# COMPILER-GROUNDED HIERARCHICAL DIAGNOSIS FOR LLM-BASED TRITON KERNEL OPTIMIZATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Recent advances in large language models (LLMs) have enabled automated kernel generation and optimization, but most existing approaches rely on surface signals such as compilation feedback and profiling metrics, which reveal that a kernel is slow but not why the backend compiler fails to realize a profitable optimization, especially on emerging accelerators such as NPUs. We therefore formulate kernel optimization as a progressive cross-layer diagnosis problem that links runtime symptoms to IR structure and compiler behavior before rewriting source, and present a compiler-grounded, hierarchical optimization framework for Triton kernels that escalates from lightweight pattern triage and profiling diagnosis to IR attribution and compiler-grounded analysis only when deeper evidence is needed. Implemented on Triton for Ascend NPUs and evaluated on 37 successfully converted entries from a standardized NPUKernelBench-derived Ascend 950 benchmark, the system attains a geometric-mean speedup of 4.35× and a median of 2.73× from the initial to optimized Triton kernel; 22/37 exceed 2× and 13/37 exceed 5×. The complete distribution ranges from near-baseline entries to large wins, motivating transparent reporting of the current system's scope and limitations.

## 1 INTRODUCTION

Efficient kernel implementation is critical for achieving high performance on modern accelerators. Domain-specific languages such as Triton have significantly lowered the barrier for writing high-performance kernels Tillet et al. (2019), and recent advances in large language models (LLMs) have inspired a rapidly growing line of work on LLM-driven kernel generation and optimization, where code agents iteratively generate, execute, and refine kernels based on runtime feedback Yu et al. (2026); flagos-ai (2026).

Most existing LLM-based approaches, including recent profile-guided optimization systems, still optimize primarily at the source-code level Li et al. (2025a); Lei et al. (2025); Wei et al. (2025). They rely on shallow signals such as compilation errors or runtime metrics, which can reveal that a kernel is slow but rarely explain why: many performance failures emerge only after the kernel is lowered through intermediate representation (IR) stages, scheduling decisions, and backend-specific compiler passes, leaving a fundamental semantic gap between operator-level code and backend performance behavior. This gap is particularly severe on emerging accelerators such as NPUs (e.g., Ascend), where the Triton-Ascend stack is still evolving and backend behavior is target-specific and relatively opaque to developers triton-lang contributors (2026). LLM agents, typically pretrained on GPU-centric code patterns, therefore behave like trial-and-error optimizers: they propose plausible source edits but lack the evidence to map a runtime symptom to a compiler-level cause.

The key insight of this paper is that kernel optimization should be treated as a *progressive cross-layer diagnosis problem*: a useful optimizer should first exploit low-cost evidence, then escalate to deeper compiler grounding only when shallower evidence is insufficient. In this view, compiler grounding is not an isolated feature; it is the deepest layer in a hierarchical reasoning process that connects *profiling signals* → *IR-level evidence* → *compiler transformations* → *source-level rewrites*.

Based on this insight, we propose a compiler-grounded optimization system for Triton kernels. the system forms a closed-loop optimization pipeline that starts from pattern triage, escalates through profiling diagnosis and IR attribution, invokes compiler-source escalation when needed, and finally proposes evidence-backed source-level rewrites. The hierarchy makes compiler grounding practical:



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

```
 group_values = tl.gather(all_scores, group_offsets, 0)
-top1 = tl.max(group_values, axis=0)
-top1_idx = tl.argmax(group_values, axis=0)
-rest = tl.where(offsets == top1_idx, -float("inf"), group_values)
-group_score = top1 + tl.max(rest, axis=0)
+pair_sums = group_values[:, None] + group_values[None, :]
+off_diagonal = offsets[:, None] != offsets[None, :]
+pair_sums = tl.where(off_diagonal, pair_sums, -float("inf"))
+group_score = tl.max(tl.max(pair_sums, axis=1), axis=0)
```

**Figure 1:** Actual source rewrite for the MoE group-score kernel. The maximum off-diagonal pair sum equals the sum of the two largest group values, so the rewrite removes index-producing reduction while preserving ties.

deeper compiler-side reasoning is invoked only when lighter-weight evidence cannot yet produce a sufficiently grounded rewrite. Beyond this per-operator online loop, the system also performs a slower retrospective synthesis pass over validated benchmark batches, so that repeated optimization histories can refine existing pattern guidance, reduce repeated diagnostic cost across operators, and introduce new reusable pattern families for later runs.

We implement the system on top of Triton for Ascend NPUs and evaluate it with an OpenCode agent backed by DeepSeek V4 Pro on a NPUKernelBench-derived Ascend 950 suite. After removing failed Torch-to-Triton conversions, the evaluation cohort contains 37 successfully converted operators, each benchmarked on five cases sampled from its original benchmark inputs (185 cases in total). The geometric-mean initial-to-optimized Triton kernel speedup is 4.35× and the median is 2.73×; the full distribution shows that 22/37 entries exceed 2× and 13/37 exceed 5×, demonstrating substantial headroom on a subset of operators but not uniform improvement. The per-round optimization trajectories further show that 22 entries first reach their best validated result after round 8, while 32 selected paths reach profiling diagnosis, 3 reach IR attribution, and 4 reach compiler-source escalation. We therefore present the paper as a systems argument with transparent benchmark and process evidence, reserving causal decomposition of the hierarchy for future controlled experiments.

In summary, this paper makes the following contributions:

- We formulate LLM-based kernel optimization as a progressive cross-layer diagnosis problem and identify the lack of compiler-grounded evidence as a key reason profiling-only agents fail on backend-specific optimizations.
- We describe a hierarchical workflow that escalates from pattern triage to profiling diagnosis, IR attribution, and compiler-source escalation before proposing evidence-backed source-level rewrites.
- We present standardized, distribution-aware evaluation evidence on a NPUKernelBench-derived Ascend 950 benchmark, reporting aggregate gains, best-round and evidence-depth accounting, and a representative cross-layer trace.

## 2 Motivation

### 2.1 A Running Example

Consider a Mixture-of-Experts group-score Triton kernel optimized in our experiment: for every group of 32 expert scores, the kernel must compute the sum of its two largest values. Its initial implementation obtains the first maximum, uses `tl.argmax` to identify the corresponding index, masks that index, and then reduces again for the second maximum; Figure 1 shows the later source rewrite selected after cross-layer analysis.

Profiling identifies a mixed execution profile: 71% vector activity but 26% scalar activity. Plausible source-level reactions—forcing a scalar-oriented SIMT mode or replacing `argmax` with value-based masking—are both insufficient: the former severely regresses because it disables the productive vector path, while the latter fails correctness on low-precision ties. IR attribution then ties 26 synchronization operations in the early lowered stages to the `argmax` reductions and index-mask construction, and compiler-source analysis explains the mechanism: vector maxima are supported natively, whereas

`argmax` requires an additional scalar index-tracking stage with synchronization. A successful rewrite must therefore avoid index-producing reduction altogether.

The resulting algebraic reformulation is shown in Figure 1: $\max_{i \neq j}(v_i + v_j)$ is exactly the sum of the two largest group values, including tied maxima. It replaces the `argmax`-and-mask chain with vectorized pairwise addition and value-only reduction, passes all 50 correctness cases, and reaches 1.42× geometric-mean speedup.

This is the failure mode we care about: a profile can identify a scalar-heavy path, but it cannot determine whether the scalar work originates in source control flow, an IR-level lowering, or a backend implementation requirement—the lowering mechanism that distinguishes these hypotheses is hidden behind compiler transformations, which is exactly where source-level trial-and-error becomes unreliable. This motivates the central challenge of the paper: *the relationship between source-level code and runtime performance is mediated by compiler transformations that are invisible to surface-level optimization signals.*

### 2.2 Limitations of Existing Approaches

Existing LLM-based kernel systems still optimize largely at the level of source generation or mutation. Reward-loop methods such as SparseRL Wang et al. (2026), Kevin Baronio et al. (2025), CUDA-L1 Li et al. (2025c), and KernelEvolve Liao et al. (2025) show that runtime reward can substantially improve correctness and throughput, but share the same abstraction: kernel text is the object of search, while the compiler is treated as a black box that returns performance or failure signals. That abstraction is powerful for broad exploration in source space, but brittle when multiple source rewrites induce similar runtime symptoms while diverging only after lowering: in the running example, forcing execution mode or masking values by equality both look reasonable under end-to-end reward, yet neither supplies the tie-correct, vectorized reformulation implied by the lowering evidence.

Profiling-guided systems such as TritonForge, PRAGMA, Astra, and KForge move one step closer to diagnosis by adding hotspots, hardware-counter summaries, and multi-agent reasoning around execution traces Li et al. (2025a); Lei et al. (2025); Wei et al. (2025); Sereda et al. (2025). However, profiling still localizes the symptom more reliably than the mechanism: it can show that a kernel is movement-heavy, scalar-heavy, or serialization-heavy, but not whether the root cause is work distribution, an IR-level layout artifact, or a backend pass precondition—an especially acute limitation for NPU-oriented generation systems such as AscendKernelGen and AscendCraft Cao et al. (2026); Wen et al. (2026).

### 2.3 Key Insight

Effective kernel optimization should therefore be cast as *progressive diagnosis across layers, with deeper compiler grounding invoked only when shallower evidence is insufficient*. This is why compiler grounding should be selective rather than always-on: it is the mechanism that turns a vague slowdown into a rewrite that is both justified and backend-aware.

## 3 System Overview

the system combines pattern triage, profiling diagnosis, IR attribution, and compiler-grounded reasoning in a unified loop: the system first asks whether a strong pattern-backed hypothesis already exists, then gathers runtime evidence from profiling, analyzes intermediate representations when needed, and finally consults compiler-grounded references when shallower analysis is insufficient, producing an evidence-backed source-level rewrite that can be evaluated and iteratively refined.

This online optimization loop is paired with a slower cross-operator knowledge-synthesis loop. After a batch benchmark sweep completes, the system rereads validated optimization histories across operators, aligns each round with an existing pattern family when possible, and promotes repeated new mechanisms into reusable guidance when no current family explains them well. Future optimization sessions therefore start from a refreshed low-cost hypothesis layer rather than from a fixed hand-authored pattern seed.

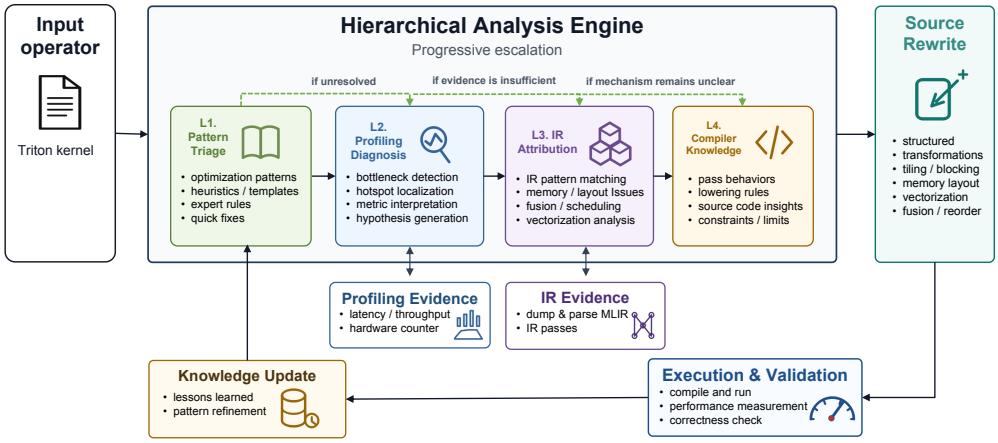



**Figure 2:** Overview of the system. The loop escalates from pattern triage to profiling, IR attribution, and compiler-grounded analysis only when lower-cost evidence is insufficient, then proposes evidence-backed source-level rewrites.

The loop is organized around explicit optimization states rather than hidden conversational context: before the first code-changing round, the system establishes a validated baseline state that fixes the executable kernel, evaluation context, and reference metrics; each later round inherits that state, records its rewrite hypothesis and supporting evidence, and completes with a validated performance comparison that defines the next decision point, so deeper analyses become reusable evidence attached to the round rather than ephemeral observations. The process can be run either as a single end-to-end session or as a sequence of externally audited rounds; in both cases, the hierarchy in Figure 2 is enforced through explicit round boundaries, round-local evidence, and validation gates. Crucially, the system does not rely on compiler-level reasoning for every opportunity: inexpensive pattern triage is applied first, and progressively deeper reasoning is invoked only when required, which improves efficiency, reduces unnecessary context overload for the agent, and preserves interpretability—compiler grounding is the deepest diagnostic stage in a progressive loop whose cheapest stage can itself improve over time through retrospective synthesis.

## 4 METHOD

the system is designed around a central systems question: how can an optimization system access enough compiler information to diagnose backend-specific failures without forcing deep compiler reasoning on every optimization round? It answers this question with a hierarchical diagnostic workflow in which each level consumes the evidence accumulated so far and either proposes a grounded rewrite hypothesis or escalates to a deeper analysis stage.

### 4.1 HIERARCHICAL DIAGNOSTIC WORKFLOW

the system organizes optimization as four levels of progressively more specific evidence, traversed as an ordered analysis ladder: *pattern triage*, *profiling diagnosis*, *IR attribution*, and *compiler-source escalation*.

At **L1** (**Pattern Triage**), the system asks whether a strong pattern-backed hypothesis already exists. The workflow first consults a curated pattern index and then opens only the one or two detailed pattern references that match the current symptom. This guidance mixes publicly documented Triton idioms (autotuning, program reordering, block pointers, compiler hints) Tillet et al. (2019); Triton contributors (2026c;a;b), selected Ascend-oriented tuned operators from prior public optimization materials Ascend (2026), and internal patterns distilled from prior validated optimization histories. If code inspection, benchmark shape, and existing validated evidence already justify a low-risk rewrite, the round may stop here without invoking profiling or IR capture, but that decision must be recorded explicitly in the round rationale. L1 is also not a fixed knowledge source: after batched benchmark runs, the system performs a retrospective synthesis pass that maps each round to an existing pattern

family when possible and creates a new reusable pattern family—with its preconditions, rewrite, evidence, and interpretation—when no current one explains the mechanism, so future sessions start from accumulated cross-operator evidence.

At **L2** (**Profiling Diagnosis**), the system gathers runtime evidence to localize the bottleneck class. It builds a structured execution view of the operator on the NPU—kernel time, data-movement behavior, API-side overhead, and utilization signals such as AIV/AIC occupancy and overlap—which determines whether the issue is likely dominated by memory behavior, pipeline overlap, or scheduling overhead, without yet explaining the compiler mechanism.

At **L3** (**IR Attribution**), the system inspects intermediate representations (IR) to translate the runtime symptom into a structural diagnosis, such as non-unit-stride loads, missed fusion, or layout-induced vectorization failures, reconstructing the operator's stage-by-stage IR trajectory as an evidence view of stage summaries, adjacent-pass changes, and performance-relevant structural signatures; if the diagnosis already maps to a known rewrite template, the system can act at this stage.

At **L4** (**Compiler-Source Escalation**), the system invokes compiler-grounded analysis to explain why the diagnosed pattern blocks backend optimization, asking a narrowed question—which pass-level constraint, legality condition, or backend assumption prevents the candidate optimization from materializing—in order to recover the source-level conditions a successful rewrite must satisfy.

the system escalates only when the current level cannot produce a sufficiently grounded rewrite hypothesis, using three concrete tests. First, the current level must identify a dominant hotspot or bottleneck class; evidence that only says "performance is still low" cannot stop the round. Second, the level must narrow the next change to one evidence-backed rewrite family; if competing explanations remain plausible, the system escalates rather than choosing arbitrarily. Third, the level must justify the rewrite from evidence already exposed at that level; if the remaining question concerns lowering behavior, legality conditions, or pass-specific constraints, the policy escalates instead of treating the effect as source-level tuning. Escalation stops once a stage can state a concrete rewrite hypothesis together with the evidence justifying it. Every completed round records its hypothesis, evidence sources, comparison target, and next recommendation; a deeper stage that yields no actionable diagnosis is recorded as inconclusive rather than forcing a speculative rewrite. Skipped evidence is thus visible and justified, failed escalations remain auditable, and deeper analysis leaves persistent diagnostic records that later rounds and batch synthesis can reuse.

### 4.2 Profiling-to-IR Mapping

Given a candidate kernel, the system first profiles the benchmarked operator under a standardized measurement protocol, converting the raw trace into operator-level timing summaries and structured execution signals such as data-transfer pressure, API overhead, and device-side utilization. When timing evidence alone is not sufficient, the system reconstructs the same operator's lowering trajectory across Triton and downstream compiler IR, producing a provenance record that ties the observed hotspot to specific stage-to-stage transformations. The resulting IR view summarizes stage-local signatures, pass-to-pass changes, and performance-relevant structural patterns; this does not make attribution fully automatic, but it narrows the diagnosis to a concrete lowering path that can support the next rewrite decision.

### 4.3 Compiler-Grounded Analysis

When benchmark, profiler, and IR evidence are no longer sufficient, the system first consults compiler knowledge already distilled from earlier source analysis into pass-oriented references, narrowed question templates, and backend-specific invariants. For example, one such reference records that Ascend vectorized memory paths usually require aligned power-of-2 block widths together with explicit contiguity or alignment information, so a non-power-of-2 tile or missing contiguity hint can explain why an apparently contiguous load sequence remains scalarized. These references let the system reformulate an IR symptom into a concrete compiler question, such as which pass family would explain a suspicious stage transition or which backend constraint would cause a vectorization loss. Only when the references leave the answer unresolved does the system escalate to a focused source inspection of the specific compiler subsystem implicated by the current evidence. The output is therefore not a general compiler explanation, but a localized account of why the current IR pattern

blocks backend optimization and what source-level conditions a successful rewrite must satisfy. Because compiler reasoning is triggered only after earlier evidence has localized the question, it acts in practice as a late explanation-and-constraint recovery stage rather than as another unconstrained search space.

## 5 IMPLEMENTATION

We implement the system as an agent-centric optimization environment built on top of existing code-agent tooling rather than as a bespoke compiler service. It is organized around two layers: a thin execution harness, in the sense of harness engineering for code agents OpenAI (2025), and a set of optimization skills aligned with the hierarchy. The harness makes evidence collection and validation reliable: it establishes the validated baseline, runs correctness tests and benchmark comparisons, gathers profiler outputs, captures IR artifacts, and records every round as an explicit state with its parent, evidence, validation outcome, and performance comparison. An explicit round-state machine enforces the lifecycle from validated baseline through proposed round to submitted validation result, and an agent hook checks the round contract—that the intended evidence depth and the validation record are present and mutually consistent—before the workflow advances. This separates workflow compliance from optimization strategy: the state machine and hook make omissions and invalid transitions visible, while the agent remains responsible for selecting the next rewrite. Because this layer is thin, the same protocol can be reused with different code agents, including Codex OpenAI (2026) and OpenCode OpenCode (2026).

The optimization logic itself is realized as a set of agent skills that mirror the hierarchy in Figure 2. Pattern-triage skills consult the curated pattern index and synthesize low-cost rewrite hypotheses; profiling skills convert execution traces into bottleneck diagnoses; IR-attribution skills reconstruct the lowering trajectory and surface performance-relevant structural changes; compiler-grounded skills turn unresolved IR symptoms into narrow compiler questions and recover the source-level constraints a rewrite must satisfy; finally, rewrite skills map these diagnoses into source-level edits validated against the round's benchmark and correctness criteria.

This organization keeps the implementation close to the paper's core claim: the harness enforces auditable optimization rounds, while the skills decide how to spend analytical depth within each round, and the retrospective synthesis pass feeds validated mechanisms back into pattern guidance without entangling reusable knowledge with the execution substrate.

## 6 EVALUATION

We evaluate the system on an NPUKernelBench-derived suite that ports selected KernelBench operators to Ascend 950 Ouyang et al. (2025); Just-it (2026), organized around three questions: **RQ1:** What kernel-level speedups does the system obtain from the initial to optimized Triton implementation? **RQ2:** Are the outcomes broadly distributed, or concentrated in a small subset of operators? **RQ3:** Do per-round optimization trajectories show the intended progression from pattern triage to deeper evidence? Table 1 answers RQ1; Figure 3 answers RQ2; and the optimization accounting plus the representative trace answer RQ3. We separately report a post-optimization comparison against Torch NPU; it is not used as the before/after optimization metric.

### 6.1 EXPERIMENTAL SETUP

For each operator, the workflow begins with a PyTorch reference and a Triton implementation whose baseline is validated before optimization. The optimization agent is OpenCode OpenCode (2026) backed by DeepSeek V4 Pro DeepSeek-AI (2026), targeting the A5 configuration of an Ascend 950 accelerator, and each operator is optimized for 15 rounds. We remove operators whose AI conversion from the Torch NPU implementation fails, leaving 37 successfully converted Triton operators. For each retained operator, we sample five cases from its original benchmark inputs to form the fixed benchmark used throughout optimization, giving 185 kernel benchmark cases in total. We take the geometric mean over the five cases for each operator and then across operators, so the main result is a kernel-level initial-to-optimized Triton comparison, not an end-to-end application latency result.

**Table 1:** Triton optimization speedup and geometric-mean kernel time summary.

| Metric | Level 1 | Level 2 | Overall |
|---|---|---|---|
| # Operators | 19 | 18 | 37 |
| Speedup (geometric mean) | 7.29× | 2.51× | 4.35× |
| Speedup (median) | 9.05× | 1.58× | 2.73× |
| Initial time (geo. mean, $\mu$s) | 142.57 | 17.72 | 51.70 |
| Optimized time (geo. mean, $\mu$s) | 19.54 | 7.05 | 11.90 |

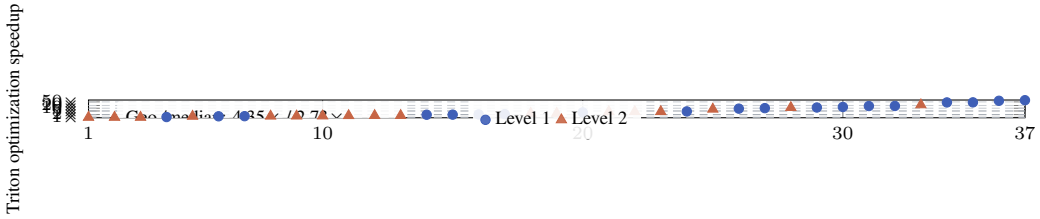



**Figure 3:** Distribution of Triton kernel optimization speedups, ordered by speedup.

### 6.2 RQ1: Aggregate Effectiveness

Across the 37 entries, the geometric-mean kernel speedup is 4.35× and the median operator speedup is 2.73×; the runtime rows of Table 1 report the associated geometric-mean kernel times before and after optimization. The level-wise geometric means in Table 1 differ substantially, but this difference is descriptive rather than causal: the two benchmark groups contain different operators and shapes, so it does not isolate a benefit of any component of the system.

### 6.3 RQ2: Outcome Distribution

Figure 3 shows a highly heterogeneous distribution: 22 of 37 entries exceed 2.0×, and 13 exceed 5.0×; the gap between the 4.35× geometric mean and the 2.73× median reflects this long-tailed behavior. The workflow can therefore produce substantial wins, but the current experiment does not establish uniform improvement or causally isolate the contribution of its four evidence levels. The remaining analyses ask a narrower, auditable question: how often did optimization need to progress beyond its shallowest evidence level?

### 6.4 Comparison with Torch NPU

The Torch NPU comparison is a separate post-optimization evaluation, not a substitute for the Triton-before/after result above: for each operator with a matching report, we compare the optimized Triton implementation with its Torch NPU counterpart over all available benchmark cases (35 operators, 1678 cases, 14–111 per operator; median 50). Table 2 reports a 2.60× geometric-mean speedup and a 1.46× median, where speedup is Torch NPU total time divided by optimized Triton total operator time; 22 of 35 entries are non-regressing, while 16 exceed 2× and 14 exceed 5×.

### 6.5 RQ3: Evidence Accounting and Mechanism

The per-round optimization trajectories let us separate final outcome measurement from process accounting. For each of the 37 entries, we consider only rounds that pass both correctness and benchmark validation, compare their geometric-mean speedups under that round's declared authority metric, and select the highest-speedup round (breaking ties by the earlier round). We assign the selected round to its primary evidence level: compiler-source, IR, and profiling evidence take precedence when attached to the round; otherwise, its explicitly stated pattern-triage label is used. Panel (b) therefore labels the evidence level of the selected best round, whereas panel (c) counts whether the complete trajectory reaches each level at least once. The best result is often not immediate: 22 of 37 entries first reach their best validated round after round 8, including 12 in the final three rounds, and the median best-round index is 10—later rounds remain empirically active.

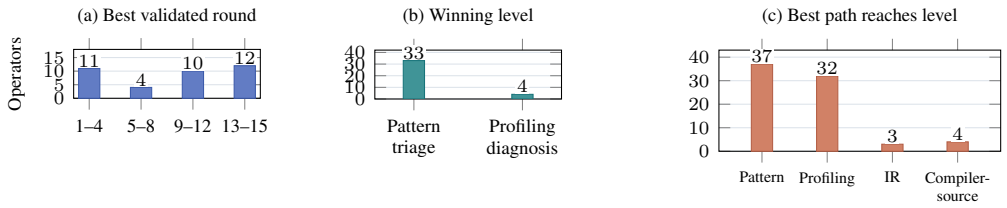



**Figure 4:** Optimization accounting. Panel (a) records the round with the best validated, comparable per-round result; panel (b) records the primary evidence level of that selected round (no selected best round is attributed to IR attribution or compiler-source escalation); panel (c) records whether the complete trajectory reaches each level at least once, so panel (b) does not measure whether deeper analysis was used before the selected round. The median best-round index is 10; 28/37 winning-round levels are explicitly labeled.

**Table 2:** Post-optimization comparison with Torch NPU (speedup = Torch NPU time / optimized Triton time).

| Group | Operators | Cases | Geo. speedup | Median |
|---|---|---|---|---|
| Level 1 | 18 | 894 | 1.41× | 0.87× |
| Level 2 | 17 | 784 | 4.96× | 5.98× |
| Overall | 35 | 1678 | 2.60× | 1.46× |

Under this accounting, 33 selected best rounds are attributed to pattern triage, 4 to profiling diagnosis, and none to IR attribution or compiler-source escalation. This does not imply that deep analysis is unused or ineffective: 32 trajectories reach profiling diagnosis, 3 reach IR attribution, and 4 reach compiler-source escalation at least once (Figure 4). The distinction reflects the hierarchy's intended role: IR and compiler-source analysis can explain a plateau, reject an incompatible pattern, or establish a backend limitation, while a subsequent retained candidate can still be a simpler pattern-level rewrite.

**Representative cross-layer trace.** The HYENA FFT-size padding operator combines profiler, IR, and compiler-source evidence in a single, validated optimization trajectory. The initial rewrite replaces a flat one-dimensional index decoder with two-dimensional row–column tiling, eliminating per-element division and remainder operations, and reaches 1.89× on the optimization benchmark. The agent then profiles the remaining kernel bottleneck and records:

*Agent profiling record.* "`aiv_scalar_ratio=48.1%, aiv_vec_ratio=12.5%`."

This directs a narrow semantic simplification rather than a blind launch-configuration search: because masked loads already supply zeros outside the live extent, the agent removes a redundant `tl.where`, reaching 2.08× geometric-mean kernel speedup on the per-round benchmark. The retained two-dimensional tiled kernel (Listing 1 in Appendix A) combines the removal of the per-element division and remainder operations with this zero-preserving simplification. When later source rewrites fail to improve this candidate, the agent escalates to IR attribution, and a compiler-source inspection then identifies the lowering mechanism:

*Agent IR record.* "`hivmave-scalar-broadcast-to-vload` pass adds 2 loads, 20 sync ops."

*Agent compiler-source record.* "Boolean 2D broadcast masks trigger `brc_scalar_core_1d`."

These findings explain why plausible alternatives—static row unrolling, split loops, a one-dimensional grid, and mask-free fast paths—all regress: they either increase program/branch overhead or add memory traffic without removing the dominant lowering cost. The agent consequently retains the simpler two-dimensional tiled kernel with the redundant conditional removed, rather than claiming that deeper analysis itself produced a new winning rewrite. Thus the trace shows the hierarchy's practical value as diagnosis and rejection: profiling identifies the symptom, IR localizes its generated cost, compiler-source analysis explains the backend constraint, and validation selects the robust implementation.

## 7 Related Work

**Compiler substrates and compiler-centric tooling.** Triton provides the immediate DSL and compilation substrate for our setting, while Triton-Ascend extends that stack to Ascend NPUs Tillet et al. (2019); triton-lang contributors (2026). Classical tensor compilers such as TVM cast operator optimization as schedule or configuration search inside a compiler-controlled transformation space Chen et al. (2018), but do not address an LLM agent that edits an existing Triton kernel while explaining the observed slowdown through compiler evidence. KPerfIR integrates profiling into a compiler workflow through IR-level abstractions Guan et al. (2025); we share the compiler-centric intuition but use that evidence to steer an agentic rewrite loop.

**LLM-driven kernel synthesis and source-level optimization.** Source-level reward-loop systems such as SparseRL Wang et al. (2026), Kevin Baronio et al. (2025), CUDA-L1 Li et al. (2025c), and KernelEvolve Liao et al. (2025) show that runtime reward can substantially improve correctness and throughput for CUDA kernels at scale, but their evidence remains dominated by execution outcomes rather than explicit compiler-level attribution. Benchmarks such as KernelBench, TritonBench, and MultiKernelBench provide the evaluation substrate for this line of work Ouyang et al. (2025); Li et al. (2025b); Wen et al. (2025).

**Profiling-guided and agentic optimization.** AVO elevates a coding agent from candidate generator to autonomous variation operator for GPU attention kernels, with long-horizon edit, test, and revision loops Chen et al. (2026), while TritonForge, PRAGMA, Astra, and KForge use profiling, execution traces, and multi-agent coordination to steer iterative optimization Li et al. (2025a); Lei et al. (2025); Wei et al. (2025); Sereda et al. (2025). These systems localize bottlenecks better than reward-only search, but profiling alone does not explain which lowering decision, legality condition, or pass transition produced the hotspot. Our work is stricter in scope: the agent must move from profiling to IR attribution and compiler-grounded explanation before rewriting source.

**NPU-specific kernel generation.** Recent NPU-focused systems such as AscendKernelGen and AscendCraft validate LLM-based kernel generation on Ascend, where vendor-specific DSLs and backend constraints make the problem materially harder than on GPUs Cao et al. (2026); Wen et al. (2026), but remain generation-centric. In contrast, our system starts from existing Triton-Ascend kernels and escalates through progressively deeper evidence before proposing rewrites, aiming to explain and repair a kernel whose bottleneck is already visible at runtime.

## 8 Discussion

the system provides evidence that compiler-grounded, hierarchical optimization can unlock substantial gains, but the results also clarify its current boundary: the workflow reaches a geometric-mean kernel speedup of 4.35× and a median of 2.73× across 37 entries, with large wins coexisting with near-baseline entries. The optimization accounting shows why the workflow should not be summarized by its final speedup table alone: the median best-round index is 10, 12 selected paths first peak in the final three rounds, and 32/3/4 paths reach profiling diagnosis, IR attribution, and compiler-source escalation. These counts show practical use of progressive evidence without claiming that any one level causes the final outcome.

What the evaluation does not yet provide is causal isolation: it does not compare progressive escalation against profiling-only or always-deep alternatives under one frozen toolchain, does not separate the retrospective synthesis loop from the seeded pattern guidance, and does not report token cost or wall-clock optimization time. Restricting rewrites to structured source transformations improves auditability but may exclude larger refactors or multi-kernel coordination, and transfer to other backends depends on how much compiler-grounded evidence their profiler, IR stack, and source references can expose. The next step is a stronger systems evaluation with frozen-toolchain baselines, explicit cost accounting, and direct ablations of the hierarchy and the retrospective synthesis loop. In conclusion, treating LLM-based kernel optimization as progressive cross-layer diagnosis—escalating from pattern triage and profiling to IR attribution and compiler-source escalation before proposing evidence-backed rewrites—provides a practical path to backend-aware kernel optimization, and the limitations above define the agenda for evaluating the hierarchy more rigorously.

### Ethics statement

This work studies automated optimization of compute kernels and does not involve human subjects, personal data, or the release of sensitive datasets. The benchmark operators used in the evaluation are derived from publicly available benchmark suites, which are cited in the paper. We do not foresee direct harmful applications of the methods described here beyond the general dual-use considerations of performance-engineering automation.

### Reproducibility statement

We describe the optimization workflow, its hierarchical evidence levels, and the round-state contract in Sections 3–5, and the evaluation protocol—including benchmark derivation, case sampling, validation gates, and accounting rules—in Section 6. All headline quantities are reported together with their full outcome distributions so that the aggregate statistics can be interpreted against per-operator results.

### AI use statement

In this work, we used generative AI tools as the optimization agents studied in the paper: the kernel rewrites, diagnostic records, and analysis notes reported in the evaluation were produced by LLM-based code agents operating inside the described workflow, and were validated through the correctness and benchmark gates described in the paper. Generative AI tools were also used to assist with editing and formatting of the manuscript. We have reviewed all AI-assisted work: experimental claims are tied to validated benchmark records, and LLM-produced kernel candidates were accepted only after passing automated correctness and performance validation. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

## References

Ascend. Triton Ascend Ops. GitHub repository, 2026. https://github.com/Ascend/triton-ascend-ops.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels, 2025. URL https://arxiv.org/abs/2507.11948. arXiv preprint arXiv:2507.11948.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, Dongyang Tao, Xiansong Huang, Fan Xu, Feidiao Yang, Yao Lu, Chang-Dong Wang, Yutong Lu, Weicheng Xue, Bin Zhou, and Yonghong Tian. AscendKernelGen: A systematic study of LLM-based kernel generation for neural processing units, 2026. URL https://arxiv.org/abs/2601.07160. arXiv preprint arXiv:2601.07160.

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. AVO: Agentic variation operators for autonomous evolutionary search, 2026. URL https://arxiv.org/abs/2603.24517. arXiv preprint arXiv:2603.24517.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026.

flagos-ai. Awesome LLM-driven kernel generation. GitHub repository, 2026. https://github.com/flagos-ai/awesome-LLM-driven-kernel-generation.

Yue Guan, Yuanwei Fang, Keren Zhou, Corbin Robeck, Manman Ren, Zhongkai Yu, Yufei Ding, and Adnan Aziz. KPerfIR: Towards an open and compiler-centric ecosystem for GPU kernel performance tooling on modern AI workloads. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025.

Just-it. NPUKernelBench. GitHub repository, 2026. https://github.com/Just-it/AscendOpGenAgent/tree/br_430/benchmarks/NPUKernelBench.

Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. PRAGMA: A profiling-reasoned multi-agent framework for automatic kernel optimization, 2025. URL https://arxiv.org/abs/2511.06345. arXiv preprint arXiv:2511.06345.

Haonan Li, Keyu Man, Partha Kanuparthy, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. TritonForge: Profiling-guided framework for automated Triton kernel optimization, 2025a. URL https://arxiv.org/abs/2512.09196. arXiv preprint arXiv:2512.09196.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025b. doi: 10.18653/v1/2025.findings-acl.1183.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning, 2025c. URL https://arxiv.org/abs/2507.14111. arXiv preprint arXiv:2507.14111; accepted by ICLR 2026.

Gang Liao, Hongsen Qin, Ying Wang, et al. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at Meta, 2025. URL https://arxiv.org/abs/2512.23236. arXiv preprint arXiv:2512.23236.

OpenAI. Harness engineering. OpenAI blog, 2025. https://openai.com/index/harness-engineering/.

OpenAI. Codex. Official product page, 2026. https://openai.com/codex/.

OpenCode. OpenCode: The open source AI coding agent. GitHub repository, 2026. https://github.com/anomalyco/opencode.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels?, 2025. URL https://arxiv.org/abs/2502.10517. arXiv preprint arXiv:2502.10517.

Taras Sereda, Tom St. John, Burak Bartan, Natalie Serrino, Sachin Katti, and Zain Asgar. KForge: Program synthesis for diverse AI hardware accelerators, 2025. URL https://arxiv.org/abs/2511.13274. arXiv preprint arXiv:2511.13274.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 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*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

Triton contributors. triton.autotune. Triton documentation, 2026a. https://triton-lang.org/main/python-api/generated/triton.autotune.html.

Triton contributors. triton.language. Triton documentation, 2026b. https://triton-lang.org/main/python-api/triton.language.html.

Triton contributors. Matrix multiplication. Triton documentation, 2026c. https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html.

triton-lang contributors. Triton-Ascend. GitHub repository, 2026. https://github.com/triton-lang/triton-ascend.

Yaoyu Wang, Hankun Dai, Zhidong Yang, Junmin Xiao, and Guangming Tan. Mastering sparse CUDA generation through pretrained models and deep reinforcement learning. OpenReview, 2026. URL https://openreview.net/forum?id=VdLEaGPYWT. ICLR 2026 Oral.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for GPU kernel performance optimization, 2025. URL https://arxiv.org/abs/2509.07506. arXiv preprint arXiv:2509.07506.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A multi-platform benchmark for kernel generation, 2025. URL https://arxiv.org/abs/2507.17773. arXiv preprint arXiv:2507.17773.

Zhongzhen Wen, Shudi Shao, Zhong Li, Yu Ge, Tongtong Xu, Yuanyi Lin, and Tian Zhang. AscendCraft: Automatic Ascend NPU kernel generation via DSL-guided transcompilation, 2026. URL https://arxiv.org/abs/2601.22760. arXiv preprint arXiv:2601.22760.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of LLMs, 2026. URL https://arxiv.org/abs/2601.15727. arXiv preprint arXiv:2601.15727.

## A Source-Level Rewrite for the Hyena Operator

This appendix shows the full source-level rewrite referenced by the representative cross-layer trace in the evaluation. The Hyena FFT-size padding operator maps a (rows $\times$ seqlen) input to a padded (rows $\times$ fft_size) output while scaling the live values. The initial implementation decodes a flat one-dimensional offset into row and column indices with per-element division and remainder operations, and applies a `tl.where` conditional to zero out the padding region. The retained rewrite replaces this with two-dimensional row–column tiling, which eliminates the per-element division and remainder chain entirely; and because masked loads already supply zeros outside the live extent, the conditional is redundant and is removed rather than re-expressed. Both changes were validated by the per-round correctness and benchmark gates described in the evaluation, and the retained kernel reaches 2.08$\times$ geometric-mean kernel speedup on the per-round optimization benchmark.

**Listing 1:** Source-level rewrite for the Hyena FFT-size padding operator (mask arguments abbreviated as `lm`/`sm`).

```
 def pad_scale(...):
-    offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
-    row, col = offs // fft_size, offs % fft_size
-    x = tl.load(x_ptr + row * seqlen + col, mask=lm, other=0.0)
-    out = tl.where(col < seqlen, x * scale, 0.0)
-    tl.store(out_ptr + offs, out, mask=sm)
+    rows = pid * BLOCK_ROWS + tl.arange(0, BLOCK_ROWS)
+    for col0 in tl.range(0, fft_size, BLOCK_COLS):
+        cols = col0 + tl.arange(0, BLOCK_COLS)
+        src = (rows * seqlen)[:, None] + cols[None, :]
+        dst = (rows * fft_size)[:, None] + cols[None, :]
+        x = tl.load(x_ptr + src, mask=lm, other=0.0)
+        tl.store(out_ptr + dst, x * scale, mask=sm)
```