# TritonParse: Multi-IR Provenance and Reproducible Debugging for Triton Kernel Compilation


Ying Li*
William & Mary
Williamsburg, USA
yli81@wm.edu

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

David Berard†
Anthropic
San Francisco, USA
dberard@anthropic.com

Xu Zhao
Meta
Menlo Park, USA
xzhao9@meta.com

Barney Huang
Meta
Menlo Park, USA
barney@meta.com

Wenyuan Chi
Meta
Menlo Park, USA
wychi@meta.com

Cheng-huan Tsai
Meta
Menlo Park, USA
agrontsai@meta.com

Prithvi Patel
Meta
Menlo Park, USA
prithvip24@meta.com

Srivatsan Ramesh
Meta
Menlo Park, USA
srir@meta.com

Manman Ren
Meta
Menlo Park, USA
mren@meta.com

Hongtao Yu
Meta
Menlo Park, USA
hoy@meta.com

James Wu
Meta
Menlo Park, USA
james.jz.wu@gmail.com

Dan Zimmerman
Meta
Menlo Park, USA
danzimm@meta.com

Nicholas J. Riasanovsky
Meta
Menlo Park, USA
njriasan@meta.com

Adam Mainz†
Google
Mountain View, USA
amainz@google.com

Bert Maher†
Anthropic
San Francisco, USA
bert@anthropic.com

Peiying Hua
Meta
Menlo Park, USA
peiying@meta.com

Daohang Shi
Meta
Menlo Park, USA
daohang@meta.com

Taylor Robie
Meta
Menlo Park, USA
taylorrobie@meta.com

Yuanwei Fang
Meta
Menlo Park, USA
fywkevin@meta.com

Shuofei Zhu
Meta
Menlo Park, USA
shuofei@meta.com

Shangdi Yu
Meta
Menlo Park, USA
shangdiy@meta.com

Nikhil Patel
Meta
Menlo Park, USA
nikhilap@meta.com

Bin Bao
Meta
Menlo Park, USA
binbao@meta.com

Yang Chen†
Anthropic
San Francisco, USA
yangchen@anthropic.com

Jan Szczepaniec
Meta
Menlo Park, USA
jansz@meta.com

Haishan Zhu†
OpenAI
San Francisco, USA
haishanz@openai.com

Patrick Li
Meta
Menlo Park, USA
patrickli.2001@gmail.com

Shane Nay
Meta
Menlo Park, USA
snay@meta.com

Adnan Aziz
Meta
Menlo Park, USA
adnanaziz@meta.com

Bill Yoshimi
Meta
Menlo Park, USA
byoshimi@meta.com

Alexey Loginov
Meta
Menlo Park, USA
loginov@meta.com

Yifan Sun
William & Mary
Williamsburg, USA
ysun25@wm.edu



## Abstract

Domain-specific languages (DSLs) for deep learning, such as Triton, reduce the development effort required to write high-performance GPU kernels. However, debugging Triton kernels remains challenging. To facilitate quick and streamlined debugging, we present TritonParse, a unified framework that systematically collects and organizes debugging evidence to help developers efficiently localize Triton kernel failures. TritonParse follows a 4R workflow: Record, Reconstruct, Review, and Reproduce. Beginning with capturing compilation and launch metadata, TritonParse then reconstructs traces into analyzable forms by aggregating data from multiple sources, supports human-in-the-loop inspection through a


*The two authors contributed equally.
†These authors contributed to this work while at Meta.




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



*ASPLOS '27, Heraklion, Greece*



© 2027 Copyright held by the owner/author(s).



ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850058

user interface, and generates reduced single-kernel reproducers (runnable programs that replay captured kernels) for manual debugging and automated regression analysis. Evaluations across NVIDIA H100 and AMD Instinct MI300 show that TritonParse reliably reproduces bugs across all known cases. Trace collection and execution replay incur modest runtime and storage overheads: 21% additional runtime and 4 MB of reconstructed storage per operator on average.

***CCS Concepts:*** • **Software and its engineering → Software testing and debugging**.

***Keywords:*** Triton, compiler debugging, GPU kernels, multi-IR provenance, reproducibility

**ACM Reference Format:**
Ying Li, Yueming Hao, David Berard, Xu Zhao, Barney Huang, Wenyuan Chi, Cheng-huan Tsai, Prithvi Patel, Srivatsan Ramesh, Manman Ren, Hongtao Yu, James Wu, Dan Zimmerman, Nicholas J. Riasanovsky, Adam Mainz, Bert Maher, Peiying Hua, Daohang Shi, Taylor Robie, Yuanwei Fang, Shuofei Zhu, Shangdi Yu, Nikhil Patel, Bin Bao, Yang Chen, Jan Szczepaniec, Haishan Zhu, Patrick Li, Shane Nay, Adnan Aziz, Bill Yoshimi, Alexey Loginov, and Yifan Sun. 2027. TritonParse: Multi-IR Provenance and Reproducible Debugging for Triton Kernel Compilation. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3845814.3850058

## 1 Introduction

Domain-specific languages (DSLs) for deep learning have emerged to make GPU kernel development easier and faster while preserving high performance, beyond the fixed functionality provided by vendor libraries such as cuBLAS and cuDNN (e.g., support for only a narrow set of tensor operations) [9, 16, 27, 47]. Among these efforts, Triton stands out as an open-source programming system that provides both an intermediate language and a compiler. With Triton, developers can express tile-based programs and compile them into highly efficient GPU code [45].

However, debugging Triton programs for both correctness and performance remains difficult. Several unique aspects of Triton magnify the challenges, including: (1) the execution environment for Triton is usually large (e.g., generative AI training) and involves a large number of kernel calls. (2) An autotuning process can compile and evaluate numerous kernel configurations to find the optimal one. And (3) A multi-stage compilation pipeline lowers code through several intermediate representations (IRs) before the final binary is compiled. This combination of dynamic kernel selection/invocation process and complex transformation creates a significant gap between the high-level source and the final executed machine code. Consequently, tracing a performance bottleneck or a correctness bug back to its origin is often an intractable manual task.

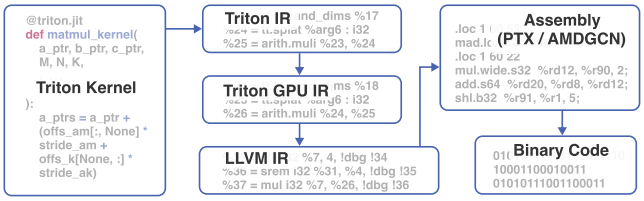



**Figure 1.** Triton's compilation pipeline performs multi-IR lowering (TTIR, TTGIR) before translating LLVM IR to device-specific assembly (PTX/AMDGPU) and packaging executable binaries.

Although kernel authors primarily interact with Triton at the Python level, some failures cannot be diagnosed from the Python kernel alone because their behavior emerges during autotuning, lowering, or backend code generation. For example, decisions such as thread/block mapping, predication/masking, memory-space placement, and numeric instruction selection may become explicit only in Triton IR (TTIR)/Triton GPU IR (TTGIR), or backend instruction set architecture (ISA). In such cases, lower-level compilation artifacts provide evidence needed to distinguish a kernel, input, or configuration issue from compiler-generated behavior. Diagnosing such cases therefore requires correlating evidence across multiple compilation stages rather than inspecting the Python kernel in isolation. For instance, instruction-selection changes between Triton versions 3.2 and 3.3 can affect BF16 rounding behavior in operations like RoPE (see §5.1).

Existing tools for debugging GPU kernels provide fragmented views of Triton's lifecycle and do not address the challenges posed by multi-level compilation and autotuning. These tools also cannot easily produce reproducible unit tests for integration into continuous integration (CI) workflows. We group the relevant prior work into three categories.

- **Vendor-specific profilers** such as NVIDIA Nsight Compute [33] and the ROCm profiler [1] deliver runtime metrics and source-assembly correlation. These profilers can be heavyweight, and provide limited cross-vendor support and alignment of Triton's IRs with the final backend code.
- **Framework-level tools** such as tlparse [40] expose PyTorch compiler IRs, but do not capture Triton's internal stages (e.g., TTIR, TTGIR) or provide stable identifiers that connect different levels of abstraction.
- **General compiler tools** operate at a single IR level and lack the Triton-specific semantics needed for holistic debugging. LLVM's bugpoint [29] reduces test cases within one IR. Alive2 [30] validates LLVM optimizations through semantic equivalence checking but remains confined to single-IR verification without preserving cross-stage provenance. Prior work on robust LLVM debug information [21] maintains consistency within traditional compiler pipelines rather than across multi-IR compilation stacks.

No existing tool offers a lightweight, portable workflow that captures the entire multi-IR compilation path for Triton kernels to produce environment-aware reproducers and support regression diagnosis.

To address the challenges of debugging Triton kernels in large-scale applications using frameworks such as PyTorch [2, 39], we propose the 4R framework, a workflow primarily designed for kernel authors and ML library developers, including those without compiler expertise.

The 4R framework synthesizes established debugging principles [4, 6, 10, 12, 32, 36, 42] into four dedicated stages: *Record*, *Reconstruct*, *Review*, and *Reproduce*. Since live debugging is impractical at scale, the Record stage first captures a complete, offline trace that includes both compilation and execution information. Because this captured data is multifaceted, the Reconstruct stage then aggregates the raw logs, builds mappings between IR stages, and analyzes differences across kernel launches. The Review stage empowers human developers to analyze Triton kernel execution by presenting this structured information through an interactive interface. Finally, the Reproduce stage automatically generates a reduced, portable reproducer using dependency analysis to capture cross-file/function dependencies. The resulting reproducer can then be used for manual debugging or automated regression analysis.

We present TritonParse, an open-source[1] framework that realizes the 4R methodology for debugging Triton kernels. TritonParse targets captured, per-launch, single-kernel failures, including performance regressions, hangs, illegal memory accesses, assertion failures, and correctness mismatches. TritonParse provides an integrated debugging framework that brings together the runtime context of each kernel invocation, compiler metadata, and multi-level IR artifacts, then performs structured analysis and dependency extraction to capture the state required for replay and generate reproducers. TritonParse's novelty lies in connecting these otherwise fragmented components into a unified debugging workflow.

No existing tool delivers this combination: vendor profilers bury developers in raw metrics without portable, reproducible evidence of a failure's context; framework-level tracers miss Triton's internal compilation stages; and general compiler tools operate on a single IR and lack Triton-specific semantics. TritonParse bridges the framework, compiler, and runtime layers to preserve and reconstruct the evidence needed to understand, reproduce, and debug a kernel failure. By collecting and correlating this evidence, TritonParse reduces the manual effort required for investigation and provides compiler metadata, multi-level IR artifacts, and a reduced reproducer for further inspection or deeper investigation by compiler developers.

In summary, this paper makes the following contributions:

- **An End-to-End Framework for Triton Kernel Debugging.** We present TritonParse, a unified framework that supports a developer-driven debugging workflow for Triton kernels. It brings together multi-IR capture, structured analysis, an interactive UI, and reproducer generation, allowing developers to correlate runtime and compilation evidence, investigate failures across compilation stages, and reproduce problematic kernel executions.
- **An Extensive Multi-IR Provenance Tracer for Triton Kernels.** We introduce a lightweight yet extensive tracer that captures Triton's full multi-stage compilation pipeline without modifying the compiler. The multi-IR trace provides unprecedented visibility into the compiler's transformations, which is important for identifying the root cause of bugs.
- **An Effective Workflow for Reduced Single-Kernel Reproducer Generation and Regression Analysis.** We demonstrate a technique for generating reduced, portable, bug-preserving reproducers that are both efficient and reliable. The reproducer achieves a 100% replay success rate on both NVIDIA and AMD GPUs, and serves as a portable artifact for manual debugging and automated regression analysis.

## 2 Background

### 2.1 Triton Language

Triton has emerged as a widely used approach for building high-performance GPU kernels without the complexity of CUDA C++ toolchains. Unlike vendor-tuned libraries (e.g., cuBLAS and cuDNN), which support only fixed operator suites and cannot keep pace with rapidly evolving AI workloads, Triton offers a programmable, tunable, and portable alternative. As Triton sees growing adoption across AI systems, robust debugging support becomes essential to sustain developer productivity and kernel reliability.

Triton is an open-source Python-embedded DSL and compiler for GPU kernels [45]. Unlike CUDA, which exposes threads, warps, blocks, and shared memory directly, Triton adopts a blocked tensor programming model in which the compiler handles scheduling, memory coalescing, and tiling. This higher abstraction accelerates kernel development, but it also hides the mapping from source code to executed instructions, which makes it difficult for developers to localize bugs and reason about performance anomalies.

### 2.2 GPU Kernel Compilation

Triton's just-in-time (JIT) compilation lowers kernels through multiple IR stages before producing hardware-specific code (see Figure 1). Each stage exposes different information and potential failure points, and kernel behavior depends on

[1] Link to source code: https://github.com/meta-pytorch/tritonparse

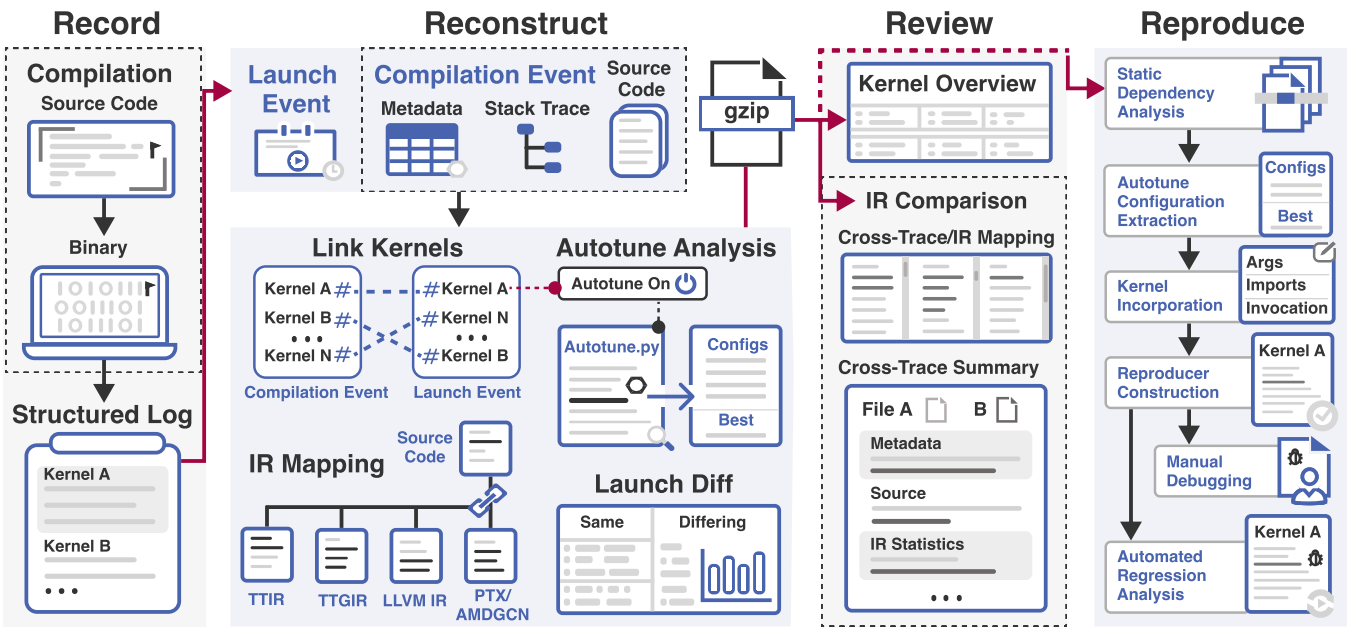



**Figure 2.** Overview of TritonParse's four-stage workflow. The Record stage captures Triton compilation and execution events. The Reconstruct stage parses logs, aligns source and IR mappings, analyzes launches and autotuning, and packages results into a compact trace file. The Review stage provides a UI for synchronized inspection across IRs, call stacks, and launches. The Reproduce stage generates a reduced single-kernel reproducer that recreates failures for debugging.

inputs, memory layouts, hardware, and cross-stage transformations. Failures often arise from subtle cross-IR or platform-specific interactions, which force developers to inspect multiple IRs simultaneously to localize a bug.

- **Triton IR (TTIR)** captures the algorithmic structure and shape-dependent indexing [45]. An off-by-one indexing error often becomes visible here when tensor shapes change.
- **Triton GPU IR (TTGIR)** makes the mapping of work to threads and blocks explicit, along with predication and memory space assignments [17, 45]. A mis-specified memory space can lead to a silent race condition that does not appear at higher levels.
- **LLVM IR** exposes target-independent control and data flow, along with early signs of register pressure [25]. Excessive register pressure surfaces here and can force spilling that impairs performance.
- **Vendor ISA** (AMDGCN for AMD [1], PTX or SASS for NVIDIA [35]) provides the definitive view of issued instructions, scheduling, and resource usage. Performance anomalies such as uncoalesced memory loads or instruction stalls are often diagnosable only at this stage.

Together, the multi-IR compilation stages illustrate that diagnosing a single bug may require inspecting all of these layers: validating high-level semantics at TTIR, checking mapping and memory behavior at TTGIR, confirming compiler transformations at LLVM IR, and verifying final execution at the vendor ISA. The unique multi-IR compilation process motivates TritonParse to provide synchronized, cross-IR views, allowing developers to follow a problem across all compilation stages until its root cause is identified.

### 2.3 Design Requirements for Kernel Debugging

Following the reasoning about unique challenges in debugging in the context of running Triton GPU kernels in large language model (LLM) workloads, we identify the following design requirements for a Triton debugging system.

**DR1: Multi-IR Traceability.** A debugging system must capture and correlate information across Triton's full compilation pipeline, from source code to final hardware instructions. Without this complete view, developers cannot trace how a high-level tensor operation transforms into the specific instruction sequence that triggers a failure.

**DR2: Environmental Reproducibility.** The framework must handle environment-sensitive failures by capturing relevant execution context—kernel configurations, input shapes, compiler flags, and hardware details. Since identical source code can produce different behaviors across GPU models, driver versions, or even launch parameters, the framework must generate portable reproducers that reliably trigger failures while clearly documenting environmental dependencies.

**DR3: Integrated Analysis.** A debugging framework must connect compilation artifacts with runtime behavior, linking low-level symptoms (memory violations, performance anomalies) to their high-level causes. This requires not just data collection but structured analysis that makes complex multi-stage transformations comprehensible to developers.

**DR4: Lightweight Operation.** The debugging system must impose low time and storage overhead for continuous use. Since many GPU kernel failures are intermittent and appear only under realistic workloads, developers need a recording mechanism that is enabled without significantly impacting performance or timing-sensitive behaviors.

## 3 TritonParse

TritonParse is a unified framework for analyzing and debugging Triton kernels, supporting a wide range of use cases including compiler-generated kernels (e.g., from TorchInductor [2]), hand-written kernels, and kernels generated by higher-level frontends such as Helion [41]. TritonParse can also capture kernels produced by other external frontends, such as Pallas [3], when they lower through Triton's standard compilation and runtime path. TritonParse (see Figure 2) is organized into four stages that follow the 4R framework.

### 3.1 Record

The Record stage captures the complete execution context of each kernel during compilation and launch (see Figure 2). For every kernel, TritonParse collects the IRs produced at each lowering stage, compilation metadata, and per-invocation launch arguments, so that the kernel's full lifecycle is preserved in a single trace (addresses **DR1**). The output is self-contained, and downstream stages consume it without additional dependencies on the original execution environment.

TritonParse attaches to Triton via its hook interface, where hooks serve as registration points in the compilation or runtime flow for attaching callbacks without modifying the compiler. A simplified example of compile-time hook registration is shown in Listing 1. Each hook wraps the target function, records the required data, and invokes the original implementation, so the recording path adds no behavioral change to the kernel under inspection. Hooks are a principled integration point because they cleanly separate kernel logic from debugging support, and Triton developers maintain a stable hook interface across releases as the designated extension mechanism. To ensure long-term compatibility, TritonParse confines all capture logic to a small number of well-isolated hooks, which keeps the integration robust to upstream changes. We verified this by maintaining compatibility across Triton versions 3.4 to 3.7 during evaluation.

TritonParse registers two hook types. (1) Compile-time hooks record IR snapshots at each lowering stage (TTIR, TTGIR, LLVM IR, PTX, and AMDGCN), together with compilation metadata such as kernel names, stack traces, and compiler and driver versions. These records are packaged as a *compilation event*. (2) Launch-time hooks record per-invocation arguments, including tensor shapes, data types, and scalar values, and optionally the raw tensor contents. These records, excluding raw tensor contents, are packaged as a *launch event*.

```python
def trace_triton(src, metadata, event_type, ...):
    ...
from triton import knobs
knobs.compilation.listener = trace_triton
```

**Listing 1.** A simplified example of compile-time hook registration.

**Algorithm 1** Multi-level IR source code mapping algorithm.

**Input:** *trace* - TTIR, TTGIR, LLVMIR, PTX/AMDGCN
**Output:** $M$ - Bidirectional mappings across all IR levels and Python source
*// Extract IR files*
$IRs \leftarrow$ ExtractIRs($trace$)
*// Build IR-to-source mappings, line -> (file, line, column)*
**for** $ir \in$ {ttir, ttgir, llvmir, ptx/amdgcn} **do**
  $M[ir] \leftarrow$ BuildIRMap($IRs[ir]$)
**end for**
*// Create bidirectional cross-IR mappings, j -> target_lines*
**for** ordered pairs $(i, j)$, where $i, j \in \{ttir, ttgir, llvmir, ptx/amdgcn\}$ and $i \neq j$ **do**
  AddLinksByLocation($M[i], M[j]$)
**end for**
*// Aggregate by Python source lines*
$M$[python] $\leftarrow$ GroupBySource($M$)
**return** $M$

TritonParse keeps traces compact through three storage decisions (addresses **DR4**). (1) Events are streamed as newline-delimited JSON (NDJSON, also known as JSON Lines/JSONL) with optional record-level compression, which enables streaming append-only logging while keeping in-memory state bounded. (2) Raw tensor contents are needed to faithfully reproduce input-dependent bugs such as numerical instabilities or shape-specific failures, but they dominate trace size. So TritonParse stores them separately from the main log and makes capture opt-in. (3) Each tensor is referenced in the main log by a content hash, which serves as a unique identifier and allows the Reproduce stage to retrieve the exact original tensor while keeping the primary log lightweight.

TritonParse is designed to limit recording overhead through selective recording: lightweight launch and compilation metadata are captured by default, while raw tensor contents are recorded only when required for full replay or data-dependent debugging. In addition, most recording work consists of CPU-side bookkeeping around asynchronous GPU launches and can overlap with the execution of previously launched kernels.

### 3.2 Reconstruct

The Reconstruct stage prepares the recorded trace for analysis and reproduction. Reconstruct runs offline after recording completes and derives three views from the raw events (addresses **DR3**): per-kernel lifecycle, cross-IR provenance, and per-launch behavioral summaries. The result is packaged

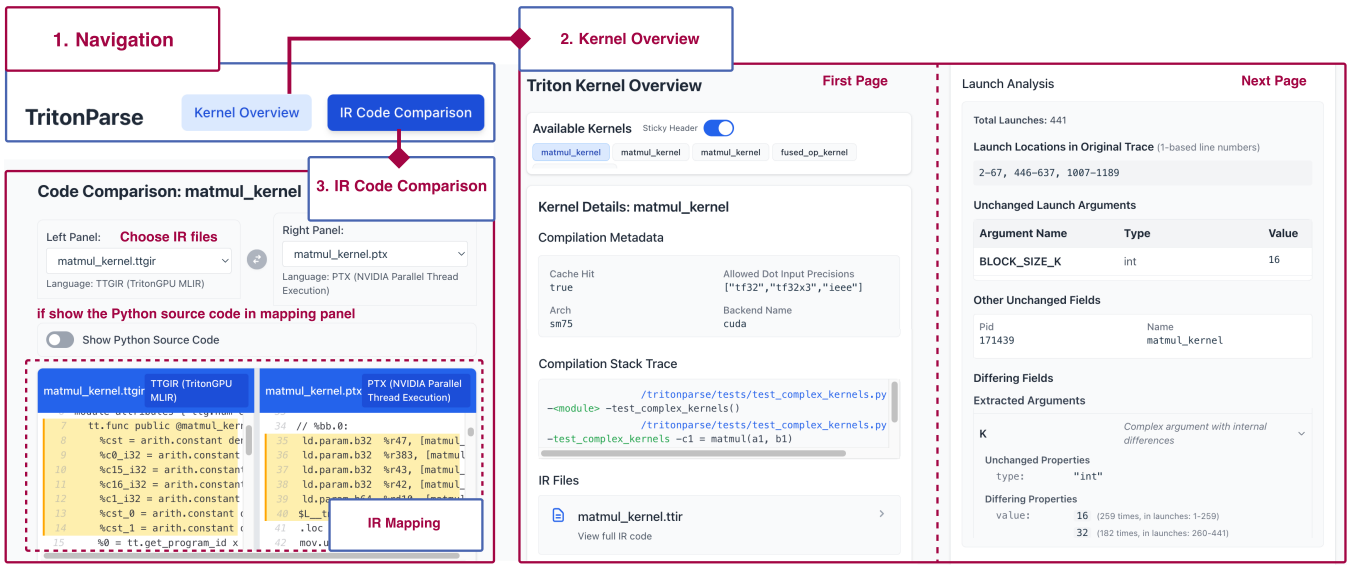



**Figure 3.** The TritonParse UI is organized into two tabs: the Kernel Overview tab and the IR Comparison tab. For clarity and space, the figure shows only a subset of the data in each panel rather than the full data available in the interface.

into a single self-contained archive that the Review and Reproduce stages consume directly.

**Per-kernel lifecycle.** The raw log records kernel compilation and launch events as independent entries. Reconstruct links compilation and launch events by kernel hash so that each kernel's full history (one compilation event and all subsequent launches) forms one navigable unit. Out-of-line tensor blobs are rejoined to their referencing launch events through their content hashes, so the archive is self-contained and downstream stages need no access to the original recording environment.

**Cross-IR provenance.** The compiler preserves source-location annotations independently in each IR, but the correspondence between IRs is never materialized in the raw log. Reconstruct rebuilds this correspondence by joining IR annotations on shared source locations, producing bidirectional mappings across TTIR, TTGIR, LLVM IR, the vendor ISA (PTX or AMDGCN), and the original Python source (addresses **DR1**).

Algorithm 1 summarizes the lightweight procedure that links IR stages using compiler-provided source-location metadata. For each IR, TritonParse first builds a map from instruction lines to `(file, line, column)` tuples by extracting location definitions and references and resolving location aliases. Ordered pairs of IR maps are then linked by joining on the shared source-location key, yielding forward and backward links between instructions that originate from the same Python source line. The links are finally aggregated by Python source line, which enables direct bidirectional navigation across all IR levels from a single click in the source view. As an inspection aid, cross-IR mapping does not perform complete semantic mapping and cannot recover source locations that are lost, merged, or distorted during compiler transformations.

**Per-launch behavioral views.** Two summaries are computed across the launch events of each kernel. The *autotune view* records each candidate configuration, its measured performance, and the final selection, so developers can inspect tuning decisions when diagnosing performance differences or generating reproducers. Users can also replay the top-performing configuration directly, which skips autotuning reruns and shortens bug reproduction time. The *launch diff view* aggregates variations in kernel arguments and grid configurations across all invocations, which exposes the parameters that vary at runtime and supports comparison of different invocations.

**Packaging.** The processed trace is packaged into a single `ndjson.gz` archive (addresses **DR4**). The file-level compression complements the record-level runtime compression applied during Record and offers better dictionary efficiency, which reduces storage size while preserving compatibility with downstream tools.

### 3.3 Review

TritonParse's Review stage (see Figure 2) provides a zero-install browser UI that renders synchronized multi-IR views, Python call stacks, and launch analysis from the processed output files of the Reconstruct stage (addresses **DR3**). The interface presents five views (shown in Figure 3), each answering a distinct debugging question.

- **Kernel Metadata** answers "what was compiled and how was it launched." The view summarizes the kernel name, launch configuration (grid and block dimensions), target

GPU device, compilation time, memory usage, and optimization flags, giving developers a single-glance snapshot of the kernel's compile and launch context.

- **Launch Diff Analysis** answers "which inputs trigger different behaviors." TritonParse automatically summarizes relevant differences across kernel launches, and the view presents the number of launches, parameters that vary across invocations, and those that remain constant, helping developers isolate the input dimensions responsible for behavioral changes.
- **Autotune Analysis** answers "why did the autotuner choose this variant." The view displays every autotuning configuration explored and the final configuration selected for each input, thereby exposing the tuning decision and enabling replay of the top-performing configuration.
- **Cross-IR Mapping** answers "how does the source code map to the executed instructions" (addresses **DR1**). The view provides synchronized side-by-side inspection of the Python source and all IR stages, with corresponding lines linked through color-coded highlights, scroll-synchronized panels, and direct interaction, so developers can follow a single source line through every transformation and locate the stage where a bug or performance anomaly first appears.
- **Cross-Trace Comparison** answers "what changed between two builds of the same kernel," for example, between traces produced by different Triton versions (addresses **DR3**). Rather than inspecting a single compilation flow, this view aligns two traces stage by stage. And the comparison proceeds from coarse-grained to fine-grained across five levels: *metadata* surfaces configuration changes, normalized *source code* surfaces program-level differences, *IR statistics* surface structural changes across stages, *source-mapped IR* localizes IR differences back to source lines, and *tensor values*, when available, surface numerical differences. The five levels run automatically and appear as structured summaries in the view.

These views can be used independently depending on the debugging scenario. A known bug-free Triton version is not required. Cross-Trace Comparison is an optional mode for regression analysis. With a single failing trace, Launch Diff Analysis helps isolate failure-triggering inputs, while Cross-IR Mapping follows the implicated source line across lowering stages to identify where divergence first appears.

Several features support detailed inspection: call stack integration, file selection menu, and synchronized highlighting. Call stack integration displays the Python call stack for each kernel, including file paths, line numbers, and function names, allowing users to pinpoint the code responsible for triggering compilation. A file selection menu lets users choose any trace file or any pair of IR stages for comparison. Synchronized highlighting ensures that selecting a line in one view instantly highlights the corresponding lines in the others, making cross-IR/trace navigation easier.

**Algorithm 2** AST-based dependency extraction.

**Input:** entry source file $f_0$, kernel function $k$
**Output:** extracted functions $F$, required imports $I$
$G_0 \leftarrow \textsc{BuildCallGraph}(f_0)$ // single-file call graph
$R \leftarrow \textsc{TransitiveClosure}(G_0, \{k\})$ // transitive closure
$Q \leftarrow \textsc{InternalImports}(R, f_0)$ // multi-file traversal
**while** $Q \neq \emptyset$ **do**
  $(f, E) \leftarrow Q.\textsc{Dequeue}()$ // (imported file, entry functions)
  $G_f \leftarrow \textsc{BuildCallGraph}(f)$
  $R_f \leftarrow \textsc{TransitiveClosure}(G_f, E)$
  $R \leftarrow R \cup R_f$ // Update reachable functions
  $Q \leftarrow Q \cup \textsc{InternalImports}(R_f, f)$
**end while**
$F \leftarrow \textsc{ExtractFunctions}(R)$ // import pruning
$I \leftarrow \textsc{PruneImports}(F)$
**return** $F, I$

### 3.4 Reproduce

The Reproduce stage of TritonParse (see Figure 2) is responsible for generating a reduced single-kernel reproducer (i.e., a runnable program that replays a captured kernel invocation and reproduces the failure with a single command, addresses **DR2**). Once generated, the reproducer can be used either for manual debugging of performance issues or for automated regression analysis. The reduced reproducer, together with linked debugging evidence, supports focused investigation, while final root-cause determination in manual debugging remains developer analysis.

The Reproduce stage runs three preparatory analyses whose outputs feed a single assembly step. **Static dependency analysis** determines *what code* to include, **autotune configuration extraction** determines *which variant* to run, and **kernel incorporation** determines *how to embed* the kernel source. Reproducer construction then combines these outputs with the recorded runtime state to produce the final file. The remainder of this section describes each step, followed by the automated regression analysis workflow that consumes the generated reproducer.

**Static Dependency Analysis.** A runnable reproducer must include not only the kernel function but also its full dependency closure so that the execution can be fully reproduced. In real-world Triton code, this closure typically spans multiple files and includes helper functions (e.g., launcher wrappers that compute grid dimensions and launch the kernel), autotune configuration generators (e.g., functions that supply the search space), custom operators (e.g., device functions that the kernel calls inline), and shared utilities (e.g., indexing or tiling helpers). Omitting any of these dependencies prevents the reproducer from compiling or causes it to execute a different code path than the original failure. After extracting the dependency closure, TritonParse validates its completeness by executing the generated reproducer.

TritonParse extracts the closure automatically through Python Abstract Syntax Tree (AST) call-graph traversal. As shown in Algorithm 2, the analysis proceeds in four steps.

- **Single-file call graph construction.** TritonParse analyzes the source file containing the target kernel and builds an initial call graph containing only functions reachable from the captured kernel, excluding unrelated application functions, builtins, and Triton-internal APIs.
- **Transitive closure.** TritonParse computes the transitive closure of reachable calls within the file to identify all in-file functions required by the kernel.
- **Multi-file traversal.** For dependencies defined in other modules, TritonParse locates the corresponding source files and continues the analysis recursively for internal project code, while preserving external libraries (e.g., `torch`, `numpy`) as imports.
- **Import pruning.** TritonParse retains only the imports referenced by the collected code and merges redundant imports when possible.

**Autotune Configuration Extraction.** Autotuning complicates reproduction. In most cases, the bug lies in the variant selected by the autotuner, and reproducing the bug only requires running that variant. Occasionally, the bug lies in the autotuning process itself, requiring the full search to be rerun. TritonParse supports both cases through two modes: autotune-off and autotune-on. In *autotune-off* mode, TritonParse reuses the autotune view produced by the Reconstruct stage, which already records the winning configuration of the original run. TritonParse writes the autotuner-selected parameters (e.g., `BLOCK_M`, `BLOCK_N`) into the reproducer as fixed constants, removes the `@triton.autotune` decorator, and inserts compilation settings that the decorator previously supplied (e.g., `num_warps`, `num_stages`) at the kernel call site. In *autotune-on* mode, TritonParse keeps the decorator and the original search space, so replay re-runs the full autotuning process. Autotune-off is the default because reproducing a single variant is the common case.

**Kernel Incorporation.** TritonParse offers three modes for placing the kernel source into the reproducer, because the right representation depends on what is available at reproduction time and where the reproducer needs to run.

- **Import mode** writes an import statement that references the kernel at its original source path. The reproducer stays small and tracks later edits to the source, but it runs only on machines where the original project tree is available.
- **Embed mode** copies the kernel function and all dependencies extracted by static analysis directly into the reproducer file. The file is fully self-contained and portable across machines, making embed mode the default choice for bug reports, CI artifacts, and regression bisection.
- **TTIR injection mode** bypasses the Triton frontend by feeding the TTIR captured during recording directly into the Triton compiler. This mode applies when the original Python source is no longer available, or when the bug lies below the TTIR level and the developer wants to isolate the post-TTIR pipeline from frontend changes.

**Reproducer Construction.** After dependency extraction, TritonParse builds a context bundle containing runtime arguments, compilation metadata, and source information, and uses it to instantiate a reproducer template with the required imports, dependent functions, argument reconstruction logic, and kernel invocation. When tensor capture is enabled, tensor inputs are restored from separately stored blobs by resolving their content hashes, enabling exact replay of recorded inputs.

The reproducer is reduced relative to the original application, while its dependency closure size is case-dependent and determined by the code reachable from the captured kernel. As a concrete example, for an H100 FP8 GEMM case from `TritonBench` [31], a benchmark suite of real-world Triton kernels, TritonParse reduces the original setup containing 3,833 functions across 500 files to a five-function dependency closure and a standalone 893-line reproducer, of which about 360 lines are kernel and dependency code.

The resulting reproducer depends only on standard Python components, Torch, Triton, and the embedded or imported kernel code. Under a fixed software and hardware environment, Triton compilation is deterministic, so the reproducer can replay the captured kernel with the same launch, compilation, and input state to reconstruct the original per-launch execution rather than an unrelated synthetic test. It can then replay the failure on a compatible software and hardware environment without requiring the environment to be identical to the original one (addressing **DR2**).

**Automated Regression Analysis.** Beyond manual failure replay, the reproducer also serves as a regression test artifact. Triton-compiled kernels often suffer from regressions, where a kernel that used to work stops producing correct results in a newer Triton version, or drops performance significantly. Regressions are difficult to localize manually because any upstream commit in LLVM or Triton itself can trigger the error, and each bisection step may require rebuilding part of the compiler stack.

TritonParse addresses this with an automated regression analysis workflow built around the generated reproducer. The reproducer is essential because bisection must run the same failing kernel against many Triton versions, and writing such a standalone test manually is prohibitively expensive when the kernel is buried inside a large model and depends on helper functions, autotune configurations, and recorded inputs. With an auto-generated reproducer, bisection becomes a simple loop: given a known-good Triton version and a known-bad Triton version, the workflow bisects the Triton commit history between them, rebuilds Triton and runs the reproducer at each candidate commit, and returns the first commit at which the reproducer's behavior changes. Readers can refer to §5.2 for how the automated regression analysis can be used in practice.

```
1   def _topk_forward(...):
2       pid = tl.program_id(0)
3   +   if pid == 0:
4   +       while pid == pid: #hang
5   +           tl.atomic_add(S, 1)
6       ...
7   +   assert stride_ym == 192, f"stride check
        failed" #assertion
8       ...
9   -   offs_y_n = tl.arange(0, N_EXPTS_ACT)
10  +   offs_y_n = tl.full((N_EXPTS_ACT,),
        N_EXPTS_ACT * 1000000, dtype=tl.int32) #IMA
```

**Listing 2.** Codes for assertion, IMA, hang error in Triton kernel, `_topk_forward` as an example.

## 4 Experiments

We use a series of experiments to evaluate whether TritonParse can reliably capture, minimize, and reproduce Triton kernel executions with minimal runtime and storage overhead, while preserving sufficient context for debugging.

### 4.1 Methodology

We use two datasets for the experiments.

**Operator Dataset.** The operator dataset consists of GPU operators from the `TritonBench` [31] benchmark suite, a collection of PyTorch operators used to evaluate Triton and its integration with PyTorch. Each operator includes a range of Triton kernels, with many designed specifically for LLM training. We include all operators that run successfully on our NVIDIA and AMD GPU platforms. We use this dataset to measure recording overhead, the effectiveness of log size reduction, and the TritonParse autotune analysis feature.

**Bug Dataset.** To systematically evaluate reproducer reliability, we use `gpt-oss-20b` [37] in our bug dataset, a state-of-the-art open-weight language model optimized for low-latency and specialized workloads (21 billion parameters with 3.6 billion active parameters). The bug dataset contains several bug categories, including hang errors, assertion failures, and illegal memory access (IMA) errors (shown in Listing 2). These injected bugs represent single-kernel failure types observed in our Triton debugging experience and are consistent with patterns reported in GitHub issues. We apply each bug to each of the five kernels used in `gpt-oss-20b`, forming 15 mutations. In each run, we test one mutation and let TritonParse generate reproducers. In addition to this mutation-based dataset, our evaluation includes two real-world cases in section 5. The two cases cover a correctness issue and a performance regression, respectively.

**Environments.** To evaluate TritonParse in realistic and diverse deployment settings, we select one AMD and one NVIDIA GPU platform (see Table 1), representing the two major GPU vendors. We keep the platforms as close as possible to make a reasonable comparison.

**Table 1.** AMD and NVIDIA platforms configuration.

| | AMD Platform | NVIDIA Platform |
|---|---|---|
| **CPU** | AMD EPYC 9654 | AMD EPYC 9654 |
| **GPU** | AMD Instinct MI300X | NVIDIA H100 |
| | 192 GB HBM3 Memory | 98 GB HBM3 Memory |
| **OS** | CentOS Stream 9 | CentOS Stream 9 |
| **SDK** | ROCm 6.4.0 | CUDA 12.8 |
| | driver 6.7.3 | driver 550.90.07 |
| **Compiler** | GCC 11.5.0 | GCC 11.5.0 |
| **Libraries** | PyTorch 2.9.0 | PyTorch 2.9.0 |
| | Triton 3.4.0 | Triton 3.4.0 |

### 4.2 Recording Overhead

The first question we explore is whether the Record and Reconstruct stages add noticeable runtime overhead. We evaluate recording's runtime overhead using the operator dataset (see subsection 4.1). For each operator, we compare three execution modes: (1) native execution without TritonParse, used as the baseline, (2) execution without runtime compression, where logs are parsed directly into the compact, structured archive format, and (3) execution with runtime compression followed by parsing into a compact, structured log archive stored as a zip file. Since compression also happens in the Reconstruct stage, this experiment includes both the Record and Reconstruct stages. Each run is repeated five times to average out CPU-side variance.

**Results.** The results (see Figure 4) show that TritonParse adds modest runtime overhead. Without runtime compression, the overhead is 21% on NVIDIA and 7% on AMD. With runtime compression enabled, the overhead rises to 22% on NVIDIA and 10% on AMD. The runtime overhead is lower on AMD because its longer baseline runtime absorbs a similar absolute tracing overhead. We compute 95% confidence intervals using Student's $t$-distribution. Across three configurations, the median confidence interval half-width is 1.15%–1.62% of the mean on AMD and 2.57%–3.05% on NVIDIA.

For a small number of operators, the measured normalized runtime is slightly below 1×. These measurements do not indicate kernel acceleration; they arise from replacing an existing listener path, overlap between CPU-side recording work and asynchronous GPU execution, and measurement noise. This level of runtime overhead is acceptable for two reasons. (1) Tracing runs only during debugging sessions and not during normal application execution. (2) The Review and Reproduce stages consume the recorded trace offline and add no further runtime overhead.

### 4.3 Storage Overhead

We evaluate how effective the compression mechanisms are in terms of reducing trace size while preserving all information required for replay, reproduction, and analysis. Using

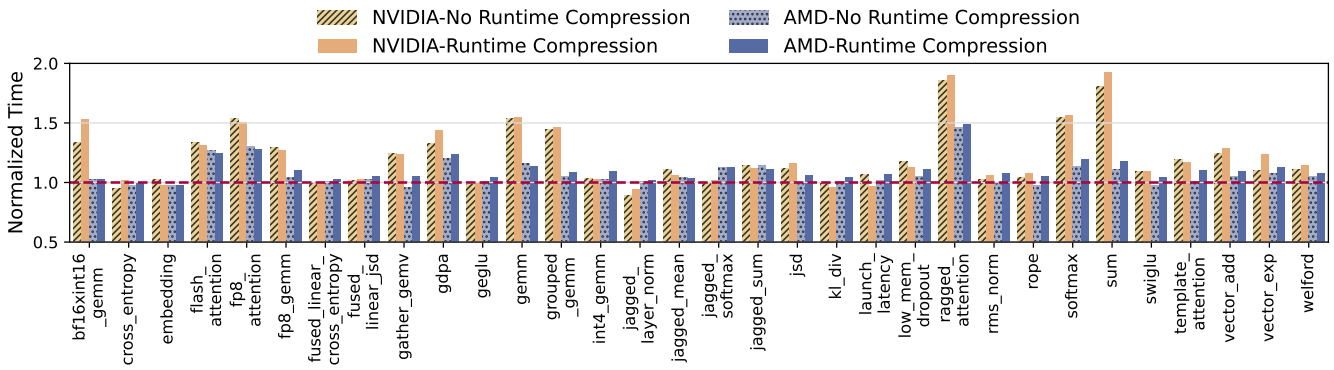



**Figure 4.** Normalized execution time with and without backend recording enabled, for AMD and NVIDIA platforms. Values close to 1.0 indicate minimal overhead for cases with runtime compression, and without runtime compression.

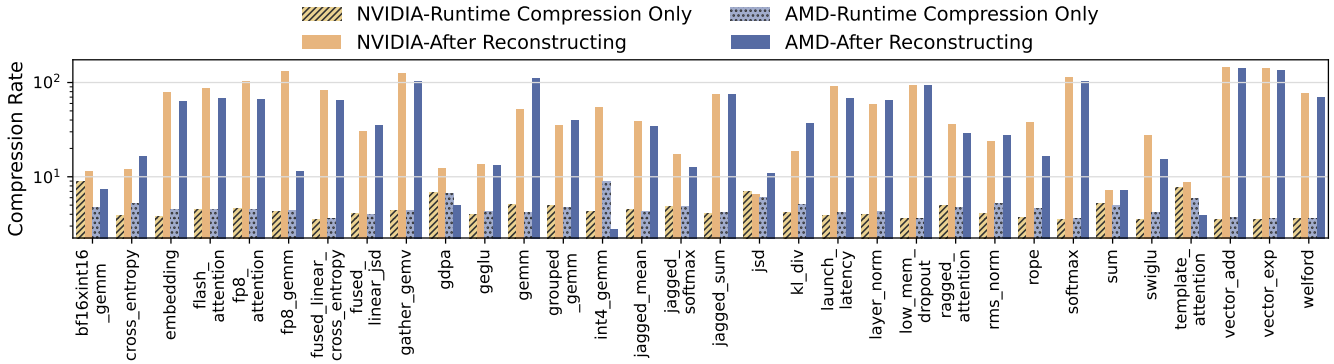



**Figure 5.** Log size comparison across AMD and NVIDIA platforms. Reported values (in log scale) represent the ratio of raw log size to logs with runtime compression only, and raw log size to logs with both runtime compression and the Reconstruct stage.

the same operator dataset and hardware platforms as in subsection 4.2, we evaluate the two-step compression: runtime compression (first step during Record stage), followed by the Reconstruct stage, packaging the logs into a structured zip archive (second step). Therefore, we compare three cases: (1) raw logs collected directly with TritonParse as baseline, (2) logs captured with runtime compression only, and (3) logs captured with runtime compression and then packaged by the Reconstruct stage. We measure the total size of each log type over five runs.

**Results.** As shown in Figure 5, runtime compression alone reduces logs to 21.9% of the raw size on NVIDIA and 21.4% on AMD. The Reconstruct stage further packages the logs into a structured archive that is only 1.85% of the raw size on average, corresponding to a 57.4× reduction on NVIDIA and a 51.2× reduction on AMD. In absolute terms, raw logs span 12 MB to 1.8 GB (median 120 MB on NVIDIA, 150 MB on AMD), while archives span 0.5 MB to 12 MB, averaging 3 MB on NVIDIA and 5 MB on AMD per operator. The gap between NVIDIA and AMD reflects differences in IR verbosity. TTGIR, AMDGCN, and PTX produce different amounts of code for the same operator, so the raw log size and the final archive size both depend on which platform generated them.

The structured archive can occasionally exceed the runtime-compressed log because the archive carries analysis outputs that the runtime-compressed log does not contain, including IR mappings, autotune data, and launch diffs. These outputs make the archive directly consumable by the Review and Reproduce stages, which justifies the occasional size increase.

### 4.4 Reproduction Reliability and Efficiency

We use the bug dataset to evaluate whether TritonParse-generated reproducers can faithfully recreate failures. TritonParse first generates the reduced single-kernel reproducer and then replays the bugs (assertion, IMA, and hang) on NVIDIA H100. Each replay run records both the success rate of reproducing the original bug and the runtime overhead of the reproduction stage. A replay is considered successful only if it exhibits the same failure signature as the recorded run. This criterion focuses on reproducing the observed failure behavior rather than establishing root-cause equivalence.

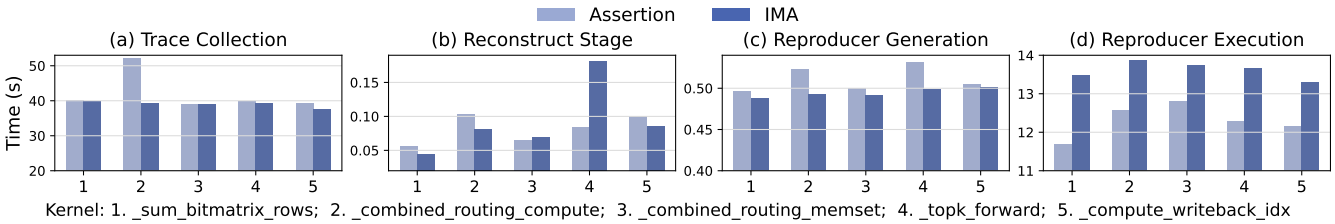



**Figure 6.** The time taken at each step of the reproducer generation process for kernels in the `gpt-oss-20b` model. In the hang case, timing measurements are omitted, since the hang alters real-time behavior and prevents reliable time reporting.

Hangs require special handling because the kernel never returns. TritonParse records traces before each kernel launch, so the trace remains complete even when the application never terminates. During reproduction, TritonParse detects the hang from the recorded annotation and applies a configurable timeout (30 seconds by default) to bound replay. The 30-second default is conservative: Figure 2 in the Liger-Kernel paper [20] shows all kernel executions completing within about 225 ms, which is more than two orders of magnitude below the timeout.

**Results.** The injected-bug benchmark provides a controlled setting for quantitative evaluation of reproducer reliability. Our evaluation shows that TritonParse successfully generates reduced single-kernel reproducers and reproduces failures in 100% of cases in this benchmark, confirming its ability to preserve error conditions across diverse execution contexts. This high rate is achievable because TritonParse records highly detailed execution traces, and Triton's deterministic compilation supports reliable replay under a controlled setup.

The entire TritonParse process is efficient, as shown in Figure 6. Trace collection takes only 40.5 seconds on average, which is close to the runtime of the unmodified `gpt-oss-20b` model (39 seconds). We reduce the raw log size of 419 MB to 4.4 MB with compression mechanisms. The Reconstruct stage completes in 0.18 seconds. Next, the reproducer is generated within 0.53 seconds. The reproducer execution finishes in about 13.88 seconds. The entire process is significantly faster than the manual reproduction creation process. For the hang case, we do not report trace collection or reproducer execution time, as the hang prevents accurate timing.

## 5 Case Studies

Next, we present two naturally occurring bugs, a correctness issue and a performance regression, as complementary real-world case studies to demonstrate how developers can effectively work with the Review and Reproduce stages in practice.

### 5.1 Debugging with Multi-IR Capture

We maintain a test for a customized Rotary Position Embedding (RoPE) [44] neural network module, which is widely used in transformer-based language models. This module consists of about 700 lines of code, including a Triton kernel of over 150 lines within the forward pass. After upgrading from Triton 3.2 to Triton 3.3, we observe that the numerical accuracy of the RoPE operator improved noticeably.

To understand this accuracy change, we use TritonParse to capture the full compilation pipeline. After extracting the multiple IRs, we compare the differences at each IR level using the cross-trace diff view. The cross-trace comparison is automatically summarized by TritonParse, with the PTX outputs revealing instruction-level differences between the two versions that help developers explain the observed accuracy change.

We use $out_rope1 = in_rope1 * cos_rope - in_rope2 * sin_rope$ as one example to illustrate how developers analyze the instruction-level differences and understand the observed accuracy change (see Listing 3). In Triton 3.2, the compiler uses a sequence of fused multiply-add (FMA) instructions to emulate multiplication and subtraction: two for the multiplications and one for the subtraction. In contrast, Triton 3.3 uses native BF16 operations [35]: `mul.bf16` to compute the products, `neg.bf16` to flip the sign, and then a standard FMA to combine results. This instruction sequence can also be further optimized into a single `mul.bf16` instruction, which directly performs the multiply using the hardware BF16 units.

These two approaches differ in rounding behavior. Native `mul.bf16` applies separate rounding for each multiply, while the FMA-based emulation uses a single rounding step. To quantify this effect, we conduct a randomized experiment on an NVIDIA H100. Out of 10,000 tests, 25.84% produce results that differ between the two implementations.

To stabilize accuracy across Triton versions, we adjust our baseline setup by upcasting BF16 inputs to FP32 before the multiply-subtract sequence and downcasting the result back to BF16. The cast removes the dependence on the compiler's BF16 instruction selection and yields consistent results in both old and new environments.

### 5.2 Regression Diagnosis via Reproducer Generation

To demonstrate the power of our reproducer and auto-regression analysis, we apply TritonParse to a real-world performance

```
{ .reg .b16 c;  // temp1 = in_rope1 * cos_rope
mov.b16 c, 0x8000U;
fma.rn.bf16 %temp1, %in_rope1, %cos_rope, c; }
{ .reg .b16 c; // temp2 = in_rope2 * sin_rope
mov.b16 c, 0x8000U;
fma.rn.bf16 %temp2, %in_rope2, %sin_rope, c; }
{ .reg .b16 c;       // result = temp1 - temp2
mov.b16 c, 0xbf80U;
fma.rn.bf16 %result, %temp2, c, %temp1; }
```

```
// temp = in_rope2 * sin_rope
mul.bf16 %temp, %in_rope2, %sin_rope;

// neg_temp = -temp
neg.bf16 %neg_temp, %temp;

// result = in_rope1 * cos_rope + neg_temp
fma.rn.bf16 %result, %in_rope1,
            %cos_rope, %neg_temp;
```

**Listing 3.** Mapping between old inline PTX (left, Triton 3.2) and simplified instruction (right, Triton 3.3).

```
mov.b32       %r0, 0f00000000;
mov.b32       %r1, %r0;
mov.b32       %r2, %r0;
mul.f32       %r3, %r4, %r5;
mul.f32       %r6, %r7, %r8;
ld.shared.v4.b32     {%r9, %r10, %r11, %r12}, [%r13];
fma.rn.f32    %r2, %r6, %r9, %r2;
fma.rn.f32    %r1, %r3, %r11, %r1;
```

```
mov.b32       %r0, 0f00000000;
mov.b64       %rd0, {%r0, %r0};
mov.b64       %rd1, %rd0;
mul.f32x2     %rd2, %rd3, %rd4;
ld.shared.v4.b32
              {%r1, %r2, %r3, %r4}, [%r5];
mov.b64       %rd5, {%r1, %r3};
fma.rn.f32x2  %rd1, %rd2, %rd5, %rd1;
```

**Listing 4.** Mapping between PTX in Triton 3.4 (left) and PTX in Triton 3.5 (right). `r`**`X`** (X is the register ID) represents a 32-bit register, while `rd`**`X`** (X is the register ID) uses two 32-bit registers to simulate one 64-bit register.

regression case to evaluate whether TritonParse can reliably reproduce the performance regression and streamline the debugging process.

**Problem.** We observe a performance regression when running the DeepSeek-R1 model [18] on the NVIDIA B200 GPU using the mixture-of-experts (MoE) benchmark [8, 22, 43] (`benchmark_moe.py`) from the vLLM library [24]. The benchmark runs with a batch size of 16384 and the `fp8_w8a8` data type. With Triton 3.4, the end-to-end execution time is 6.75 ms, whereas with Triton 3.5, it increases to 10.26 ms.

**Record and Reconstruct.** To investigate the cause, we use TritonParse to record the compilation paths and launch traces on one version, because the same reproducer is sufficient to trigger the issue in both versions. After recording, TritonParse processes the raw logs into structured archives, which support the subsequent reproduction and regression analysis without further execution of the original application. The processed trace files reveal two `fused_moe_kernel` instances with the same names, but different autotune configurations.

**Reproduce.** TritonParse extracts the MoE kernel and its cross-function dependencies to produce a reduced single-kernel reproducer. During both the reproducer generation and subsequent replay, there is no need to download the original ~500 GB DeepSeek-R1 model or install its runtime dependencies (e.g., vLLM). The isolated kernel runs in 2.85 ms on Triton 3.4 and 4.22 ms on Triton 3.5 (averaged over ten runs), confirming that the regression is reproducible in isolation.

Using the generated reproducer as the test script, we invoke TritonParse's automated regression analysis workflow to examine the relevant commit history and bisect the regression across LLVM revisions. The automated bisection localizes the regression to a change in the NVIDIA PTX backend where packed-`f32` instructions are added for NVIDIA Blackwell and newer GPU architectures. We then analyze the localized compiler change and its generated code. Concretely, the new optimization lowers pairs of `f32` arithmetic operations (e.g., `fadd`, `fmul`, `fma`) into a single `f32x2` (two-float) instruction. While this consolidation may reduce instruction count in ideal code paths, in our case, it results in additional register pressure and triggers register spills to GPU local memory that degrade end-to-end throughput.

**Review.** Using the cross-trace IR view, we confirm that the only observable difference between the two versions is in the generated PTX, as shown in Listing 4. Triton 3.5 uses one `fma.rn.f32x2` instead of two `fma.rn.f32` instructions in Triton 3.4, and replaces two `mov.b32` register preparation instructions with one `mov.b64`. In the example, **`X`** denotes the register ID, `r`**`X`** denotes a 32-bit register, and `rd`**`X`** a 64-bit register spanning two adjacent 32-bit registers. Although PTX register IDs are virtual and not directly comparable, `rd` registers consume adjacent register pairs, which can reduce allocation flexibility and increase register pressure.

**Solution.** After the issue is identified, we rerun autotuning with a larger configuration set to recover a better setting. In this case, TritonParse helps to narrow down the root cause and reduces the overall time for localization, validation, and repair. We reported the identified issue to the LLVM repository, where it was subsequently fixed and released in a later version, providing additional upstream validation of our diagnosis.

**Table 2.** Comparison of TritonParse with similar debugging tools or frameworks across different features. Prior tools target isolated stages: compiler IR ( tlparse [40]), GPU profiling (Nsight Compute [33], Nsight Systems [34], and ROCm profiler [1]), IR reduction (LLVM bugpoint [29]), environment reproduction (ReFrame [23], EasyBuild [11, 15], ReproZip [5], and Spack [13]), or CPU record-replay (BugNet [32], iReplayer [28], rr [36], REPT [7], Kernel REPT [14], Rerun [19], PRES [38], DoublePlay [46]). The table uses a color scheme of blue, light blue, and white to indicate yes, partial support, and no support, respectively.

| Capability | tlparse | GPU Profilers | LLVM bugpoint | CPU Record–Replay | Environment Reproduction | TritonParse |
|---|---|---|---|---|---|---|
| **Multi-IR Capture** | Kernel only | No | No | No | No | Full pipeline |
| **Kernel Debugging** | No | Perf metrics only | No | CPU only | No | GPU only |
| **Reproducer Generation** | No | No | LLVM IR only | Program level | Environment only | Kernel level |
| **Automated Bisection** | No | No | LLVM only | No | No | Triton & LLVM |
| **GPU Support** | NVIDIA & AMD | NVIDIA or AMD | No | No | No | NVIDIA & AMD |

This case study shows how TritonParse's workflow enables end-to-end regression analysis: from recording a workload, through reproducing the issue in isolation, to automatically identifying the specific LLVM commit responsible, after which the developer uses the preserved debugging evidence to analyze the change and determine the repair.

## 6 Related Work

We organize related work by the 4R framework in this paper.

**Record.** LLVM and Multi-Level IR Compiler Framework (MLIR) [26] expose pass pipelines and IR dumps for inspection and transformation. For Triton specifically, tlparse [40] provides debugging options, exposes PyTorch compiler IRs, and generates kernel code. However, they operate only at the PyTorch compilation layers and kernel entry points; they do not capture Triton's internal IR stages or provide stable cross-stage identifiers to connect multi-IR to backend code.

**Record and Review.** Vendor tools such as Nsight Compute [33], Nsight Systems [34], and ROCm Profiler [1] provide detailed runtime metrics and source-to-assembly mappings, but are heavyweight, contain only limited IR stages, and lack cross-vendor multi-IR alignment to backend code.

**Reconstruct.** Test-case minimization has been widely studied in software testing, notably with *delta debugging* [48]. In compilers, LLVM's bugpoint [29] reduces IR programs to isolate miscompilation bugs. However, it operates on a single IR level and assumes a fixed backend. Systems such as ReFrame [23], EasyBuild [11, 15], ReproZip [5], and Spack [13] support software-environment reconstruction for regression testing and reproducible deployment. But they do not preserve the IRs and artifacts needed for kernel-level debugging.

**Record and Reproduce.** Record/replay systems such as BugNet [32], iReplayer [28], rr [36], REPT [7], Kernel REPT [14], Rerun [19], PRES [38], and DoublePlay [46] support deterministic replay for reproducible debugging, but primarily on CPUs and operating systems.

We provide a comparative table summarizing the differences between TritonParse and existing debugging tools or frameworks across various feature dimensions (see Table 2). To our best knowledge, no prior system captures, correlates, and deterministically reproduces multi-stage GPU kernel compilation artifacts. TritonParse complements them by bridging the gap between them, rather than replacing existing debugging or profiling tools.

## 7 Conclusion

Debugging GPU kernels in Triton has lagged behind the rapid adoption of the language itself, leaving developers to reason about correctness and performance through fragmented tools and manual inspection of opaque compilation pipelines. TritonParse closes this gap with a unified 4R framework that records full compilation context, reconstructs cross-IR provenance, supports interactive review, and produces portable reproducers that travel from a developer's workstation to CI and back. Experiments show high replay fidelity with modest runtime and storage overhead, and two case studies demonstrate the framework on a real numerical regression and a real performance regression.

More broadly, TritonParse establishes that multi-stage DSL compilers can be made debuggable without compiler modification or vendor-specific instrumentation, and that the same trace can serve human inspection and automated bisection from a single capture. As DSL-based GPU programming continues to expand across AI workloads, integrating debugging support with agentic systems for generating or optimizing Triton kernels is a promising future direction. At the same time, debugging frameworks such as TritonParse will become increasingly important infrastructure rather than optional tooling.

## Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank the open-source software contributors, including Guanda Zhu, for improving TritonParse's flexibility and portability across hardware platforms. This work is supported partially by the National Science Foundation (US) under Awards CCF-2402804 and OAC-2441804.

## References


[1] AMD. 2025. *AMD ROCm*. Retrieved July 20, 2025 from https://rocm.docs.amd.com/en/latest

[2] 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. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, 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/jax-ml/jax

[4] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 473–484.

[5] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, 2085–2088. doi:10.1145/2882903.2899401

[6] James Clause and Alessandro Orso. 2007. A technique for enabling and supporting debugging of field failures. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 261–270.

[7] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. {REPT}: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 17–32.

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

[9] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*. PMLR, 2024–2033.

[10] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1017–1032.

[11] EasyBuild Community. 2025. *EasyBuild: Building Software with Ease*. Retrieved August 15, 2025 from https://easybuild.io/

[12] Jakob Engblom. 2012. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 1–6.

[13] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R De Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[14] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 281–292.

[15] Markus Geimer, Kenneth Hoste, and Robert McLay. 2014. Modern scientific software management using easybuild and lmod. In *2014 First International Workshop on HPC User Support Tools*. IEEE, 41–51.

[16] Scott Gray, Alec Radford, and Diederik P Kingma. 2017. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224* 3, 2 (2017), 2.

[17] Yue Guan, Yuanwei Fang, Keren Zhou, Corbin Robeck, Manman Ren, Zhongkai Yu, Yufei Ding, and Adnan Aziz. 2025. KPerfIR: Towards an Open and Compiler-centric Ecosystem for GPU Kernel Performance Tooling on Modern AI Workloads. *arXiv preprint arXiv:2505.21661* (2025).

[18] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[19] Derek R Hower and Mark D Hill. 2008. Rerun: Exploiting episodes for lightweight memory race recording. *ACM SIGARCH computer architecture news* 36, 3 (2008), 265–276.

[20] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989* (2024).

[21] Shan Huang, Jingjing Liang, Ting Su, and Qirun Zhang. 2025. Robustifying Debug Information Updates in LLVM via Control-Flow Conformance Analysis. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 527–549.

[22] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.

[23] Vasileios Karakasis, Theofilos Manitaras, Victor Holanda Rusu, Rafael Sarmiento-Pérez, Christopher Bignamini, Matthias Kraushaar, Andreas Jocksch, Samuel Omlin, Guilherme Peretti-Pezzi, João P. S. C. Augusto, Brian Friesen, Yun He, Lisa Gerhardt, Brandon Cook, Zhi-Qiang You, Samuel Khuvis, and Karen Tomko. 2020. Enabling Continuous Testing of HPC Systems Using ReFrame. In *Tools and Techniques for High Performance Computing (Communications in Computer and Information Science, Vol. 1190)*, Guido Juckeland and Sunita Chandrasekaran (Eds.). Springer International Publishing, Cham, Switzerland, 49–68. doi:10.1007/978-3-030-44728-1_3

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[27] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021.

[28] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-replay for Multithreaded Applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 344–358.

[29] LLVM Project. 2025. *LLVM Bugpoint Tool*. Retrieved August 7, 2025 from https://llvm.org/docs/Bugpoint.html

[30] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.

[31] Meta PyTorch Team. 2025. *TritonBench*. Retrieved August 10, 2025 from https://github.com/pytorch-labs/tritonbench

[32] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. Bugnet: Continuously recording program execution for deterministic replay

debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 284–295.

[33] NVIDIA. 2025. *NVIDIA Nsight Compute*. Retrieved August 7, 2025 from https://developer.nvidia.com/nsight-compute

[34] NVIDIA. 2025. *NVIDIA Nsight Systems*. Retrieved August 7, 2025 from https://developer.nvidia.com/nsight-systems

[35] NVIDIA. 2025. *PTX ISA*. Retrieved August 14, 2025 from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[36] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

[37] OpenAI. 2025. *GPT-OSS-120b & GPT-OSS-20b Model Card*. Model Card/Technical Report. OpenAI. https://cdn.openai.com/pdf/419b6906-9da6-406c-a19d-1bb078ac7637/oai_gpt-oss_model_card.pdf

[38] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. 2009. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 177–192.

[39] 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).

[40] Pytorch. 2025. *tlparse*. Retrieved July 30, 2025 from https://github.com/pytorch/tlparse

[41] PyTorch Compilers Team, Meta. 2025. *Helion*. Retrieved August 25, 2025 from https://github.com/pytorch/helion

[42] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[43] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[44] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[45] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[46] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–24.

[47] Dewei Wang, Wei Zhu, Liyang Ling, Ettore Tiotto, Quintin Wang, Whitney Tsang, Julian Opperman, and Jacky Deng. 2025. ML-Triton, A Multi-Level Compilation and Language Extension to Triton GPU Programming. *arXiv preprint arXiv:2503.14985* (2025).

[48] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* 28, 2 (2002), 183–200.

# A Artifact Appendix

## A.1 Abstract

TritonParse implements the Record, Reconstruct, Review, and Reproduce workflow for Triton kernels. The artifact contains the tool, a reference trace, and automated checks of its core functionality on reduced workloads. The checks run on one NVIDIA GPU; an optional browser interface supports visual inspection. This appendix describes how to run the artifact, interpret its outputs, and understand its coverage of the paper's experiments.

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

- **Algorithm:** Record/reconstruct/review/reproduce workflow.
- **Program:** TritonParse 0.5.1, Triton 3.7.1, PyTorch 2.13.0.
- **Compilation:** Prebuilt Python wheels; optional UI build using Node.js.
- **Transformations:** Three bug patches and transcribed PTX excerpts.
- **Binary:** None bundled; dependencies installed at pinned versions.
- **Data set:** Reference trace (~420 KB), with provenance.
- **Run-time environment:** Linux x86-64 with a shell and `curl`; an NVIDIA driver supporting CUDA 13.0 (validated on 580.126.09).
- **Hardware:** One NVIDIA GPU, compute capability ≥ 8.0, with ≥ 16 GB memory.
- **Run-time state:** Cache state affects elapsed time.
- **Execution:** No root privileges, credentials, model downloads, exclusive GPU access, or pinning required.
- **Metrics:** IR coverage, storage reduction, reproducer size and exit status, fault class, BF16 divergence, and test pass counts.
- **Output:** PASS/FAIL summaries, logs, and CSV measurements.
- **Experiments:** Two required scripts and one optional UI build; acceptance criteria in subsection A.6.
- **How much disk space required (approximately)?:** 5.1 GB for the environment; 5 MB for the source tree, plus build and scratch output.
- **How much time is needed to prepare workflow (approximately)?:** Allow 5 minutes; measured setup took 78 seconds with a warm pip cache and 236 seconds without.
- **How much time is needed to complete experiments (approximately)?:** 2 minutes for basic checks and 8 for GPU checks; 15 minutes including setup. The optional UI build adds about 1 minute, followed by manual inspection.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD-3-Clause.
- **Data licenses (if publicly available)?:** BSD-3-Clause.
- **Workflow automation framework used?:** Shell and Python.
- **Archived (provide DOI)?:** Yes. The artifact is publicly archived on Zenodo and is available at https://doi.org/10.5281/zenodo.21853863.

## A.3 Description

**A.3.1 How to access.** The artifact is public in the TritonParse repository. Obtain the fixed version with:

```
git clone \
  https://github.com/meta-pytorch/tritonparse
cd tritonparse
git checkout f9634c64f16a
```

Run subsequent commands from this repository root. Scripts, manifests, the reference trace, and measurements are in `asplos2027-ae/`. Its README provides detailed instructions and claim-to-script mappings. The artifact includes an optional standalone UI and instructions for accessing the archived version. Computational checks and reference data are provided to validate the artifact's functionality and selected paper claims. Git-based runs record the branch, commit, and local-change status. Runs from an archive are identified as a source archive; its filename and archival record identify the commit.

**A.3.2 Hardware dependencies.** Paper claim C9 requires BF16 instructions available from `sm_80`; SM75 GPUs cannot complete it. Suitable GPUs include A100, L4, L40S, and RTX 3090. Native `mul.bf16` requires `sm_90`. On SM80–SM89, the Triton 3.2 sequence in Listing 3 executes as inline PTX while the Triton 3.3 side uses a semantic model. On SM90 or newer, both sequences execute on hardware and are checked against their models. The H100, MI300X, and B200 configurations used for the paper's experiments are not required. The transcribed PTX sequences are shipped under `claims/C9_rope/`, so they can be inspected without installing the historical compiler versions.

**A.3.3 Software dependencies.** The conda environment supplies Python and Node.js. Setup uses the versions listed in `requirements-pinned.txt` and installs TritonParse from PyPI. PyTorch bundles CUDA userspace libraries; no separate CUDA toolkit or C++ compiler is needed. Bug patches also have a Python applier, so `patch` is optional. The optional UI uses npm; its File Diff editor loads Monaco from `cdn.jsdelivr.net` and requires access to that host.

Alternatively, a virtualenv on Python ≥ 3.11 can install the same requirements; Node.js must then be supplied separately for the UI. The README lists `cu126` and `cu129` wheel alternatives for older drivers; PyTorch 2.13.0 has no `cu128` build.

Unpacked source archives use the default PyPI installation without Git metadata. Manual installation from an archive requires setting `SETUPTOOLS_SCM_PRETEND_VERSION_FOR_TRITONPARSE=0.5.1` before invoking `pip install .` at the repository root.

## A.4 Installation

The entry point invokes `setup_env.sh` automatically. It reuses conda 23 or newer, or downloads and SHA-256-verifies Miniforge into `asplos2027-ae/.conda`. The script does not run `conda init` or edit shell startup files. Miniforge uses conda-forge and avoids the Anaconda `defaults` channel's

Terms-of-Service prompt; the README covers that error for existing conda installations.

By default, the `tritonparse-ae` environment is recreated from the manifest. Set `AE_KEEP_EXISTING_ENV=1` to retain it. For environment-only setup, run `./asplos2027-ae/setup_env.sh`.

### A.5 Experiment workflow

`./asplos2027-ae/run_all.sh`

The entry point runs the following stages, stopping at the first failure. Each run saves logs, package versions, and CSV results in a timestamped directory under `asplos2027-ae/logs/`.

The optional UI build can be omitted with `--skip-ui`. Use `--skip-setup` when the environment is already activated. To run the remaining stages after a check fails, pass `--keep-going`; use `--quiet` to send detailed output to the log files. Each stage can also run separately and activates the artifact environment itself.

**Basic checks.** The script `00_kick_the_tires.sh` reconstructs the reference trace into an IR-mapped archive and queries its kernels and launches (C2, subsection 3.2); compares compilations with source-line attribution (C3, subsection 3.3); generates a reproducer (C4, subsection 3.4); and runs 501 CPU tests. After installation, these checks use the shipped trace and local packages, without a GPU or network access. The compilation comparison exercises the multi-IR attribution used in both case studies.

**GPU checks.** The script `01_gpu_core.sh` records TTIR, TTGIR, LLIR, PTX, and launch metadata (C1, subsection 3.1); executes a fresh reproducer consisting of one Python file from a clean directory (C4); measures storage reduction (C6, subsection 4.3); reproduces assertion, illegal-memory-access, and hang faults (C8, subsection 4.4); tests BF16 rounding (C9, subsection 5.1); and runs 28 GPU tests. The reproducer uses the pinned environment without importing the original workload. By default, two 30-second timeouts cover the original and reproduced hangs. The CPU and GPU suites also run in project CI.

**Visual inspection.** The optional script `02_review_ui.sh` builds a multi-file site in `website/dist/`. Use the printed commands to activate the artifact environment and start an HTTP server, then open http://127.0.0.1:4173/ and load the supplied trace. Check IR selection, linked Python/IR highlighting, and File Diff between compilations. Build success and browser inspection are separate checks. The interface is also hosted at https://meta-pytorch.org/tritonparse/; trace processing occurs in the browser.

### A.6 Evaluation and expected results

The paper's experiments use PyTorch 2.9.0, Triton 3.4.0, and CUDA 12.8 (Table 1). The artifact pins newer software and uses different hardware and workloads. Its checks assess functionality and numerical mechanisms without requiring equality with the paper's measurements.

**Acceptance criteria.** Structural checks require all expected IR stages, successful standalone execution, and matching fault classes. Claim C6 requires mean raw-to-reconstructed size reduction $\geq 10\times$ over three repetitions. C9 requires mean divergence of 15–40 % over five seeds with 10,000 samples each, and zero mismatches between the executed PTX sequences and their semantic models. On SM80–SM89, this independent consistency check applies to the Triton 3.2 side. Reported variability is mean ± half-range. Differences from `data/reference/` marked `DIFF` are informational; failed checks or missing required outputs still fail the run.

**Reference measurements.** On an H100, C6 produces 74–77× reduction, compared with the paper's 57.4× (Figure 5); C9 produces 24.2 % ± 0.7 divergence, compared with 25.84 %. Runtime gzip leaves about 50 % of raw log size, compared with the paper's 21.9 %. It compresses each record separately, and the artifact's synthetic autotuned-matmul workload differs from the paper's 37-operator TritonBench mix. For comparison, whole-file gzip of the same workload's log leaves about 0.84 % of raw. These gzip measurements are informational; the reconstruction ratio determines whether C6 passes.

**Coverage limits.** C8 applies the three mutations from Listing 2 to a 30-line kernel, exercising 3/3 fault classes rather than the paper's 15 mutations across five `gpt-oss-20b` kernels. Illegal-memory-access checks match fault classes because allocator-dependent addresses can produce different CUDA error codes. One possible code is `cudaErrorIllegalAddress`. Another code accepted in the same class is `cudaErrorInvalidAddressSpace`. C9 uses transcribed PTX and validates rounding behavior; it does not re-run the Triton 3.2-to-3.3 upgrade, which the current tool cannot instrument (`triton` > 3.3.1 is required). Recording overhead (subsection 4.2) is not covered: the paper measures 37 TritonBench operators on H100 and MI300X, and a smaller single-GPU run cannot substantiate those hardware- and workload-dependent percentages. The MoE regression and LLVM bisection (subsection 5.2) require a B200, vLLM, and repeated LLVM builds, and are also outside scope.

### A.7 Experiment customization

Set `SKIP_UNITTESTS=1` to omit the test-suite stages. The variable `HANG_TIMEOUT` sets the hang limit (30 seconds by default). The flag `--assume-no-native-mul` makes the C9 study exercise the SM80–SM89 model path even on SM90 or newer GPUs. The README documents the directory and trace-selection options. To regenerate an equivalent reference trace on the installed stack and GPU, run `scripts/make_reference_trace.py`.