KLINEAGE: RECOVERING THE MISSING *When* OF KERNEL OPTIMIZATION BY DEOPTIMIZING EXPERTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based agents are increasingly used to generate GPU kernels, but they often struggle to determine when an optimization is sound because its required code state and dependencies are implicit in expert implementations. We introduce KLINEAGE, which learns this missing "when" knowledge from expert kernels: instead of relying on forward rollouts, KLINEAGE walks expert implementations backward through validation-gated simplifications and reverses each accepted step into a reusable optimization skill. Each skill records not only the optimization intent, but also when to apply the optimization technique, including where it applies in code, what conditions made it valid, what effect it has, and what failures its assumptions avoid. A downstream LLM materializes these skills on new code surfaces under the same compile/correctness/profile gate. This guidance on when to apply each optimization can help downstream models to generate higher-performance kernels. On five expert workloads across two NVIDIA architectures, these lineage-derived skills serve as an effective optimization curriculum, exceeding recent memory-based LLM-kernel baselines in both final kernel quality and optimization efficiency under the same fixed budget. We also demonstrate that the KLINEAGE framework extends beyond NVIDIA GPUs to Ascend NPUs.

1 INTRODUCTION

LLM-based agents are becoming a practical interface for GPU-kernel generation: they can draft an implementation, run it, inspect compiler or execution feedback, and iterate. Benchmarks such as KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025b) have made this setting measurable, while recent agents improve kernels through feedback, search, post-training, or skill memory (Andrews & Witteveen, 2025; Wang et al., 2025; Lange et al., 2025; Du et al., 2026a; Sun et al., 2026; Du et al., 2026b). The setting is also increasingly multi-surface: agents may write CUDA, Triton, CuTe/CUTLASS-style templates, TileLang programs, or a restricted kernel DSL, each exposing different scheduling knobs and failure modes (Li et al., 2025b; Wang et al., 2025; Saba et al., 2026; Li et al., 2025a; Hari et al., 2026; Chu et al., 2025; Qu et al., 2026). However, prior studies document a gap between attempting an optimization and applying it effectively. KernelBench reports that providing optimization examples can encourage more aggressive transformations while increasing execution failures (Ouyang et al., 2025).

The problem is often not that the model lacks the names of useful optimizations. It can suggest tiling, shared-memory staging, vectorized loads, software pipelining, or hardware intrinsics. What is missing is the condition under which each optimization is valid and useful. Vectorized loads require alignment and valid tail handling; pipelining assumes staged data movement; matrix intrinsics require compatible tile shapes, layouts, and hardware support. A forward-search agent may therefore apply a familiar optimization to the wrong state, producing a compile error, a correctness failure, or a slower kernel. Pretraining captures much of the *what* of optimization, but not reliably the *when*.

We treat this missing "when" knowledge as the object to be learned. The memory item we need is not a free-form hint such as "use vectorized loads", but a skill in a narrow sense: a verified, code-anchored specification of an optimization intent, including where the intent acts on a code or IR surface, what carrier can guide implementation, the context in which it has been observed to work, the effect it tends to have, and the risks when its assumptions are violated. Existing kernel-agent memories expose parts of this signal through failure rules, profile-state knowledge bases, or external

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.

human expert skill snippets (Du et al., 2026b; Cao et al., 2026; Dong et al., 2026; Sun et al., 2026), while post-training approaches may absorb it into model weights (Du et al., 2026a; Baronio et al., 2025; Li et al., 2026). These memories are useful, but when they are built from forward rollouts, their coverage is limited by the trajectories the agent was able to discover. The optimization experience they accumulate depends on what the backbone model can propose and successfully implement. When high performance requires unfamiliar combinations of hardware features and optimization steps, forward rollouts can remain concentrated on familiar implementation patterns, leaving the relevant code states and dependencies unexplored within a fixed budget. Expert kernels supply successful implementations directly, giving KLINEAGE a concrete basis for recovering optimization paths and applicability conditions that the agent may fail to discover on its own. Our goal is to externalize the missing "when" knowledge from expert artifacts instead.

We introduce KLINEAGE, a framework for inducing reusable skills from expert GPU-kernel lineages. The key observation is that expert kernels already contain the missing conditional knowledge, but only implicitly. Expert kernels are developed through a sequence of interdependent optimization decisions involving tiling, vectorization, parameter selection, and hardware features. However, the final code typically leaves the intermediate code states, optimization order, and conditions that make each step valid implicit.KLINEAGE recovers these decisions by walking an expert kernel backward through named, validation-gated deoptimization steps. Each accepted simplification is stored as its inverse forward transition, yielding a curriculum from a naive implementation toward the expert kernel; the forward transitions observed across experts become evidence for reusable skills. To our knowledge, KLINEAGE is the first LLM-kernel memory constructed by deoptimizing expert implementations rather than accumulating forward-rollout states.

At reuse time, KLINEAGE retrieves skills by scope. Given a new target, it retrieves skills along the target case, surface language, platform, and already-applied optimizations. The target state need not be naive: KLINEAGE can start from any current kernel attempt and use the retrieved lineage to complete missing intents or repair mis-materialized ones. It then lets a downstream LLM materialize selected skills on the target code surface using their anchors and carriers. This keeps the LLM in the role where it is useful—instantiating a validated code-anchored intent and repairing local mismatches under feedback—while the memory constrains it with validated preconditions and observed risks.

We evaluate KLINEAGE on two NVIDIA platforms (H100 and RTX PRO 6000) and on Ascend910B. On each platform, KLINEAGE induces a skill library from five expert kernels through round-trip validation. The induced skills generalize to held-out operators, where KLINEAGE outperforms memory-based kernel optimization agents (AdaExplore (Du et al., 2026b) and AccelOpt (Zhang et al., 2025)) as well as few-shot prompting with the same expert kernels given as raw source. On broader workloads, KLINEAGE raises the base agent's mean speedup over `torch.compile` from $1.05\times$ to $1.32\times$ on 100 KernelBench Level 2 problems (Ouyang et al., 2025; Wen et al., 2025) on NVIDIA, and over eager PyTorch from $0.63\times$ to $2.09\times$ on Ascend. These results show that conditioned skills make expert optimization knowledge reusable beyond the kernels from which it was extracted.

Contributions. This paper introduces KLINEAGE and makes three technical contributions:

- **Conditional skills that encode when to optimize.** We make the "when" of optimization explicit by pairing each optimization intent with the conditions under which it is valid and useful. Each skill includes code anchors and implementation carriers to guide materialization, together with effects, risks, verification evidence, and scope over workloads, languages, platforms, and prior optimization states.
- **Guided deoptimization for constructing conditional skills.** We walk expert kernels backward through validation-gated simplifications and reconstruct accepted steps as validated forward transitions, forming an executable curriculum from naive to expert. We then lift these transitions into conditional skills, using the intermediate code states, optimization dependencies, and validation outcomes as evidence for their applicability conditions.
- **Using conditional skills to guide when to optimize.** We retrieve conditional skills according to the target kernel's current state and context, using their applicability conditions to guide which optimizations to apply at each stage. Only skills admitted through held-out roundtrip validation are reused; a downstream LLM materializes the selected skills into optimized kernels under a compile/correctness/profile gate.

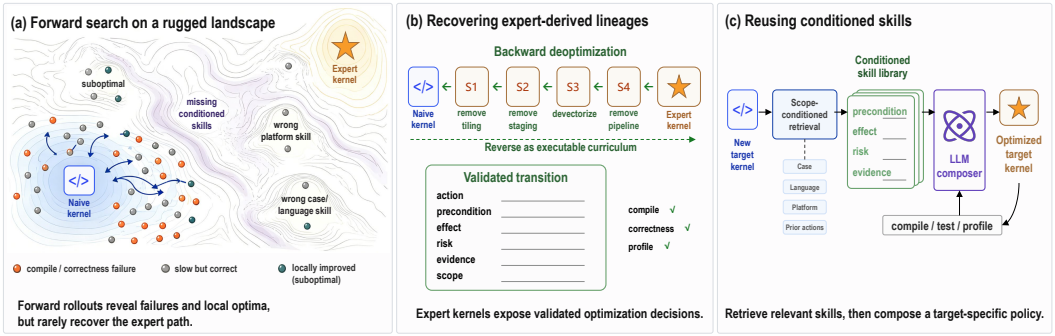


Figure 1: **Motivation of KLINEAGE. (a)** Forward search often knows which optimizations to try but not when their preconditions hold. **(b)** KLINEAGE walks expert kernels backward to recover validated forward transitions from simpler states toward expert states. **(c)** The recovered skills are reused only as code-anchored candidates, with compile/test/profile gates deciding admission on new targets.

2 RELATED WORK

LLM agents for GPU kernel optimization. Recent benchmarks such as KernelBench Ouyang et al. (2025), TritonBench Li et al. (2025b), and production-oriented FastKernels (Oliaro et al., 2026), together with kernel substrates such as ThunderKittens Spector et al. (2025), have made GPU-kernel generation a common testbed for LLM agents. Recent systems improve kernels through compiler/profile feedback, multi-agent refinement, DSL or minimal-program interfaces, static-analysis guidance, post-training, and search over CUDA, Triton, CuTe/CUTLASS-style, and related surfaces (Andrews & Witteveen, 2025; Wang et al., 2025; Lange et al., 2025; Zhu et al., 2026; Wei et al., 2025; Saba et al., 2026; Li et al., 2025a; Hari et al., 2026; Chu et al., 2025; Qu et al., 2026; Lou & Muller, 2024; Davis et al., 2026; Baronio et al., 2025; Li et al., 2026; 2025c). KernelFoundry (Wiedemann et al., 2026), for example, uses quality-diversity search, prompt evolution, and template tuning to explore kernels under hardware profiling feedback, while AKO4ALL (Xie et al., 2026) drives an agent through a fixed optimization protocol and Atrex Kernel Agent (Yang et al., 2026) works a profile-driven loop over operators sampled from production inference traces. These systems establish realistic compile/profile loops, but they largely acquire optimization knowledge by moving forward from an initial program.

Reusable experience and memory. Closest to our work are kernel agents that reuse optimization experience: AdaExplore Du et al. (2026b) converts failures into textual rules; AccelOpt (Zhang et al., 2025) summarizes a run's evolving state; K-Search (Cao et al., 2026) co-evolves kernels with an intrinsic world model; KernelSkill Sun et al. (2026) organizes expert-written skills; and KernelBlaster (Dong et al., 2026) stores forward-rollout experience in a profile-centric memory. CuBridge (Ma et al., 2026) is also related in using LLMs to understand and reconstruct expert attention kernels. KLINEAGE differs in how the memory is constructed and represented: to our knowledge, it is the first LLM-kernel memory construction method that automatically derives reusable lineages by deoptimizing expert implementations, and each skill records code/IR anchors, carriers, preconditions, effects, risks, evidence, and scope rather than only a forward trace, textual rule, or pre-existing snippet.

3 METHOD

KLINEAGE constructs reusable, code-anchored optimization memory from expert evidence. Guided deoptimization walks an expert kernel backward but stores accepted steps as inverse forward transitions (§ 3.2); a lift turns recurring transitions into candidate skill hypotheses (§ 3.3); held-out roundtrip materialization decides which hypotheses become reusable skills (§ 3.4).

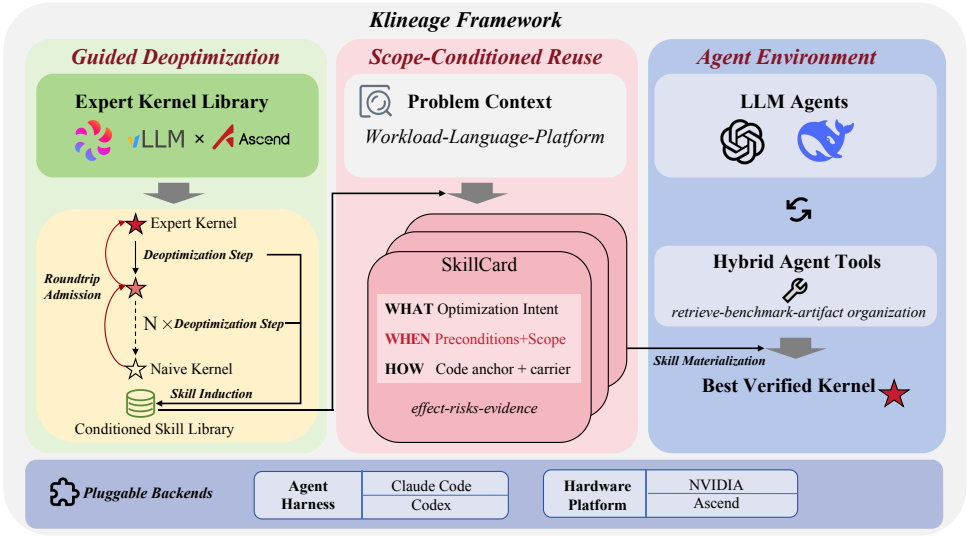


Figure 2: Overview of KLINEAGE.

3.1 MOTIVATION

As illustrated in Figure 1, expert kernels entangle many compatible optimization decisions, and walking them backward under validation exposes each decision as an explicit forward transition. This direction is more natural in practice because high-quality expert kernels in existing repositories are often not paired with corresponding naive implementations. We make this construction precise below.

KLINEAGE represents this evidence at three levels as shown in Figure 1(b). A *backward deoptimization step* $b_i : K_i \to K_{i-1}$ removes or weakens one optimization and records the edit, locus, and validation evidence for the simplified predecessor. KLINEAGE stores the inverse $\tau_i = b_i^{-1} : K_{i-1} \to K_i$ as a *forward concrete transition*: a validated before/after edit from a simpler state toward the expert state. A *lineage* is the resulting forward sequence

$$L = (K_0, \tau_1, K_1, \ldots, \tau_n, K_n = K^*),$$

an executable curriculum from naive to expert, not a claim about the expert's historical writing path.

Concrete transitions are still tied to one source kernel. KLINEAGE lifts them into an *optimization intent*: the abstract pass family being expressed, such as staging reused tiles, vectorizing aligned contiguous loads, or exposing tunable parameters. A *candidate skill hypothesis* $\hat{\sigma}$ makes the intent actionable by adding a code/IR anchor, a carrier, conditions, and evidence:

$$\hat{\sigma} = \big(\iota,\ \text{anchor},\ \text{carrier},\ \text{pre},\ \text{effect},\ \text{evidence},\ \text{risk},\ \text{scope},\ \text{ver}\big),$$

where ι is the intent, anchor identifies the code or IR surface on which it acts, and carrier is an actionable but not necessarily executable representation such as a diff sketch, pseudocode, annotated kernel snippet, schedule fragment, or natural-language code annotation. The other fields record preconditions, expected effects, evidence, risks, scope over case/language/platform/prior-action context, and a verification log ver (§ 3.3). A hypothesis becomes an admitted *skill* only after at least one trial succeeds. At reuse time, an agent *materializes* the skill on a target code surface; the candidate is accepted only if it passes the compile/correctness/profile gate.

3.2 GUIDED DEOPTIMIZATION

Starting from $K_n = K^*$, KLINEAGE repeatedly proposes a simplification (e.g., remove software pipelining, devectorize, scalarize an intrinsic), patches the kernel with an LLM rewriter, and accepts

the patch only if it passes validation. The inverse of an accepted simplification becomes a forward transition in the lineage; rejected simplifications become precondition-violation evidence for later risk fields.

Operationalizing $\tau_i = b_i^{-1}$. The forward transition τ_i is not a textual reversal of the diff that produced b_i. The LLM rewriter re-derives a forward edit on K_{i-1} functionally equivalent to K_i, given the action category and locus from the backward step. The candidate forward edit is then run through the compile/correctness/profile gate; only edits whose output kernel matches K_i's validation status are stored as τ_i. This makes τ_i an executable forward primitive grounded in K_{i-1}'s state, not a syntactic inversion that may not even compile.

During induction we group accepted backward steps by action category to anchor the lift, over an open vocabulary rather than a closed set of actions. Multiple intents may be lifted from one category, the same intent may admit different carriers in different languages, and any partial order over actions is treated as a soft prior: if a simplification violates the order but still passes validation, its inverse transition is kept.

3.3 From Transitions to Skills

The library is induced from forward concrete transitions. Transitions sharing an action category and similar locus / effect signatures are aggregated, and an LLM receives their forward diffs, loci, profile deltas, and local code context, then lifts them into an intent ι, an anchor, a carrier, and a portable $(\mathrm{pre}, \mathrm{effect}, \mathrm{risk})$ tuple. The intent describes the manipulated code construct and exploited platform property; the anchor and carrier keep it actionable.

Intent, carrier, and materialization. The lift separates the general idea from its realization. The intent names the optimization, the carrier gives it a code-bearing form, and materialization instantiates the admitted skill on a target language and code surface. One intent can therefore have different realizations across cases, languages, and platforms.

Roundtrip admission. An induced $\hat{\sigma}$ starts as a hypothesis. Admission requires at least one held-out roundtrip in which an agent, given only $\hat{\sigma}$, materializes it on a fresh state and reproduces its expected effect; success is recorded in ver. Only admitted skills $\sigma \in \mathcal{S}$ are reused.

3.4 Scope-Conditioned Reuse

KLineage supplies a verified, scope-conditioned candidate pool rather than a new search policy. Given a target T and a root submission, the downstream agent selects admitted skills, materializes them on the target code surface, and submits candidates under the same fixed-budget compile/correctness/profile loop used by all baselines. In the online prompt, each retrieved SkillCard is serialized as the fields in $\hat{\sigma}$—intent, anchor, carrier, precondition, effect, evidence, risk, scope, and verification log—with the carrier copied verbatim as the code-bearing hint to instantiate, not as an executable tool. The root submission may be a compiler baseline, a partially optimized agent attempt, or a previous best kernel. To bound the candidate pool, KLineage retrieves with a multi-faceted retriever: given $T = (c_T, l_T, p_T, d_T)$, it returns the union of per-dimension top-k skills under similarities over $(\mathcal{C}, \mathcal{L}, \mathcal{P}, \mathcal{D})$, using each skill's verified rather than declared scope.

3.5 KLineage Framework

Building on these ideas, we implement KLineage as an agent framework for GPU kernel programming. As shown in Figure 2, KLineage induces conditioned skills from expert-written kernels such as FlashInfer Ye et al. (2025) through guided deoptimization. Each skill records not only an optimization, but also the conditions under which it applies, such as data layout, hardware features, memory placement, and pipeline schedule. Given a new workload with the specified language and target platform, KLineage guides LLM agents to retrieve and materialize skills whose conditions match the current context. We also provide hybrid agent tools for LLM Agents in the environment, including shared interfaces for skill retrieval, kernel artifact organization, and benchmarking, together with platform-specific profiling tools such as `ncu` and `msprof`. KLineage also provides a

unified abstraction for pluggable integration of different backends and agent harnesses. This design makes agent trajectories and generated artifacts traceable while leaving agents free to profile kernels, diagnose bottlenecks, and explore platform-specific optimizations.

4 EVALUATION

We structure our evaluation around four research questions:

RQ1: How do kernels generated by KLINEAGE compare with those produced by existing memory-based agents and with expert implementations?

RQ2: Does a KLINEAGE improve kernel generation over providing expert source code directly?

RQ3: How well does KLINEAGE generalize to non-NVIDIA accelerators?

RQ4: How effectively can the induced skills be transfered to other operators and broader benchmarks?

4.1 SETUP

Platforms and languages. We evaluate on NVIDIA H100 PCIe (SM90), NVIDIA RTX PRO 6000 Blackwell (SM120), and Ascend 910B. On Nvidia, we use PyTorch 2.11.0+cu130, FlashInfer 0.6.11, TileLang 0.1.8, and FLA 0.5.0. FLA replaces FlashInfer as the GDN reference on SM120. On Ascend, we use CANN 9.1.0 and Torch-NPU 2.10.0. We build a separate skill library for each platform and record the verified architecture in each skill's scope. The target languages are CUDA C on NVIDIA and AscendC on Ascend.

Expert kernels and benchmarks. We select five expert kernels per platform, covering GEMM, Conv2d, FMHA, GDN, and Top-K. These operators are common in production, and their expert implementations contain a range of optimization techniques. Appendix A lists the workload shapes, data types, and reference implementations. The transfer targets and benchmark problems do not contribute to skill induction. We evaluate the transferability of the induced skills using 100 KernelBench Level 2 problems on NVIDIA GPUs and 100 problems from MultiKernelBench's Fusion category on Ascend NPUs.

Agents and budgets. We use Claude Code with Claude Opus 4.6 as the harness and LLM backbone in KLINEAGE for offline skill induction and online kernel generation on both Nvidia and Ascend platforms. The online generation budget for these experiments is $10 per workload; induction costs are reported separately. For operator transfer and benchmark evaluation on both platforms, we use Codex with DeepSeek-V4.1-Flash. Each transfer target has a two-hour wall-clock budget. Each problem in KernelBench or MultiKernelBench has a 30-minute wall-clock budget. All reported costs are calculated using the corresponding model providers' official API pricing.

4.2 RQ1: KERNEL GENERATION UNDER INDUCED SKILL LIBRARY

We induce one library for each NVIDIA platform by the procedure of § 3.2 and § 3.3: walk each expert kernel backward under the checks, keep the reverse of every accepted step, and admit a lifted skill only once an agent handed that skill alone reproduces its effect on the deoptimized kernel.

As shown in Table 1, this procedure yields 52 admitted SkillCards on H100 and 36 on RTX PRO 6000 from the five expert kernels per platform listed in Table 5. The roundtrip admission rate is lower on H100: 72.2% $(52/72)$, compared with 80.0% $(36/45)$ on RTX PRO 6000. This difference is consistent with the more complex optimizations in the H100 experts, which use features such as distributed shared memory and warp-group tensor-core (WGMMA) instructions. Each SkillCard records the code and preconditions for applying an optimization, such as the layout requirements for WGMMA on H100. Induction is a one-time offline cost of $131 on H100 and $92 on RTX PRO 6000.

What published memory-based agents reach on the same five kernels. Two systems already build a reusable memory for kernel optimization: AdaExplore (Du et al., 2026b), which turns failures

Platform	Skills in library	Roundtrip Admission	Induction cost (USD)
H100 (SM90)	52	52/72	131
RTX PRO 6000 (SM120)	36	36/45	92

Table 1: The number of induced skills, roundtrip admission rate and induction cost of KLINEAGE on 2 Nvidia platforms across 5 expert kernels.

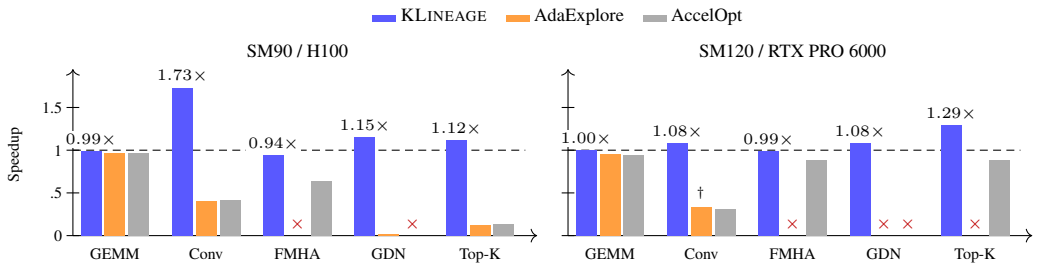


Figure 3: Generated CUDA-kernel speedup over the platform-specific vendor reference.

into rules and searches with MCTS, and AccelOpt (Zhang et al., 2025), which keeps an LLM-summarized memory over the kernels it has generated. On the same five expert workloads, they reach $0.25\times$ and $0.54\times$ against the vendor reference and return no correct kernel at all on four and two of the ten workload–platform pairs; KLINEAGE reaches $1.12\times$ and is correct on all ten (Figure 3, absolute latencies in Table 6). Neither baseline converts more budget into a structurally different kernel, so we settle the comparison here and read the rest of the evaluation as KLINEAGE against the expert code it was induced from.

In Figure 3, a red $\times$ marks a workload with no correct kernel within the budget, and † marks the post-hoc adapter audit of AdaExplore's SM120 Conv2d kernel after excluding fallback wrappers.

4.3 RQ2: Expert Source versus Conditioned Skills

We replace the conditioned skill library with expert kernel sources, keeping the rest of the setup fixed and allowing iterative refinement in both settings.

Generation from expert source is slower on all ten workload–platform pairs: 5.83–$14.48\times$ on Conv2d and 37.43–$105.18\times$ on GDN. The gap is smaller on GEMM and Top-K, at 1.12–$1.48\times$. Per-workload latencies are reported in Table 7 (Appendix C).

Even after repeated refinement, agents given expert source code often converge to simpler implementations that omit complex combinations of optimizations. On Conv2d, the agent reaches the appropriate tensor-core intrinsic but fails to combine it with the required shared-memory staging, swizzling, and pipelining. On SM90 GDN, it recovers the local `cumsum` transpose but omits the subsequent WGMMA, pipelining, and TMA optimizations. In both cases, the agent recovers an individual optimization while missing the intermediate code states needed to compose it with others. These results show that conditioned skills record these prerequisites and guide the agent in building the intermediate states needed for more complex optimizations. Roundtrip admission validates individual skills so that the LLM can tackle long-horizon optimization through small, verified steps, progressively implementing expert strategies while exploring alternative implementations.

4.4 RQ3: Generalization beyond NVIDIA GPUs

On accelerators with less publicly available kernel code, coding agents may have less prior knowledge of platform-specific optimizations. We use Ascend 910B to test whether KLINEAGE can obtain useful optimization knowledge from the same five expert kernels on a non-NVIDIA platform. To this end, we induce the library from expert implementations in `torch-npu`, `cann-ops` and `vllm-ascend`, using the same five operator categories and workload shapes as on NVIDIA. Of 109 candidate skills, 67 pass roundtrip admission, yielding a rate of 61.5%, compared with 72.2% on H100 and

80.0% on RTX PRO 6000 (Table 1). Reaching a correct naive kernel on Ascend also requires more deoptimization steps than on NVIDIA. These difficulties are consistent with weaker model familiarity with Ascend programming, reflecting more limited coverage in pretraining data. The one-time offline induction cost is $197.

We then generate AscendC kernels for the five workloads from naive implementations using the induced skills. The generated kernels outperform the expert baselines on all five workloads, with speedups of 1.048×–1.496× (Table 2).

Time (μs)	GEMM	Conv	FMHA	GDN	Top-K
KLINEAGE	424.2	17.74	6518.3	3393.4	15.48
Expert Baseline	444.6	26.54	7531.0	4764.4	19.83
Speedup	1.048×	1.496×	1.155×	1.404×	1.281×

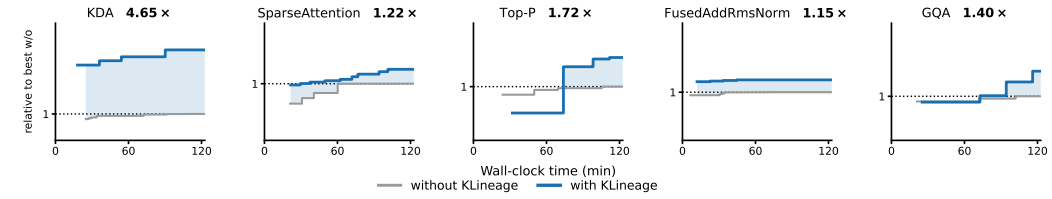
Table 2: Generated AscendC-kernel speedup over the platform-specific vendor reference.

These results suggest that KLINEAGE can build on the small set of expert kernels available early in a new hardware platform's development. Engineers typically prioritize core operators such as GEMM, and guided deoptimization allows KLINEAGE to recover the hardware-specific knowledge in these implementations and reuse it to optimize other operators.

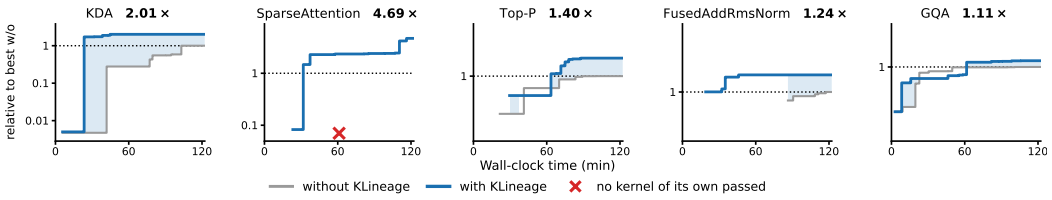
4.5 RQ4: TRANSFER TO NEW OPERATORS AND BENCHMARK SUITES

Operator transfer. On H100 and Ascend 910B, we evaluate transfer to five new operators: KDA, Sparse Attention, Top-P, Fused Add-RMSNorm, and GQA. None contributed to skill induction. We compare KLINEAGE with the Codex baseline, which receives no conditioned skills.

At the end of the budget, KLINEAGE retains faster kernels on every target where both settings produce valid kernels. Relative to Codex baseline, the geometric-mean speedup is 1.73× over all five H100 targets and 1.41× over the four Ascend targets with valid results from both settings. On the remaining Ascend target, Sparse Attention, KLINEAGE produces a valid kernel while Codex baseline fails. Figure 4 shows the optimization trajectories, and Appendix E reports the corresponding token and cost trajectories. These results demonstrate that the skill libraries induced by KLINEAGE transfer to kernel-generation tasks for operators not used during induction.



(a) H100 (SM90)



(b) Ascend 910B

Figure 4: KLINEAGE improve generation across the five transfer targets on H100 and Ascend 910B within two hours.

Optimizations recovered on Ascend. Table 3 shows that KLINEAGE increases the total number of optimizations implemented across the five Ascend targets from 40 to 55, while preserving all

optimizations found by the Codex baseline. Blue checkmarks indicate optimizations found only with KLINEAGE.

	KDA		Top-P		GQA		Sparse Attn.		Add-RMSNorm	
Optimization	w/o	w/	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Cube matmul for attention products	×	×	×	×	✓	✓	×	✓	×	×
Cube products split from Vector softmax	×	×	×	×	✓	✓	×	✓	×	×
Work-aware launch partition	×	×	×	×	×	✓	×	×	×	✓
Two-slot / ping-pong staging	✓	✓	✓	✓	✓	✓	×	✓	✓	✓
Prefetch ahead of compute	×	✓	×	×	×	×	×	✓	✓	✓
Pack or transpose for downstream access	×	✓	×	×	✓	✓	×	×	×	×
Event-ordered overlap across streams	×	×	×	×	×	✓	×	✓	×	×
Workspace reuse across phases	×	×	×	×	×	✓	×	✓	×	×
L2 bypass for streaming writes	×	×	×	×	×	×	×	×	×	✓
Row-max subtraction before exp	×	×	×	×	×	✓	✓	✓	×	×
GatherMask compaction of threshold band	×	×	×	✓	×	×	×	×	×	×
Nine further optimizations appear in both settings; the full audit is in Appendix F.										
Total (of 20)	7	**9**	8	**9**	10	**14**	7	**13**	8	**10**

Table 3: KLINEAGE retains all 40 optimizations found without skills and adds 15 across the five Ascend transfer targets.

4.5.1 GENERALITY TO BROADER OPERATORS

We test generality beyond the expert workloads on two benchmarks: 100 Level 2 problems from KernelBench (Ouyang et al., 2025) on NVIDIA and 100 problems from the Fusion category of MultiKernelBench (Wen et al., 2025) on Ascend 910B. Each platform's skill library remains unchanged, and none of these problems contributed to induction. Speedup is measured against `torch.compile` on NVIDIA and eager PyTorch on Ascend, then averaged over all valid solutions in each setting.

Table 4 reports both evaluations. On NVIDIA, both settings produce correct kernels for all 100 problems; KLINEAGE raises mean speedup from $1.05\times$ to $1.32\times$ and the number of problems faster than `torch.compile` from 26 to 42. On Ascend, correctness rises from 68 to 100 problems, mean speedup from $0.63\times$ to $2.09\times$, and the number faster than eager from 22 to 93.

NVIDIA runs incur higher API costs because faster kernel execution and profiling allow more agent turns within the same wall-clock budget.

Method	**Mean speedup**	**Correct (%)**	**Faster than reference (%)**	**Cost (USD)**
KernelBench Level 2 — NVIDIA, CUDA				
without KLINEAGE	$1.05\times$	100	26	28.77
with KLINEAGE	$\mathbf{1.32}\times$	100	**42**	31.19
MultiKernelBench Fusion — Ascend 910B, AscendC				
without KLINEAGE	$0.63\times$	68	22	21.92
with KLINEAGE	$\mathbf{2.09}\times$	**100**	**93**	24.05

Table 4: KLINEAGE improves performance on both broader operator benchmarks while producing correct kernels for all evaluated problems.

5 CONCLUSION

KLINEAGE recovers the missing *when* of GPU-kernel optimization from expert artifacts by walking them backward under validation-gated deoptimization and lifting accepted forward transitions into code-anchored, scope-tagged skills. Expert kernels become not only outputs to imitate but evidence from which the conditional structure of optimization can be externalized and reused.

AI USE STATEMENT

We used AI to construct conditioned skill memory for each of the three hardware platforms and to generate the final kernel code. We also used AI to polish the paper's writing. We reviewed the LLM-generated code and take responsibility for the final content of this paper.

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, 2025.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model, 2026. URL https://arxiv.org/abs/2602.19128.

Ruifan Chu, Anbang Wang, Xiuxiu Bai, Shuai Liu, and Xiaoshe Dong. Gpu kernel optimization beyond full builds: An llm framework with minimal executable programs. *arXiv preprint arXiv:2512.22147*, 2025.

Joshua H Davis, Klaudiusz Rydzy, Srinivasan Ramesh, Aadit Nilay, Daniel Nichols, Swapna Raj, Nikhil Jain, and Abhinav Bhatele. Keet: Explaining performance of gpu kernels using llm agents. *arXiv preprint arXiv:2605.04467*, 2026.

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, 2026. URL https://arxiv.org/abs/2602.14293.

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, 2026a.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre Wang He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, Yiming Yang, and Sean Welleck. Adaexplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation, 2026b.

Siva Kumar Sastry Hari, Vignesh Balaji, Sana Damani, Qijing Huang, and Christos Kozyrakis. Improving efficiency of gpu kernel optimization agents using a domain-specific language and speed-of-light guidance. *arXiv preprint arXiv:2603.29010*, 2026.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and 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. *arXiv preprint arXiv:2512.09196*, 2025a.

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, 2025b.

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, 2025c.

Xiaoya Li, Albert Wang, Guoyin Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning, 2026. URL https://arxiv.org/abs/2507.14111.

Mark Lou and Stefan K Muller. Automatic static analysis-guided optimization of cuda kernels. In *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 11–21, 2024.

Xing Ma, Yangjie Zhou, Wu Sun, Zihan Liu, Jingwen Leng, Yun Lin, Shixuan Sun, Minyi Guo, and Jin Song Dong. Cubridge: An llm-based framework for understanding and reconstructing high-performance attention kernels. *arXiv preprint arXiv:2605.05023*, 2026.

Gabriele Oliaro, Yichao Fu, May Jiang, Owen Lu, Junli Wang, Zhihao Jia, Hao Zhang, and Samyam Rajbhandari. Fastkernels: Benchmarking gpu kernel generation in production, 2026. URL https://arxiv.org/abs/2605.23215.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient GPU kernels?, 2025.

Qiuyi Qu, Yicheng Sui, Yufei Sun, Rui Chen, Xiaofei Zhang, Yuzhi Zhang, Haofeng Wang, and Ge Lan. A two-stage gpu kernel tuner combining semantic refactoring and search-based optimization. *arXiv preprint arXiv:2601.12698*, 2026.

Tara Saba, Zhiyang Chen, Jikai Jason Li, Anne Ouyang, Xujie Si, and Fan Long. Cutegen: An llm-based agentic framework for generation and optimization of high-performance gpu kernels using cute, 2026. URL https://arxiv.org/abs/2604.01489.

Benjamin Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Dan Fu, and Christopher Re. Thunderkittens: Simple, fast, and adorable kernels. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 2097–2130, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/05dc08730e32441edff52b0fa6caab5f-Paper-Conference.pdf.

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.

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 and evaluation benchmarks, 2025.

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. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. URL https://openreview.net/forum?id=IZKZIcPaHz.

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.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. Kernelfoundry: Hardware-aware evolutionary gpu kernel optimization. *arXiv preprint arXiv:2603.12440*, 2026.

Shuxiao Xie, Shuyang Xie, Dezhi Ran, Wei Yang, and Tao Xie. AKO: Agentic kernel optimization. https://tongminglaic.github.io/AKO, 2026. Technical report; AKO4ALL release at https://github.com/TongmingLAIC/AKO4ALL.

Lingyun Yang, Yuxiao Wang, Shenghao Liang, Linfeng Yang, Daocheng Ying, Chunbo You, Rui Zhang, Luping Wang, Yinghao Yu, Guodong Yang, and Liping Zhang. Are LLM-generated GPU kernels production-ready? a trace-driven benchmark and optimization agent, 2026. URL https://arxiv.org/abs/2607.14541.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025. URL https://arxiv.org/abs/2501.01005.

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. *arXiv preprint arXiv:2511.15915*, 2025.

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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(34):29168–29176, 2026. doi: 10.1609/aaai.v40i34.40155.

A Workload Details

Table 5 lists the shape, dtype and reference for the five expert kernels and five transfer targets on each platform. Expert kernels are the sources from which KLineage induces skills; transfer targets contribute no lineage.

Role	Kernel	Shape (dtype)	Reference
NVIDIA — expert kernels			
source	GEMM	$M{=}N{=}K{=}4096$ (BF16)	cuBLAS
source	Conv2d	$N{=}8$, $C{=}64$, $H{=}W{=}56$, $F{=}128$, $K{=}3$, s=1, p=1 (FP16)	cuDNN
source	FMHA	batch 8, heads 64, seq 2048, dim 128 (FP16)	FlashInfer
source	GDN	$h_{qk}{=}16$, $h_v{=}48$, dim 128, chunk 64, seq 4096 (BF16)	FlashInfer / FLA
source	Top-K	batch 64, seq 4096, top-$k{=}512$ (FP32)	FlashInfer
NVIDIA — transfer targets			
target	KDA	batch 1, tokens 4096, heads 96, dim 128 (BF16)	FLA
target	Sparse Attention	queries 8192, heads 128, 2048 sparse indices	FlashInfer
target	Top-P	batch size 32, vocabulary 128256	FlashInfer
target	Fused Add-RMSNorm	batch size 8192, hidden 7168	Torch
target	GQA	tokens 16384, 32 query / 8 KV heads	FlashInfer
Ascend — expert kernels			
source	GEMM	$M{=}N{=}K{=}4096$ (BF16)	Torch-NPU
source	Conv2d	$N{=}8$, $C{=}64$, $H{=}W{=}56$, $F{=}128$, $K{=}3$, s=1, p=1 (FP16)	AscendC custom op
source	FMHA	batch 8, heads 64, seq 2048, dim 128 (FP16)	Torch-NPU
source	GDN	$h_{qk}{=}16$, $h_v{=}48$, dim 128, chunk 64, seq 4096 (BF16)	vLLM-Ascend
source	Top-K	batch 64, seq 4096, top-$k{=}512$ (FP32)	Torch-NPU
Ascend — transfer targets			
target	KDA	batch 1, tokens 4096, heads 96, dim 128 (BF16)	Torch-NPU
target	Sparse Attention	queries 8192, heads 128, 2048 sparse indices	AscendC custom op
target	Top-P	batch size 32, vocabulary 128256	Torch-NPU
target	Fused Add-RMSNorm	batch size 8192, hidden 7168	Torch-NPU
target	GQA	tokens 16384, 32 query / 8 KV heads	AscendC custom op

Table 5: The workload suite separates five expert kernels used for induction from five targets used for transfer on each platform.

B Source-Tier Reconstruction and Memory-Based Kernel Agents

This appendix backs Figure 3 with absolute latencies, cost trajectories and the per-step record of one baseline run. AdaExplore (Du et al., 2026b) builds failure-rule memory with MCTS-style exploration; AccelOpt (Zhang et al., 2025) maintains an LLM-summarized memory over generated kernels. Both are re-run in our harness with their search policy and memory schema preserved (Appendix D). These numbers are a reconstruction check, not a transfer claim: the operators here are the ones whose lineages the library was built from.

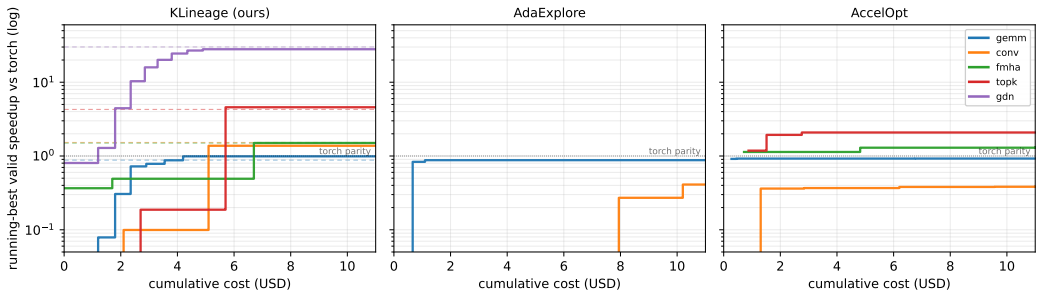


Figure 5: KLINEAGE reaches its best SM120 kernels within $4–$7 of online optimization cost.

C MAIN-TIER PER-INSTANCE LATENCIES

Table 6 gives the absolute per-kernel latencies behind Figure 3. "FAIL" indicates that the method did not produce a correct kernel within the budget; † marks a post-hoc adapter audit rather than an as-submitted valid kernel.

Platform	Kernel	KLINEAGE (ms)	AdaExplore (ms)	AccelOpt (ms)	Reference (ms)
SM90	GEMM	0.3130	0.3198	0.3190	0.3086
SM90	Conv2d	0.0157	0.0684	0.0658	0.0272
SM90	FMHA	3.5183	FAIL	5.1499	3.3121
SM90	GDN	0.2439	17.7673	FAIL	0.2816
SM90	Top-K	0.0143	0.1338	0.1276	0.0160
SM120	GEMM	0.3681	0.3896	0.3933	0.3688
SM120	Conv2d	0.0235	0.0760†	0.0822	0.0253
SM120	FMHA	3.2151	FAIL	3.6366	3.1932
SM120	GDN	0.6348	FAIL	FAIL	0.6867
SM120	Top-K	0.0131	FAIL	0.0288	0.0169

Table 6: KLINEAGE produces correct kernels on all ten CUDA workloads, while AdaExplore and AccelOpt fail on four and two, respectively.

Expert-source comparison. Table 7 gives the per-workload reconstruction latencies for the comparison in § 4.3. The Gen. column denotes expert-source prompting, and slowdown is its latency divided by the corresponding KLINEAGE latency.

Cost over the budget. Figure 5 reads the same comparison as a function of cumulative LLM-API cost on SM120. Loading the induced library costs around $0.7 in retrieval and load — the flat plateau at the start of each KLINEAGE curve — and is paid once per workload, since lineages and SkillCards stay in a single prefix-cached chat session. After the load, KLINEAGE reaches its eventual best between $4 and $7, well under the cap, and each step lines up with one named, verified skill rather than a black-box submission.

D BASELINE EXECUTION AND HARNESS NOTES

We evaluate AdaExplore (Du et al., 2026b) and AccelOpt (Zhang et al., 2025) in the same external harness used for KLINEAGE. This harness standardizes the parts that determine the reported score—input shapes, correctness tests, profiling, vendor references, and cost accounting—while preserving each baseline's own search policy and memory representation. In other words, we do not replace either baseline with a KLINEAGE-style memory or planner; each method still decides what to submit using its original in-context state.

All three methods use Claude Opus 4.6 as the online materialization backbone, the per-workload $10 optimization-cost budget, the same compile/correctness/profile gate, and the same vendor reference

Platform	Kernel	Gen. (ms)	KLINEAGE (ms)	Slowdown
SM90	GEMM	0.4070	0.3130	1.30×
SM90	Conv2d	0.2273	0.0157	14.48×
SM90	FMHA	27.7556	3.5183	7.89×
SM90	GDN	25.6522	0.2439	105.18×
SM90	Top-K	0.0211	0.0143	1.48×
SM120	GEMM	0.4781	0.3681	1.30×
SM120	Conv2d	0.1369	0.0235	5.83×
SM120	FMHA	11.8063	3.2151	3.67×
SM120	GDN	23.7612	0.6348	37.43×
SM120	Top-K	0.0147	0.0131	1.12×

Table 7: Conditioned skills outperform direct expert-source prompting on all ten CUDA workloads under the same $10 budget.

per platform (§ 4.1, Table 5). We use a dollar budget rather than a fixed submission count because the methods differ in session structure and cache reuse: KLINEAGE keeps one prefix-cached chat session per workload, whereas the baselines either restart sessions per submission or use looser caching. The goal is therefore to compare optimization usefulness under the same online cost, not to measure each method's asymptotic ceiling under an unconstrained search horizon.

AdaExplore. For AdaExplore, we use the released failure-rule memory, failure-tag extraction, MCTS-style exploration, action vocabulary, and per-step prompt templates. The failure-rule memory is reset at the start of each workload, matching the published per-task setting and preventing rules from one workload from leaking into another. The only harness-level changes are that submissions are routed through our evaluator and budget meter, so that compile status, correctness, latency, and LLM cost are measured consistently with KLINEAGE and AccelOpt. We do not modify AdaExplore's search policy, rule-extraction prompt, or memory schema.

Non-kernel wrapper outputs. A practical issue arises in some AdaExplore runs: several submissions reach the evaluator as wrapper programs rather than executable generated kernels. The agent emits a Triton kernel definition, but the surrounding `ModelNew` catches the kernel-call failure and returns the PyTorch reference. Such submissions can pass correctness, and may even appear fast, but the measured program is then the fallback reference path rather than the generated kernel. We verified this by replacing the fallback return path with a `raise`; under the same workload interface, these submissions no longer pass.

We therefore exclude fallback-wrapper scores from the generated-kernel comparison. To avoid under-crediting AdaExplore, we additionally perform a post-hoc audit for SM120 Conv2d by wiring a thin adapter that directly calls the underlying generated Conv2d kernels with the case-side (x, w) tensors. This audit recovers two real kernels, but they are much slower than cuDNN (best about $0.41\times$ vs. torch and $0.33\times$ vs. cuDNN near the end of the run), and the adapter itself is not produced by AdaExplore within its workflow. We mark this number with † in Table 6: it credits the real audited kernel rather than the wrapper's fallback score. We do not otherwise repair AdaExplore submissions or introduce task-specific adapters, since such harness-level repair is outside the published AdaExplore loop and the shared budget.

AccelOpt. For AccelOpt, we use the released iterative refine-with-summary loop and its per-step memory-summary prompt. The memory state is initialized empty for each workload, matching the published per-task setting. As with AdaExplore, submissions are routed through our evaluator and budget meter, but the baseline's memory remains an LLM-summarized rolling state over its own forward submissions. We do not introduce SkillCards, lineage retrieval, expert-derived transitions, or any KLINEAGE-specific memory schema into AccelOpt.

Shared scoring protocol. All three methods share the online backbone, the per-workload cost budget, the evaluator (`triton.testing.do_bench` with `warmup=25` and `rep=100`, both in milliseconds), the correctness check (three random seeds), the speedup definition ($T_{\text{ref}}/T_{\text{gen}}$), and the run count (5 runs per workload, reported as success-only geometric mean). The vendor reference is fixed

per platform and workload: cuBLAS for GEMM, cuDNN for Conv2d, FlashInfer for FMHA and Top-K, and FlashInfer or FLA for GDN (Table 5). Thus, the comparison keeps the model, evaluator, budget, and references fixed; what varies is the optimization memory each method carries between submissions.

Scope of the comparison. This comparison evaluates fixed-cost online optimization, not unlimited-horizon search. Search-heavy methods may improve with additional budget, and KLINEAGE could also use additional budget for repair and local exploration. Our goal in the main comparison is to ask how useful each memory representation is under the same online materialization cost and validation gate. Per-workload absolute latencies for both baselines are listed alongside KLINEAGE in Table 6.

What an AdaExplore trajectory looks like at the step level. Figure 6 shows the per-step trajectory for two of the AdaExplore SM120 cases. Panel (a) is GEMM, where 21 valid kernels are produced over the $10 budget at speedups ranging from $0.57\times$ to $0.88\times$, plus 8 failed steps; the running-best reaches $0.88\times$ at $1.1 and stays flat afterwards, and most subsequent valid kernels regress below the running-best rather than improve it. Panel (b) is Conv2d, where most steps fail outright (16 failed) or emit non-kernel wrappers that compute the right answer through PyTorch (6 flagged in the raw trajectory). A thin post-hoc adapter can benchmark two underlying kernels as real Conv2d kernels, reaching about $0.41\times$ near the end of the run, while the remaining wrappers are excluded. The two panels jointly motivate why we report running-best over a wrapper-filtered step set rather than picking the wrapper's reported speedup at face value: the latter would have credited the Conv2d wrappers with about $0.96\times$ on this case.

Green circles denote valid submissions, green diamonds denote adapter-audited kernels, red crosses denote excluded fallback wrappers, and grey triangles denote failed steps. The blue line tracks the best wrapper-filtered speedup, including the audited Conv2d kernels.

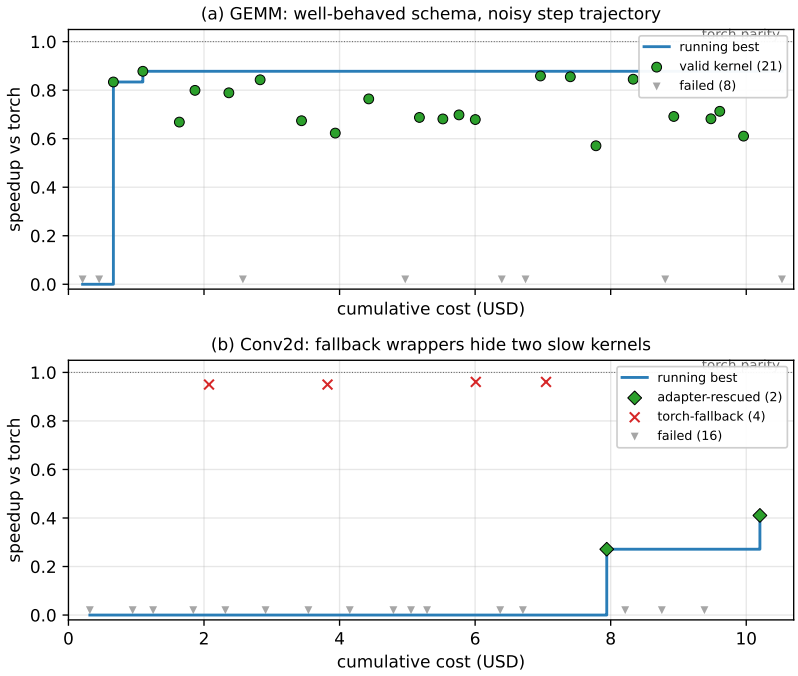


Figure 6: AdaExplore's SM120 trajectories show repeated regressions on GEMM and frequent failures or fallback wrappers on Conv2d.

E CUDA AND ASCEND TRAJECTORIES AGAINST TOKENS AND COST

Figure 4 shows the CUDA and Ascend transfer trajectories against wall-clock time in the top and bottom panels, respectively. Each operator is normalized by the best kernel reached without KLINEAGE, except Ascend Sparse Attention, which uses the eager reference because the run without KLINEAGE produced no valid kernel. To complement the time-based view, Figure 7 and Figure 8 plot the same runs against cumulative API tokens and monetary cost, respectively, with CUDA above Ascend in each figure. Token counts include input and output tokens, including cache hits; normalization and shading follow the corresponding wall-clock plots. These axes better capture agent-side optimization effort when a session is idle, interrupted, or resumed, where wall-clock time may not faithfully reflect the amount of model interaction.

For the cost analysis, we compute API cost using the DeepSeek V4.1-Flash peak rates: $0.30 per million cache-miss input tokens, $0.006 per million cache-hit input tokens, and $1.20 per million output tokens, charged per API call. This provides an upper bound on the actual cost of these runs, since only the Sparse Attention pair was executed during the peak-pricing window.

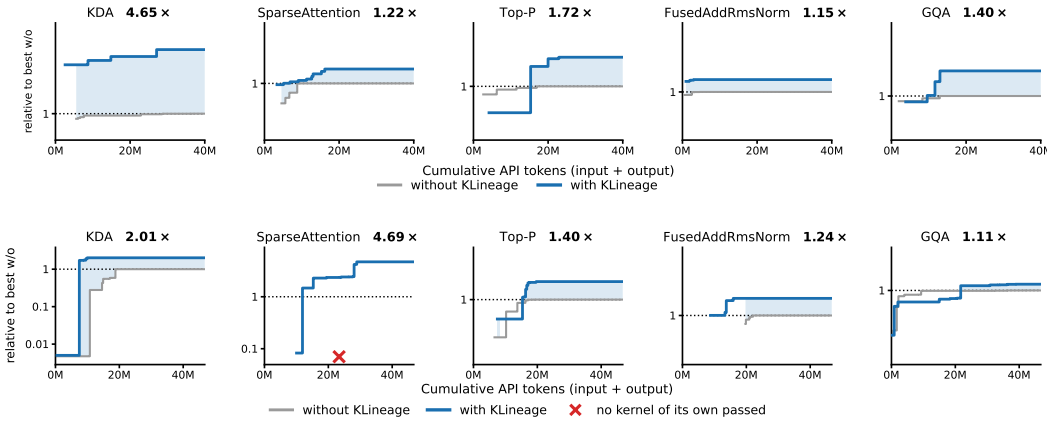


Figure 7: Token-based trajectories show the transfer gains from conditioned skills on both CUDA and Ascend.

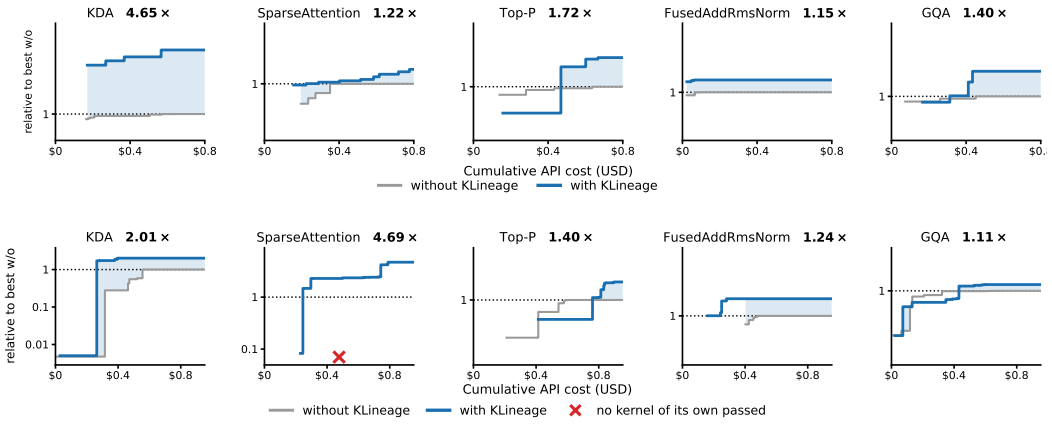


Figure 8: Transfer trajectories against API cost show faster final kernels with conditioned skills on both platforms.

F AscendC Technique Matrix

Table 8 audits the final submission of each setting on the five Ascend transfer targets for the optimization mechanisms visible in its source. A mechanism is counted when its implementation is present in the selected bundle; the audit describes code, not an isolated performance effect, and it does not establish that any one mechanism caused a speedup. The setting with KLineage contains everything the setting without it found on every target: no mechanism is lost, and fifteen are added. Blue checkmarks identify optimizations present only in the KLineage setting.

	KDA		**Top-P**		**GQA**		**Sparse Attn.**		**Add-RMSNorm**	
Optimization	w/o	w/	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Compute mapping										
AIV vector arithmetic and reductions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cube matmul for attention products	×	×	×	×	✓	✓	×	✓	×	×
Cube products split from Vector softmax	×	×	×	×	✓	✓	×	✓	×	×
Core-parallel row / token / head-slice split	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Work-aware launch partition	×	×	×	×	×	✓	×	×	×	✓
Memory and pipeline										
GM-to-UB staging (DataCopy, LocalTensor)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Two-slot / ping-pong staging	✓	✓	✓	✓	✓	✓	×	✓	✓	✓
Prefetch ahead of compute	×	✓	×	×	×	×	×	✓	✓	✓
Pack or transpose for downstream access	×	✓	×	×	✓	✓	×	×	×	×
Event-ordered overlap across streams	×	×	×	×	×	✓	×	✓	×	×
Workspace reuse across phases	×	×	×	×	×	✓	×	✓	×	×
L2 bypass for streaming writes	×	×	×	×	×	×	×	×	×	✓
Explicit DMA / Vector / Cube synchronization	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Numerics and data semantics										
FP32 vector intermediates	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Row-max subtraction before exp	×	×	×	×	×	✓	✓	✓	×	×
Compensated FP32 sum in cutoff search	×	×	✓	✓	×	×	×	×	×	×
Invalid-index / causal / tail masking	×	×	✓	✓	✓	✓	✓	✓	×	×
GatherMask compaction of threshold band	×	×	×	✓	×	×	×	×	×	×
Operator-specific reuse										
Recurrent state kept in local memory	✓	✓	×	×	×	×	×	×	×	×
Fused row sum reused for both outputs	×	×	×	×	×	×	×	×	✓	✓
Total	7	**9**	8	**9**	10	**14**	7	**13**	8	**10**

Table 8: KLineage preserves every optimization found by expert-source prompting and adds further optimizations on all five Ascend targets.

G Case Study: Gated Delta Net

We use GDN to inspect what a lineage records beyond a final latency number. The operator decomposes into three FlashQLA sub-kernels (`cumsum`, `fused_fwd`, `kkt_solve`) with different optimization profiles. Figure 9 shows the deoptimization chains; its backward edges remove optimizations, and KLineage stores the inverse directions as forward skill candidates. The admitted SkillCards are listed with their verification evidence in Table 9 (App. H).

The point of the trace is not only that these optimizations exist, but that the intermediate code states make them learnable. A memory item that is too concrete copies one expert surface and is hard to reuse; one that is too abstract names an optimization but gives the agent no actionable edit. KLineage first validates each concrete before/after transition by roundtrip execution, then aggregates validated transitions into intents with anchors, carriers and preconditions. This is why the resulting skill can remain actionable on code while still transferring across cases and languages.

Three sub-kernels, three dominant skills. The largest drop in each trace isolates a different reusable intent. For `cumsum`, the dominant skill is a *shared-memory transpose*: the scan axis is strided in input memory, and transposing each chunk into [head, seq] before scanning cuts $0.034 \rightarrow 0.012$ ms,

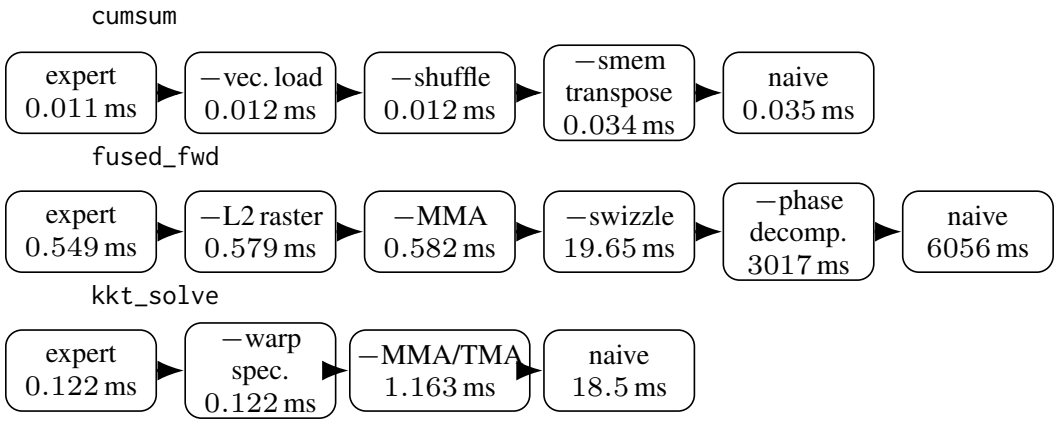


Figure 9: GDN deoptimization identifies shared-memory transpose, phase decomposition, and MMA/TMA layout as dominant optimizations in its three sub-kernels.

while vectorized access, warp-shuffle scan and intra-block parallelism contribute little at this shape. For `fused_fwd`, it is `gemm_phase_decomposition`, which restructures per-token $O(C^3D)$ work into cooperative $O(C^2(D{+}VT))$ GEMM phases and accounts for the largest single improvement in the chain; the remaining MMA/swizzle skill matters only once the phase structure is in place. For `kkt_solve`, cooperative threading pays before the MMA/TMA intrinsic skill becomes profitable.

The *conditional dependency* is the point: on `cumsum` the layout transpose makes the on-chip scan worth doing at all; on `fused_fwd` phase decomposition is a precondition for the tensor-core swizzle to matter; on `kkt_solve` the cooperative-threading layout has to be in place before the MMA/TMA intrinsics fit. Each of these is a precondition a card records and a label does not.

H GDN SkillCards (Case Study)

Table 9 lists the admitted SkillCards from the GDN case study (Appendix G). Each row is one admitted skill; the verification-evidence column is the speedup recorded in the held-out roundtrip trial that admitted the card.

Skill	**Source sub-kernel**	**Verification evidence**
`vectorized_global_load_store`	`cumsum`	$3.9\times$ over naive
`warp_shuffle_scan`	`cumsum`	$2.7\times$ over naive
`smem_transpose`	`cumsum`	$2.9\times$ over naive
`multi_threaded_block_parallelism`	`cumsum`	$1.1\times$ over naive
`mma_ldmatrix_stmatrix_swizzle`	`fused_fwd`	$34\times$ over scalar
`gemm_phase_decomposition`	`fused_fwd`	$194\times$ over naive
`multi_threaded_block_parallelism`	`fused_fwd`	$1.6\times$ over naive
`mma_ldmatrix_stmatrix_swizzle_tma`	`kkt_solve`	$16\times$ over scalar
`multi_threaded_block_parallelism`	`kkt_solve`	$16\times$ over naive

Table 9: GDN's three sub-kernels yield verified skills for memory layout, parallelism, and matrix computation.