DeepContext: A Context-aware, Cross-platform, and Cross-framework Tool for Performance Profiling and Analysis of Deep Learning Workloads

Qidong Zhao*
North Carolina State University
Raleigh, NC, United State
qzhao24@ncsu.edu

Hao Wu*
George Mason University
Fairfax, VA, United State
hwu27@gmu.edu

Yueming Hao†
Meta
Menlo Park, CA, USA
yhao@meta.com

Zilingfeng Ye
Independent
Beijing, China
yipzlf@gmail.com

Jiajia Li
North Carolina State University
Raleigh, NC, United State
jiajia.li@ncsu.edu

Xu Liu
North Carolina State University
Raleigh, NC, United State
xliu88@ncsu.edu

Keren Zhou
George Mason University
Fairfax, VA, United State
kzhou6@gmu.edu

Abstract

Effective performance optimization of deep learning models requires comprehensive profiling across heterogeneous computing environments, yet existing tools fail to bridge the semantic gap between high-level operations and low-level execution. This paper presents DeepContext, a novel profiling system that correlates program contexts across Python code, deep learning frameworks, C/C++ libraries, and GPU execution. DeepContext features a framework-agnostic shim layer that seamlessly correlates the behavior of the deep learning framework with hardware performance metrics. Furthermore, DeepContext provides an automated performance analyzer that offers actionable optimization guidance based on its holistic view of the entire software stack of deep learning applications. DeepContext works for mainstream deep learning frameworks and runs on modern CPU+GPU architectures with low overhead. Our evaluation demonstrates that DeepContext uncovers previously hidden performance bottlenecks in real-world deep-learning applications. Guided by DeepContext, we are able to fix multiple performance issues, achieving speed-ups between 1.06× and 1.66×.

*Both authors contributed equally to this research.
†The author's contribution to this project was carried out while he was a Ph.D. student at North Carolina State University.

CCS Concepts: • **General and reference → Performance**; **Measurement**; • **Software and its engineering → Dynamic analysis**.

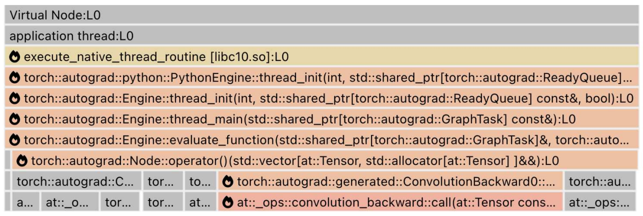
Keywords: Profiling Tool; Deep Learning; Performance Tuning; Heterogeneous Computing; Context-aware Profiling; Automated Analysis; Cross-framework Profiling

ACM Reference Format:
Qidong Zhao, Hao Wu, Yueming Hao, Zilingfeng Ye, Jiajia Li, Xu Liu, and Keren Zhou. 2025. DeepContext: A Context-aware, Cross-platform, and Cross-framework Tool for Performance Profiling and Analysis of Deep Learning Workloads. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676642.3736127

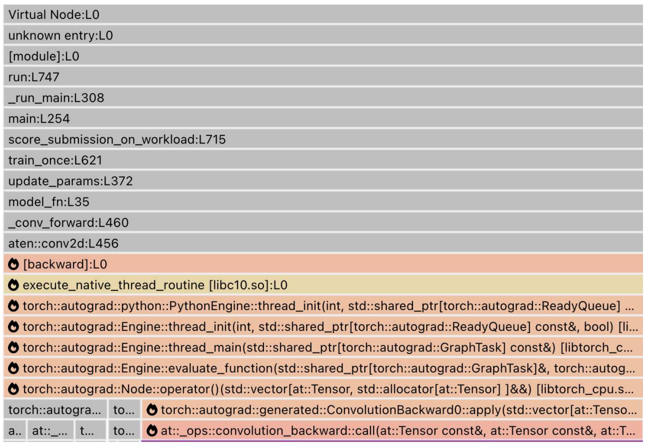
1 Introduction

The rapid advancement of deep learning has led to increasingly complex models [3, 19, 46] deployed across diverse and heterogeneous computing environments. Optimizing the training and inference of these models is critical for improving performance and reducing computational costs [27, 33]. However, the sophisticated interactions between CPUs and GPUs, coupled with the diversity of frameworks [11, 45] and compilation modes [6], pose significant challenges for developers seeking to identify and address performance bottlenecks effectively.

Profiling tools play a critical role in identifying performance issues in deep learning workloads. These tools include framework-specific solutions, such as the PyTorch profiler [47] and the JAX profiler [32], as well as those provided by vendors, such as Nsight Systems [41], rocprof [5],


Virtual Node:L0
application thread:L0
execute_native_thread_routine [libc10.so]:L0
torch::autograd::python::PythonEngine::thread_init(int, std::shared_ptr[torch::autograd::ReadyQueue]...
torch::autograd::Engine::thread_init(int, std::shared_ptr[torch::autograd::ReadyQueue] const&, bool):L0
torch::autograd::Engine::thread_main(std::shared_ptr[torch::autograd::GraphTask] const&):L0
torch::autograd::Engine::evaluate_function(std::shared_ptr[torch::autograd::GraphTask]&, torch::auto...
torch::autograd::Node::operator()(std::vector[at::Tensor, std::allocator[at::Tensor]]&&):L0
torch::autograd::C...
tor...
to...
torch::autograd::generated::ConvolutionBackward0::...
torch::au...
a...
at::_o...
tor...
tor...
at...
at::_ops::convolution_backward::call(at::Tensor cons...
at::_ops:...

(a) The hot call path w/o framework context


Virtual Node:L0
unknown entry:L0
[module]:L0
run:L747
_run_main:L308
main:L254
score_submission_on_workload:L715
train_once:L621
update_params:L372
model_fn:L35
_conv_forward:L460
aten::conv2d:L456
[backward]:L0
execute_native_thread_routine [libc10.so]:L0
torch::autograd::python::PythonEngine::thread_init(int, std::shared_ptr[torch::autograd::ReadyQueue] ...
torch::autograd::Engine::thread_init(int, std::shared_ptr[torch::autograd::ReadyQueue] const&, bool) [li...
torch::autograd::Engine::thread_main(std::shared_ptr[torch::autograd::GraphTask] const&) [libtorch_c...
torch::autograd::Engine::evaluate_function(std::shared_ptr[torch::autograd::GraphTask]&, torch::autog...
torch::autograd::Node::operator()(std::vector[at::Tensor, std::allocator[at::Tensor]]&&) [libtorch_cpu.s...
torch::autogra...
to...
torch::autograd::generated::ConvolutionBackward0::apply(std::vector[at::Tenso...
a..
at::_...
t...
to...
at::_ops::convolution_backward::call(at::Tensor const&, at::Tensor const&, at::T...

(b) The hot call path w/ framework context

Figure 1. Comparison of highlighted call paths w/o and w framework contexts in two framegraphs [23].

and VTune [29]. The primary functionality of these tools is tracing, which captures metrics associated with individual CPU and GPU operations and displays them on a comprehensive timeline to assist users in investigating performance bottlenecks.

The landscape of profiling tools presents significant challenges for comprehensive performance analysis and optimization of deep learning workloads. Table 1 summarizes the features and limitations across five categories of existing tools: system, hardware, framework, Python, and deep learning profilers.

System profilers such as Linux perf and eBPF provide strong CPU profiling capabilities and cross-framework support, but lack Python and framework context that is critical for deep learning workloads. Hardware profilers (i.e., profilers provided by hardware vendors) such as Nsight Systems, rocprof, and Intel VTune offer hardware-specific insights but have varying limitations: Nsight Systems lack framework context; rocprof is particularly limited in its contextual awareness; and all have constraints in cross-device profiling. Framework profilers like JAX and PyTorch profilers provide good Python integration and some device context, but are framework-specific. Dedicated Python profilers such as Scalene offer Python visibility, but lack framework-specific insights. Existing deep learning profilers such as XSP provide framework and device context but still miss the full stack view.

These limitations significantly hinder thorough performance analysis. For example, Figure 1 illustrates a key shortcoming. In Figure 1a, only the C++ code is visible in the call path, obscuring where convolution operations originate, especially problematic since frameworks like PyTorch launch backward and forward operators from different CPU threads. This limitation makes it challenging to pinpoint performance bottlenecks in complex models that may invoke hundreds of convolution operations [26]. In contrast, Figure 1b reveals both the Python call path and the associated deep learning operators, providing deeper insights for optimization.

Additionally, most existing tools [5, 32, 35, 47] trace and record every operation, generating enormous profiles for long-running training workloads. This not only consumes excessive memory resources, but also makes bottleneck identification tedious and manual. While some tools can aggregate metrics postmortem for individual kernels, they cannot streamline metrics aggregation online to reduce profile size or differentiate between instances called from different contexts for automated problem identification.

The incompatibility across platforms and frameworks further complicates analysis. Vendor tools such as Nvidia's Nsight suite work with multiple frameworks on Nvidia GPUs, but are incompatible with AMD GPUs. Similarly, framework-specific tools such as the PyTorch profiler support both AMD and Nvidia GPUs, but cannot handle JAX workloads. This siloed approach increases the learning curve and limits cross-referencing across frameworks and hardware, making it difficult for users to determine optimal configurations for their specific workloads.

To address these challenges, we introduce DeepContext, a novel profiler that delivers comprehensive performance insights for deep learning workloads. As shown in Table 1, DeepContext captures all critical program context information across the entire software and hardware stack, enabling the identification of performance issues from high-level Python code down to the device code on GPUs. It provides cross-GPU profiling capabilities and maintains visibility across multiple frameworks—supporting both PyTorch and JAX on AMD and Nvidia GPUs, as well as x86 and ARM CPUs. By effectively collecting, attributing, and aggregating performance metrics, DeepContext facilitates an in-depth analysis of performance bottlenecks and reveals optimization opportunities at the architectural and system levels that conventional profiling tools typically miss.

This paper presents the design, implementation, and evaluation of DeepContext and makes the following contributions:

Table 1. Comparison of DeepContext with existing profiling tools. (✓ = supported; ✗= not supported.)

Profiling Tools		Python	Framework	C++	Device	Cross	Cross	CPU
Category	**Name**	**Context**	**Context**	**Context**	**Context**	**GPUs**	**Frameworks**	**Profiling**
System Profilers	**Linux perf [36]**	✗	✗	✓	✗	✗	✓	✓
	eBPF [30, 31]	✗	✗	✓	✗	✗	✓	✓
Hardware Profilers	**Nsight Systems [41]**	✓	✗	✓	✗	✗	✓	✓
	rocprof [4]	✗	✗	✗	✗	✗	✗	✗
	Intel VTune [29]	✗	✗	✓	✗	✗	✓	✓
Framework Profilers	**JAX profiler [32]**	✓	✗	✗	✗	✓	✗	✓
	PyTorch profiler [47]	✓	✓	✗	✗	✓	✗	✓
Python Profilers	**Scalene [8]**	✓	✗	✗	✗	✗	✓	✓
Deep Learning Profilers	**XSP [35]**	✗	✓	✗	✓	✗	✓	✗
	DeepContext (our tool)	✓	✓	✓	✓	✓	✓	✓

- We introduce a "shim" layer—DLMonitor—that converts deep learning framework-specific data into a framework-agnostic format, enabling seamless integration of framework information with third-party performance tools.
- We design an automated performance analyzer that provides actionable optimization suggestions based on performance metrics and program contexts. Such optimization suggestions include fusing operators, changing data layouts, modifying hardware configurations, and many others.
- We adopt a series of optimizations to reduce the runtime overhead and memory and storage requirements of Deep Context to enable it running on real deep learning applications, which typically have complex code hierarchies and long execution time.

We evaluated DeepContext using a diverse range of deep learning workloads across various platforms and frameworks, demonstrating that DeepContext significantly saves memory and disk space usage with similar runtime overhead compared to the state-of-the-art performance tools. Through case studies, we show that DeepContext can effectively identify performance issues and suggest insightful code changes, enabling straightforward optimization of deep learning models. Even users with limited experience in deep learning frameworks or CPU and GPU architectures can achieve speedups ranging from 1.06× to 1.66×.

2 Related Work

In this section, we review existing approaches about deep learning profilers, call path profiling, and automated performance analysis.

Deep Learning Profilers. Many profilers can measure deep learning workloads. These tools include vendor-provided tools such as Nsight Systems [41], VTune [29], rocprof [4] and DLProf [13], as well as framework-based tools such as the JAX profiler [32] and the PyTorch profiler [47]. However, these tools often focus on specific frameworks or platforms with limited applicability.

Some workload-specific profilers, such as RL-Scope [22] and XSP [35], analyze interactions across layers of the deep learning stack to identify bottlenecks that are not obvious when examining individual layers. DeepContext advances these tools by employing a generic solution for different frameworks, GPUs, and platforms.

There are profilers that post-process metrics from existing profiling tools. For instance, Hotline Profiler [50] introduces a multi-scale timeline with annotations for DNN training, based on the postmortem analysis of results from the PyTorch profiler [47]. Similarly, DLProf [13] analyzes the results collected from Nsight Systems. Since these tools do not modify the runtime, they suffer from the same limitations as trace-based profilers, which incur significant memory and disk overhead.

Skyline [56], as the most related tool to our tool, offers an interactive profiling experience by integrating the profiler into the development environment. However, unlike Deep Context that intercepts "native" C/C++ operations, Skyline uses monkey patching [9] for PyTorch Python operations, which introduces overhead and prevents it from obtaining native call path information. Additionally, it does not interact with vendor-provided profiling substrates, limiting its ability to gather abundant low-level information using performance counters.

Call Path Profilers. General performance tools such as HPCToolkit [59], TAU [49], perf [36], and DrCCTProf [58] offer call path profiling and performance analysis for low-level languages such as Fortran and C/C++. These tools provide deep insights into the complex behaviors of the underlying software stack and operating system, offering a complementary perspective to what deep learning profilers may miss. Additionally, some of these tools can sample a large set of CPU performance counters, going beyond coarse-grained metrics. However, these tools often lack integration with

Python runtime and deep learning frameworks, limiting their effectiveness in profiling multi-language environments.

On the other hand, Python-specific profiling tools like Scalene [8] and cProfile [21] provide effective analysis of Python call paths but lack the ability to analyze call paths in lower-level languages. Moreover, they can only collect limited information about accelerators. DeepContext addresses these shortcomings by providing a comprehensive call context that spans every level of the software stack, effectively bridging the gap between native language profiling and high-level language analysis.

Automated Performance Analysis. Existing automated performance analysis tools often target specific or limited domains. For instance, tools such as NCU [14] and GPA [61] focus on pattern matching of GPU kernels using expert-defined rules based on fine-grained metrics. These tools provide insights into how and where to modify kernel source code to improve performance. On the other hand, tools such as Nsight Systems [41] and DLProf [13] analyze trace patterns to provide coarse-grained recommendations. They offer insights into which GPU operations are expensive and whether the CPU is causing performance bottlenecks. However, these tools do not consider both low- and high-level contexts simultaneously. In contrast, DeepContext introduces a pattern matching system that allows flexible rules to be defined for analysis, incorporating low- and high-level contexts based on fine- and coarse-grained metrics. This approach provides a more holistic view of performance issues, enabling more flexible analysis and effective optimizations.

3 Overview

DeepContext is designed to achieve multiple objectives, including providing a holistic view that spans high- and low-level contexts, enabling cross-framework profiling, and supporting fast or even automated performance analysis. Each of these objectives presents unique challenges: (1) High- and low-level contexts are obtained through different methods, as Python is interpreted while C++ is compiled, and framework-specific information cannot be obtained simply by examining either C/C++ or Python code alone. (2) Different frameworks are implemented in vastly different ways, making direct instrumentation of their source code to capture information both unstable and unmaintainable. (3) For complex deep learning workloads, the profiling database will include extensive program context, numerous GPU kernel invocations, and various metrics, making it challenging to manually identify performance issues.

To address these challenges, DeepContext is structured into four primary components, as shown in Figure 2, each designed to handle distinct aspects of performance profiling for deep learning workloads. The *profiler* is responsible for collecting, attributing, and aggregating performance metrics. It gathers performance metrics from Nvidia and AMD GPUs

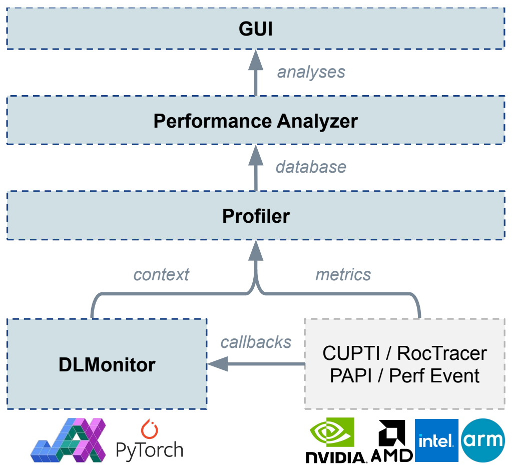


Figure 2. Major components of DeepContext.

through the CUPTI [40] and RocTracer [5] APIs, and from CPUs through Linux system calls, Perf events [36], and the PAPI API [55]. Once the program context is obtained from DLMonitor, the profiler associates the metrics collected with the program context and aggregates the metrics within the same context when necessary. DLMonitor serves as a "shim" layer between the profiler and the deep learning framework. It allows the profiler to intercept framework operations during the execution of deep learning frameworks and provides a comprehensive context when the profiler calls DLMonitor. Next, the *performance analyzer* processes the collected data postmortem, identifying performance bottlenecks and optimization opportunities. Finally, the *GUI* presents the analyzed data in an intuitive format, compatible with Visual Studio Code [37], to facilitate interpretation and inspection of performance issues.

4 Design and Implementation

This section is organized into five subsections: four dedicated to the aforementioned modules of DeepContext, and a fifth that discusses how DeepContext's modular design facilitates updates as profiling APIs evolve, while also explaining its support for diverse GPU generations and features.

4.1 DLMonitor

DLMonitor is a key component of DeepContext, providing a unified interface for obtaining call paths and registering callbacks within deep learning frameworks. Profilers interact with deep learning frameworks by invoking DLMonitor's APIs. The core APIs include:

- `dlmonitor_init`: Initializes DLMonitor's shared library, `libdlmonitor.so`, which is typically loaded at the start of execution using utilities like `LD_PRELOAD`.
- `dlmonitor_callback_register`: Registers the callback specified by profilers to intercept operations in a specific

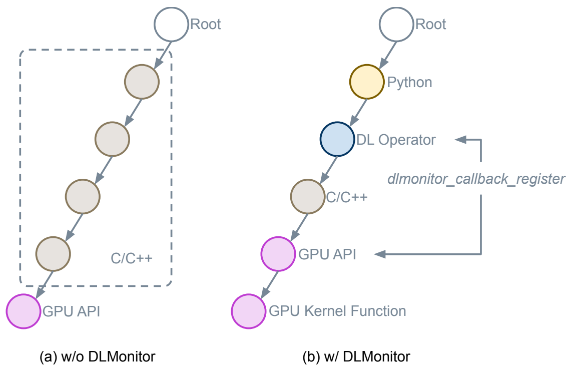


Figure 3. Comparison between call paths w/ and w/o DLMonitor.

domain, such as deep learning frameworks and GPU runtime.
- dlmonitor_finalize: Disables DLMonitor monitoring and releases all interceptions.
- dlmonitor_callpath_get: Constructs and returns a multi-layer call path to the profiler.

In Figure 3, we illustrate how DeepContext provides multi-layer insights that differentiate it from comparable tools like Nvidia Nsight and perf/eBPF. Figure 3 (a) shows the limited call path visibility provided by traditional profilers, which only capture C/C++ frames (native call path) without contextual information about Python code, deep learning framework operations, or GPU kernel execution details. In contrast, Figure 3 (b) demonstrates DeepContext's comprehensive call path construction through DLMonitor, which seamlessly integrates frames from all critical layers: high-level Python code, framework operators, native C/C++ functions, GPU kernel invocations, and even execution metrics within GPU kernels. This multi-layer visibility enables DeepContext to establish precise correlations between high-level operations and low-level performance bottlenecks while supporting runtime metric aggregation across the entire software stack. By maintaining these cross-layer relationships, DeepContext can attribute performance issues to specific framework operators or Python functions—a capability that remains beyond the reach of hardware-focused tools like Nvidia Nsight or system-level profilers like perf/eBPF. In the following, we describe implementation details about how DLMonitor intercepts operations and constructs call paths.

Intercepting Framework Operations. DLMonitor intercepts operations in PyTorch and JAX, allowing profilers to register callbacks using the dlmonitor_callback_register function before and after each operation. Here, the domain provided to the function is DLMONITOR_FRAMEWORK. Interception points include individual deep learning operators (e.g., torch.matmul), the start and end of compute graph compilation, and tensor memory allocation/deallocation. At these points, profilers can access information such as operators inputs and outputs, and retrieve the full call path via the dlmonitor_callpath_get function.

DLMonitor employs different mechanisms to support PyTorch and JAX, ensuring compatibility with frameworks installed via pip wheels *without source code modifications*. For PyTorch, it leverages PyTorch's aten::addGlobalCallback interface, which allows for invoking customizable callbacks at various points. Unlike PyTorch, implementing DLMonitor for JAX presents two challenges. First, JAX does not inherently support registering callbacks before and after each deep learning operator. Second, while PyTorch's eager mode is widely used, where each operator is executed individually, JAX compiles operators into computation graphs with fused operations before execution. Once compiled, the runtime call path of each operator differs from its call path in the original code where it was compiled from. To address these challenges, we implemented a lightweight binary instrumentation utility that allows JAX to provide profiling information comparable to that of PyTorch. Specifically, to address the first challenge, we utilize binary instrumentation to intercept JAX's compilation function for passes of computation graphs and insert callbacks before and after each JAX operator after the very last pass. For the second challenge, we record the mappings between fused operators to original ones (Figure 4) in the operator fusion pass. The call path of each original operator is recorded during the compilation phase, while the call path of the fused operator is recorded at runtime. In the GUI, we display all possible original call paths associated with the runtime call path of each fused operator.

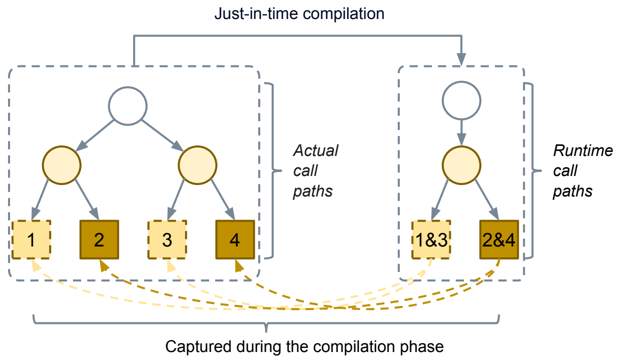


Figure 4. DLMonitor intercepts JAX's compilation phase to map fused operators to original ones.

Intercepting GPU APIs. In addition to intercepting framework operations, DLMonitor can intercept GPU APIs, such as kernel launches, memory copies, and memory allocation/free operations. To enable it, profilers indicate the domain as DLMONITOR_GPU to the dlmonitor_callback_register function. Profilers that register callbacks for GPU APIs can capture not only arguments and results about lower-level APIs, but can obtain frames between the low-level APIs and framework operations. DLMonitor registers callbacks using CUPTI for Nvidia GPUs and RocTracer for AMD GPUs. To extend DLMonitor for hardware that does not have a vendor-provided callback mechanism, users can define the function signature of the driver function and its in a configuration file. DLMonitor will register custom callbacks using LD_AUDIT for all functions recorded in the configuration file.

Call Path Integration. One key innovation of DLMonitor is its ability to assemble a unified call path that spans from high-level Python code to low-level GPU kernel execution. At each interception point, if dlmonitor_callpath_get is called, DLMonitor retrieves call paths from multiple sources. The Python call path is obtained using CPython's PyFrame-related APIs. The "native" call path, which includes C/C++ function symbols, is captured using *libunwind* [38]. The framework call path is maintained via a shadow stack in each CPU thread. DLMonitor updates the stack for operators as they are entered and exited, along with their corresponding memory locations. dlmonitor_callpath_get also allows users to choose which specific call path source to integrate or ignore to reduce overhead.

Next, DLMonitor integrates these three call paths into a single comprehensive call path. It traverses the native call path in a bottom-up direction, matching the address of each frame with the recorded addresses of deep learning operators. If a match is found, DLMonitor inserts the operator name under the caller frame. If a frame's address falls within the libpython.so address space (recorded using LD_AUDIT), all frames above it are replaced with the Python call path. If we are at a GPU kernel launch callback, we read parse the function object (e.g., CUfunction) to obtain the kernel name and insert it to the bottom of the call path.

Optimizations. We have implemented optimizations to provide more insights and reduce the overhead of DLMonitor.

Forward and backward operator association. In PyTorch, when training is enabled, the backward method initiates backward propagation from the leaf operators to the root of the computation graph. New CPU threads, called *backward threads*, are created for each GPU device to handle this process. As a result, the native call path of an operator obtained from a backward thread loses the original context of the forward operator, with no Python source code available. This issue is particularly problematic for GPU kernels without meaningful names, such as the elementwise kernel in PyTorch, as it becomes difficult to trace which source code triggered these calls. To address this, we record each forward operator's call path and sequence ID. In PyTorch, all backward operators associated with a forward operator share the same sequence ID. Using the sequence ID, the backward thread looks up the forward CPU thread, fetches the Python and framework call path of the corresponding forward operator, and integrates it with the native call path of the backward operator obtained from the backward thread itself.

Call path caching. Unwinding call paths from multiple sources is expensive, particularly when GPU APIs are frequently invoked in deep learning workloads. To mitigate this overhead, we leverage the observation that multiple GPU kernels launched within the same deep learning operator typically share identical Python and operator call paths. Based on this insight, we implement a caching mechanism that stores both the Python call path and the deep learning operator information in a thread-local variable upon the operator's first invocation.

To accommodate varying performance requirements, we support two operational modes with different overhead:

1. **Lightweight Mode:** This mode produces a simplified call path that includes Python context while avoiding expensive native unwinding. Since native call path collection is disabled in this mode, we concatenate the shadow call path, GPU API, and GPU kernel function with the cached Python call path from the thread-local variable, bypassing the costly native stack unwinding process entirely.
2. **Comprehensive Mode:** This mode generates a complete cross-language call path that captures both Python and C++ execution contexts. We utilize libunwind's unw_step to retrieve native frames incrementally, constructing the native call path in a bottom-up fashion until we reach the cached deep learning operator. We then combine this native segment with the previously cached Python call path to form a comprehensive execution trace.

4.2 Profiler

In this section, we describe how DeepContext's profiler collects GPU and CPU metrics and attributes them to a calling context tree as shown in Figure 5.

Calling Context Tree. The calling context tree is constructed by inserting call paths obtained from DLMonitor and collapsing frames that refer to the same locations. For C/C++, GPU API, and GPU kernel frames, two frames are considered the same if they share the same library path and program counter (PC). For Python frames, they are compared by file path and line number, while framework-based frames are checked by operator names. Each node in the calling context tree maintains a list of metrics, which are aggregated by sum, minimum, average, and standard deviation for metrics of the same type. Once a metric has been updated at the bottom of a call path, it is propagated to the root node of the

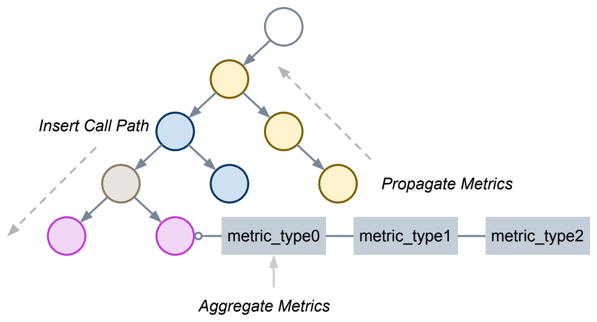


Figure 5. Operations for building a calling context tree and adding performance metrics.

calling context tree, updating the metric along the entire call path.

GPU Metrics. DeepContext can collect both coarse- and fine-grained metrics on Nvidia and AMD GPUs. Example coarse-grained metrics include time, parallelism, and shared memory usage, and fine-grained metrics include instruction samples of GPU kernels. It first registers callbacks for GPU APIs using DLMonitor, then specifies the metrics to profile by calling CUPTI or RocTracer APIs. At each callback, the profiler emits a unique correlation ID, retrieves the call path, and associates the correlation ID with the call path. GPU metrics are gathered asynchronously without blocking GPU API calls from the CPU. When the GPU buffer storing metrics is full, DeepContext flushes the metrics, using the correlation ID to link and aggregate them with the corresponding call path. Note that if fine-grained metrics, such as instruction samples, are collected, we will extend the call path by inserting the PC of each instruction collected.

CPU Metrics. DeepContext can profile GPU metrics and CPU metrics in the same run using Linux system calls or CPU measurement substrates. For example, DeepContext invokes the `sigaction` system call to registers a signal callback for `CPU_TIME` and `REAL_TIME` events. Once a sample is triggered, it will get the current CPU or REAL time, subtract the previous timestamp from it, and use the result as the interval between two samples. Next, DeepContext will obtain the current call path by calling `dlmonitor_callpath_get` and associate the interval with the call path. The profiler can also register Linux perf events or invoke PAPI API to obtain metrics from hardware counters.

4.3 Performance Analyzer

The performance analyzer of DeepContext provides a comprehensive framework for analyzing the profile results and identifying potential performance issues in deep learning workloads. It initializes the analysis environment by mapping GPU/CPU instructions back to the source code using the DWARF information. We have designed a flexible Python interface allows analysis from three key dimensions, typically executed through a workflow of *call path search*, *metrics analysis*, and *visualization*. The process starts with ① *Program Structure Analysis*, which traverses calling contexts and matches call paths given patterns. This is followed by ② *Model Analysis* and ③ *Operator Analysis*, where performance metrics are queried for semantic nodes (such as training, inference, loss functions, and individual operators) and custom filters are applied to detect potential issues. Finally, the detected problems are flagged with warning messages and reported in the GUI.

Example Analyses. Beyond custom analysis created by users, DeepContext implements a set of example analyses to detect common performance issues using the analysis API. Below we demonstrate some of the example analyses:

❶ *Hotspot Identification*
This analysis identifies the nodes that spend more time than a given threshold and returns their call paths.

```
total_time = call_tree.root.time
for n in call_tree.kernels:
  if n.time / total_time > hotspot_threshold:
    flag_hotspot(n)
```

❷ *Kernel Fusion Analysis*
This analysis detects potential inefficiencies caused by the launch of many small kernels by identifying frames that contain a large number of kernels with short GPU execution times.

```
for n in bfs(call_tree.nodes):
  if n.gpu_time / n.count < gpu_threshold:
    flag_issue(n, "Small␣GPU␣kernels")
```

❸ *Forward/Backward Operator Analysis*
This analysis identifies deep learning operators whose backward pass takes significantly longer than the forward pass, potentially indicating optimization opportunities because backward phase shouldn't take significantly longer than its forward counterpart.

```
for n in call_tree.operators:
  if n.backward.time / n.forward.time > 2:
    flag_issue(n, "Backward␣abnormality")
```

❹ *Fine-grained Stall Analysis*
This analysis identifies fine-grained stall reasons within hotspot GPU kernels. Stall reasons for each GPU instruction can be collected using the instruction sampling APIs available on Nvidia and AMD GPUs.

```
hotspots = hotspot_analysis(call_tree)
stalls = []
for n in hotspots:
  for c in n.children:
    if c.stalls > stall_threshold:
      stalls.append(c)
stall_reasons = topk(stalls)
flag_issue(n, "Kernel␣is␣mainly␣stalled␣by␣{
  stall_reasons}")
```

❺ *CPU Latency Analysis*
This analysis traverses the calling context tree in the top-down manner to identify frames whose CPU time is significantly higher than GPU time, indicating potential imbalanced workload or synchronization issues.

```
for n in bfs(call_tree.nodes):
  if n.cpu_time / n.gpu_time > cpu_threshold:
    flag_issue(n, "CPU␣time␣abnormality")
```

4.4 GUI

DeepContext integrates a lightweight GUI built on the VSCode API, supporting VSCode and other compatible IDEs such as VSCodium [2] and Eclipse Theia [1]. It features interactive flame graphs [23] with both top-down and bottom-up views for visualizing calling context trees, and highlights performance hotspots identified by the performance analyzer. Users can click on frames to inspect detailed metrics and navigate directly to relevant source code, streamlining the performance debugging process.

4.5 API and Hardware Adaptability

DeepContext's modular architecture enables efficient adaptation to evolving frameworks and hardware through clear separation of component dependencies. Referring to the component division shown in Figure 2, DLMonitor component only requires updates when framework APIs change, while the Profiler component is updated only when hardware profiling interfaces evolve. Meanwhile, the Performance Analyzer and GUI remain framework and hardware-agnostic, operating solely on abstracted performance data. This targeted update approach minimizes maintenance overhead as deep learning frameworks like PyTorch and JAX frequently modify their internal structures.

DeepContext achieves robust cross-generation hardware support through abstracted profiling interfaces that leverage common APIs across GPU generations (CUPTI for Nvidia, RocTracer for AMD). The Profiler's abstraction layer maps vendor-specific metrics to a unified internal representation, automatically adapting to hardware capabilities at runtime. Additionally, DeepContext supports extension mechanisms including configuration-based hardware support through `LD_AUDIT`, a plugin architecture for new metric collection mechanisms, and framework-agnostic analysis patterns. This modular design ensures DeepContext can evolve alongside the rapidly changing landscape of deep learning frameworks and hardware accelerators while maintaining long-term viability.

5 Evaluation

Platforms. We evaluated DeepContext's overhead on two platforms: one with an AMD EPYC 7543 CPU, 256 GB RAM, and an NVIDIA A100 SXM GPU (80 GB, 108 SMs, 156 TF32 TFLOP/s, 2 TB/s bandwidth), and the other with the same CPU, 2048 GB RAM, and an AMD MI250 GPU (64 GB, 208 Compute Units, 362.1 FP16 TFLOP/s, 3.2 TB/s bandwidth).

Workloads. We used DeepContext to profile the MLCommons Algorithm Efficiency benchmark [16], implemented in both PyTorch [6] and JAX [11]. We evaluated the eager mode of PyTorch and the JIT mode of JAX. We run each model for 100 iterations using different profiling tools.

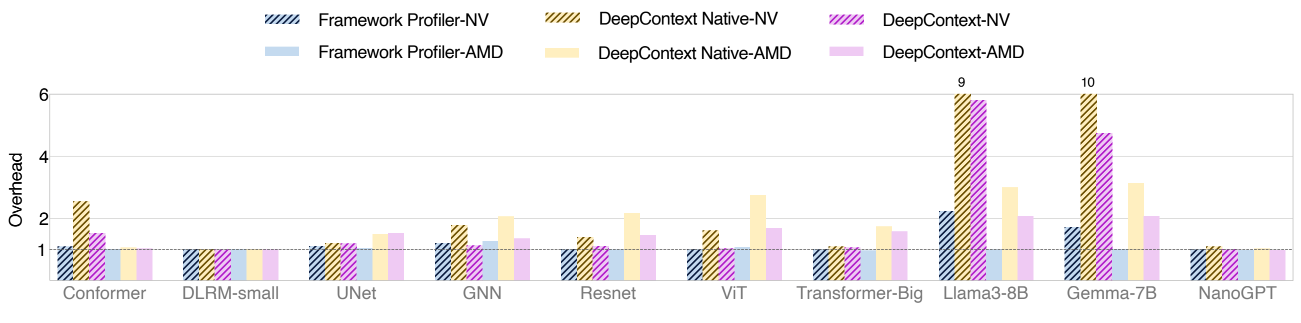
The following workloads and datasets were used.

- Conformer [25] with the LibriSpeech [44] dataset.
- DLRM-small [39] with the Criteo 1TB [15] dataset.
- U-Net [48] with the fastMRI [57] dataset.
- GNN [7] with the OGBG-MOLPCBA [28] dataset.
- ResNet [26] with the ImageNet [17] dataset.
- Vision Transformer [18] with the ImageNet dataset.
- Transformer-Big [54] with the WMT[10] dataset.
- Llama 3 [19] inference with a sample prompt from huggingface official example.
- Gemma [52] with the same prompt as Llama 3.
- nanoGPT [34] with the same prompt as Llama 3.

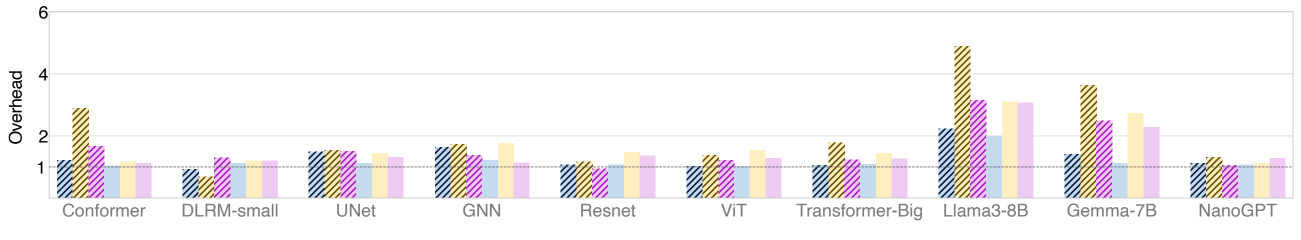
Results. We measured the end-to-end running time of each workload under three circumstances: without profiler enabled, DeepContext with Python and framework call paths obtained from DLMonitor, and DeepContext with Python, deep learing framework, and native C/C++ call paths. Then we divide the running time of DeepContext enabled by the running time without DeepContext enabled to calculate the overhead, as shown in Figure 6.

We measured the end-to-end running time of each workload under four circumstances: without profiler enabled, with framework profiler (PyTorch profiler or JAX profiler) enabled, DeepContext operating in Lightweight Mode (collecting only Python and framework call paths), and DeepContext operating in Comprehensive Mode (collecting Python, deep learning framework, and native C/C++ call paths). Then we divide the running time of DeepContext or framework profiler enabled by the running time without profiler enabled to calculate the overhead, as shown in Figure 6. In the figure, "Framework Profiler" refers to the PyTorch profiler in PyTorch workload tests and the JAX profiler in JAX workload tests, "DeepContext" corresponds to our tool running in Lightweight Mode, and "DeepContext Native" represents our tool operating in Comprehensive Mode.

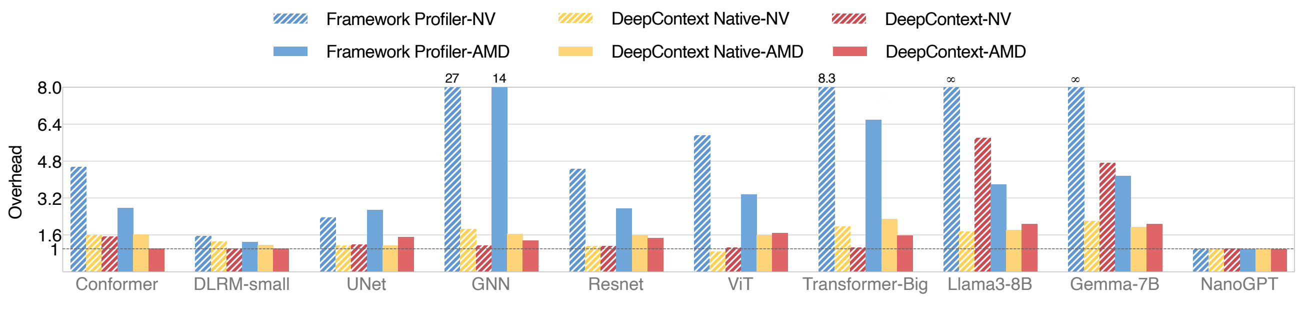
The median running time overhead of DeepContext is 1.12× and 1.50× for PyTorch on Nvidia and AMD GPUs, respectively. For JAX, its median overhead is 1.33× and 1.28× on Nvidia and AMD GPUs, respectively. When the C/C++ native call path is not collected, we observed median overheads of 1.50× and 1.90× for PyTorch, and 1.60× and 1.46× for JAX, on Nvidia and AMD GPUs, respectively. The overhead with native call path is higher than the variant without the native call path due to the additional overhead in unwinding



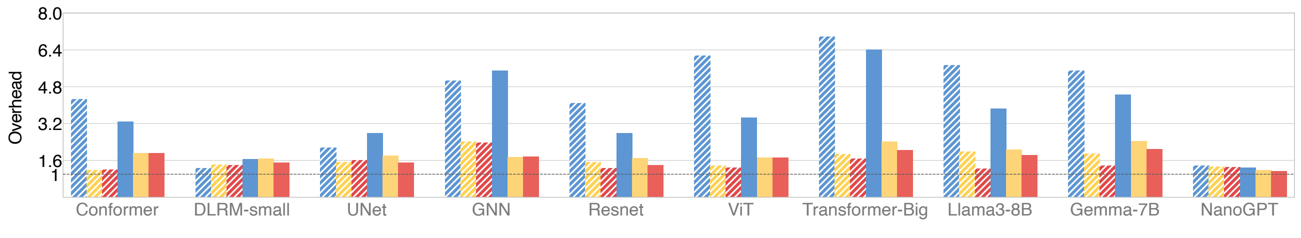
(a) Time overhead of PyTorch workloads using different profilers on Nvidia and AMD GPUs.



(b) Time overhead of JAX workloads using different profilers on Nvidia and AMD GPUs.



(c) Memory overhead of PyTorch workloads using different profilers on Nvidia and AMD GPUs.



(d) Memory overhead of JAX workloads using different profilers on Nvidia and AMD GPUs.

Figure 6. Comparison of time and memory overheads (normalized to baseline without profiling) across various workloads using DeepContext, the PyTorch profiler, and the JAX profiler.

C/C++ call paths and concatenating them with Python and framework call paths.

The median running time overhead of DeepContext in Lightweight Mode is 1.12× and 1.50× for PyTorch on Nvidia and AMD GPUs, respectively. For JAX, its median overhead in Lightweight Mode is 1.33× and 1.28× on Nvidia and AMD GPUs, respectively. When operating in Comprehensive Mode, we observed median overheads of 1.50× and 1.90× for PyTorch, and 1.60× and 1.46× for JAX, on Nvidia and AMD GPUs, respectively.

In comparison, PyTorch profiler incurs a median overhead of 1.06× and 1.01× on Nvidia and AMD GPUs, respectively. JAX profiler incurs a median overhead of 1.17× and 1.10× on Nvidia and AMD GPUs, respectively. Without native call path collection, the overhead of DeepContext is comparable to that of framework profilers. We do observe a much higher time overhead from profiling Llama3 and Gemma-7B using PyTorch. The overhead is caused by two factors: our frame unification system, which identifies the same file path

Table 2. Summary of case studies.

Deep Learning Model	Dataset	Platform	Analysis Client	Optimization Method	Speedup
DLRM-small	Criteo 1TB	Nvidia	❸ Forward/Backward Operator Analysis	replace `aten::index` with `aten::index_select`	1.66×
GNN	OGBG-MOLPCBA	Nvidia	❸ Forward/Backward Operator Analysis	replace `aten::index` with `aten::index_select`	1.07×
UNet	fastMRI	Nvidia	❶ Hotspot Identification	Avoid `channel_first` to `channel_last` conversion	1.28×
UNet	fastMRI	Nvidia	❺ CPU Latency Analysis	Match `worker_num` with `#CPU cores`	1.15×
Transformer-Big	WMT	Nvidia	❷ Kernel Fusion Analysis	Fuse small kernels using `torch.compile`	1.06×
Llama3	Sample Prompt	Nvidia	❹ Fine-grained Stall Analysis	Use fast data type conversion instructions	1.25×
UNet	fastMRI	AMD & Nvidia	❶ Hotspot Identification	Use pinned memory	N/A
DLRM-small UNet	Criteo 1TB fastMRI	Nvidia-JAX Nvidia-PyTorch	❷ Kernel Fusion Analysis	Fuse small kernels using `torch.compile`	1.11× 1.02×

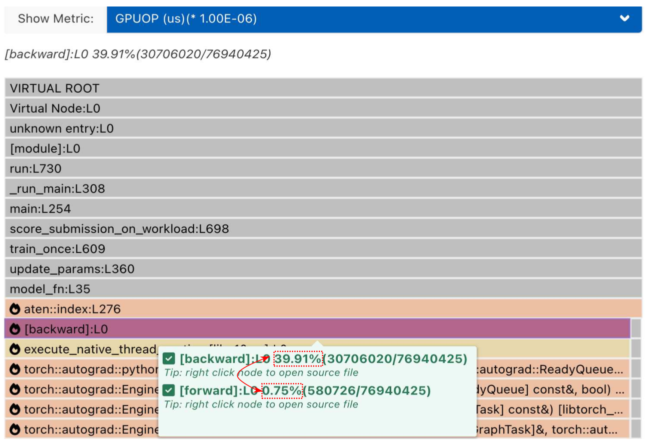
and line number, and our metrics aggregation and propagation mechanism along the call paths, introduces additional overhead. These two factors are especially significant with such workloads launch many small kernels.

The median memory overhead of DeepContext is 1.00×-2.44×, compared with that of 1.29×-27.28× and 1.27×-6.98× of PyTorch and JAX profilers, respectively. Note that the memory overhead of the framework profilers will increase with the increase in the number of iterations. Also, the PyTorch profiler encountered out-of-memory issues when exporting the profiling database to disk, failing to provide any insights for optimization. DeepContext incurs significant lower memory overhead compared to these tools because it aggregates metrics at runtime and thus is more feasible for long-running workloads.

6 Insights obtained by DeepContext

In this section, we present seven performance optimization cases conducted using additional insights obtained from DeepContext, which were not discoverable through vendor-specific profiling tools alone. By comparing DeepContext's cross-layer analysis capabilities with industry-standard tools like Nvidia Nsight Systems, Nsight Compute, and AMD rocprof—each selected as the most capable tool for its respective case—we demonstrate how DeepContext's unified view of Python code, framework operations, and hardware execution reveals critical optimization opportunities that remain hidden when using conventional profilers. These cases span diverse workloads including DLRM [39], U-Net [48], Transformer [54], GNN [7], and Llama3 [19] models across both Nvidia and AMD platforms, resulting in performance improvements of 6-40%, as summarized in Table 2. For each case, we highlight the specific advantage of DeepContext's multi-layer approach over the most relevant vendor tool, focusing on the comparison that best illustrates DeepContext's added value.

6.1 Forward/backward Operator Analysis



```
Show Metric: GPUOP (us)(* 1.00E-06)

[backward]:L0 39.91%(30706020/76940425)

VIRTUAL ROOT
Virtual Node:L0
unknown entry:L0
[module]:L0
run:L730
_run_main:L308
main:L254
score_submission_on_workload:L698
train_once:L609
update_params:L360
model_fn:L35
aten::index:L276
[backward]:L0
execute_native_thread
torch::autograd::python                    :autograd::ReadyQueue...
torch::autograd::Engine                    dyQueue] const&, bool) ...
torch::autograd::Engine                    Task] const&) [libtorch_...
torch::autograd::Engine                    raphTask]&, torch::aut...

[backward]:L0 39.91%(30706020/76940425)
Tip: right click node to open source file
[forward]:L0 0.75%(580726/76940425)
Tip: right click node to open source file
```

Figure 7. Forward-Backward association view of the DLRM-small workload.

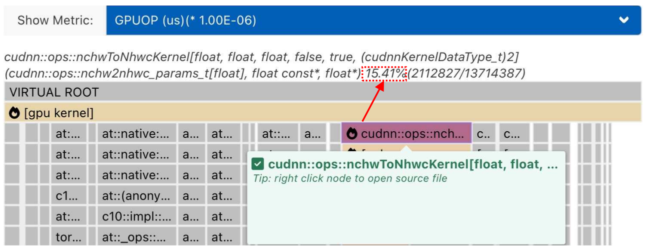
We profiled the DLRM-small workload using the Criteo 1TB dataset on the A100 platform. In DeepContext's bottom up view, we noticed that the hotspot is on the `indexing_backward_kernel` kernel (30.5s), which takes 39.6% of the total GPU kernel time. Using the DeepContext's framework call path, which associates the forward call path with the corresponding backward kernels, we found that this GPU kernel is triggered by the backward computation of `aten::index` called by `embedding_table[idx_lookup]`, as illustrated in Figure 7.

It should be noted that while the backward computations of `aten::index` take 39.9% of the total time, the forward computation takes only 0.8% of the total time. This discrepancy is caused by the deterministic nature [24] of `aten::index`, which serializes GPU threads accessing the same memory location and is unnecessary in this workload if determinism is not required. To optimize the code, we substituted `aten::index` with a non-deterministic operator `aten::index_select`, which uses atomic operations in the backward phase to avoid serialization and **reduced the total GPU time from 73.2s to 44.0s**. We also observed the same problem in the GNN workload; applying the same optimization **reduced the total GPU time from 3.97s to 3.71s**.

Comparison with Nsight Systems. Although Nsight Systems can capture the `indexing_backward_kernel` as a GPU hotspot, it does not automatically correlate backward kernels with their corresponding forward operators in a unified top-down or bottom-up hierarchy. Consequently, users must manually traverse event traces to identify the invocation context of the corresponding forward operator `aten::index` (with DL framework expertise) to make optimizations. In contrast, DeepContext integrates framework call paths, providing a direct link between the backward kernel and its forward invocation site, facilitating the determination of the feasibility of non-deterministic optimization and streamlining the modification of the source code.

6.2 Hotspot Identification with Call Path

When profiling U-Net using the fastMRI dataset on the A100 platform, we observed that the `cudnn::nchwToNhwcKernel` kernel takes 15.4% of the GPU time. Using DeepContext's


Show Metric: GPUOP (us)(* 1.00E-06)

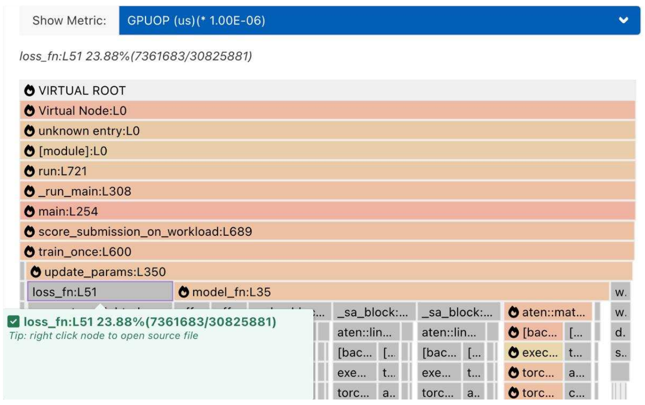
cudnn::ops::nchwToNhwcKernel[float, float, float, false, true, (cudnnKernelDataType_t)2](cudnn::ops::nchw2nhwc_params_t[float], float const*, float*) 15.41%(2112827/13714387)

cudnn::ops::nchwToNhwcKernel[float, float, ...

Tip: right click node to open source file

Figure 8. The bottom-up view of U-Net.

framework and Python call paths, we identified every PyTorch operator that invokes the conversion. In addition, with the help of native call paths, we also identified that the input tensor's memory format is converted from PyTorch's default `channels_first` layout [53] to the `channels_last` layout—a layout that is more efficient for CUDNN—and then reverted back to `channels_first` after the computations, introducing excessive overhead. To address memory format


Show Metric: GPUOP (us)(* 1.00E-06)

loss_fn:L51 23.88%(7361683/30825881)

loss_fn:L51 23.88%(7361683/30825881)

Tip: right click node to open source file

Figure 9. Top-down view of Transformer-Big.

conversion issues, we optimized the code by storing input tensors with `channels_last` layout before computations, and refactored LayerNorm and InstanceNorm layers to store weights in the `channels_last` layout to avoid conversion. This optimization reduced the **end-to-end time of 100 iterations from 54s to 42s**.

Comparison with Nsight Systems. While Nsight Systems can display CUDA kernels (such as `cudnn::nchwToNhwcKernel`) in its trace timeline, it does not associate them with PyTorch operators that invoke these kernels. As a result, users have to manually locate where these conversion kernels are triggered in the timeline and correlate them with PyTorch operators based on the kernels invoked before and after, as well as the events that happened on the CPU, demanding considerable manual effort and deep DL expertise. In contrast, DeepContext associates each kernel with its PyTorch and native C/C++ call path in a unified call path to provide insights about where and why layout conversion happens, expediting the diagnosis of low-level performance issues.

6.3 Kernel Fusion Analysis

Using DeepContext, we profiled the Transformer-Big workload on the A100 platform. DeepContext can gather multiple metrics in a single run, such as the number of invocations, the number of warps and blocks, as well as the number of shared memory and registers, in addition to the GPU time. These metrics are attributed to the corresponding frames in the call paths to assist performance analysis.

For instance, from the top-down view in Figure 9, we observed that `loss_fn` takes 7.36s, which is 23.9% of total time. Under this frame, there are three different kernels invoked: including `softmax`, `copy`, and `nll_loss`, each with the same number of invocations. The kernel fusion analysis suggests an opportunity for optimization by combining small kernels to reduce overall time. Further analysis expanding the call paths also reveals that the `softmax` kernel has relatively low register usage, which implies that fusing this kernel will not cause significant register overhead. Based on the suggestion and observation, we manually fused these small kernels into a single and more efficient kernel. After optimization, the **total GPU time is decreased from 30.5s to 23.9s.**

Comparison with Nsight Systems. When profiling the Transformer-Big workload using Nsight Systems, small kernels like `softmax` or `copy` may appear as fragmented events in the trace view. Without explicit NVTX annotations [42] around `loss_fn`, it is labor-intensive to cluster these kernels and identify a fusion opportunity. In contrast, DeepContext not only flags these sub-kernels under the same Python call path with its automated kernel fusion analysis but also reports their resource usage collectively, making it evident that fusion can be beneficial and does not exceed register limits.

6.4 CPU Latency Analysis

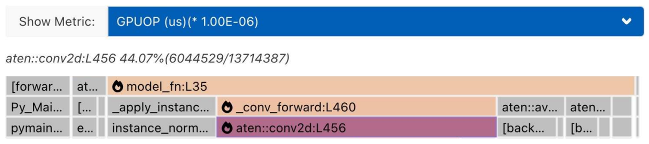
We enabled both CPU and GPU metrics to profile the U-Net workload. The CPU latency analysis highlighted that the call path to the `data_selection` function takes 69% of the CPU time with 16 threads running concurrently, while the GPU time of the same frame is only 1.3 seconds. Further investigation shows that the first iteration of loading data from the disk to the memory takes 10 seconds, and the GPU remains idle. By expanding the call paths of `data_selection`, we noted that an inefficient setting of parallel threads has been invoked. Our allocated node only has 6 physical CPU cores, but the data loader is hard coded with 16 threads to load the data, causing significant scheduling overhead. After we reduced the thread number to 8, we reduced **the end-to-end time of 100 iterations by 7s, from 54s to 47s**.

Comparison with Nsight Systems. Although Nsight Systems offers CPU profiling features, correlating CPU events directly with GPU idle times typically requires inspection across multiple traces and manual alignment. DeepContext automatically integrates both CPU and GPU metrics within their respective contexts, enabling users to immediately identify where CPU inefficiencies cause GPU idle periods.

6.5 AMD *vs* Nvidia

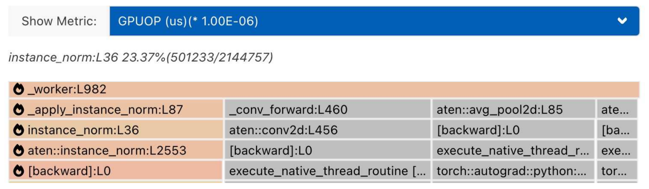
We profiled the U-Net workload on both AMD and Nvidia GPUs. From the top-down view shown in Figure 10, we can see differences between these two platforms. Figure 10a shows that on Nvidia GPUs, the performance hotspot is on the convolution operator `aten::conv2d`; on the other hand, Figure 10b shows that on AMD GPUs, the performance hotspot lies on the instance norm operator `aten::instance_norm`. In order to find out the cause of performance degradation on AMD GPU, we checked the low-level call paths of `aten::instance_norm` provided by DeepContext and found that the implementation of `aten::instance_norm` for AMD GPUs provided by PyTorch reused the same kernel template—`batch_norm_backward_cuda_template`—as that for Nvidia GPUs [12]. However, through further investigation, we observed that performance can vary drastically when using different PyTorch versions (i.e., 2.7.0 and 2.6.0) and ROCm versions (i.e., 6.3, 6.2, and 6.0). Performance regressions were observed, where using newer versions of software can yield up to 18.29× slowdown on this kernel compared to older versions. While DeepContext can detect that this kernel suffers from significant memory stalls, it does not provide sufficient visibility into the exact underlying cause, which could involve page faults or other system-level issues. Using pinned memory can offer a workaround to achieve performance comparable to Nvidia GPUs. We also employed other tools, including rocprof and PyTorch profiler, but they did not offer additional clarity into the performance discrepancy as well. We have reported this issue to AMD for further investigation.

Comparison with rocprof. Nsight Systems is primarily optimized for Nvidia GPUs, whereas its counterpart, rocprof, lacks an equivalent visualization view. This discrepancy makes it challenging to compare performance across different GPUs. Furthermore, while rocprof can collect basic CPU and GPU events, it lacks call path correlation with Python and C/C++ native frames. DeepContext provides consistent multi-layer profiling across both AMD and Nvidia backends, enabling an apples-to-apples comparison of operator hotspots such as `instance_norm`.


Show Metric: GPUOP (us)(* 1.00E-06)

aten::conv2d:L456 44.07%(6044529/13714387)

(a) Flame Graph of Nvidia GPU. Hotspot is on operator `aten::conv2d`, which is as expected.


Show Metric: GPUOP (us)(* 1.00E-06)

instance_norm:L36 23.37%(501233/2144757)

(b) Flame Graph of AMD GPU. Hotspot is on operator `aten::instance_norm`, which is abnormal.

Figure 10. Performance comparison on AMD and Nvidia.

6.6 JAX *vs* PyTorch

We compared the performance of JAX and PyTorch across two datasets/models: DLRM-small and U-Net. Our results show that JAX significantly outperforms PyTorch in all tasks, achieving performance improvements exceeding 50%. By comparing the number of kernel operations, we observed that the JAX version consistently requires fewer operations than its PyTorch counterpart. This substantial performance gap is primarily attributed to the advantages of JAX's XLA compiler, which effectively fuses operators to reduce redundant memory access and overlap compute and memory instructions. To narrow this performance gap, we can use `torch.compile` to optimize some workloads, and **we observed a** 1.11× **speedup and a** 1.02× **speedup in DLRM-small and UNet, respectively.** However, we observed that it cannot successfully compile all PyTorch modules, yields slowdowns in cases such as GNN and ResNet, and may cause significant autotuning overhead, especially with the `max-autotune` mode.

Comparison with Nsight Systems. While Nsight Systems can reveal that JAX and PyTorch versions execute different kernels with varying counts, it provides no mechanism to correlate these differences with high-level framework operations or optimization strategies. Users must manually analyze kernel patterns to hypothesize potential optimization opportunities, requiring specialized knowledge of both frameworks. In contrast, DeepContext's automated analysis compares average kernel execution times and identifies patterns indicative of suboptimal kernel fusion in the PyTorch implementation. This cross-framework analysis capability enables DeepContext to automatically recommend specific optimization approaches for PyTorch workloads based on observed patterns in JAX's execution, effectively bridging the semantic gap between different framework implementations.

6.7 Fine-grained Stall Analysis

We profiled the Llama3 workload running low-precision including `float16` and `float8` using fine-grained instruction sampling. On both AMD and Nvidia GPUs, we have identified time spent on the data conversion operators (i.e., `torch.to`) in the `LlamaRMSNorm` module [20], due to the fact that the variance calculation has to be done in `float32` for numeric stability [51]. The fine-grained stall analysis identifies non-trivial constant memory misses due to the loading of constants for each CTA. Since the input is small, there is a relatively high overhead in reading the constant memory compared to loading the data itself. Additionally, we observed math dependency-related stalls caused by non-vectorized data conversion instructions from/to `float32`. To optimize the kernel, we can (1) ensure that each block loads the minimum number of bytes required to use vectorized data conversion instructions, and (2) fuse the conversion operator with other operators to ensure that the constant memory overhead is minimized. We adopted the former to enable vectorized data conversions between float32 and float16/float8 and achieved non-trivial speedups. For example, **the conversion from float8 to float32 is 3.47× faster with each thread loading and converting four float8 elements simultaneously, with the `LlamaRMSNorm` module improved by 1.25×.** On Nvidia Ada or later architectures, we can further adopt `cvt` with `e5m2x2` [43] to first convert multiple float8 elements using a single instruction without workarounds such as byte permutation.

Comparison with Nsight Compute (NCU). Both NCU and DeepContext can collect instruction samples, but NCU is approximately 100×[1] slower than DeepContext for several reasons. First, NCU synchronizes the CPU and GPU after each GPU kernel instance, whereas DeepContext avoids synchronization and parses binary information offline. Additionally, NCU often applies kernel replay to collect additional metrics along with instruction samples. The profiling view in NCU displays each kernel instance separately without aggregating metrics, making it difficult to identify performance hotspots in source code. In contrast, the context view in DeepContext clearly reveals that all kernels in the `LlamaRMSNorm` kernel are affected by data conversion, leading to an increased memory and compute overhead.

7 Limitations and Future Work

While DeepContext offers rich profiling for deep-learning workloads, its primary limitation is the overhead incurred when unwinding call paths for workloads with many small kernels. We will mitigate this by adopting call-path caching techniques [60]. In the future, we also plan to extend DeepContext to PyTorch workloads that employ `torch.compile`, leveraging the JAX-style tracing already in place to capture operator call paths.

8 Conclusions

DeepContext addresses a critical gap in performance profiling for deep learning workloads in heterogeneous computing environments, where the interaction between CPUs, GPUs, and deep learning frameworks is inherently complex. DeepContext fulfills this need by providing a multi-level, automated analysis that bridges the different layers of the software and hardware stack. Our detailed case studies and evaluations show that DeepContext improves the ability to identify and resolve performance bottlenecks in deep learning workflows.

Acknowledgments

We would like to thank George Mason University and AMD for providing the compute facility. This work is supported by a donation from AIGCSEMI LLC and by NSF Awards 2411134, 2411136, and 2417469.

[1]We only compared the two tools in this case but not all due to the extremely high overhead caused by NCU.

References

[1] 2024. Eclipse Theia: Cloud and Desktop IDE Platform. https://theia-ide.org/. Accessed: 2024-10-17.

[2] 2024. VSCodium: Open-Source Visual Studio Code Without Microsoft Branding. https://vscodium.com/. Accessed: 2024-10-17.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023).

[4] AMD. 2025. Using rocprofv3. https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/how-to/using-rocprofv3.html. Accessed: 2025-03-02.

[5] AMD Inc. 2024. ROCm Tracer (Roctracer). https://github.com/ROCm-Developer-Tools/roctracer. Accessed: 2024-10-12.

[6] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. (2024).

[7] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261 (2018).

[8] Emery D Berger. 2020. Scalene: Scripting-language aware profiling for python. arXiv preprint arXiv:2006.03879 (2020).

[9] Bimal Biswal. [n. d.]. Monkey Patching in Python. Software Technology Tips ([n. d.]). Archived from the original on 22 August 2012. Retrieved 9 December 2013.

[10] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Shujian Huang, Matthias Huck, Philipp Koehn, Qun Liu, Varvara Logacheva, et al. 2017. Findings of the 2017 conference on machine translation (wmt17). Association for Computational Linguistics.

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax

[12] PyTorch Contributors. 2024. Normalization.cuh. https://github.com/pytorch/pytorch/blob/44483972bdd3dcd0c047020694817210846b5d70/aten/src/ATen/native/cuda/Normalization.cuh#L356. Accessed: 2024-06-26.

[13] NVIDIA Corporation. 2021. NVIDIA DLProf: Deep Learning Profiler. https://developer.nvidia.com/DLProf. Accessed: 2024-10-16.

[14] NVIDIA Corporation. 2024. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. Accessed: 2024-10-16.

[15] Criteo A. I. Lab. 2014. Criteo 1TB Click Logs dataset. https://labs.criteo.com/2014/12/download-terabyte-click-logs/.

[16] George E. Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, Juhan Bae, Justin Gilmer, Abel L. Peirson, Bilal Khan, Rohan Anil, Mike Rabbat, Shankar Krishnan, Daniel Snider, Ehsan Amid, Kongtao Chen, Chris J. Maddison, Rakshith Vasudev, Michal Badura, Ankush Garg, and Peter Mattson. 2023. Benchmarking Neural Network Training Algorithms. arXiv:2306.07179

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition. Ieee, 248–255.

[18] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929 (2020).

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

[20] Hugging Face. 2024. LLaMA Model Implementation in Transformers Library. https://github.com/huggingface/transformers/blob/main/src/transformers/models/llama/modeling_llama.py#L69. Accessed: 2024-10-19.

[21] Python Software Foundation. 2024. cProfile: Python Code Profiler. https://docs.python.org/3/library/profile.html. Accessed: 2024-10-16.

[22] James Gleeson, Moshe Gabel, Gennady Pekhimenko, Eyal de Lara, Srivatsan Krishnan, and Vijay Janapa Reddi. 2021. RL-Scope: Cross-stack profiling for deep reinforcement learning workloads. Proceedings of Machine Learning and Systems 3 (2021), 783–799.

[23] Brendan Gregg. 2016. The flame graph. Commun. ACM 59, 6 (2016), 48–57.

[24] Sam Gross. 2020. Issue #41162: Advanced indexing gradient is extremely slow when there are many duplicate indices. https://github.com/pytorch/pytorch/issues/41162#issuecomment-655834491. Meta.

[25] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. 2020. Conformer: Convolution-augmented transformer for speech recognition. arXiv preprint arXiv:2005.08100 (2020).

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

[27] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. arXiv preprint arXiv:2203.15556 (2022).

[28] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. Advances in neural information processing systems 33 (2020), 22118–22133.

[29] Intel Corporation. 2024. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. Accessed: 2024-10-12.

[30] IO Visor Project. 2024. BCC (BPF Compiler Collection). https://github.com/iovisor/bcc. Accessed: 2024-03-10.

[31] IO Visor Project. 2024. bpftrace: High-level tracing language for eBPF. https://github.com/iovisor/bpftrace. Accessed: 2024-03-10.

[32] JAX Team. 2024. JAX Profiler. https://jax.readthedocs.io/en/latest/profiler.html. Accessed: 2024-10-12.

[33] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 (2020).

[34] Andrej Karpathy. 2022. NanoGPT. https://github.com/karpathy/nanoGPT.

[35] Cheng Li, Abdul Dakkak, Jinjun Xiong, Wei Wei, Lingjie Xu, and Wen-mei Hwu. 2020. XSP: Across-stack profiling and analysis of machine learning models on GPUs. In 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 326–327.

[36] Linux Kernel Organization. 2024. Linux Perf: Performance Monitoring for Linux. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2024-10-12.

[37] Microsoft Corporation. 2024. Visual Studio Code. https://code.visualstudio.com. Accessed: 2024-10-12.

[38] David Mosberger. [n. d.]. Libunwind. http://www.nongnu.org/libunwind. Accessed: 2024-10-12.

[39] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit

Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. arXiv preprint arXiv:1906.00091 (2019).

[40] NVIDIA Corporation. 2024. NVIDIA CUPTI: CUDA Profiling Tools Interface. https://developer.nvidia.com/cupti. Accessed: 2024-10-12.

[41] NVIDIA Corporation. 2024. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2024-10-12.

[42] NVIDIA Corporation. 2025. NVIDIA Tools Extension (NVTX) Library. https://docs.nvidia.com/tools-extension/index.html. Accessed: 2025-03-02.

[43] NVIDIA Corporation. 2025. Parallel Thread Execution ISA: Data Movement and Conversion Instructions (cvt). https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-cvt. Accessed: 2025-03-02.

[44] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: an asr corpus based on public domain audio books. In 2015 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE, 5206–5210.

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

[46] William Peebles and Saining Xie. 2023. Scalable diffusion models with transformers. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 4195–4205.

[47] PyTorch Team. 2024. PyTorch Profiler. https://pytorch.org/tutorials/recipes/recipes/profiler.html. Accessed: 2024-10-12.

[48] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18. Springer, 234–241.

[49] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. The International Journal of High Performance Computing Applications 20, 2 (2006), 287–311.

[50] Daniel Snider, Fanny Chevalier, and Gennady Pekhimenko. 2023. Hotline Profiler: Automatic Annotation and A Multi-Scale Timeline for Visualizing Time-Use in DNN Training. Proceedings of Machine Learning and Systems 5 (2023), 104–126.

[51] stas00. [n. d.]. [Feature Request] Implement RMSNorm Fused CUDA Kernel. https://github.com/NVIDIA/apex/issues/1271. Accessed: 2025-03-02.

[52] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. arXiv preprint arXiv:2403.08295 (2024).

[53] PyTorch Team. 2024. Memory Format Tutorial. https://pytorch.org/tutorials/intermediate/memory_format_tutorial.html. Accessed: 2024-06-23.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. Advances in neural information processing systems 30 (2017).

[55] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring energy and power with PAPI. In 2012 41st international conference on parallel processing workshops. IEEE, 262–268.

[56] Geoffrey X Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive in-editor computational performance profiling for deep neural network training. In Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology. 126–139.

[57] Jure Zbontar, Florian Knoll, Anuroop Sriram, Tullie Murrell, Zhengnan Huang, Matthew J Muckley, Aaron Defazio, Ruben Stern, Patricia Johnson, Mary Bruno, et al. 2018. fastMRI: An open dataset and benchmarks for accelerated MRI. arXiv preprint arXiv:1811.08839 (2018).

[58] Qidong Zhao, Xu Liu, and Milind Chabbi. 2020. DrCCTProf: a fine-grained call path profiler for ARM-based clusters. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–16.

[59] Keren Zhou, Laksono Adhianto, Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Mark Krentel, Yumeng Liu, Xiaozhu Meng, and John Mellor-Crummey. 2021. Measurement and analysis of GPU-accelerated applications with HPCToolkit. Parallel Comput. 108 (2021), 102837.

[60] Keren Zhou, Jonathon Anderson, Xiaozhu Meng, and John Mellor-Crummey. 2022. Low overhead and context sensitive profiling of gpu-accelerated applications. In Proceedings of the 36th ACM International Conference on Supercomputing. 1–13.

[61] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU performance advisor based on instruction sampling. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 115–125.

A Artifact Appendix

A.1 Abstract

DeepContext is a cross-framework, context-aware profiler that correlates execution contexts from high-level Python code, deep-learning frameworks, underlying C/C++ libraries, and GPU kernels, delivering both coarse- and fine-grained performance metrics to developers and researchers. Version v1 of the artifact (49.2 MB, DOI 10.5281/zenodo.15589616) packages the profiler together with example traces, the browser-based GUI, and the automated analyzer, and is released under the MIT License. DeepContext supports PyTorch, JAX, and other CUDA/HIP-enabled frameworks across Nvidia and AMD GPUs, as well as x86 and ARM CPUs, allowing users to pinpoint multi-layer hotspots and receive optimization suggestions directly in the interface. Through representative case studies, the artifact demonstrates how unified context correlation and automatic insight generation accelerate the diagnosis and tuning of complex training and inference workloads, making DeepContext a practical tool for performance engineers tasked with optimizing heterogeneous deep-learning pipelines that span diverse compute environments.

A.2 Artifact check-list (meta-information)

- **DOI:** https://doi.org/10.5281/zenodo.15589616
- **Version:** `v2.0`
- **Archive size:** `49.3 MB`
- **License:** MIT License[2]

A.3 Dependencies

Config 1: x86 + NVIDIA AMD EPYC 7543 (32c/64t, Zen 3) with 256 GB DDR4 and an NVIDIA A100 80 GB GPU.

Config 2: Arm + NVIDIA Arm Neoverse-V2 (64 cores) with 480 GB LPDDR5 and an NVIDIA GH200 (Grace Hopper) 96 GB GPU.

[2]https://opensource.org/license/mit

Config 3: x86 + AMD AMD EPYC 7643 (48c/96t, Zen 3) with 2 TB DDR4 and an AMD Instinct MI250 accelerator (128 GB HBM2e).

All experiments reported in the paper were conducted on at least one of the configurations listed above; the profiler's instrumentation code is ISA-agnostic and requires only Python ≥ 3.10 and `ROCm` 6.0+ or `CUDA` 11.8+ on the host.

A.4 Getting the Artifact

Retrieve with:

```
wget https://zenodo.org/record/15589616/files/
    DeepContext.zip
unzip DeepContext.zip
# README.md in the root describes directory
    layout
```

A.5 Reuse Policy

This artifact is released under the MIT License.