EvokeAgent: A Hardware-Aware, Knowledge-Base-Co-Driven Self-Evolving Agent System for GPU Kernel Optimization

Anonymous authors
Paper under double-blind review

Abstract

Automated GPU operator optimization is critical for accelerating deep learning workloads, prompting the growing use of Large Language Models (LLMs) to generate and tune high-performance kernels. While existing systems increasingly incorporate runtime profiling feedback, the selection of hardware evidence is often not explicitly coupled to optimization actions; relying on the model alone to analyze all such hardware data makes it difficult to pinpoint the right optimization opportunities amid noisy metrics. We present EvokeAgent, an NCU-native GPU operator optimization agent built around a self-evolving, bottleneck-indexed knowledge base (KB). EvokeAgent establishes a bottleneck-guided routing layer that contracts voluminous NCU metrics into hypothesis-relevant diagnostic evidence downward while mapping the diagnosis to targeted KB policy retrieval upward. Evaluated on all three KernelBench levels using an untuned LLM, EvokeAgent achieves 100% correctness with geometric-mean speedups of 1.93×, 2.65×, and 2.94× on CUDA, and 1.68×, 2.56×, and 1.64× on Triton across L1–L3, respectively. Furthermore, ablation shows that this routing-and-retrieval design outperforms the non-KB baseline by 1.08×, 1.48×, and 1.55× on L1–L3 under capped optimization rounds.

1 Introduction

GPU operator optimization is a key performance challenge in modern AI systems—matrix multiplications, attention mechanisms, convolutions, and their fused combinations are recurring performance hotspots in deep-learning workloads Ouyang et al. (2025); Jaber & Jaber (2026). While cuBLAS and torch.compile provide strong baselines for common operators, the space of operator-fusion patterns grows rapidly as model architectures vary, motivating LLM-based automatic optimization Yu et al. (2026). A typical LLM-driven loop generates a kernel, benchmarks it, profiles it with NCU, analyzes the counters, modifies the code, and measures again Ouyang et al. (2025); Jaber & Jaber (2026); Lei et al. (2025); Dong et al. (2026a).

Existing systems already incorporate runtime measurements and, in some cases, NCU profiling into iterative optimization Lei et al. (2025); Zhang et al. (2025). However, they remain limited in how profiling evidence is selected, how bottlenecks are diagnosed, and how optimization experience is retrieved across tasks. Traditional profiling tools often expose a large and redundant set of hardware counters, creating information overload; asking the model to analyze this unfiltered stream on its own makes it difficult to identify decisive evidence that discriminates among competing bottleneck explanations, leaving optimization vulnerable to blind trial and error. Moreover, evidence collection, bottleneck localization, and historical policy retrieval are rarely coupled, which limits precise reuse of cross-task experience. EvokeAgent addresses this gap by using the diagnosed bottleneck as a routing variable for both evidence selection and policy retrieval. In our full-loop comparison, the combined With-KB and bottleneck-guided routing configuration outperforms its control (Section 5.4.1), supporting the joint routing-and-retrieval design.

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.

EvokeAgent makes the profiler an active diagnostic interface threaded through the optimization loop, built around a self-evolving, bottleneck-indexed knowledge base. Its key design lets the bottleneck type serve as an evidence–action intermediate semantic layer: coarse-grained counters first form a bottleneck hypothesis, and the resulting type is a routing decision—downward, it contracts the high-dimensional NCU output into the evidence needed to answer the current diagnostic question; upward, it translates the diagnosis into a policy retrieval key. The bottleneck type provides a potentially transferable routing signal: similar measured symptoms may suggest related intervention directions across operators, although the effectiveness of a transferred policy can still depend on shape, layout, dtype, and fusion context. The contribution is therefore a policy for using NCU, not another metric set: the model receives an interpretable action prior consistent with the current hardware state, rather than a fixed answer, and can still generate new transformations whose validated experience is fed back into the KB. The EvokeAgent KB starts from hand-written seeds and incrementally records LLM-selected policies from successful trials in dialect-specific bottleneck hierarchies; results should therefore be interpreted under this continual-evaluation protocol. Because the bottleneck-driven loop is independent of the emitted kernel dialect, EvokeAgent is designed for multi-language generation: CUDA kernels are the primary evaluation target, while Triton kernels provide a secondary backend evaluation, each backed by its own dialect-specific KB.

Our contributions are as follows:

1. An evidence–action intermediate semantic layer: the bottleneck diagnosed from coarse-grained NCU counters jointly selects fine-grained evidence and the KB policies shown to the LLM. This turns profiling from a post-hoc report into an interpretable routing interface that can be revised when new evidence conflicts with the current hypothesis.
2. The EvokeAgent optimization framework: we build a two-stage optimization loop consisting of parallel multi-direction initialization and iterative refinement. In each iteration, bottleneck-guided NCU evidence routing, policy retrieval, LLM-selected transformations, and knowledge updates connect hardware diagnosis to action selection and validation.
3. A self-evolving, bottleneck-indexed knowledge base: the KB starts from hand-written seeds and incrementally records LLM-selected policies only after successful trials, organizing them by backend and diagnosed bottleneck. Each record retains its description, parameter range, provenance, and speedup statistics, so later tasks receive reusable action context while the LLM remains free to adapt existing policies or design new ones.
4. Experimental validation: with an untuned LLM on all three KernelBench levels, the CUDA target achieves 100% correctness under the stated gate and 1.93×/2.65×/2.94× geometric-mean speedups (L1/L2/L3), while Triton provides a second-backend evaluation. The primary ablation compares the combined routing-and-retrieval design with a control that removes bottleneck-indexed policy access and the proposed routing mechanism, yielding ratios of 1.08/1.48/1.55.

2 Related Work

LLM-driven GPU kernel generation has grown into a rich research area, surveyed comprehensively by Yu et al. Yu et al. (2026). We organize existing work by methodological lineage; Table 9 (Appendix B) situates all systems in a single design space. Training-driven methods encode optimization knowledge into model weights via SFT or RL: KernelLLM Fisches et al. (2025) fine-tunes on approximately 25K PyTorch–Triton pairs; ConCuR Kong et al. (2025) constructs CUDA-kernel data with reasoning traces and uses it to train KernelCoder; InCoder-32B Yang et al. (2026) is a multi-stage industrial code foundation model; Kevin Baronio et al. (2026) introduces multi-turn RL with reward attribution; CUDA Agent Dai et al. (2026) scales agentic RL; AutoTriton Li et al. (2025b) applies SFT+GRPO; TritonRL Woo et al. (2025) aims to mitigate reward hacking; Dr. Kernel Liu et al. (2026) addresses multi-round RL estimation bias; QiMeng-Kernel Zhu et al. (2026) separates macro-level reasoning

from micro-level coding; and Kernel-Smith Du et al. (2026) combines evolutionary search with evolution-oriented post-training. These methods introduce additional training and data-construction costs. Encoding experience in model weights also does not by itself provide an explicit, auditable mechanism for cross-run knowledge updates without additional memory or adaptation components. Agent-driven methods use the LLM as the reasoning engine inside the loop and can be organized by capability. Iterative refinement: Caesar (built into KernelBench) performs simple loop refinement; AutoKernel Jaber & Jaber (2026) uses a keep/revert-style search; K-Search Cao et al. (2026) co-evolves a world model; MaxCode Ou et al. (2026) applies maximum-reward optimization; and GPU Kernel Scientist Andrews & Witteveen (2025) uses end-to-end runtime feedback. External memory: KernelSkill Sun et al. (2026) uses long-term skill memory and short-term trajectory memory for cross-task reuse and multi-round optimization; KernelBlaster Dong et al. (2026b) maintains persistent cross-task optimization experience organized by performance state and uses profiler-derived states for retrieval; EvoKernel Zheng et al. (2026) employs value-driven memory for NPU kernels; AccelOpt Zhang et al. (2026) maintains external optimization memory for AI accelerators; daVinci-kernel Fu et al. (2026) co-evolves a skill library with RL; and ReGraphT Gong et al. (2026) represents knowledge as a reasoning graph. Hardware profiling integration: PRAGMA Lei et al. (2025) integrates NCU profiling into multi-agent reasoning; CudaForge Zhang et al. (2025) uses hardware feedback with a metric subset selected offline; TritonForge Li et al. (2025a) adds profiling feedback for Triton; and KernelBand Ran et al. (2026) uses hardware-aware bandit steering. Our design studies a specific coupling between profiler-guided evidence selection and bottleneck-indexed policy presentation. Multi-agent orchestration: STARK Dong et al. (2026a), Astra Wei et al. (2025), AKG Du et al. (2025), and GEAK Wang et al. (2025) use multiple roles or delegated search. These systems occupy overlapping design points, so Table 9 reports only properties directly supported by the cited sources.

Positioning. EvokeAgent couples profiling routing and knowledge retrieval on the same causal axis: the same NCU-diagnosed bottleneck type both determines which fine-grained metrics are shown to the model and which policies are retrieved from the KB, while the LLM curates the KB, which grows in a data-driven manner with each run.

3 Method

3.1 System Overview

EvokeAgent's architecture (Figure 1) has two optimization stages—parallel initialization and iterative refinement—implemented through three functional components: bottleneck diagnosis, policy retrieval, and knowledge evolution. Three candidates are generated in parallel from the same human-authored initialization skills under memory-bandwidth, compute-throughput, and latency/occupancy hypotheses; the fastest correct candidate initializes one optimization trajectory. Each subsequent iteration invokes the NCU diagnostic procedure, retrieves policies using the resulting label, applies an LLM-selected transformation, and checks correctness and performance. The following subsections separate these components: NCU diagnosis and evidence selection, KB construction, and KB retrieval with evolution.

3.2 NCU Use: Bottleneck Diagnosis and Evidence Routing

NCU exposes many fine-grained hardware metrics; dumping all of them into the prompt adds noise and wastes tokens on nondiagnostic signals. EvokeAgent therefore performs two-level diagnostic filtering. Level 1 uses a small set of coarse metrics—memory-bandwidth utilization, compute utilization, and occupancy—through ordered deterministic rules to select one of three NCU parent categories. Level 2 uses fine-grained metrics such as warp-stall distributions, cache hit rates, register counts, and instruction mix to select a backend-specific diagnostic sub-category: 12 categories for CUDA and 16 rule-defined categories for Triton. Only metrics relevant to the selected sub-category enter the prompt. For example, a memory-bound kernel whose warp stalls are dominated by long_scoreboard is sub-classified as memory latency, whereas one with both L1 and L2 hit rates low is sub-classified as

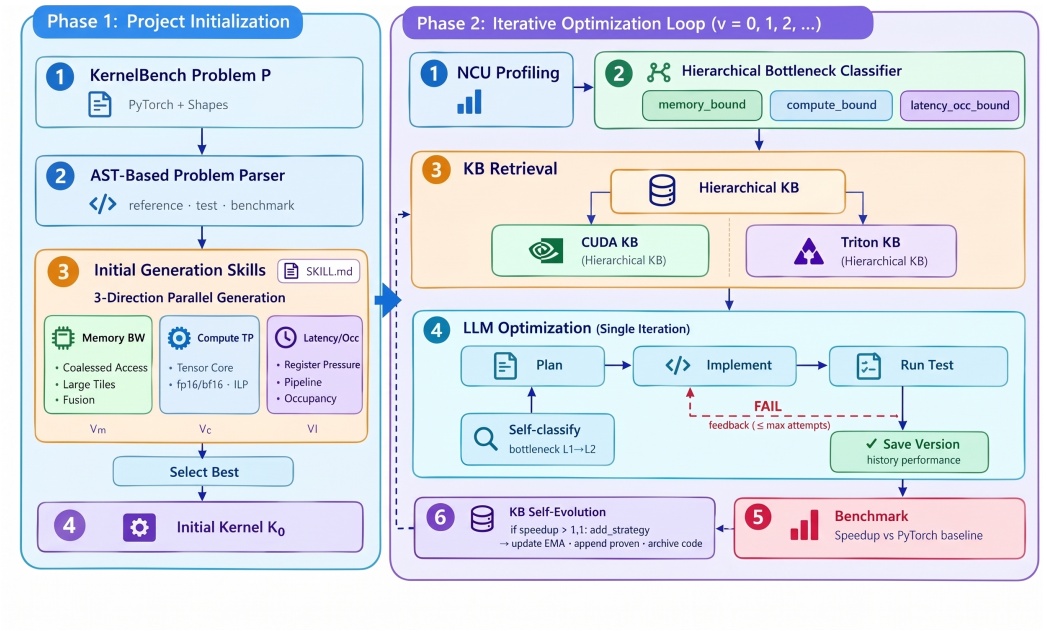


Figure 1: EvokeAgent overview. NCU profiling is an active diagnostic interface threaded through the optimization loop rather than an after-the-fact report: coarse-grained counters first form a bottleneck hypothesis, and the resulting type acts as an evidence–action intermediate semantic layer—downward, it selects the fine-grained metrics needed by the current diagnostic question; upward, it becomes the retrieval key for the bottleneck-indexed knowledge base (KB) that grows through LLM-curated absorption.

bandwidth saturation. The resulting diagnostic label and evidence subset are passed to the retrieval stage, where the label is combined with the backend target to form the lookup key.

3.3 KB Construction: Backend-Specific Hierarchy and Taxonomy

EvokeAgent maintains two parallel knowledge bases, one for each generation target: the Triton target uses the Triton KB and the CUDA target the CUDA KB. The two KBs share the same three-parent hierarchy—memory, compute, and latency/occupancy—but their policy code and provenance remain dialect-specific. The CUDA KB contains 12 diagnostic sub-categories, while the Triton KB contains 16 rule-defined sub-categories; each KB is initialized with hand-written seed policies. A policy record stores the technique, description, concrete parameter ranges, validation provenance, and EMA speedup. These design choices separate shared diagnostic semantics from backend-specific executable code and provenance.

3.4 KB Retrieval and Evolution: Policy Access and LLM-Curated Updates

At each iteration, the backend target and the diagnostic label produced in Section 3.2 form the lookup key for the corresponding KB sub-category. The retriever presents the policies stored under that key to the LLM together with their descriptions, parameter ranges, provenance, and EMA speedups. Retrieval provides action context rather than a fixed action: the LLM may reuse a policy, adapt it, or propose a new transformation. Hardware symptoms can serve as a transferable routing signal across related operators, but policy effectiveness may still depend on shape, layout, dtype, and fusion context.

LLM-curated absorption instead of automatic absorption. A successful code change is recorded only when the LLM explicitly invokes add_strategy.py after a correctness-passing optimization, specifies the bottleneck type and sub-category, and reports a positive relative gain $g > 1.1\times$. Here $g = \overline{t_{\text{PyTorch eager}}}/\overline{t_{\text{generated}}}$. Compilation or runtime failures have no valid speedup and therefore do not enter the KB or update its EMA; the implementation

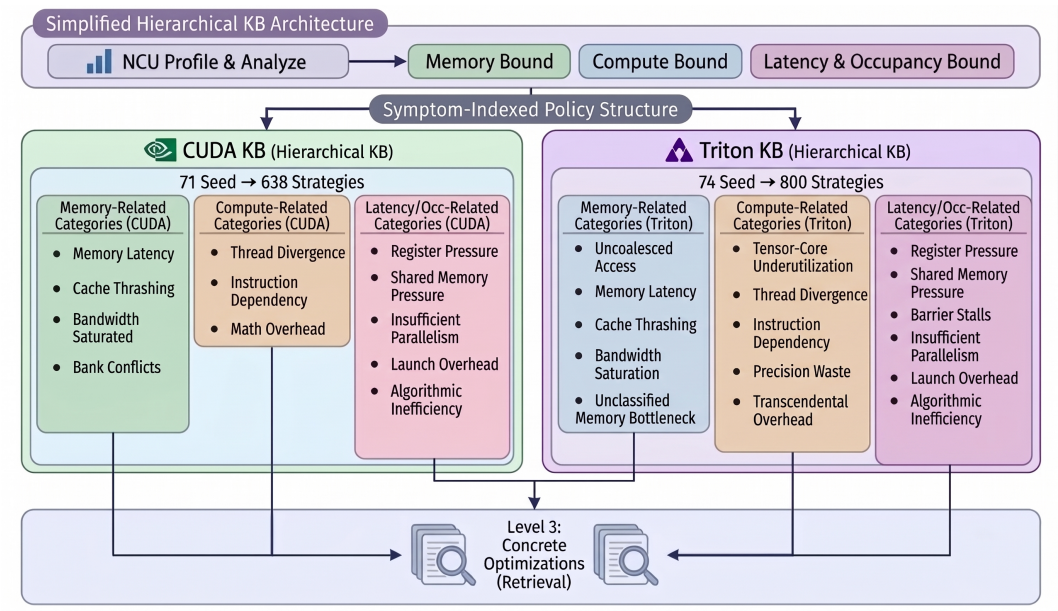


Figure 2: Simplified hierarchical knowledge-base architecture. The CUDA and Triton KBs maintain separate backend-specific policy structures under three parent bottlenecks, with 12 CUDA sub-categories and 16 Triton rule-defined sub-categories; each record stores an executable policy together with its provenance and measured outcomes.

has no explicit penalty or eviction mechanism for stored policies. This makes the KB a success-only memory whose negative evidence remains in per-version histories rather than in cross-task retrieval.

4 Experimental Setup

4.1 Benchmark, Metrics, Hardware, and Model

We evaluate on KernelBench Ouyang et al. (2025): 250 PyTorch modules across three levels—L1 (100 single operators), L2 (100 fused operators), and L3 (50 model architectures)—reporting correctness, fast_p (fraction of correct problems with speedup $\geq p\times$), and geometric-mean speedup relative to the PyTorch eager baseline. All experiments run on a single NVIDIA H20-3e GPU (compute capability 9.0, 78 SMs, 96GB HBM3); all generation components use the untuned GLM-5.2, isolating the contribution of the combined profiler-routing and KB design from additional model tuning.

Aggregation metrics. For each problem, we first average repeated measurements of the reference and generated kernels separately, and define the problem-level speedup as

$$s_i = \frac{\overline{t_i^{\text{eager}}}}{\overline{t_i^{\text{generated}}}}.$$

Our primary metric is the geometric-mean speedup $\text{GM} = (\prod_i s_i)^{1/N}$. To align with some public implementations, Table 1 also reports the arithmetic mean $\text{AM} = N^{-1}\sum_i s_i$. Because the arithmetic mean is sensitive to extreme speedups, it is not used alone to support superiority claims.

4.2 Generation Targets

EvokeAgent's bottleneck-driven loop and bottleneck-indexed retrieval are independent of the emitted kernel dialect, so the same loop drives both a CUDA-kernel target (primary) and

a Triton-kernel target (secondary backend evaluation). The CUDA target generates native CUDA kernels, retrieves policies from the CUDA KB, and is measured against the PyTorch CUDA baseline. The Triton target generates Triton kernels with retrieval from the Triton KB; its speedup is defined according to the KernelBench v0 PyTorch-eager convention, with PyTorch eager runtime in the numerator and Triton-kernel runtime in the denominator under the same input shape. Both targets use the full 250-problem suite. For each problem, three candidates are generated in parallel under memory-bandwidth, compute-throughput, and latency/occupancy hypotheses, followed by at most 10 optimization rounds. The best candidate initializes one agent optimization trajectory; each timing measurement nevertheless uses 10 warm-up runs and 100 measured iterations. The CUDA KB starts with 71 seed policies, adapted from KernelBlaster-inspired strategies and project-specific optimization experience; the Triton KB starts with 74 seed policies from the project's routine optimization traces. After the evaluation runs, the KBs contain 638 and 800 policies, respectively.

4.3 Ablation Design

Our primary ablation runs the same capped loop on L1/L2/L3 in two groups: With-KB uses bottleneck-indexed policy retrieval and the proposed routing mechanism, whereas No-KB removes that policy access and routing mechanism. Both groups use the same LLM, hardware, initial candidate winner, correctness checks, and optimization-round budget. The comparison therefore evaluates the effectiveness of the combined routing-and-retrieval design under a matched optimization protocol; it is not intended to identify separate causal effects of evidence selection and retrieval.

5 Experimental Results

5.1 Main Results on KernelBench

Table 1: Main KernelBench results on a single NVIDIA H20 GPU with an untuned LLM. The CUDA kernel is the primary target; the Triton kernel is the multi-language generalization validation; both run the same bottleneck-driven loop, each with its own KB. fast_p: fraction of correct problems with speedup $\geq p\times$; correctness is 100% in all cells; "Degraded": fraction with speedup $< 1.0\times$.

Target	Metric	L1 (100 prob.)	L2 (100 prob.)	L3 (50 prob.)
CUDA kernel (primary target)	Geo. mean speedup	1.93×	2.65×	2.94×
	Arith. mean speedup	3.22×	41.07×	5.00×
	Maximum speedup	37.07×	1093.11×	42.10×
	$\text{fast}_{1.0}$	78.0%	82.0%	92.0%
	$\text{fast}_{2.0}$	40%	43%	62%
	Degraded	22.0%	18.0%	8.0%
Triton kernel (multi-language generalization)	Geo. mean speedup	1.68×	2.56×	1.64×
	Arith. mean speedup	2.65×	10.25×	2.21×
	Maximum speedup	35.40×	245.97×	13.15×
	$\text{fast}_{1.0}$	79.0%	82.0%	76.0%
	$\text{fast}_{2.0}$	38%	42%	42%
	Degraded	21.0%	18.0%	24.0%

Table 1 shows that the two backends are comparable on L1/L2, whereas CUDA is stronger on L3 (2.94× versus 1.64× geometric mean and 92% versus 76% $\text{fast}_{1.0}$). This supports portability of the diagnosis–retrieval loop, but not backend-independent superiority: code-generation expressiveness and library interactions remain relevant. The L2 arithmetic mean is outlier-sensitive (41.07× versus 2.65× geometric mean), so we treat the geometric mean and $\text{Fast}_{1.0}$ as the primary summaries.

Comparison with a representative existing method. KernelBlaster Dong et al. (2026b) is a closely related agent-driven design that maintains persistent cross-task optimization experience organized by performance state. EvokeAgent differs in the particular coupling

it studies between two-level NCU metric filtering and bottleneck-indexed policy presentation. Table 2 juxtaposes reported numbers only as contextual reference points; the setups differ in hardware, model, baseline, timing method, task coverage, and correctness criteria. The mechanistic claims of this paper therefore rest on the same-system comparison in Section 5.4.1, not on cross-paper ranking.

Table 2: Reported KernelBench results of EvokeAgent and KernelBlaster. Setups differ; the numbers are contextual reference points, not a controlled comparison. KernelBlaster's $\text{fast}_{1.0}$ is the range across its GPUs; its L3 covers a per-model subset.

System	L1	L2	L3	$\text{fast}_{1.0}$ (L1 / L2)	Correct
KernelBlaster Dong et al. (2026b)	1.43×	2.50×	1.50×	59.7–70.4% / 72.6–84.9%	67–95%
EvokeAgent (CUDA, primary)	1.93×	2.65×	2.94×	78.0% / 82.0%	100%
EvokeAgent (Triton, generalization)	1.68×	2.56×	1.64×	79.0% / 82.0%	100%

Relative to KernelBlaster, EvokeAgent CUDA reports higher geometric means on L1/L2/L3 (1.93/2.65/2.94× vs. 1.43/2.50/1.50×), higher L1/L2 $\text{Fast}_{1.0}$ (78%/82% vs. 59.7–70.4%/72.6–84.9%), and 100% correctness vs. 67–95%. The especially large L3 gap is consistent with bottleneck-indexed experience helping complex workloads, but remains contextual; the matched ablation is the mechanistic evidence.

5.2 Contextual Comparison with Publicly Reported Results

To address comparability with recent work, we explicitly treat cross-paper numbers as a contextual comparison, not as a controlled state-of-the-art comparison. Because recent papers often report "average speedup" without specifying whether the task-level aggregation is arithmetic or geometric, we use $\text{fast}_{1.0}$ as the primary cross-paper metric: the fraction of tasks whose generated kernel is at least as fast as the Torch baseline. KernelSkill Sun et al. (2026) defines this metric explicitly in its Table 3. We retain published average-speedup values as secondary context, but do not assume a common aggregation rule: for KernelSkill's own public results, the released KernelMem implementation computes a per-problem ratio of mean reference latency to mean generated-kernel latency and then takes the arithmetic mean over problems, with failed tasks represented by a zero score; the paper itself does not label this value as arithmetic or geometric. The STARK, QiMeng, and other external values are reproduced from the KernelSkill table and its cited sources, so their average-speedup aggregation semantics are not treated as standardized. The two evaluations also differ in GPU hardware, LLM, task execution, optimization budget, and measurement details; our controlled claims remain restricted to the internal comparison under matched hardware, model, and budget.

Table 3: Contextual comparison with all methods listed in the KernelSkill paper table. $\text{Fast}_{1.0}$ is the primary comparison metric and denotes the fraction of tasks at least as fast as Torch Eager. "Published avg." reproduces the reported average-speedup values as secondary context; aggregation conventions are not assumed to be shared across papers. External results were not obtained under matched conditions.

System	L1		L2		L3	
	$\text{Fast}_{1.0}$	Published avg.	$\text{Fast}_{1.0}$	Published avg.	$\text{Fast}_{1.0}$	Published avg.
Kevin-32B Baronio et al. (2026)	16%	1.18×	61%	1.74×	2%	0.32×
Astra Wei et al. (2025)	43%	1.48×	73%	0.99×	35%	0.90×
PRAGMA Lei et al. (2025)	45%	1.49×	74%	1.02×	36%	0.92×
CudaForge Zhang et al. (2025)	54%	1.45×	89%	2.10×	68%	1.28×
QiMeng Zhu et al. (2026)	59%	2.20×	66%	1.22×	40%	0.73×
STARK Dong et al. (2026a)	71%	3.03×	100%	2.69×	87%	1.58×
KernelSkill Sun et al. (2026)	62%	5.44×	100%	2.82×	82%	1.92×
EvokeAgent (CUDA, ours)	78%	3.22×	82%	41.07×	92%	5.00×

The comparison is not uniformly favorable across levels: EvokeAgent reaches 78%/82%/92% $\text{Fast}_{1.0}$ on L1/L2/L3, while STARK and KernelSkill report 100% on L2, and CudaForge reports 89%. This does not by itself show that EvokeAgent is less effective. $\text{Fast}_{1.0}$ is a threshold statistic and is sensitive to task subsets, baseline headroom, correctness filtering, and optimization budgets; our 2.65× L2 geometric mean nevertheless indicates substantial typical gains. The lower L2 coverage may reflect a harder boundary case: a few fused operators can gain substantially without crossing the 1× threshold. Since these evaluations are not matched, the difference is a limitation of the comparison rather than evidence for a method-level ranking.

5.3 Where Do the Gains Come From?

Gains vary across operator and model families. We decompose CUDA results by category; the complete breakdown is in Appendix A. On L1, Pooling and Reduction/Normalization obtain the largest geometric-mean speedups (3.41× and 2.82×), whereas Matmul variants obtain 1.31×. On L2, ConvTranspose3d and Gemm fusions reach 4.38× and 3.47×, versus 1.70× for ConvTranspose2d fusions. On L3, sequence-model families obtain 4.13–4.43×, compared with 2.36× for CNNs.

The pattern is consistent with baseline optimization headroom. Pooling, reduction, and fused three-dimensional or dense operators expose memory-access inefficiencies, intermediate materialization, or launch overhead, whereas optimized matrix-multiplication and CNN primitives leave less room. This is an observed category pattern, not a causal category effect: categories differ in size and composition, and each problem has one optimization trajectory. Detailed tables and assignments appear in Appendix A.

5.4 Ablation Study

5.4.1 KB Policy Retrieval: The Full Multi-Round Optimization Loop

We run the design of Section 4.3 on CUDA. Both groups share the winner of With-KB's three-direction initialization; subsequent differences therefore reflect the combined change in bottleneck-guided evidence selection and policy access. Table 4 reports endpoint outcomes under the capped protocol. With-KB improves geometric mean from 1.79× to 1.93×, 2.65×, and 2.94× on L1/L2/L3, respectively, while $\text{fast}_{1.0}$ rises from 67/69/38 to 78/82/46 problems. Thus, the combined design improves both typical aggregate performance and the number of problems crossing the Torch baseline, with larger gains on the more structured L2/L3 workloads. This is consistent with heterogeneous bottlenecks making targeted evidence and reusable policies more informative, but it is an association rather than a causal claim about task complexity because the levels contain different problem sets and sample sizes. Since the control changes both evidence selection and policy access, the result supports the joint design rather than KB retrieval alone.

Table 4: With-KB vs. No-KB control on the full multi-round optimization loop (CUDA target, CUDA KB; same untuned LLM, up to 10 iterations; 100% correctness in all cells).

	With-KB			No-KB control		
Level	L1	L2	L3	L1	L2	L3
Geo. mean speedup	1.93×	2.65×	2.94×	1.79×	1.79×	1.90×
Arith. mean speedup	3.22×	41.07×	5.00×	2.76×	17.18×	3.21×
$\text{fast}_{1.0}$ (count)	78	82	46	67	69	38

5.4.2 Round-by-Round Convergence Under a Limited Budget

Table 5 reports the round-by-round values.

We track the best-so-far geometric mean at equal budgets. The first 2–3 rounds are close (ratios 1.06 and 1.10), but the ratio grows to 1.28, 1.35, and 1.41 at rounds 6, 8, and 10.

Table 5: Round-by-round convergence on L2 fused operators (CUDA target): best-so-far geometric-mean speedup of the two groups at equal iteration budget k (relative to PyTorch, all 100 problems; problems that terminate early are censored at their best value at termination). With-KB leads from round 4 onward in the reported capped protocol, and the displayed ratio increases through round 10.

Budget k	With-KB	No-KB control	Ratio
2	1.06×	1.00×	1.06
3	1.36×	1.24×	1.10
4	1.51×	1.33×	1.14
6	1.93×	1.51×	1.28
8	2.26×	1.67×	1.35
10	2.49×	1.76×	1.41

This is consistent with both groups exploiting generic initial transformations early, after which With-KB can use bottleneck-specific evidence and validated policies to avoid redundant interventions while No-KB performs a less structured search. The trajectory therefore supports reduced later-stage trial-and-error, not merely a better initial candidate. It is not a causal proof of either component: the ablation changes evidence selection and policy access, and each problem has one optimization trajectory. Benchmarking inadvertently saved extra versions for some operators, so the endpoint and round-wise tables use different states and need not match.

Does the KB constrain the LLM's exploration? We tracked the actual KB usage of every iteration in the main runs: about 53% of rounds read the KB policy list, 69% further consulted validated code instances (72% for the union), yet even when the policy list lies on the retrieval path a substantial fraction of rounds design a new policy—the KB suggests rather than constrains (Section 6.2).

6 Discussion

6.1 Mechanisms for Reducing Trial-and-Error Cost

The main results improve CUDA $\text{fast}_{1.0}$ from 67% to 78%, 69% to 82%, and 76% to 92% on L1/L2/L3. Together with the L2 round-wise ratio reaching 1.41 at round 10, this is consistent with bottleneck-indexed evidence and policies reducing unproductive trial-and-error. Since the ablation changes both evidence selection and KB access, it supports the combined design rather than either component alone.

6.2 Why Does the LLM Design New Policies in Nearly Half of the Rounds?

The KB provides context without constraining exploration; new-policy creation may reflect instance mismatch or taxonomy gaps and requires controlled testing. Thresholds and profiling require adaptation across GPUs and vendors; vendor primitives can limit headroom, and one trajectory per problem cannot characterize search variance.

7 Conclusion

We presented EvokeAgent, a GPU operator optimization framework centered on bottleneck-indexed, self-evolving knowledge. Its evidence–action routing layer selects NCU evidence and KB policies from the diagnosed bottleneck. Under the capped protocol, the combined design improves over its control, with geometric-mean ratios of 1.08/1.48/1.55 on L1/L2/L3; the conclusions are specific to the evaluated H20-3e hardware and single-trajectory protocol, and retrieved policies guide rather than constrain exploration.

AI Use Statement

In this work, generative AI was used as an integral component of the EvokeAgent system for initial CUDA and Triton kernel generation, iterative kernel modification, profiler-guided NCU bottleneck analysis, optimization-policy selection, and knowledge-base policy curation. Generative AI also assisted with software implementation, bug fixing, English manuscript editing, and interpretation of experimental results. Generated kernels and policy updates were checked through compilation, numerical correctness tests, and benchmark measurements under the protocol described in Appendix C. The authors reviewed the AI-assisted code, bug fixes, manuscript text, interpretations, experimental claims, figures, and references, and take responsibility for the final contents of this submission.

Reproducibility Statement

The paper reports the hardware and software environment, measurement protocol, bottleneck-classification rules, metric inventory, retrieval format, failure handling, and aggregation procedure used to obtain the reported results. The complete benchmark composition, experimental configuration, classifier rules, knowledge-base initialization, and additional implementation details are provided in the main text and appendices. Because LLM sampling, kernel compilation, and shared-GPU load can introduce nondeterminism, independent runs may produce different per-problem trajectories; the reported aggregate results should therefore be interpreted under the specified single-trajectory evaluation protocol.

References

Martin Andrews and Sam Witteveen. Gpu kernel scientist: An llm-driven framework for iterative kernel optimization, 2025.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. In International Conference on Learning Representations, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/86ba836d4c5dd859d795a172911745e2-Paper-Conference.pdf.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model, 2026.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation, 2026.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. Stark: Strategic team of agents for refining kernels. In International Conference on Learning Representations, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/229608737599da1f292d4cf5a18c99f0-Paper-Conference.pdf.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. Kernelblaster: Continual cross-task cuda optimization via memory-augmented in-context reinforcement learning, 2026b.

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, Yining Li, Jiaxing Xie, Huanan Dong, Yaguang Wu, Xiangjun Huang, Jian Yang, Hui Wang, Bowen Zhou, Bowen Li, Qipeng Guo, and Kai Chen. Kernel-smith: A unified recipe for evolutionary kernel optimization, 2026.

Jinye Du, Quan Yuan, Zuyao Zhang, Yanzhi Yi, Jiahui Hu, Wangyi Chen, Yiyang Zhu, Qishui Zheng, Wenxiang Zou, Xiangyu Chang, Zuohe Zheng, Zichun Ye, Chao Liu, Shanni Li, Renwei Zhang, Yiping Deng, Xinwei Hu, Xuefeng Jin, and Jie Zhao. Akg kernel agent: A multi-agent framework for cross-platform kernel synthesis, 2025.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm: Making kernel development more accessible, June 2025. URL https://huggingface.co/facebook/KernelLLM.

Dayuan Fu, Mohan Jiang, Tongyu Wang, Dian Yang, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. davinci-kernel: Co-evolving skill selection, summarization, and utilization via rl for gpu kernel optimization, 2026.

Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. From large to small: Transferring cuda optimization expertise via reasoning graph. In International Conference on Learning Representations, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/c4e40d310af1be55f2c0f32cb16a9a16-Paper-Conference.pdf.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search, 2026.

Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. Concur: Conciseness makes state-of-the-art kernel generation, 2025. URL https://arxiv.org/abs/2510.07356.

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.

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.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. Autotriton: Automatic triton programming with reinforcement learning in llms, 2025b.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations. In Proceedings of the 43rd International Conference on Machine Learning, volume 306 of Proceedings of Machine Learning Research. ML Research Press, 2026. URL https://icml.cc/virtual/2026/poster/65609.

Jiefu Ou, Sapana Chaudhary, Kaj Bostrom, Nathaniel Weir, Shuai Zhang, Huzefa Rangwala, and George Karypis. Maxcode: A max-reward reinforcement learning framework for automated code optimization, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In Proceedings of the 42nd International Conference on Machine Learning, volume 267 of Proceedings of Machine Learning Research, pp. 47356–47415. PMLR, 2025. doi: 10.5555/3780338.3782232. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. Kernelband: Steering llm-based kernel optimization via hardware-aware multi-armed bandits. In Proceedings of the 43rd International Conference on Machine Learning, 2026. URL https://icml.cc/virtual/2026/poster/62803.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. Kernelskill: A multi-agent framework for gpu kernel optimization, 2026. URL https://arxiv.org/abs/2603.10085.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks, 2025. URL https://arxiv.org/abs/2507.23194.

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.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. Tritonrl: Training llms to think and code triton without cheating, 2025.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Shawn Guo, Haowen Wang, Weicheng Gu, Yaxin Du, Joseph Li, Fanglin Xu, Yizhi Li, Lin Jing, Yuanbo Wang, Yuhan Gao, Ruihao Gong, Chuan Hao, Ran Tao, Aishan Liu, Tuney Zheng, Ganqu Cui, Zhoujun Li, Mingjie Tang, Chenghua Lin, Wayne Xin Zhao, Xianglong Liu, Ming Zhou, Bryan Dai, and Weifeng Lv. Incoder-32b: Code foundation model for industrial scenarios, 2026.

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. In Proceedings of the Thirty-Fifth International Joint Conference on Artificial Intelligence, pp. 8069–8077, 2026. doi: 10.24963/ijcai.2026/895. URL https://www.ijcai.org/proceedings/2026/895. Survey Track.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. Accelopt: A self-improving llm agentic system for ai accelerator kernel optimization. In Proceedings of Machine Learning and Systems, volume 8, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/0f8426558905746fc38da5e335700aec-Abstract-Conference.html.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, 2025.

Yujie Zheng, Zhuo Li, Shengtao Zhang, Jiaqian Wang, Junjie Sheng, Junchi Yan, Weinan Zhang, Ying Wen, Bo Tang, and Muning Wen. Towards cold-start drafting and continual refining: A value-driven memory approach with application to npu kernel synthesis. In Proceedings of the 43rd International Conference on Machine Learning, volume 306 of Proceedings of Machine Learning Research, 2026. URL https://icml.cc/virtual/2026/poster/63035.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 40, pp. 29168–29176. AAAI Press, 2026. doi: 10.1609/aaai.v40i34.40155. URL https://ojs.aaai.org/index.php/AAAI/article/view/40155.

A Operator Category Definitions

Tables 6, 7, and 8 group KernelBench problems by operator category / architecture family. This appendix gives the tables, the category definitions, and the operators contained in each, making the decomposition reproducible.

Table 6: L1 performance by operator category (CUDA primary target, same run as the CUDA column of Table 1). Two vector-reduction/scalar operators (problems 4, 5) belong to no category and are counted only in the overall summary.

Category	#Prob.	Geo. Mean	≥1×	Max
Pooling	6	3.41×	83%	9.86×
Reduction/Normalization	19	2.82×	100%	37.07×
Elementwise	22	2.26×	77%	24.85×
Convolution	35	1.55×	77%	19.71×
Matmul variants	16	1.31×	56%	4.00×

Table 7: L2 performance by fusion category (CUDA primary target, same run as the CUDA column of Table 1).

Category	#Prob.	Geo. Mean	≥1×	Max
ConvTranspose3d fusions	19	4.38×	79%	1093.11×
Gemm fusions	19	3.47×	89%	896.04×
Conv3d fusions	12	2.92×	75%	614.21×
Matmul/BMM fusions	18	2.36×	89%	346.76×
Conv2d fusions	21	1.73×	90%	7.88×
ConvTranspose2d fusions	11	1.70×	55%	81.51×

Table 8: L3 performance by architecture family (CUDA primary target, same run as the CUDA column of Table 1). The 50 full-model problems grouped by architecture family.

Family	#Prob.	Geo. Mean	≥1×	Max
Other sequence models (NetVLAD/UNet/Mamba)	5	4.43×	100%	11.45×
RNN/LSTM/GRU	10	4.13×	100%	42.10×
Transformer/Attention	8	3.23×	88%	8.03×
CNN (classic and modern)	24	2.36×	88%	35.70×
MLP variants	3	2.12×	100%	4.32×

L1 categories (single operators). Each L1 problem is a single PyTorch operator. Five categories partition all 100 L1 problems by the operator's computational role:

- Convolution (35): all standard, transposed, depthwise, pointwise, dilated, strided, padded, and grouped convolution variants in 1D/2D/3D (problems 50, 54–87).
- Matmul variants (16): matrix and batched/tensor matrix multiplications and their shape variants (square, standard, batched, large/small K, irregular, tall-thin, diagonal, symmetric, upper/lower triangular, transposed A/B/double-transposed, 3D/4D tensor–matrix) (problems 1–3, 6–16, 18).
- Reduction/Normalization (19): all normalization operators (BatchNorm, InstanceNorm, GroupNorm, RMSNorm, Frobenius/L1/L2/LayerNorm, problems 33–40), explicit reductions along dimensions (Sum, Mean, Max, Min, Argmax, Argmin, problems 47–49, 51–53), and cumulative operators (cumsum, cumprod, cumsum_reverse, cumsum_exclusive, masked_cumsum, problems 89–93).
- Elementwise (22): pointwise activations (ReLU, LeakyReLU, Sigmoid, Tanh, Softmax, LogSoftmax, Swish, GELU, SELU, HardSigmoid, Softplus, Softsign, ELU, HardTanh, MinGPTNewGelu, problems 19–32, 88), loss functions (MSE, CrossEntropy, Huber, KLDiv, TripletMargin, Hinge, problems 94, 95, 96, 98, 99, 100), and the ScaledDotProductAttention operator (problem 97), placed here because it is dominated by pointwise-scaled dot products.
- Pooling (6): max and average pooling on 1D/2D/3D (problems 41–46).

Two problems (4: matrix–vector multiplication, 5: matrix–scalar multiplication) are vector-reduction/scalar operators that do not fit cleanly into the five categories; they are included in the L1 overall summary (Table 1) but omitted from the category tables, rather than being forced into ill-fitting bins.

L2 categories (fused operators). Each L2 problem fuses two or more operators; we group by the dominant operator family in the fusion, since fusion gains are determined by the dominant reduction/compute pattern:

- Conv2d fusions / Conv3d fusions / ConvTranspose2d fusions / ConvTranspose3d fusions: fusions whose primary operator is a 2D/3D standard or transposed convolution, respectively.
- Gemm fusions: fusions whose primary operator is a dense GEMM.
- Matmul/BMM fusions: fusions whose primary operator is a matrix or batched matrix multiplication (BMM) other than dense GEMM.

The distinction between "Gemm" and "Matmul/BMM" follows KernelBench's own naming: operators named Gemm_* are grouped into Gemm fusions, and operators named Matmul_*/BMM_* into Matmul/BMM fusions. This explains why the convolution grouping in L2 is finer than in L1 (which merges all convolutions into one category): in L2, the bottleneck profiles of fused kernels differ enough across convolution dimensionality and direction (2D vs. 3D, standard vs. transposed) to warrant separate rows.

L3 families (full models). Each L3 problem is a full model or its core module; the 50 problems fall into five architecture families:

- CNN (classic and modern blocks/networks) (24): LeNet5, AlexNet, GoogLeNet (Inception module and V1), ResNet (basic block, 18, 101), VGG16/19, DenseNet (transition layer, dense block, 121, 201), SqueezeNet (Fire module and full network), MobileNetV1/V2, EfficientNet (MBConv, B0–B2), ShuffleNet (unit and full network), RegNet (problems 4–27).
- RNN/LSTM/GRU (10): VanillaRNN and its hidden-state variants, LSTM and its h/c/bidirectional variants, GRU and its hidden/bidirectional variants (problems 33–42).
- Transformer/Attention (8): VisionTransformer, SwinMLP, SwinTransformerV2, VisionAttention, ConvolutionalVisionTransformer, MinGPT causal attention and block, ReLU self-attention (problems 28–32, 43, 44, 50).
- MLP variants (3): Mlp, ShallowWideMLP, DeepNarrowMLP (problems 1–3).
- Other sequence models (5): UNetSoftmax, NetVLAD (with and without ghost clusters), Mamba2 (returning Y and returning the final state) (problems 45–49).

B Taxonomy of LLM-Driven GPU Kernel Optimization Methods

Table 9: A taxonomy of LLM-driven GPU kernel optimization methods. ○ = partial support; ● = full support; – = no additional training or not applicable.

Method	NCU	Memory	KBench	Triton	Memory Index	Training
Training-driven						
Kevin, Dr.Kernel, Kernel-Smith	–	–	●	●	–	RL
Agent-driven — pure iteration						
AutoKernel, K-Search	–	–	●	●	–	–
Agent-driven — external memory						
KernelSkill	●	●	●	–	Skill	–
KernelBlaster	●	●	–	–	Bottleneck*	–
AccelOpt	–§	●	–	○	–¶	–
daVinci-kernel	–	●	●	●	BM25+LLM	RL
Agent-driven — hardware profiling						
PRAGMA, CudaForge	●	–	–	●/CUDA	–	–
Agent-driven — multi-agent						
STARK, Astra, GEAK	○	–	○	●	–	–
EvokeAgent (ours)	●	●	●	●	Bottleneck†	–

*Organized by performance state (bottleneck type) and retrieved with NCU-extracted states as keys, but within-state selection is weighted random search that does not exploit policies' historical validation records. †Organized by bottleneck, deterministically retrieved, carrying provenance and EMA-speedup records; the knowledge base evolves data-driven with optimization runs. §AccelOpt uses the AWS Neuron Profile (Trainium NPU), not NCU. ¶A capacity-limited FIFO queue injected wholesale into the prompt; no indexing/retrieval mechanism; applies only within a single problem's iterations.

C Experimental Environment and Configuration Details

C.1 Software and Hardware Versions

Table 10: Experimental environment. All optimization runs and benchmark measurements are performed on the same GPU of the same machine.

Component	Version / configuration
GPU	NVIDIA H20-3e (sm_90, 96GB HBM3, 78 SMs)
Driver	595.45.04
CUDA Toolkit	13.2
PyTorch	2.12.1+cu132
Triton	3.7.1
NCU (Nsight Compute)	2026.1.1
Operating system	Linux 5.15

C.2 LLM Configuration

All generation components (initial generation, iterative optimization, bottleneck analysis, knowledge curation) use the same untuned GLM-5.2, invoked through its API in an agentic loop (the model can read and write files and execute commands within the loop). No SFT or RL is applied; sampling parameters are not explicitly set and follow the API defaults.

Iteration budget. Each problem uses three initial candidate generations to select the starting version v_0, followed by at most 10 optimization rounds. No reported result includes rounds beyond this cap. The per-problem optimization timeout is 1800 seconds.

C.3 Benchmark Measurement Protocol

Speedup measurement follows KernelBench convention, using the same measurement script for all versions and baselines:

- Timing: CUDA-event timing; 10 warmup runs followed by 100 measured runs; the mean is taken after trimming 10% from each tail (outlier removal);
- Correctness gate: a speedup is counted only after passing the correctness check. Tolerances align with KernelBench v0: atol = 10^{-2} and rtol = 10^{-3} uniformly for fp32/tf32/fp16/bf16;
- Input shapes: each problem uses its single default input shape (as shipped with the KernelBench problem); correctness and speedup are measured on the same shape;
- Baseline: PyTorch eager (the problem's original Model class), measured back-to-back in the same process as the kernel under test, to eliminate cross-time load drift in the shared-GPU environment.

Measurement noise in a shared single-GPU environment. The experimental machine is a multi-GPU shared server, and the same card occasionally hosts background processes with fluctuating load. To control noise: (1) kernels that fail correctness are counted as no-speedup rather than excluded; (2) speedup comparisons (e.g., the two ablation groups, round-by-round convergence) all come from back-to-back measurements of the same problem on the same card; (3) we do not report precise single-problem number comparisons; the aggregate statistics (geometric means, fast$_p$) have limited variance contribution at the 100-problem scale.

C.4 KB Retrieval Configuration

Each retrieval during an iteration returns all policies of the current bottleneck sub-category (no top-k truncation). Policies are presented in the prompt with their EMA speedup and provenance records, and the LLM decides whether to adopt, adapt, or create a new policy.

Thus, the NCU bottleneck sub-category is the retrieval key, while the retrieved policies provide action guidance rather than a fixed action.

Failure and EMA behavior. The code updates EMA only through add_strategy.py. Compilation or correctness failures remain in version history but are not inserted into the KB; runtime failures provide no valid benchmark and do not update EMA. The code formula would lower EMA if a lower speedup were submitted, but the prompt gate requests the write only when the relative gain exceeds 1.1×, so ordinary negative iterations do not reach the update entry point. Failed attempts are therefore not represented as KB samples, and there is no explicit penalty or eviction mechanism for stored policies; retention and preference are mediated by provenance and EMA in the prompt.

D Bottleneck Classifier: Full Rules and Metric Inventory

This appendix gives the full implementation details of the two-level NCU metric filtering described in Section 3.2. The rule-based classifier uses deterministic rules for the thresholded categories and is instantaneous and reproducible. The bank-conflict entry is an exception: its counters are injected as evidence and the LLM adjudicates whether to use that sub-category. The code resides in core/analysis/hierarchical_classifier.py.

D.1 Decision Procedure

At each iteration, we apply the following procedure to the current kernel's NCU profile:

1. The shape profile with the largest N (element count) among all shape profiles is taken as the representative profile (the one that best reflects throughput characteristics);
2. Level 1: the representative profile's 3 coarse-grained metrics (memory throughput %, SM throughput %, achieved occupancy %) determine the top-level bottleneck category according to the rule table 11;
3. Level 2: within that category, fine-grained metrics (warp-stall distribution, cache hit rates, register counts, instruction mix, etc.) determine the sub-category per rule table 12;
4. The sub-category's fine-grained metric groups and the retrieved policies are injected (Appendix D.5).

D.2 Level-1 Rules (Coarse-Grained, 3 Metrics)

Thresholds: $T_{\text{mem}} = 60\%$, $T_{\text{sm}} = 60\%$, $T_{\text{occ}} = 40\%$ (calibrated for the NVIDIA H20-3e configuration). Rules match in order, stopping at the first hit:

Table 11: Level-1 bottleneck classification rules (matched in order).

#	Condition	Verdict
1	Memory throughput $> 60\%$ and SM throughput $< 40\%$	Memory bottleneck
2	SM throughput $> 60\%$ and memory throughput $< 60\%$	Compute bottleneck
3	Occupancy $< 40\%$ or (SM $< 40\%$ and memory $< 40\%$)	Latency/occupancy bottleneck
4	Remaining (both moderate)	Latency/occupancy bottleneck

D.3 Triton Level-2 Rules (Fine-Grained, 16 Diagnostic Sub-Categories)

The Level-2 rules distinguish mechanisms rather than merely enumerating symptoms. The 16 rule-defined categories shown here correspond to the Triton KB. CUDA uses a separate 12-category KB tree; fallback routes in Table 13 do not introduce additional CUDA KB keys. Placing retrieval at the diagnostic sub-category granularity provides a structured routing key.

Table 12: Level-2 sub-classification rules. Each row returns on hit; "Default" is the fallback sub-category when no rule hits.

Parent	Sub-category	Condition	Metrics
Memory bottleneck	Uncoalesced access	Sectors/request < 3.5	Global-load sectors/request (ideal = 4)
	Memory latency	long_scoreboard $> 30\%$	Warp-stall distribution
	Cache thrashing	L2 hit $> 50\%$ and L1 $< 50\%$	L1/L2 sector hit rates
	Bandwidth saturation	L1 $< 50\%$ and L2 $< 30\%$	Same as above
	Unclassified Memory Bottleneck	—	—
Compute bottleneck	Tensor-core underutilization	Tensor-pipe active $< 10\%$	Compute-pipe utilization
	Thread divergence	Control-flow instructions $> 15\%$	SASS instruction mix
	Instruction dependency / low ILP	IPC < 1.0	Instructions per cycle
	Precision waste	fp16 instructions $< 10\%$ and fp32 $> 50\%$	SASS instruction mix
	(Default) Transcendental overhead	—	—
Latency/ occupancy bottleneck	Register pressure	Occupancy $< 40\%$ and registers/thread > 128	Launch configuration
	Shared-memory pressure	Shared memory/block > 100KB	Launch configuration
	Barrier stalls	Barrier stalls $> 15\%$	Warp-stall distribution
	Insufficient parallelism	Total grid blocks < 78 (SM count)	Launch configuration
	Launch overhead	Blocks < 156 ($2\times$ SM count)	Launch configuration
	Algorithmic inefficiency	SM $< 40\%$ and memory $< 40\%$	Both coarse metrics low

D.4 CUDA Level-2 Subclasses and Rules

The CUDA target uses the deterministic Level-1 rules in Table 11 and the CUDA-specific Level-2 routing rules in Table 13. Its KB retrieval tree contains 12 sub-categories. Within each parent category, deterministic rules are evaluated in order and return on the first hit. The compute parent uses math_overhead as its deterministic default when no earlier compute rule matches. The only model-adjudicated exception is bank conflicts: the corresponding NCU counters are injected as evidence, and the LLM decides whether to use the bank-conflicts sub-category. Fallback routes reuse existing CUDA KB keys and do not create additional sub-categories.

Fallback routing. If no memory rule matches, the route reuses memory_bound>bandwidth_saturated; if no compute rule matches, the compute parent returns its deterministic default compute_bound>math_overhead; the latency/occupancy fallback uses resident autotune and does not create a CUDA KB key. These routes reuse existing CUDA KB keys rather than adding sub-categories.

The CUDA KB starts with 71 seed policies and contains 638 policies after the evaluation runs. The Triton KB starts with 74 seed policies and contains 800 policies after the runs. CUDA seeds were adapted from KernelBlaster-inspired strategies and project-specific optimization experience; Triton seeds came from the project's routine optimization traces.

Table 13: CUDA Level-2 subclasses and classification rules. The CUDA KB contains 12 sub-categories. Fallback routes are described below the table and do not introduce additional CUDA KB keys; math_overhead is the deterministic compute default, whereas the bank-conflicts sub-category is adjudicated by the LLM from injected shared-memory bank-conflict counters.

Subclass	Condition	Selected NCU evidence	KB key
Memory latency	long_scoreboard $> 30\%$	warp-stall distribution	memory_bound> memory_latency
Cache thrashing	L2 hit $> 50\%$ and L1 $< 50\%$	L1/L2 sector hit rates	memory_bound> cache_thrashing
Bandwidth saturation	L1 hit $< 50\%$ and L2 $< 30\%$	L1/L2 sector hit rates	memory_bound> bandwidth_saturated
Bank conflicts	model-adjudicated from bank-conflict evidence	l1tex_data_bank_ conflicts	memory_bound> bank_conflicts
Thread divergence	control-flow instructions $> 15\%$	SASS instruction mix	compute_bound> thread_divergence
Instruction dependency	IPC < 1.0	instructions per cycle	compute_bound> instruction_ dependency
Math overhead (default)	no earlier compute rule matches	–	compute_bound> math_overhead
Register pressure	occupancy $< 40\%$ and registers/thread > 128	launch configuration	latency_occ>register_ pressure
Shared-memory pressure	shared memory/block > 100KB	launch configuration	latency_occ>shared_ mem_pressure
Insufficient parallelism	total grid blocks < 78	launch configuration	latency_occ> insufficient_parallelism
Launch overhead	blocks < 156	launch configuration	latency_occ>launch_ overhead
Algorithmic inefficiency	SM $< 40\%$ and memory $< 40\%$	two coarse metrics	latency_occ> algorithmic_ inefficiency

D.5 Metric Inventory Injected into the LLM

NCU collection proceeds at two levels: 14 coarse-grained metrics are always collected, while the Level-1 decision uses the three metrics specified in Table 11; 49 fine-grained metrics in 7 groups are selectively injected according to the Level-2 classification result. The 7 groups are:

Table 14: Fine-grained NCU metric groups (49 in total). At injection time, only the groups relevant to diagnosing the current bottleneck sub-category are selected.

Group	Metrics	Representative metrics
Compute pipeline	6	FMA/Tensor/ALU pipe activity, IPC, SM active cycles
Memory hierarchy	11	L1/L2 sector hit rates, DRAM read/write bytes, sectors/request, shared bank conflicts
Warp stalls	12	long/short_scoreboard, wait, barrier, mio_throttle, etc., 12 stall-reason fractions
Scheduler	3	Issue-active rate, eligible warps/cycle, active warps
Warp state	3	Average warp latency (issued/executed), thread instructions/executed instructions
Instruction mix	6	SASS integer/fp32/fp16/memory/control-flow/fp64 instruction fractions
Launch configuration	8	Registers/thread, shared memory, grid/block, occupancy limiters

Relation to the ablation control group. The control group removes bottleneck-indexed policy access and the proposed routing mechanism while keeping the LLM, hardware, initial candidates, correctness checks, and optimization budget unchanged. Thus, the primary ablation evaluates the complete routing-and-retrieval design rather than treating policy retrieval as an isolated component.

D.6 Retrieval Path and Policy Presentation

After classification, the retriever fetches from the target KB all policies under the identified bottleneck sub-category. Each policy is presented to the LLM with the following fields: technique name, natural-language description, concrete parameter ranges (numerically specified feasible regions rather than abstract advice), provenance (which operators the policy was validated on, and with what speedup), and the EMA speedup (0.7× historical value + 0.3× new validation, damping single-run outliers). Below is the complete field set of one real policy (excerpted from the CUDA KB):

> technique: multi_dimensional_grid
> description: Replace a flat 1D CUDA grid with a 3D grid for index decoding. Under a 1D grid, each thread needs three divisions and three modulo operations to reconstruct (n, c, oh, ow). Instead, use blockIdx.z=N×C to select a plane and a 2D block (32×8) to tile the (oh, ow) output; coordinates are then obtained directly from block and thread IDs, freeing the ALU pipeline from integer index arithmetic (NCU measured 46.95 integer instructions versus 7.82 FP32 instructions). Mapping the x dimension to consecutive ow positions also ensures coalesced loads.
> parameters: 3D grid=(ow tiles, oh tiles, N×C); block size=32×8, with x-dimension alignment of at least 128 B for coalesced access; prefer this scheme for 4D outputs.
> actual_speedup: 2.69 (EMA)
> proven_on: convtranspose3d_logsumexp_hardswish_subtract_clamp, 5.05×, 2026-08-04; max_pooling_2d, 1.90×, 2026-07-30; convtranspose3d_layernorm_gelu_scaling, 2.25×, 2026-08-03