signaling potential slowdowns in training jobs. In the rest of this section, we introduce the five metrics and explain how they are used to detect and narrow down the root causes of fail-slows and regressions.

### 5.2.1 Detecting Fail-slows with Macro Metric

**1 Training throughput for detecting fail-slows.** Fail-slows caused by low-level hardware changes, such as GPU underclocking or network jitter, are apparent and can be detected solely through comparisons across training steps. To this end, FLARE measures training throughput by timing the rate at which input data is consumed by the training pipeline. It is achieved by instrumenting the dataloader API of Pytorch.

### 5.2.2 Detecting Regression with Micro Metrics

Regressions caused by software changes introduced by algorithm and infrastructure teams are often subtle and difficult to detect. Identifying them usually requires manual comparisons across historical training jobs. However, directly comparing training throughput between monitored and historical jobs does not reliably reveal regressions, since model size, input datasets, and other factors may differ significantly. To address this, FLARE detects regressions using micro metrics. This is because when macro metrics do not show sudden changes, abnormalities in micro metrics often indicate potential persistent performance regressions.

**2 FLOPS and 3 bandwidth.** FLOPS and bandwidth capture the performance of instrumented computation and communication kernels, and can be used to detect regressions. For example, computation kernels with large input sizes but much lower FLOPS than the theoretical value indicate a regression in computation performance.

FLARE monitors the FLOPS of instrumented critical computation kernels, leveraging timing data and input layout. It also 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, FLARE calculates the communication bandwidth by utilizing the start and end timestamps of the final communication kernels issued across all participating ranks.

Notably, when analyzing FLOPS and bandwidth to detect regressions, FLARE accounts for the overlap of communication and computation kernels, which is common in training MoE-based LLMs [36]. This ensures that computation kernels with falsely low FLOPS are not mistakenly flagged.

While FLOPS and bandwidth 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, FLARE’s tracing also omits the monitoring of these operations. To detect their potential

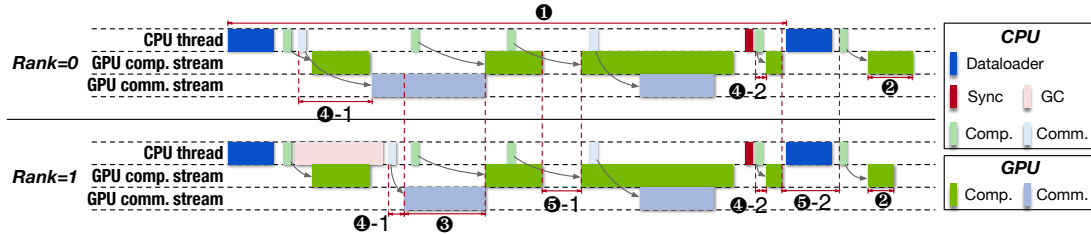


Figure 7: A timeline of a distributed training job annotated with aggregated metrics used for diagnosing slowdowns in FLARE.

contributions to regressions, 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 regression detection: issue latency distribution for intra-step CPU operations and void percentage for inter-step CPU operations and minority GPU kernels.

**④ Issue latency distribution for kernel-issue stall.** In a well-optimized parallel backend, 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 a regression anomaly known as a kernel-issue stall, leading to GPU idle time within the training step.

④-1 in Figure 7 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. ④-2 in Figure 7 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 regression 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, FLARE 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 regression due to kernel-issue stall, the kernel-issue latencies of unhealthy training jobs should be

much shorter than those of a healthy training job.

Ahead of deployment, FLARE learns healthy kernel issue distributions from historical data for each backend type and cluster scale. FLARE use the maximum Wasserstein distance [37] between these healthy distributions as a threshold. At runtime, Flare compares the collected data against the healthy distribution by computing the Wasserstein distance between them. A warning is triggered when the distance exceeds the learned threshold.

**⑤ 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 7. Consequently, FLARE introduces a metric, termed the void percentage, to identify slowdowns caused by these factors.

As for inter-step CPU operations, as depicted by ⑤-2 in Figure 7, FLARE measures the latency between the last kernel preceding the dataloader and the first kernel following the same dataloader. FLARE then computes the void percentage for inter-step CPU operations using the following equation:

$$V_{inter} = T_{inter} / T_{step} \quad (1)$$

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 ⑤-1 in Figure 7, FLARE 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 FLARE’s tracing mechanism. FLARE subsequently accumulates these slots for each training step and computes the void percentage using the following equation:

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

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 backend, FLARE annotates the training job with potential regressions attributed to inter-step CPU operations or minority GPU kernels.

### 5.2.3 Diagnosing Root Causes for Fail-slows

While changes in training throughput between steps of the same job indicate fail-slows, FLARE cannot directly diagnose the specific factors behind these slowdowns. Following prior work, FLARE investigates the root causes using two micro

metrics mentioned above: FLOPS and bandwidth.

By comparing the FLOPS of identical kernels across different ranks, FLARE 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. The captured communication bandwidth is compared with offline profiled data. If low-bandwidth communication is detected, FLARE 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.

### 5.2.4 Diagnosing Root Causes for Regressions

Pinpointing the root causes of regressions in computation and communication kernels is straightforward. FLARE directly forwards the traced data, such as input shape and layout, to the infrastructure team for resolution. When FLARE detects regressions related to kernel-issue stalls, it further narrows down the root causes by analyzing the invocation of relevant Python APIs recorded by the tracing daemon. Specifically, FLARE checks for APIs such as Python GC invoked just before communication kernels with abnormal issue distributions. The same approach applies to regressions detected by void percentage. If relevant APIs are found, FLARE directly pinpoints the potential root causes and first routes them to the corresponding algorithm teams. If the algorithm teams cannot resolve them independently, or if no relevant Python APIs are found, FLARE forwards the regressions to the infrastructure team for further investigation. In this case, both algorithm and infrastructure teams validate and resolves regressions with the root causes already narrowed down.

## 6 Evaluation

FLARE consists of 11.4K lines of code: 2,197 lines in Python, 7,447 lines in C++, and the rest in supporting code to facilitate its usage. Beyond the automatic diagnostic workflow described above, FLARE also provides rich information to assist manual optimizations, e.g., visualized distributed training timeline. In this section, we present experiments to demonstrate FLARE’s effectiveness from various perspectives.

### 6.1 Functionality Comparison

Since most related works [2, 3, 14] are closed-source or only partially open-sourced, we begin with a functionality comparison between FLARE and them, followed by in-depth evaluations of FLARE. Table 2 summarizes the comparison.

Firstly, in terms of tracing mechanisms, only MegaScale and FLARE provide full-stack tracing. Other works, such as C4D, Greyhound, and Holmes, trace only communication or computation kernels. However, due to different design assumptions, FLARE ensures backend extensibility, whereas MegaScale requires intrusive modifications to backend codebases, e.g., patching FSDP before tracing.

Secondly, in diagnostics, only FLARE offers automated de-

Table 2: Functionality comparison between FLARE and other existing works. “Comm.” denotes communication.

Category	Features	MegaScale	C4D	Greyhound	Flare
User experience	Full-stack tracing	✓	✗	✗	✓
	Backend-extensible	✗	✓	✓	✓
	Easy-to-play interfaces	✓	✗	✗	✓
	Automated diagnostics with aggregated metrics	✗	✗	✗	✓
	Distributed visualization	✓	✗	✗	✓
Hang error	Non-comm. hang	✓	✓	✗	✓
	Comm. hang	≥ 30min	≥ 30min	✗	≤ 5min
Slowdown	Critical kernels	✓	✗	✓	✓
	Overlapping of Comp. and Comm.	✓	✗	✗	✓
	Comm. kernels	✓	✓	✓	✓
	Kernel-issue stall	Only GC	✗	✗	✓
	Less critical operations	✗	✗	✗	✓

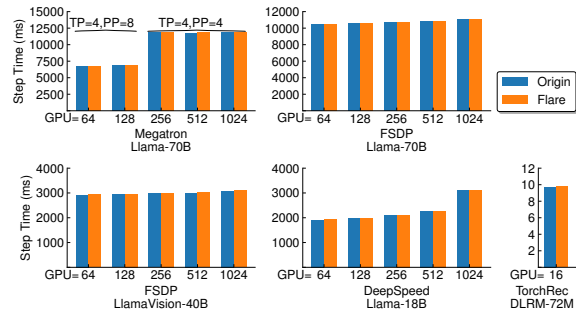


Figure 8: Runtime overhead in terms of latency with various models, backends, and number of GPUs.

tection of performance regression anomalies, mainly from algorithm and infrastructure teams, enabled by the proposed micro metrics. While MegaScale supports full-stack tracing, it only provides distributed visualization for manual regression investigation. C4D, Greyhound, and Holmes diagnose only errors and fail-slows. Although FLARE also detects these anomalies, it does not enhance their diagnostic mechanisms beyond providing intra-kernel inspection for fast communication hang-error diagnosis.

In a nutshell, FLARE is the first diagnostic framework to combine full-stack tracing with backend extensibility and provide automated regression anomaly diagnostics.

### 6.2 Runtime Overhead

We evaluate FLARE on four parallel backends: 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. We directly compare FLARE with the original execution, as it serves as the lower bound. As shown, FLARE incurs a latency overhead of 0.43% for three LLM training backends and 1.02% for TorchRec. Across other GPUs like

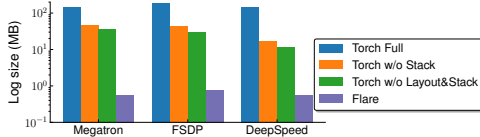


Figure 9: Memory consumption of dumped logs per GPU per step using the PyTorch profiler and FLARE while training a Llama-70B model on 16 A100 GPUs.

A100, FLARE shows consistent results.

We also compared the runtime overhead of FLARE with existing works [2, 14]. Compared with MegaScale, FLARE incurs similar runtime overhead, as both selectively trace key code segments. The main difference is that FLARE avoids intruding into backend codebases, achieving better backend extensibility. Originally, Greyhound only traces start timestamps of communication kernels. We extend Greyhound’s tracing mechanism to support full-stack tracing. In this case, Greyhound incurs unacceptable overhead, reaching 35% latency with Llama-8B trained on just 8 GPUs. This is because Greyhound’s tracing mechanism is tailored for fail-slows and does not cover regressions.

The memory overhead experiment is conducted on two setups, which are 16 A100 GPUs on 2 nodes and 1536 H800 GPUs on 192 nodes. The testbed slightly differs from that used in the latency overhead evaluation, as we conducted the experiments on available idle GPUs due to heavy load on our training cluster. We compare FLARE with Torch Full, Torch w/o Stack, Torch w/o Layout&Stack, and FLARE. 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 9 shows the memory overhead results on 16 A100 GPUs. FLARE consumes only 0.39%, 1.76%, and 2.48% of memory overhead for the respective configurations of PyTorch profiler. FLARE generates a maximum of 0.78MB of tracing logs per GPU. Besides, in a real-world Llama-20B training job on 1536 H800 GPUs, FLARE 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, FLARE consistently maintains an extremely low runtime overhead in terms of both latency and memory. The lightweight selective tracing facilitates FLARE’s deployment within our training cluster, serving as a diagnostic framework for diverse training jobs.

### 6.3 Effectiveness of Intra-kernel Inspecting

We evaluate the intra-kernel inspecting 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. We customize the training script composed solely of communication kernels, with one GPU intentionally suspended to simulate a hang error caused by communication issues.

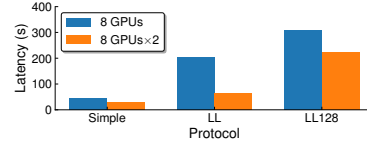


Figure 10: Latency for pinpointing the erroneous GPUs causing a hang error in ring-allreduce with different protocols.

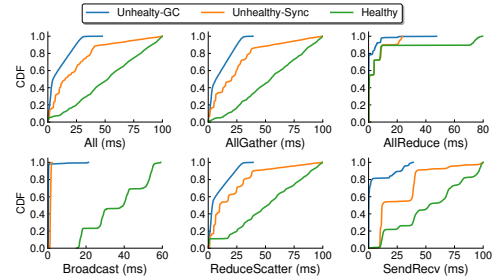


Figure 11: Issue distribution for a Llama-20B model trained with Megatron and 256 GPUs, including the overall CDF and the CDFs for each type of communication kernel, respectively.

Figure 10 illustrates the pinpointing latencies for intra- and inter-server communication. The figure presents the latency results for three communication protocols [38] and cross-node configurations. As shown, FLARE requires 29.4~309.2s to detect erroneous GPUs across different scenarios. Among the protocols, FLARE performs best when the SIMPLE protocol is used for communication. This is because, with the SIMPLE protocol, FLARE only needs to scan the first thread of each thread block to check the steps, whereas the other two protocols require scanning the entire thread block.

When comparing intra-server and inter-server results, FLARE 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, FLARE scans fewer thread blocks for error diagnosis in inter-server scenarios.

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

### 6.4 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 11 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,

Table 3: Typical errors detected by FLARE.

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	

GC is efficiently managed by the parallel backend, 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 11. 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 `Unhealthy-GC` is worse than that of `Unhealthy-Sync`.

## 7 Deployment & Case Studies

### 7.1 Cluster-wide Deployment

FLARE has been deployed and running continuously in a training cluster with 6,000 GPUs for over eight 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.

### 7.2 Errors

Table 3 presents a subset of error and fail-slow anomalies detected by FLARE. As shown in the table, FLARE effectively identifies OS- and hardware-related anomalies, including crashes, and hangs. While these issues are typically conspicuous and could be detected by existing methods based on noticeable training interruptions [9, 14], FLARE’s novelty lies in providing richer error information through techniques like intra-kernel inspecting. Such runtime information helps ease and accelerate the attribution process of low-level issues for operations teams.

### 7.3 Fail-slows and Performance Regressions

Table 4 summarizes fail-slow and regression slowdowns diagnosed by FLARE using its aggregated metrics. We omit evaluation of FLARE in fail-slow diagnostics, as it employs similar mechanisms to prior works [3, 14].

To evaluate regression diagnostics, we collect 113 real-

Table 4: Fail-slows and regressions diagnosed by FLARE, with “**Details**” showing training job specifics and associated MFU decline. We show the diagnosed regression anomalies by FLARE in bold.

Metric	Attribution	Details
FLOPS	GPU underclocking	480 GPUs, Llama-65B, 14% ↓
	<b>Backend migration</b>	<b>1856 GPUs, Llama-80B</b> , 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	<b>Python GC</b>	2048GPUs, Llama-80B, 10% ↓ 280GPUs, LlamaVision-11B, 60% ↓ ...
	<b>Unnecessary GPU Sync</b>	<b>256GPUs, Llama-20B</b> , 2.66% ↓ ...
	<b>Package checking</b>	<b>280GPUs, LlamaVision-20B</b> , 30% ↓
	<b>Frequent GPU mem. management</b>	<b>1344GPUs, Llama-176B</b> , 19% ↓
Void percentage	<b>Dataloader</b>	<b>512GPUs, Llama-80B</b> , 41% ↓

world training jobs submitted within a week in a 6,000-GPU cluster. During this period, the cluster maintained an average GPU usage of 80%. The jobs included popular LLMs, multi-modal LLMs, and recommendation models. We first diagnose them with FLARE, then validate results against human-labeled ground truth. It successfully diagnosed 9 true regressions based on issue latency distribution and void percentage, with only 2 false ones. This corresponds to a false positive rate of 1.9% and a true positive diagnostic accuracy of 81.8%.

We further investigated the 2 false positives. The first was a multi-modal LLM trained with FSDP, where input data contained images of varying resolutions. This caused imbalanced computation across ranks, leading to abnormal issue latency distribution. The second was a recommendation model using CPU-based embeddings. Its void percentage was higher than that of GPU-based ones, causing misclassification.

Although these 2 cases were falsely diagnosed as regressions, they highlight opportunities to refine FLARE’s micro-metrics with historical data. For example, relaxing latency distribution thresholds for imbalanced multi-modal inputs and adjusting void percentage thresholds for CPU-based embedding models could reduce such false positives. After these refinements, FLARE no longer misclassifies these job types in real-world cluster deployments.

Now, we present several typical diagnosed regressions.

#### 7.3.1 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 backend 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 FLARE is a training job of Llama-20B 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 regression was observed in training throughput, FLARE successfully detects the abnormal issue latency distribution. After removing these unnecessary synchronizations by disabling the timer, the MFU of the training job persistently improves from 41.4% to 42.5%, representing a 2.66% increase. Such regression is minor, which could not be directly observed through macro metrics like training throughput in a large training cluster with a lot of noise. However, there is an obvious drift in micro metrics like issue latency distribution, enabling FLARE to detect such obscured regression anomalies.

In addition to the above two cases, FLARE also detects other kernel-issue stalls, such as unnecessary package version checking, frequent CUDA memory management within the PyTorch runtime, and others.

### 7.3.2 Case-2: Migration between Backends

Different parallel backends are suited to varying hardware conditions. In this context, algorithm teams may migrate an LLM between backends to meet their specific demands. However, this migration process can potentially introduce regressions. A typical scenario encountered by FLARE 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 regression specifically stems from a matrix layout change. The weight dimension of the LLM’s FFN (feed forward network) 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.

While the algorithm team does not report this regression, FLARE detected and routed it to our infrastructure team. Following FLARE’s diagnosis, our infrastructure team customizes a kernel that pads 8484 to 8512. Figure 12 illustrates the FLOPS of the same operator before migration, after migration, and post-optimization guided by FLARE. As shown, the operator experiences a 65.3% decline in FLOPS after migration, and FLARE 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.

### 7.3.3 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 often introduces regressions. Firstly, algorithm teams generally modify position embeddings (PE), activation functions (ACT), and normalization

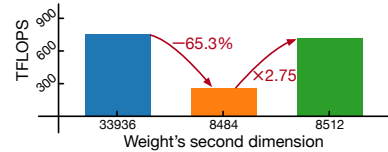


Figure 12: The change in computation TFLOPS when migrating Llama-80B from FSDP to Megatron.

Table 5: Changes in detected  $V_{minority}$  and normalized TFLOPS when different minority kernels are not optimized.

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

operators (NORM), while preserving the core structure of the transformer. Table 5 illustrates the detected changes in  $V_{minority}$  caused by these operator modifications during daily deployments. In this table, the parallel backend 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 FLARE,  $V_{minority}$  increases proportionally with their computational complexity. Our infrastructure team leverages FLARE’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, FLARE eliminates the need for manual identification, thereby accelerating anomaly diagnosis.

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

After routing this anomaly with the root cause identified in the dataloader to the algorithm team, they validated that the regression was due to the attention mask generation process within the dataloader. 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 Experience

### 8.1 Practical Usages

By leveraging FLARE, algorithm teams can independently identify and resolve regression anomalies caused by inefficient code without requiring intervention from the infrastructure team. With FLARE’s lightweight logging, the infrastructure team also collects sufficient runtime data to analyze

submitted jobs and independently discover new optimization opportunities. Only regression anomalies that cannot be resolved by algorithm teams are routed to the infrastructure team. Overall, the frequency of collaboration on recurrent regressions, such as unnecessary synchronizations, decreased by 63.5% within a one-week deployment, accelerating model evolution for algorithm teams and allowing the infrastructure team to focus more on system optimization.

## 8.2 Using Historical Data

Historical traces are essential for enhancing the effectiveness of regression detection. While runtime data is crucial, it is insufficient on its own for accurate regression diagnosis. FLARE detects issues by comparing real-time data against historical data. E.g., when using issue latency distributions to detect kernel-issue stalls, FLARE relies on historical data from specific backends operating on specific hardware in Figure 11. However, regression detection is inherently difficult to ascertain, even with historical data. Therefore, in practice, we adopt a conservative policy: we report potential regressions and their root causes without directly terminating training jobs.

## 8.3 Hardware Extending

Currently, FLARE has supported the extensibility of additional hardware, particularly NPUs dedicated to DL training. Since FLARE directly instruments key code segments at the Python and C++ runtime levels, extending it is straightforward. We already support internal CUDA-native NPUs, incurring less than 0.5% runtime overhead when running on 450 NPUs. We also investigate the extensibility of FLARE's intra-kernel inspecting on NPUs. Our study shows that the methodology is largely extensible, as NPUs also use dedicated hardware cores for cross-device communication. Inspecting their register status provides additional information for the operations team to address hang-error anomalies, but this requires additional engineering effort.

## 8.4 Scopes & Limitations

FLARE operates within certain scopes and has limitations.

*Firstly, FLARE targets large-scale private GPU clusters for LLM training and relies on historical data to calibrate its diagnostic metrics.* This assumption is reasonable. Accessing historical data is practical in a private GPU clusters. In our internal cluster, we also profiled typical LLMs and parallel backends at different scales to obtain ground-truth data for building the diagnostic mechanism.

*Secondly, FLARE cannot detect regressions in training jobs with major architecture updates, e.g., switching from a dense LLM to a sparse MoE LLM.* In such cases, FLARE must collect new historical data to refine its diagnostic engine. This requirement is usually reasonable, as FLARE is designed to relieve the infrastructure team from repetitive efforts on recurrent regression anomalies, allowing them to focus instead

on optimizing LLMs with new architectures.

## 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, 39–41]. 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. Greyhound [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 backend, limiting their generality.

**Anomaly diagnosis in large-scale datacenter.** Many research efforts [42–46] focus on anomaly diagnosis at different levels of the datacenter, including runtime, network, and storage. AND [43] 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 [44] proposes a PathDebugging technique to trace fault linkages between the middle network and autonomous systems. Researchers [45] 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 FLARE in large-scale distributed LLM training.

## 10 Conclusion

In this paper, we introduce FLARE, a real-time diagnostic framework for LLM training. By addressing challenges such as lightweight long-term monitoring, root cause detection, and backend extensibility, FLARE provides a comprehensive solution for anomaly diagnostics across large-scale GPU clusters. With its novel intra-kernel inspecting mechanism and holistic aggregated metrics, FLARE 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.

## Acknowledgments

This work is partially sponsored by the National Key Research and Development Program of China (2024YFB4505700), National Natural Science Foundation of China (62232011) and Natural Science Foundation of Shanghai Municipality (24ZR1430500) and CAAI-Ant Group Research Fund. We thank the anonymous reviewers and our shepherd Andrew Moore for their constructive feedback and suggestions.

## 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. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 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. Enhancing large-scale AI training efficiency: The C4 solution for real-time anomaly detection and communication optimization. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1246–1258, March 2025.
- [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*, pages 8026–8037, Red Hook, NY, USA, December 2019. Curran Associates Inc.
- [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, Mostafa 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, Wen-chao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. GREYHOUND: Hunting fail-slows in hybrid-parallel training at scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 731–747, 2025.
- [15] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zun-ling Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, Zekun He, Yachen Wang, Xianneng Zou, and Junchen Jiang. Holmes: Localizing irregularities in LLM training with mega-scale GPU clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 523–540, 2025.
- [16] Qinlong Wang et al. Dlover: An automatic distributed deep learning system. <https://github.com/intelligent-machine-learning/dlover>, 2023. Accessed: 2025-01-13.
- [17] Lf ai & data foundation. <https://lfaidata.foundation/>. Accessed: 2025-01-13.
- [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] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [20] OpenAI. Creating video from text. <https://openai.com/index/sora/>, 2024. Accessed: 2024-10-20.
- [21] OpenAI. Learning to reason with llms. <https://openai.com/index/learning-to-reason-with-llms/>, 2024. Accessed: 2024-10-20.
- [22] 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.
- [23] Hugging Face. Llama 3.1: A new milestone for open large language models. <https://huggingface.co/blog/llama31>, 2024. Accessed: 2024-10-20.
- [24] 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.
- [25] 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.
- [26] NVIDIA Corporation. Nvidia ampere architecture. <https://www.nvidia.com/en-us/data-center/ampere-architecture/>, 2020.
- [27] NVIDIA Corporation. Nvidia hopper architecture. <https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/>, 2022.
- [28] NVIDIA Corporation. Nvidia nvswitch: The world's highest-bandwidth on-node switch. <https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf>, 2018.
- [29] NVIDIA Corporation. Nvidia gb200 nv172: Hpc & ai gpu for data centers. <https://www.nvidia.com/en-us/data-center/gb200-nv172/>, 2024.
- [30] NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines. <https://github.com/NVIDIA/cutlass>, 2024.
- [31] 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.
- [32] NVIDIA Corporation. Cuda profiling tools interface (cupti) user guide. <https://docs.nvidia.com/cupti/index.html>, 2024.
- [33] NVIDIA Corporation. Cuda runtime api - event management. [https://docs.nvidia.com/cuda/cuda-runtime-api/group\\_\\_CUDA\\_\\_EVENT.html](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__EVENT.html), 2024.
- [34] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, Huntsville Ontario Canada, October 2019. ACM.
- [35] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. 2021.
- [36] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, and et al. DeepSeek-V3 technical report. (arXiv:2412.19437), December 2024.
- [37] Aaditya Ramdas, Nicolás García Trillos, and Marco Cuturi. On wasserstein two-sample testing and related families of nonparametric tests. *Entropy*, 19(2):47, 2017.
- [38] 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.
- [39] 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.
- [40] 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.

- [41] 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.
- [42] 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.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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.