EXPERIENCE IS NOT ENOUGH: KNOWLEDGE-GROUNDED EVOLUTIONARY SEARCH FOR ASCEND C KERNEL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents have become capable kernel engineers on mainstream GPU stacks, but they struggle on emerging accelerators such as Ascend NPUs, whose programming model is barely represented in pretraining data. Recent NPU agents address this gap largely through experience: they accumulate memory from their own generation and optimization trials and reuse it on later tasks. We argue that experience alone is not enough for kernel generation. An agent can only remember what it has tried, and it rarely tries a mechanism it does not know. As a result, such memory offers little help when a new operator calls for a different mechanism, and even familiar operators raise unseen situations as optimization proceeds. We present ASCENDEVOLVE, which grounds every step of an evolutionary kernel search in a large external knowledge base built from platform documentation, toolkit headers, and operator repositories. Because this corpus far exceeds any context window, ASCENDEVOLVE indexes it and routes each agent to a small, task-specific slice of evidence instead of the corpus itself. The retrieved knowledge seeds diverse implementation strategies, guides repair when a candidate fails, and constrains how strong candidates are recombined. On 53 Ascend C operators from CANN-Bench spanning four difficulty levels, ASCENDEVOLVE achieves an overall score of 71.39, improving over Reflexion and OpenEvolve by 11.6% and 17.2%.

1 INTRODUCTION

Trial-and-error agents learn from the programs they actually generate and evaluate. As a result, a memory built from past trials is naturally limited by what the agent has explored: it can reuse and refine discovered mechanisms, but does not guarantee access to mechanisms that have never appeared in its search. This motivates knowledge beyond self-generated experience.

Recent work has demonstrated substantial progress in LLM-based kernel generation and optimization for GPUs. On CUDA and related programming stacks, iterative workflows that combine generation with compilation, execution, and profiling feedback can progressively improve both correctness and performance (Ouyang et al., 2025; Zhang et al., 2025; Wei et al., 2025; Baronio et al., 2025). These results establish iterative execution feedback as an effective ingredient for kernel optimization on mature GPU ecosystems. The problem becomes considerably more challenging on specialized accelerator programming stacks for which general-purpose LLMs have demonstrated substantially lower capability—a difficulty well documented for domain-specific programming languages more broadly (Joel et al., 2025). On Huawei's Ascend NPU (Liao et al., 2021), recent work reports that general-purpose LLMs achieve very low success rates on complex kernel-generation tasks without domain-specific adaptation (Cao et al., 2026).

Recent systems have addressed this difficulty in part by accumulating reusable experience from previous generation and optimization trajectories. Value-driven memory substantially improves functional correctness over zero-shot prompting (Zheng et al., 2026), and experience mined from optimization trajectories yields consistent speedups on existing operators (Wu et al., 2026; Zhang et al., 2026). These results provide strong evidence that reusable experience can substantially improve both feasibility and optimization efficiency. However, these systems necessarily construct their reusable evidence from trajectories that the agent has already proposed and evaluated. This suggests

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.

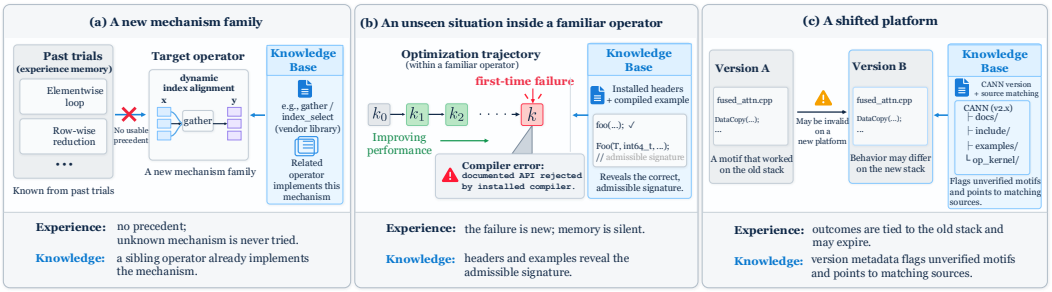


Figure 1: Three situations in which an agent's own trials offer no usable precedent while primary platform sources do: (a) a mechanism family absent from past trials, (b) a first-time failure midway through optimization, and (c) a motif carried across toolkit versions. Illustrative, not measured.

a complementary limitation: the usefulness of experience depends not only on how much is retained, but also on what the agent has managed to explore.

> We refer to this coverage constraint as *experience closure*: the reusable evidence in a self-generated memory is derived from trajectories the agent has actually proposed and evaluated. As a result, such memory may provide limited guidance for mechanisms that have not yet appeared in those trajectories—particularly on platforms where the model's familiarity with the programming stack is limited to begin with.

Figure 1 shows three settings where self-generated experience provides limited guidance:

(i) **Coverage gap.** An operator may require a mechanism family that no previous trajectory has explored; since such a mechanism is rarely proposed, it is also unlikely to be recorded.

(ii) **Repair gap.** A familiar operator may encounter, midway through optimization, a constraint that has not appeared before—compiler diagnostics often identify the violated constraint without directly providing a valid alternative implementation.

(iii) **Validity gap.** Experience collected under one software or hardware configuration may not transfer unchanged to another.

Self-generated experience is not the only source of implementation evidence. Accelerator ecosystems expose additional information through programming guides, API definitions, installed headers, reference implementations, and vendor-maintained operator libraries (NVIDIA Corporation, 2017; Zhao et al., 2021). These sources can provide mechanism and constraint information before the agent has encountered the corresponding trajectory itself. We therefore view primary platform knowledge and execution experience as complementary: platform knowledge can broaden the set of mechanisms the agent considers, while execution experience provides empirical evidence about which candidates are correct and efficient on the target device (Wen et al., 2026). Our goal is not to replace trial-and-error with retrieval, but to inform it.

We present AscendEvolve, a system that grounds every stage of an evolutionary kernel search in primary platform sources. The design follows three principles:

① **Hierarchical and version-aware.** Platform information is organized hierarchically and with version awareness: a shared, versioned corpus of documentation, installed headers, and vendor operator libraries is distilled into per-operator evidence before search begins, avoiding repeated processing of a large heterogeneous corpus.

② **Demand-driven retrieval.** Rather than placing the corpus in context, AscendEvolve routes each concrete implementation or optimization question to a small set of relevant sources.

③ **Bidirectional interaction.** Retrieved platform evidence and execution experience interact bidirectionally: retrieved mechanisms seed candidate implementations, each trial records an empirical verdict on the knowledge it was grounded in, and outcomes influence subsequent retrieval.

Because retrieved evidence may suggest multiple plausible implementation mechanisms rather than a single canonical solution, ASCENDEVOLVE maintains several mechanism-distinct implementation lines and progressively selects and recombines promising design choices through evolutionary search.

On 53 Ascend C operators spanning four difficulty levels, ASCENDEVOLVE achieves an overall score of **71.39**, improving over Reflexion (Shinn et al., 2023) and OpenEvolve (Sharma, 2025) by **11.6%** and **17.2%**, respectively, with the largest gains on the operator categories where prior agents have the lowest correctness coverage. On L4 operators, ASCENDEVOLVE reaches **35.0%** correctness coverage compared with 4.38% for Reflexion and 1.25% for OpenEvolve. Removing the knowledge base nearly halves the L3 score, showing how platform knowledge helps generate correct kernels. Our contributions are as follows:

① **Experience closure.** We identify experience closure—a coverage limitation of experience-driven kernel agents on platforms outside the model's familiarity—and characterize three settings where it is most pronounced.

② **Method.** We propose ASCENDEVOLVE, which couples primary platform knowledge with procedural experience through hierarchical, demand-driven retrieval and bidirectional feedback within an evolutionary search that maintains mechanism-diverse implementation lines.

③ **Evaluation.** We evaluate on 53 Ascend C operators from CANN-Bench (Gao et al., 2026) and show that knowledge grounding yields substantial gains, particularly on hard operators where experience-only methods have very low correctness coverage. We isolate the contribution of the knowledge base through ablation.

2 PROBLEM FORMULATION

Notation. Let o denote an operator specified by an executable reference implementation and a set of test cases $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$, where each $(\mathbf{x}_i, \mathbf{y}_i)$ is an input–output pair. A candidate kernel x is a compilable Ascend C program that implements o. We write $C(x) \in [0, 1]$ for the fraction of test cases that x passes under a fixed numerical-tolerance criterion, and $L(x)$ for the wall-clock latency of x measured on the target device under a fixed profiling protocol. $\mathcal{K}_G$ is a shared, static knowledge base of platform documentation, vendor libraries and machine-observed facts. $\mathcal{K}_o$ is a per-operator knowledge base that fixes the task contract and links it to $\mathcal{K}_G$. $\mathcal{E}$ is a dynamic experience store that accumulates trial records during search. $R(q \mid \mathcal{K}_o, \mathcal{K}_G, \mathcal{E})$ is a retrieval function that, given a concrete question q, returns a small *evidence pack* $\mathcal{C}$ from the knowledge bases, filtered by experience.

Problem setup. The goal is to produce a kernel $x^\star$ that is correct on all test cases and as fast as possible on the target device. We compare candidates with a correctness-first fitness

$$F(x) = \big(C(x),\, -L(x)\big), \tag{1}$$

ordered lexicographically: latency is compared only between candidates of equal correctness. The search is conducted by an LLM-based agent that, at each step t, conditions on an evidence pack

$$\mathcal{C}_t = R\big(q_t \mid \mathcal{K}_o, \mathcal{K}_G, \mathcal{E}_{<t}\big) \tag{2}$$

to propose, evaluate and refine candidates. The agent must maximize F within T LLM calls.

3 METHOD

3.1 OVERVIEW

Existing agents search the space of candidate kernels with an LLM policy conditioned on experience $\mathcal{E}_{<t}$ gathered from earlier trials. On a platform the LLM barely knows, this conditioning is not enough: the policy can only revisit and recombine what it has already tried, and a mechanism that lies outside its prior is never sampled in the first place. ASCENDEVOLVE adds a second source of information that does not depend on the agent's own behavior: every decision is conditioned on the evidence pack $\mathcal{C}_t$ (Eq. 2), which retrieves version-checked platform facts and expert implementation patterns from the knowledge bases $\mathcal{K}_G$ and $\mathcal{K}_o$, filtered by accumulated experience. The rest of this section describes how the knowledge bases are built (§3.2), how R retrieves from a corpus far larger

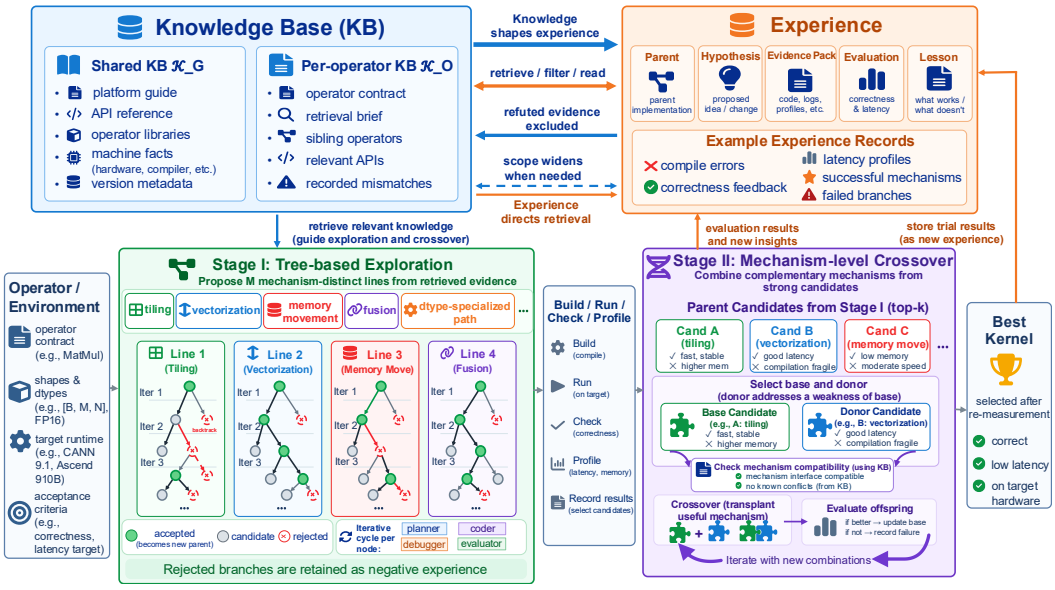


Figure 2: Overview of AscendEvolve. Retrieval draws a task-specific evidence pack from $\mathcal{K}_G$ and $\mathcal{K}_o$. Stage I explores M mechanism-distinct lines as trees, keeping rejected branches as negative experience; Stage II transplants a donor mechanism into the best base after a compatibility check. Each trial writes a record that steers later retrieval.

than any context window (§3.3), how knowledge and experience interact (§3.4), and how both drive a two-stage evolutionary search (§3.5). Figure 2 gives an overview.

3.2 Knowledge Base Construction

Writing a kernel requires answers to three questions: what the operator must compute, what the platform permits, and how experts implement similar operators on it. The first is specific to each task. The other two are shared across tasks. We organize knowledge along this line.

Shared knowledge base. $\mathcal{K}_G$ holds platform knowledge and is built from primary sources, not from hand-written summaries. It contains the official programming guide, API reference and compilable examples; the vendor's open-source operator libraries, which expose production implementations across operator families; and facts recorded from the target machine, such as installed headers and the compiler in use. Each source $d \in \mathcal{K}_G$ carries metadata $m(d)$ that records its toolkit version, hardware scope and trust level; this metadata induces an authority order $\succ$ used during retrieval (§3.3). $\mathcal{K}_G$ is immutable during search. Appendix A lists the sources and their statistics.

Per-operator knowledge base. $\mathcal{K}_o$ fixes the task and tells the agent where to look. It has two parts. The *contract* freezes the operator semantics, the executable reference, the shape and dtype inventory, the target environment and the acceptance criteria; it is the sole authority on what the kernel must compute. The *retrieval brief* links the operator to $\mathcal{K}_G$: it names the operator family, the closest sibling implementations and the relevant API entries, and it records any disagreement found between these sources and the contract. $\mathcal{K}_o$ is produced fully automatically by a preparation pass that runs once per operator before search begins, with no human editing, and it stores references into $\mathcal{K}_G$, never copies. In one of our tasks, for example, the pass found that a closely related vendor operator places a numerical-stability constant differently from the executable reference. The brief records this mismatch, so later agents borrow the vendor's tiling without its arithmetic.

3.3 Retrieval from a Large Knowledge Base

$\mathcal{K}_G$ exceeds the context window by orders of magnitude, and most of it is irrelevant to any single decision. Two properties of kernel code make standard dense retrieval (Lewis et al., 2021) a poor fit. Queries are frequently exact symbols, and a nearly identical passage from a mismatched toolkit

version is actively harmful, since it leads to code that looks right and does not compile. We therefore design R as a hierarchical, demand-driven procedure with three steps.

① **Route.** Retrieval is triggered by one concrete question q_t, such as how a given API handles tail elements, and never by the task as a whole. The retrieval brief in $\mathcal{K}_o$, backed by a global index over $\mathcal{K}_G$, maps q_t to a short list of candidate sources. This step reduces the search space from the whole corpus to a handful of files.

② **Filter.** Candidates are checked against the target environment using $m(d)$. Sources of a mismatched version are either dropped or kept as explicitly labeled leads. When two candidates disagree, the authority order $\succ$ breaks the tie: facts observed on the target machine outrank official sources of the matching version, which outrank other versions and informal material. The disagreement itself is retained in the pack so that the agent is aware of it.

③ **Read.** The agent reads the surviving sources directly, locating the relevant region by exact symbol or path. Implementation patterns are returned with their surrounding code, because a tiling loop is meaningless without the declarations it relies on.

If a question remains unresolved after these steps, the scope widens one level along the index. Each item in $\mathcal{C}_t$ keeps its source identity, so that decisions in the final kernel are traceable to evidence. Appendix B gives the full procedure and an example pack.

3.4 Interaction between Knowledge and Experience

Knowledge and experience answer different questions. Knowledge states what is possible and legal; experience states what worked, on this operator and this device. Neither is sufficient alone. Knowledge without experience cannot say which of several legal mechanisms is fastest, and experience without knowledge is confined to the mechanisms the LLM already knows. AscendEvolve couples the two in both directions.

Knowledge shapes experience. Every trial starts from a hypothesis h_t that is grounded in an evidence pack, and the record written after evaluation keeps this link:

$$e_t = \big(p_t,\ h_t,\ \mathcal{C}_t,\ F(x_t),\ \ell_t\big), \tag{3}$$

where p_t is the parent candidate and ℓ_t is a short lesson. An experience record is thus more than a score attached to a code change. It is an empirical verdict on a piece of knowledge in a specific context, such as "the vectorized path from a sibling operator is valid here and reduces latency on large shapes" or "this API is documented but rejected by the installed compiler." Because trials are seeded by retrieved mechanisms, the experience the agent accumulates covers regions of the design space that it would not have reached from its prior.

Experience directs retrieval. The retrieval function in Eq. 2 is conditioned on $\mathcal{E}_{<t}$: the outcome of the last trial determines the next question (a compilation error routes to headers, a numerical mismatch to the reference and dtype constraints, a regression to profiles); evidence refuted for this operator is excluded from later packs; and when accumulated failures show that the current evidence is insufficient, retrieval widens its scope. In this way experience handles situations that arise only during optimization and that no static brief could anticipate. When knowledge and measurement disagree, the on-device result overrides documentation for the task at hand; the verdict is written to the task-local experience, never to $\mathcal{K}_G$, keeping platform facts free of candidate-specific outcomes.

3.5 Knowledge-Grounded Evolutionary Search

The search has two stages (Algorithm 1). Stage I explores mechanism-distinct implementation lines, and Stage II recombines the best of them.

Stage I: tree exploration. The planner first proposes M implementation lines that differ in mechanism, for example in tiling scheme, vectorization strategy or dtype-specialized path. Diversity comes from the different mechanisms found in the retrieved evidence, not from prompt perturbation. Each line then runs for K iterations. In one iteration, following the role decomposition common to iterative code agents (Madaan et al., 2023; Shinn et al., 2023), a planner selects an accepted parent and states

Algorithm 1 ASCENDEVOLVE

Require: operator o, shared KB $\mathcal{K}_G$, lines M, iterations K, crossover budget B

$\mathcal{K}_o \leftarrow \textsc{Prepare}(o, \mathcal{K}_G)$; $\mathcal{E} \leftarrow \emptyset$
propose M mechanism-distinct lines from $R(\cdot \mid \mathcal{K}_o, \mathcal{K}_G, \mathcal{E})$
{Stage I: tree exploration}
for each line $i = 1, \ldots, M$ **and** $k = 1, \ldots, K$ **do**
  select an accepted parent p; form question q from $\mathcal{E}$
  $\mathcal{C} \leftarrow R(q \mid \mathcal{K}_o, \mathcal{K}_G, \mathcal{E})$ ▷ route, filter, read
  plan hypothesis h; generate child x; repair if needed
  evaluate $F(x)$; $\mathcal{E} \leftarrow \mathcal{E} \cup \{(p, h, \mathcal{C}, F(x), \ell)\}$
  accept x as new parent if F improves, else backtrack
end for
$x^\star \leftarrow$ best candidate over all lines
{Stage II: mechanism-level crossover}
for $b = 1, \ldots, B$ **do**
  choose donor x' whose mechanism addresses a weakness of $x^\star$ ▷ from $\mathcal{E}$
  if mechanisms are incompatible under $\mathcal{K}_o, \mathcal{K}_G$ **then continue**
  $x'' \leftarrow \textsc{Crossover}(x^\star, x')$; evaluate; update $\mathcal{E}$
  if $F(x'') \succ F(x^\star)$ **then** $x^\star \leftarrow x''$
end for
return $x^\star$ after re-measurement

one falsifiable hypothesis with its expected effect; a coder realizes it as a child candidate; a debugger applies bounded repairs that preserve the hypothesis; and an evaluator runs the build, correctness and latency gates in order. If the child improves F, it becomes the next parent. Otherwise the line backtracks to an earlier accepted node and the planner forms a different hypothesis. The history of a line is therefore a tree. Rejected branches are kept as negative experience (Andrychowicz et al., 2017), which prevents a harmful change from contaminating later candidates and records which assumptions are already falsified.

Stage II: mechanism-level crossover. Stage I returns the best correct candidate of every line together with a summary of its mechanism, its per-case latency and its known weaknesses. Stage II treats these candidates as a population. The best candidate under F serves as the base, and a donor is selected only if its mechanism addresses a specific weakness of the base, such as a slow shape regime or an inefficient tail path. Here the two information sources play complementary roles again: experience identifies *which* mechanism wins in which regime, and the knowledge base verifies *whether* the two mechanisms can coexist, in terms of tiling structure, dtype behavior and platform constraints. Crossover transplants the donor mechanism into the base instead of concatenating implementations. An offspring that improves F becomes the new base, and a failed combination is recorded so that the pairing is not tried again. The search ends when the budget B is spent or no compatible hypothesis remains, and the finalists are re-measured under the same protocol before the best kernel is returned.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Dataset. We evaluate on CANN-Bench (Gao et al., 2026), using all 53 public operators across four difficulty levels, with 20 test cases per operator (1,060 cases in total). The benchmark progresses from relatively simple elementwise and Vector-Core operators (L1–L2) to increasingly complex fused Vector–Cube workloads (L3–L4), allowing us to examine how the methods behave as implementation requirements become more specialized.

Baselines. We compare against two test-time search paradigms that form a natural progression. **Reflexion** (Shinn et al., 2023) represents sequential experience-based refinement using execution feedback. **OpenEvolve** (Sharma, 2025) represents population-based evolutionary search. All methods use GPT-5.5 with medium reasoning effort as the backbone, with Codex as Reflexion's agent harness, and the same benchmark inputs and evaluator throughout.

Table 1: Main results on CANN-Bench. Build is reported over operators; runtime, correctness and performance coverage are reported over test cases. Best score per level in **bold**.

Level	Method	Build (%)	Runtime cases (%)	Correct cases (%)	Perf. cases (%)	Score ↑
L1 (8)	Reflexion	100.00	100.00	96.25	96.25	109.58
	OpenEvolve	100.00	100.00	100.00	100.00	100.37
	AscendEvolve	100.00	100.00	95.63	95.63	**111.18**
L2 (16)	Reflexion	100.00	100.00	99.06	91.56	81.38
	OpenEvolve	100.00	100.00	77.50	65.94	71.27
	AscendEvolve	100.00	100.00	85.63	78.75	**87.84**
L3 (21)	Reflexion	100.00	85.48	61.19	41.90	53.56
	OpenEvolve	85.71	79.76	66.43	47.86	**57.84**
	AscendEvolve	100.00	90.48	59.76	48.33	56.94
L4 (8)	Reflexion	87.50	48.13	4.38	0.00	10.94
	OpenEvolve	50.00	41.25	1.25	0.63	8.78
	AscendEvolve	87.50	86.88	35.00	35.00	**36.63**
Overall (53)	Reflexion	98.11	86.42	69.34	58.77	63.98
	OpenEvolve	86.79	83.11	65.00	54.06	60.91
	AscendEvolve	98.11	94.25	69.25	62.64	**71.39**

Evaluation. We follow the staged evaluation logic of CANN-Bench (Gao et al., 2026) but recalibrate its performance component for our target NPU. *Build* is a binary result for each operator's selected kernel. *Runtime* measures whether each test case executes without failure, and *correctness* checks its output against the reference implementation. A case receives *performance* credit only if it passes correctness and both the candidate and `torch_npu` have valid positive latency measurements. Build coverage is operator-level; all other coverage is case-level.

Since the published CANN-Bench performance anchors are hardware-specific, we use the latency of the corresponding `torch_npu` implementation as a common baseline. Let b_o indicate whether operator o builds, and let $r_{o,i}$ and $c_{o,i}$ indicate whether case i runs and passes correctness, respectively. We define $R_{o,i} = b_o r_{o,i}$ and $A_{o,i} = R_{o,i} c_{o,i}$, so a build failure gives all 20 cases zero credit. The first score component is $\sum_{i=1}^{20} R_{o,i}$: each case that runs after a successful build earns one point. The score for operator o is

$$S_o = 100 \left[\frac{1}{20} \sum_{i=1}^{20} \left(0.2 R_{o,i} + A_{o,i} \left(0.3 + 0.5 P_{o,i} \right) \right) \right], \tag{4}$$

where $P_{o,i} = \mathrm{clip}\left(\sqrt{T_{\mathrm{torch},o,i} / T_{\mathrm{cand},o,i}},\ 0.5,\ 4 \right)$ when both latencies are valid and positive, and $P_{o,i} = 0$ otherwise. Thus, 100 corresponds to full correctness with `torch_npu`-level performance when all cases have valid latencies, while faster correct kernels may score above 100. We macro-average operator scores so each contributes equally.

4.2 Main Results

AscendEvolve achieves the highest overall score (71.39), improving over Reflexion and OpenEvolve by 11.6% and 17.2%, respectively (Table 1). The source of the gain changes with difficulty. On L1–L2, where executability is already high, improvements come from stronger optimization rather than broader coverage. On L4, the bottleneck shifts to implementation feasibility: AscendEvolve reaches 35.0% correctness coverage, compared with 4.38% for Reflexion and 1.25% for OpenEvolve. L3 is more mixed: OpenEvolve obtains a slightly higher final score, while AscendEvolve achieves substantially higher build and runtime coverage.

Observation 1. External knowledge becomes increasingly important when the challenge is no longer merely optimizing a valid implementation, but **finding a valid implementation mechanism**.

Table 2: Overall score decomposition. Successful runs contribute $\sum_i R_{o,i}$ points per operator, one per case that runs after a successful build.

Method	Successful runs (pts)	Correct. score	Perf. score	Score ↑	Build (%)	Runtime cases (%)	Correct cases (%)	Perf. cases (%)
AscendEvolve	**18.85**	20.77	**31.77**	**71.39**	98.11	**94.25**	69.25	**62.64**
Reflexion	17.28	**20.80**	25.90	63.98	98.11	86.42	**69.34**	58.77
OpenEvolve	16.62	19.50	24.79	60.91	86.79	83.11	65.00	54.06

Table 3: Level 3 knowledge-base ablation.

Metric	No KB	With KB	Δ
Build coverage	61.90%	**100.00%**	↑38.10
Runtime-case coverage	57.86%	**90.48%**	↑32.62
Correct-case coverage	40.24%	**59.76%**	↑19.52
Performance-case coverage	17.86%	**48.33%**	↑30.48
Score ↑	29.49	**56.94**	↑27.45

Table 4: Stage-wise synthesis outcomes.

Level	Stage 2 vs. Stage 1 relative gain (%) ↑	Adoption rate (%) ↑
L1	4.02	62.5
L2	3.26	87.5
L3	1.88	25.0

4.3 Where Do the Gains Come From?

The gain over Reflexion comes primarily from optimization rather than additional correctness: the two methods have nearly identical correctness coverage (69.25% vs. 69.34%), while AscendEvolve's performance score increases from 25.90 to 31.77 and accounts for 79.2% of the overall improvement (Table 2). Against OpenEvolve, runtime, correctness and performance all improve.

Observation 2. Knowledge grounding serves **two roles**: it improves the quality of valid kernels and prevents search from being lost in invalid implementations.

4.4 Does External Knowledge Help Beyond Experience?

Removing the knowledge base causes failures throughout the evaluation pipeline: build coverage falls from 100% to 61.9%, and performance coverage from 48.3% to 17.9% (Table 3). The effect is therefore not limited to making correct kernels faster. External knowledge allows substantially more implementations to survive compilation, execution and correctness validation before performance optimization can even begin. As a result, the L3 score nearly doubles from 29.49 to 56.94.

This effect is also visible in strategy diversity (Table 6): the KB increases planned strategies only from 5.25 to 6.75 per operator, but increases full-correct strategies from **0.25** to **1.30**—a more than **5**× increase. Knowledge therefore does not merely make the agent propose more strategies; it makes a much larger fraction of distinct strategies survive platform and correctness constraints.

Observation 3. Knowledge improves ***effective***, not merely nominal, exploration.

4.5 When Does Mechanism-Level Synthesis Help?

Stage 2 improves the final set over Stage 1 by 4.02% on L1, 3.26% on L2 and 1.88% on L3 (Table 4). Adoption is highest on L2 at 87.5%, followed by L1 at 62.5% and L3 at 25.0%. The results indicate that synthesis is most effective when optimization mechanisms can be transferred while preserving validated semantics.

Stage 2 is effective when a donor mechanism matches a recurring input regime and can be integrated without changing the validated computation. Cases outside that regime retain the Stage 1 path and show little change. Crossover can be neutral or harmful when it addresses only a narrow correctness edge case, replaces a parallel implementation with a less efficient path, or introduces no substantive mechanism change. This pattern is consistent with the high adoption rate at L2 and the lower adoption and limited uplift at L3. Appendix F gives representative stage-wise case studies.

Observation 4. Synthesis is most effective when mechanisms can be **composed without altering correctness-sensitive computation**.

Table 5: Archive-visible implementation diversity on 20 common Level 2–3 operators. ASCENDEVOLVE is evaluated using its Stage 1 warmup candidates.

Method	Strategy sig./op ↑	Kernel src./op ↑	Pairwise src. disagr. ↑	Pairwise struct. dist. ↑
ASCENDEVOLVE (S1)	6.75	13.90	95.1%	0.303
Reflexion	1.60	2.70	50.7%	0.081
OpenEvolve	1.30	1.65	19.6%	0.049

Table 6: Correctness-effective planned-strategy diversity during warmup on 20 Level 3 operators.

Metric	With KB	No KB
Planned strategy sig. / op	6.75	5.25
Full-correct-eff. sig. / op	1.30	0.25
Full-correct strategy survival	19.3%	4.8%
Ops with ≥ 1 full-correct	40.0%	20.0%

4.6 DOES KNOWLEDGE ENABLE MORE EFFECTIVE EXPLORATION?

ASCENDEVOLVE explores substantially more mechanism-distinct implementations than Reflexion and OpenEvolve (Table 5). More importantly, this diversity is not merely syntactic: with knowledge grounding, substantially more planned mechanisms survive correctness filtering (Table 6). The KB increases full-correct strategy survival from 4.8% to 19.3%, supporting the role of the knowledge base as a source of viable mechanisms rather than simply additional prompting context.

5 RELATED WORK

LLM-based kernel generation. LLM-based kernel generation has progressed rapidly on CUDA and Triton. KERNELBENCH (Ouyang et al., 2025) established a systematic benchmark for functional correctness and performance, while subsequent work explored multi-agent refinement (Zhang et al., 2025; Wei et al., 2025), reinforcement learning with execution feedback (Baronio et al., 2025; Li et al., 2025; Woo et al., 2025; Zhu et al., 2025), and evolutionary search (Novikov et al., 2025; Liao et al., 2025). Kernel generation is more challenging on specialized accelerator stacks such as Ascend C, where general-purpose LLMs show limited out-of-the-box capability (Cao et al., 2026). Recent work addresses this gap through domain adaptation (Cao et al., 2026), constrained intermediate representations (Wen et al., 2025), external knowledge augmentation (Wen et al., 2026; Qiu et al., 2026), and reusable experience accumulated from previous search trajectories (Zheng et al., 2026; Wu et al., 2026). Our work focuses on the **complementarity** between the latter two directions.

Test-time search and experience for code agents. Test-time scaling improves agent performance through additional refinement, search, memory, or population-based exploration (Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2023; Wang et al., 2023; Novikov et al., 2025). For kernel generation, compilation and execution provide strong verification signals, motivating iterative and evolutionary search guided by previous outcomes (Liao et al., 2025; Zhang et al., 2026; Zheng et al., 2026). However, increasing the number of trials does not necessarily increase the diversity of mechanisms explored when new candidates are generated primarily from previously discovered trajectories. Our method **complements** experience-driven search with retrieved platform knowledge, using external evidence to seed mechanism-diverse implementation lines and execution feedback to evolve them. Appendix C relates our setting to automated kernel optimization before LLMs and to retrieval-augmented generation for code.

6 CONCLUSION AND LIMITATIONS

ASCENDEVOLVE combines trial experience with hierarchical, demand-driven platform knowledge for evolutionary kernel search. On 53 Ascend C operators, it improves over Reflexion and OpenEvolve by 11.6% and 17.2%; ablations show that knowledge expands both the pool of candidates eligible for optimization and the diversity of correct mechanisms.

Limitations. Our gains are not uniform across difficulty. On the hardest operators, knowledge grounding is considerably more effective at making candidates executable than at making them numerically correct: retrieved evidence supplies mechanisms and platform constraints, but not the assurance that a particular composition of them preserves the intended semantics.

REFERENCES

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. *arXiv preprint arXiv:2507.11948*, 2025.

Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *The ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, et al. AscendKernelGen: A systematic study of LLM-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*, 2026.

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

Xue-Jian Gao, Deng Pan, Yueming Su, Jiasheng Li, Bin Du, Fengming Zhu, Chengdi Ma, Junyi Fan, Qichen Liao, Chengqiu Hu, et al. Cann bench: Benchmarking agent generated kernels against real npu and algorithmic limits. *arXiv preprint arXiv:2607.20518*, 2026.

Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages, 2025. URL https://arxiv.org/abs/2410.03981.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks, 2021. URL https://arxiv.org/abs/2005.11401.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. AutoTriton: Automatic triton programming with reinforcement learning in LLMs. *arXiv preprint arXiv:2507.05687*, 2025.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 789–801, 2021. doi: 10.1109/HPCA51647.2021.00071.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. ReACC: A retrieval-augmented code completion framework, 2022. URL https://arxiv.org/abs/2203.07722.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

NVIDIA Corporation. CUTLASS: CUDA templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2017. Accessed: 2026-05-07.

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

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2719–2734, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.232. URL https://aclanthology.org/2021.findings-emnlp.232/.

Junhao Qiu, Zidong Wang, Yansong Sun, Zhitong Ma, Ping Guo, and Qingfu Zhang. Agenticcann: Automated ascend c operator generation via knowledge-augmented agentic evolution. *arXiv preprint arXiv:2607.26661*, 2026.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *The 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. https://github.com/algorithmicsuperintelligence/openevolve, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

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. *arXiv preprint arXiv:2509.07506*, 2025.

Junyi Wen, Ruiyan Zhuang, Yongjia Xu, Pengtu Li, Rui Zou, Hongyi Chen, Chingman Wan, Puxu Yang, Wuhui Chen, and Yanlin Wang. Hawk: Harnessing hardware-aware knowledge for high-performance npu kernel generation. *arXiv preprint arXiv:2607.01590*, 2026.

Zhongzhen Wen, Shudi Shao, Zhong Li, Yu Ge, Tongtong Xu, Yuanyi Lin, and Tian Zhang. Ascendcraft: Automatic ascend npu kernel generation via dsl-guided transcompilation. *arXiv preprint arXiv:2601.22760*, 2025.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. TritonRL: Training LLMs to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025.

Jiehao Wu, Zixiao Huang, Wenhao Li, Chuyun Shen, Junjie Sheng, and Xiangfeng Wang. Ascendoptimizer: Episodic agent for ascend npu operator optimization. *arXiv preprint arXiv:2603.23566*, 2026.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. TLP: A deep learning-based cost model for tensor program tuning. In *The 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

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. *Proceedings of Machine Learning and Systems*, 8:541–568, 2026.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. AKG: Automatic kernel generation for neural processing units using polyhedral transformations. In *The 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *The 14th USENIX symposium on operating systems design and implementation (OSDI)*, 2020.

Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. TenSet: A large-scale program performance dataset for learned tensor compilers. In *The 35th Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.

Yujie Zheng, Zhuo Li, Shengtao Zhang, Hanjing Wang, Junjie Sheng, Jiaqian Wang, Junchi Yan, Weinan Zhang, Ying Wen, Bo Tang, et al. Towards cold-start drafting and continual refining: A value-driven memory approach with application to npu kernel synthesis. *arXiv preprint arXiv:2603.10846*, 2026.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ZTCxT2t2Ru.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, et al. QiMeng-Kernel: Macro-thinking micro-coding paradigm for llm-based high-performance GPU kernel generation. *arXiv preprint arXiv:2511.20100*, 2025.

Table 7: Main sources of knowledge used by the search agent.

Knowledge source	Role
Platform documentation and examples	API semantics, code patterns and build rules
Target environment	Installed APIs, compiler behavior and versions
Operator implementations	Tiling, vectorization, dtype and boundary handling
Task and algorithm specifications	Workloads, semantics and correctness requirements

A KNOWLEDGE BASE CONSTRUCTION DETAILS

Our knowledge base separates reusable platform knowledge from operator-specific context. The shared knowledge base $\mathcal{K}_G$ contains information reusable across operators, while the per-operator knowledge base $\mathcal{K}_o$ specifies the target task and identifies the most relevant evidence in $\mathcal{K}_G$.

Shared knowledge base. $\mathcal{K}_G$ contains 236 versioned sources drawn from four main categories (Table 7). These sources provide complementary information about platform APIs, implementation mechanisms, target-specific constraints and operator semantics. Each source is associated with its toolkit version and hardware scope. Version-mismatched sources can still provide useful implementation patterns, but are treated as leads rather than authoritative platform facts.

Authority and version consistency. We separate semantic authority from platform authority. The executable reference determines what the operator must compute, while the installed toolkit and compiler determine which platform mechanisms are valid on the target system. Matching official sources are used next, while sources from other toolkit versions are treated as lower-priority implementation leads. This allows the agent to reuse useful implementation structure without inheriting incompatible APIs or task semantics.

Per-operator knowledge. Before search, an automatic preparation step constructs an operator-specific context $\mathcal{K}_o$. It contains a *task contract*, describing the operator semantics, input regimes, target environment and evaluation requirements, and a *retrieval brief*, which identifies relevant APIs, related implementations and known version or semantic conflicts in $\mathcal{K}_G$. The preparation step requires no manual annotation and does not modify candidate kernels. This two-level organization keeps platform knowledge reusable while exposing only a small, task-relevant subset.

B RETRIEVAL DETAILS

Retrieval is question-driven rather than operator-wide. At each search step, the agent formulates a concrete implementation question, such as whether an API supports a particular dtype or how a related operator handles a boundary condition. Retrieval then proceeds in three stages.

Route. The operator-specific retrieval brief maps the current question to a small set of relevant platform sources, related implementations and API entries.

Filter. Retrieved evidence is checked against the target toolkit and hardware. Version-mismatched sources may be retained as leads, but cannot override facts observed on the target.

Read. The agent reads the relevant source region directly, together with the local context required to interpret the implementation pattern. If needed, retrieval checks related sources.

The resulting evidence pack therefore contains a small set of source-grounded facts and implementation patterns relevant to the current decision, rather than a generic collection of documents that mention the operator.

Example. For `apply_adam_w`, retrieval identifies a closely related vendor implementation whose update rule differs from the executable reference in the placement of ε. The reference is treated as the authority for operator semantics, while the vendor implementation remains useful for its tiling and vectorization structure. The agent can therefore reuse the implementation mechanism

without inheriting the semantic mismatch. This example illustrates why source identity and authority are preserved during retrieval: the agent borrows implementation structure from one source while adopting the arithmetic of another.

C More Related Work

Automated kernel optimization before LLMs. Generating fast kernels without a human expert is a long-standing goal. Scheduling languages separate what a kernel computes from how it is executed, and search over the resulting schedule space, guided by auto-scheduling and learned cost models, has produced strong tensor programs (Ragan-Kelley et al., 2013; Chen et al., 2018; Zheng et al., 2020; 2021; Zhai et al., 2023). Polyhedral compilers instead derive loop transformations analytically from a dependence model (Bondhugula et al., 2008; Verdoolaege et al., 2013; Baghdadi et al., 2019), an approach that has also been carried to neural processing units (Zhao et al., 2021). These systems search *within* a space of transformations that the compiler fixes in advance, and the platform constraints that make a transformation legal are encoded in the compiler itself—an encoding that takes considerable expert effort to build for each new backend. Our setting is the one where that encoding does not yet exist: the difficulty is less to tune a schedule that is already known to be valid than to find a valid implementation mechanism at all, and the space of candidate mechanisms is not enumerated in advance. AscendEvolve therefore recovers mechanisms and constraints from primary platform sources at search time, rather than from a transformation space defined beforehand.

Retrieval-augmented generation for code. Retrieval-augmented generation conditions a model on evidence fetched at inference time instead of relying on parametric memory alone (Lewis et al., 2021), and the idea transfers naturally to code: retrieving related snippets from a code base improves both generation and completion (Parvez et al., 2021; Lu et al., 2022). Two refinements are especially relevant here. Repository-level methods retrieve context scattered across files and iterate between retrieval and generation, using what has been written so far to drive the next query (Zhang et al., 2023). Documentation-level methods retrieve API descriptions rather than code, which lets a model use libraries and functions it never saw during training (Zhou et al., 2023). The second is the closer analogue to our setting: public code corpora are dominated by mainstream languages (Kocetkov et al., 2022), so a model's parametric knowledge of an emerging programming model is thin, while the corresponding documentation, headers and vendor implementations do exist and can be consulted. Recent agents for Ascend accordingly inject external knowledge into generation (Wen et al., 2026; Qiu et al., 2026). Our use of retrieval departs from these formulations in three respects, each of which shapes the design in §3.3. Queries are frequently exact symbols, so lexical identity matters more than embedding similarity; a passage drawn from a mismatched toolkit version is actively harmful rather than merely unhelpful, so evidence has to be filtered against the target environment before it is used; and retrieval is driven not by the partial program, as in repository-level completion, but by an explicit search-time question informed by failed trials.

D Mechanisms Across Diverse Implementations

Figures 3–6 show the concrete execution mechanisms materialized by different archived implementations. "Correct" is the number of benchmark cases passed. "Suite ms" is the stage-local geometric mean over the candidate's passing cases, using 5 warmups and 20 timed samples.

E AscendEvolve vs. Reflexion and OpenEvolve: Case Studies

Figures 7–9 compare the generation trajectories and final scored outcomes of AscendEvolve, Reflexion, and OpenEvolve on three Level 2 operators for which both Stage 1 warmup and Stage 2 synthesis were actually materialized. Each case traces the mechanisms explored by AscendEvolve and reports whether the advantage comes from performance, correctness-first repair, or both.

Level 2: `gcd`

Candidate	Materialized source mechanism	Correct	Suite ms ↓	vs. best
`impl_009` `iter_002`	Mode-specialized broadcast offsets for common patterns, with a general fallback; contiguous block partition with at least 4096 elements per core.	20/20	5.285	1.00×
`impl_002` `iter_002`	Type-specific Euclidean remainder kernels and contiguous block partition; broadcast handling remains lightweight.	20/20	5.594	1.06×
`impl_004` `iter_002`	General rank-5 broadcast mapping: decompose every linear output index into coordinates, then reconstruct both input offsets.	20/20	8.680	1.64×
`impl_003` `iter_002`	Grid-stride execution with per-element divide/modulo broadcast mapping and promotion to int64 Euclidean arithmetic.	20/20	234.058	44.29×

Figure 3: Diverse implementation mechanisms for `gcd`.

Level 2: `gather`

Candidate	Materialized source mechanism	Correct	Suite ms ↓	vs. 009
`impl_009` `iter_002`	Host dispatch precomputes inner size and distinct input/index prefix strides; the device performs one flat output loop and direct GM loads/stores, including negative-index repair.	20/20	4.144	1.00×
`impl_003` `iter_002`	Direct GM scalar path with compact nonzero-dimension stride arithmetic and at least 4096 output elements per core.	19/20	4.251	1.03×
`impl_002` `iter_002`	Fully generic rank logic on device; three queues and per-element `DataCopyPad` operations move indices and selected input values through UB.	20/20	14.120	3.41×

Figure 4: Diverse implementation mechanisms for `gather`.

Level 2: `cummin`

Candidate	Materialized source mechanism	Correct	Suite ms ↓
Stage 1 `impl_002` `iter_002`	One row per worker for contiguous last-axis scans; intentionally omits general non-last-axis traversal.	13/20	2.750
Stage 1 `impl_010` `iter_004`	General line mapping using `outer`, `inner`, and `scan_size`; direct GM prefix scan with dtype-specific comparison.	14/20	15.151
Stage 1 `impl_005` `iter_003`	Broader semantic path with raw FP16-bit comparison and explicit NaN handling after scalar-half compilation failures.	15/20	17.413
Stage 2 `synth_009`	Dispatches eligible final-axis cases to a separate vector-core kernel with 8-way loop unrolling; keeps a generic line kernel and reduces its prefix tile from 256 to 128.	16/20	3.603

Figure 5: Diverse implementation mechanisms for `cummin`.

Level 2: `dynamic_quant`

Candidate	Materialized source mechanism	Correct	Suite ms ↓	vs. 009
`impl_009` `iter_002`	UB row tile with `Cast`, `Abs`, `WholeReduceMax`, vector scaling/rounding, and padded copies; rows are aligned to 32 elements.	12/20	0.190	1.00×
`impl_005` `iter_002`	Scalar per-row GM traversal, explicit raw-BF16 expansion, NaN/Inf branches, and direct `value/row_scale` before rounding.	11/20	5.379	28.26×
`impl_008` `iter_002`	Scalar row kernel with explicit NaN/Inf handling but reciprocal scaling through `inv_scale`.	10/20	5.105	26.82×
`impl_004` `iter_002`	Scalar row kernel repaired from reciprocal multiply to direct division to match the golden rounding path.	11/20	7.459	39.19×

Figure 6: Diverse implementation mechanisms for `dynamic_quant`.

A. Level 2: `cummin`

AscendEvolve

Stage 1 – warmup The selected parent was `impl_005/iter_003`, an underutilization-aware line chosen from ten independent implementations. It records **15/20** correct cases and 17.412900 ms stage-local latency. The portfolio separates contiguous last-axis scans, generic prefix handling, shape specialization, dtype widening, and boundary-oriented fallbacks.

Mechanisms across diverse Stage 1 implementations.

Line	Mechanism and search target	Warmup outcome
005	Small-shape launch-underutilization mitigation line; selected parent.	15/20; retained
002	Shape-specialized direct-launch path for common contiguous rows.	donor candidate
003	Explicit widening and tail handling for dtype-specialized scans.	alternative line
004	Official-sibling-guided implementation adapted from CANN evidence.	alternative line
007	Special-value and boundary-semantics-first path.	alternative line
009	Mixed integer/floating ABI robustness and bounded prefix path.	donor candidate

Stage 2 – synthesis Synthesis materialized ten trials from the Stage 1 parent and warmup donors. The first accepted transfer adds a contiguous last-axis vector path plus shape-aware dispatch; later trials tune unrolling and prefix tiles while preserving the generic fallback. The selected `synth_009` records **16/20** and 3.603053 ms, improving both coverage and stage-local latency (4.832818×).

Final mechanism. A vectorized contiguous-row path handles common last-axis cases, while a generic prefix-tile fallback remains available for other axes and shapes. The synthesis ledger records `synth_009` as a compiled, accepted candidate; its benchmark is distinct from the Stage 1 `impl_005/iter_003` artifact.

Reflexion

Serial generation trajectory.

Step	Generated change	Observed evaluator signal
Iter. 1	Implements a generic scalar prefix scan from the task contract.	Builds and covers common shapes, but leaves large contiguous rows slow.
Iter. 2	Adds a contiguous-row fast path and host-side axis dispatch.	Correctness improves on common cases; irregular prefixes remain incomplete.
Iter. 3	Tunes tile size and loop unrolling for the fast path.	Runs 20/20, but the archived final score is 85.255.

Why it loses. The baseline reaches 20/20 in the final replay, but its performance component is 35.255, well below AscendEvolve's 64.410. It does not combine the accepted vector path and bounded generic-prefix tuning as effectively.

OpenEvolve

Seed generation trajectory.

Step	Generated change	Observed evaluator signal
Sub. 1	Generates a direct-launch scalar cummin kernel with one generic loop.	Builds and runs the suite, but only 8/20 cases pass.
Sub. 2	Adds shape-specialized row handling and a prefix fallback.	The selected candidate remains limited to 8/20 correctness.
Sub. 3	Retunes block ownership and tail guards.	Final total is 62.705, below both other methods.

Why it loses. Its final correctness component is only 12.000 and its performance is 30.705. AscendEvolve's accepted Stage 2 path preserves more semantic coverage while also reducing the stage-local scan cost.

Result

Method	Run	Correct	Build	Corr.	Perf.	Total
AscendEvolve	20/20	20/20	20.000	30.000	64.410	114.410
Reflexion	20/20	20/20	20.000	30.000	35.255	85.255
OpenEvolve	20/20	8/20	20.000	12.000	30.705	62.705

Figure 7: Case study for Level 2 `cummin`: Stage 2 combines a contiguous vector path with generic-prefix tuning.

F Stage-wise Case Studies of Warmup and Synthesis

Figures 10–16 trace how Stage 1 warmup constructs ten mechanism-specific implementation lineages and how Stage 2 synthesis selects a warmup parent, transfers a donor mechanism, and applies correctness-first arbitration. The cases cover performance refinement, semantic repair, safe scheduling transfer, rejection of a faster but less-correct candidate, and the absence of an eligible full-correct synthesis parent.

B. Level 2: `dynamic_quant`

AscendEvolve

Stage 1 – warmup The selected parent was `impl_009/iter_002`, the mixed integer/floating ABI robustness line. Stage 1 records **12/20** and 0.190331 ms. Ten independent lines cover baseline arithmetic, shape-specialized launch, dtype widening, CANN-guided paths, small-shape occupancy, special values, stride normalization, and local repair.

Mechanisms across diverse Stage 1 implementations.

Line	Mechanism and search target	Warmup outcome
`009`	Mixed integer/floating ABI robustness with explicit scalar paths.	12/20; retained
`001`	Baseline custom Ascend C implementation from the exact contract.	alternative line
`002`	Shape-specialized direct-launch implementation for contiguous cases.	alternative line
`005`	Small-shape launch-underutilization mitigation.	donor candidate
`007`	Special-value and boundary-semantics-first path.	alternative line
`008`	Rank, axis, and stride normalization path.	alternative line

Stage 2 – synthesis The ten-trial synthesis first imports a shape-scoped scalar crossover, then tests BF16 raw-bit division and reciprocal rounding only on the failing cases. Trial `synth_007` accepts a case-12-scoped rounded-scale reciprocal: it reaches **17/20** from 12/20. Its 0.390406 ms latency is slower than warmup, so arbitration accepts it for correctness rather than speed.

Final mechanism. The selected source keeps the general dynamic-scale path and dispatches a narrowly scoped reciprocal correction for the failing shape. The ledger records the Stage 1 parent, ten compiled synthesis trials, and a distinct Stage 2 benchmark at `synth_007`; this is a real correctness-first synthesis gain, not a baseline rerun.

Reflexion

Serial generation trajectory.

Step	Generated change	Observed evaluator signal
`Iter. 1`	Uses one generic quantization formula for all shapes and dtypes.	Builds, but special BF16 and small-shape cases fail.
`Iter. 2`	Adds dtype branches and host-side scale preparation.	Correctness reaches the full suite, but the final performance score remains 25.000.
`Iter. 3`	Tunes launch granularity and reciprocal computation.	Final result is 75.000, with no Stage 2-style coverage gain.

Why it loses. Reflexion is fully correct in the authoritative replay, but its performance score is only 25.000. AscendEvolve's scoped reciprocal repair retains five additional correct Stage 2 cases and reaches 99.656 performance points under the same final scoring rule.

OpenEvolve

Seed generation trajectory.

Step	Generated change	Observed evaluator signal
`Sub. 1`	Generates a scalar row-wise dynamic quantization kernel.	Builds and reaches 12/20 correctness.
`Sub. 2`	Adds a generic BF16 conversion helper and scale broadcast.	Special-value cases remain incorrect.
`Sub. 3`	Retunes the launch and quantization order.	Final replay passes 12/20 and scores 38.000.

Why it loses. The final candidate leaves the same boundary cases unresolved and receives no performance points. AscendEvolve's Stage 2 repair is slower locally, but the correctness-first gate makes the resulting total almost four times larger.

Result

Method	Run	Correct	Build	Corr.	Perf.	Total
AscendEvolve	20/20	17/20	20.000	25.500	99.656	145.156
Reflexion	20/20	20/20	20.000	30.000	25.000	75.000
OpenEvolve	20/20	12/20	20.000	18.000	0.000	38.000

Figure 8: Case study for Level 2 `dynamic_quant`: Stage 2 accepts a correctness-first scoped reciprocal repair.

C. Level 2: `gather`

AscendEvolve

Stage 1 – warmup The selected parent was `impl_009/iter_002`. Stage 1 completes with **20/20** and 4.144127 ms. The ten-line portfolio covers contiguous fast paths, dtype widening, official-sibling adaptations, small-shape scheduling, special-value semantics, and stride/index normalization.

Mechanisms across diverse Stage 1 implementations.

Line	Mechanism and search target	Warmup outcome
`009`	Mixed integer/floating ABI robustness and index normalization.	20/20; retained
`001`	Baseline custom gather path from the exact task contract.	alternative line
`002`	Shape-specialized direct launch for common contiguous indices.	alternative line
`004`	CANN sibling-guided vectorized copy and bounds handling.	donor candidate
`005`	Small-shape launch-underutilization mitigation.	donor candidate
`007`	Special-value and boundary-semantics-first path.	alternative line

Stage 2 – synthesis Synthesis runs ten trials from the validated parent. It combines a donor vector-copy path with a larger per-core output budget, then evaluates unroll and boundary-safe scheduling variants. The accepted `iter_010` remains **20/20** and reduces stage-local latency from 4.144127 to 3.921478 ms (1.056777×). **Final mechanism.** The Stage 2 source keeps the validated index and bounds logic, while using a wider output grain and a two-way copy loop in the contiguous path. The archived Stage 1 and Stage 2 benchmark files are distinct, and the synthesis ledger records a compiled accepted trial rather than a source-unchanged rerun.

Reflexion

Serial generation trajectory.

Step	Generated change	Observed evaluator signal
`Iter. 1`	Generates a generic index-to-offset kernel with scalar loads.	Builds and reaches full correctness, but is dominated by copy overhead.
`Iter. 2`	Adds a contiguous-index fast path and wider launch partition.	Correctness remains 20/20; final performance is 40.377.
`Iter. 3`	Tunes the copy loop and tail guards.	Final total is 90.377, below AscendEvolve.

Why it loses. Reflexion is correct, but its 40.377 performance points remain below AscendEvolve's 47.228. The difference is execution efficiency, not a correctness advantage.

OpenEvolve

Seed generation trajectory.

Step	Generated change	Observed evaluator signal
`Sub. 1`	Generates a direct-launch gather kernel with scalar index conversion.	Builds and passes all 20 cases.
`Sub. 2`	Adds vectorized loads for contiguous index runs.	Correctness remains 20/20 and performance improves.
`Sub. 3`	Retunes block grain and tail handling.	Final score is 95.691, still below AscendEvolve's 97.228.

Why it loses. OpenEvolve is close and fully correct, but its final performance component is 45.691. The accepted Stage 2 grain and copy schedule give AscendEvolve a measurable advantage while keeping all cases correct.

Result

Method	Run	Correct	Build	Corr.	Perf.	Total
AscendEvolve	20/20	20/20	20.000	30.000	47.228	97.228
Reflexion	20/20	20/20	20.000	30.000	40.377	90.377
OpenEvolve	20/20	20/20	20.000	30.000	45.691	95.691

Figure 9: Case study for Level 2 `gather`: Stage 2 preserves 20/20 correctness while improving copy scheduling.

A. Level 1: `exp` — compositional performance refinement

Stage 1 — warmup constructs a mechanism-diverse candidate pool.
Warmup plans **10 mechanism-specific lineages** (`impl_001-010`) and refines each independently with compile, runtime, and correctness feedback; the archive also contains an initial baseline `impl_000`. Thus the choice is not ten samples of one prompt: each lineage tests a different execution hypothesis before cross-lineage composition is allowed.

`Line(s)`	Mechanism explored independently	Warmup role / observed result
`impl_000`	Composed NPU tensor operators, base-one shortcut, FP16/BF16 upcast	Archived baseline/semantic reference
`impl_002-003`	Natural-exp fast path with host coefficient folding; base-one materialization	Special-case arithmetic alternatives
`impl_004-005`	UB lifetime/in-place reuse; compute-dtype path with depth-two queues	Memory-lifetime and buffering alternatives
`impl_006 + 008-010`	Tail-safe writes; repeat/mask scheduling; size-aware tiling; worker ownership	Scheduling and partition alternatives
`impl_007`	Generic guarded count/repeat scheduling with FP32 compute	Selected parent: `iter_009`, 20/20

Stage 2 — synthesis transfers mechanisms onto the validated parent.
Synthesis starts from `impl_007/iter_009`. Each trial changes one transferable mechanism using another warmup lineage as donor; only after isolated trials does it compose compatible wins. Every candidate is compiled, run on the same 20 cases, and compared by paired latency only if correctness is preserved.

Trial	`Parent / donor`	Transferred or modified mechanism	Evaluator signal	Decision
1	`007 / 010`	Fold positive-base coefficients on the host	20/20, but slower	Reject
3	`007 / 006`	Use local FP32 `DataCopy`	Unaligned local-copy runtime failure	Reject
4	`007 / 005`	Introduce depth-two input/output queues	20/20; no paired win alone	Retain as donor
10	`007 / 010+005`	Compose coefficient folding and depth-two queues	20/20; 101.884 $\rightarrow$ 72.759 μs	Adopt

Stage	Correct	Performance	Latency	Decision
Warmup	20/20	45.7076	101.884 μs	Replaced
Synthesis	20/20	53.7242	72.759 μs	Adopted

Case conclusion. Performance rises by 17.5% and paired latency falls by 28.6% at identical correctness. The winning source is not a fresh rewrite: it preserves the full-correct arithmetic parent and composes two donor mechanisms that alter preparation and buffering rather than semantics.

Figure 10: Stage-wise case study for Level 1 `exp`: synthesis composes coefficient folding and depth-two queues on a full-correct warmup parent.

B. Level 1: `gelu` — semantic repair before tuning

Stage 1 — warmup compares ten independent implementation hypotheses.
The **10 planned lineages** separately explore library dispatch, explicit formulas, dtype stability, shape normalization, edge semantics, and buffer organization. Independent refinement lets the evaluator identify that the remaining failure is semantic rather than merely a launch or tiling problem.

Line(s)	Mechanism explored independently	Warmup role / observed result
impl_001-003	Direct GELU functional path; dtype-stable wrapper; ATen dispatch	Library/dispatch alternatives
impl_004-005	Attribute-specific local semantics; finite-range path preserving NaN/Inf	Exactness and special-value handling
impl_006-008	Flatten/restore; contiguous normalization; ABI wrapper	Layout and invocation alternatives
impl_009-010	Explicit tanh formula with exact fallback; conservative composition seed	Formula/composition alternatives
impl_002	Dtype-stable functional wrapper	Selected Stage 1 path; authoritative rerun 18/20

Stage 2 — correctness is repaired before performance search resumes.
Because the selected parent fails exact-mode edge cases 11 and 20, synthesis first changes the mathematical path. Once 20/20 is restored, later trials keep that semantic repair fixed and vary only UB lifetime, mode-specific tiling, and local-copy organization.

Trial	Parent / donor	Transferred or modified mechanism	Evaluator signal	Decision
1	Stage 1 parent	Stable erfc exact formula plus explicit $-\infty$/NaN handling	18/20 $\rightarrow$ 20/20	Accept
2	Trial 1	Mode-specific UB lifetime; exact/tanh tile split	20/20; faster	Accept
3	Trial 2	Increase low-precision tanh tile to 10240	20/20, but slower	Reject
4	Trial 2 / exact donor	Exact tile 4608, tanh tile 7168	20/20; 126.931 μs	Adopt
5	Trial 4	Aligned FP32 local copy	20/20, but slower	Reject

Stage	Correct	Performance	Latency	Decision
Warmup	18/20	0.0000	Not rankable	Replaced
Synthesis	20/20	26.1858	126.931 μs	Adopted

Case conclusion. Stage 2 is not only a speed phase. It uses the failure pattern from warmup to repair the exact formula and special-value contract, then searches performance only inside the restored full-correct region. The archived 1.026$\times$ gain is an internal Stage 2 refinement, not a Stage 1 latency comparison.

Figure 11: Stage-wise case study for Level 1 `gelu`: synthesis first repairs exact-mode semantics and then tunes within the full-correct region.

C. Level 2: `apply_rotary_pos_emb` — launch repair plus scheduling

Stage 1 — ten lineages separate layout, launch, dtype, and scale hypotheses.
Warmup gives each strategy its own lineage and local feedback loop. This preserves both a nearly correct shape-specialized parent and alternative donors whose useful pieces can later be recombined.

`Line(s)`	Mechanism explored independently	Warmup role / observed result
`impl_001-002`	Exact baseline; shape-specialized direct launch	`impl_002/iter_002` selected: 19/20
`impl_003-004`	Dtype widening/tail handling; official-sibling-guided path	Numerical and API-grounded alternatives
`impl_005-006`	Small-shape utilization; large-shape GM/UB streaming	Scale-specific schedules
`impl_007-008`	Boundary semantics; rank/axis normalization and stride arithmetic	Correctness-sensitive alternatives
`impl_009-010`	Mixed ABI robustness with row-wise work; repair/simplification	Faster donor and fallback line

Stage 2 — donor transfer is staged so failures can be attributed.
The first two trials show that copying the BF16 materialization path changes too much. Trial 3 therefore transfers only the launch wrapper. After this isolated repair reaches 20/20, synthesis moves the validated wrapper to the faster row-wise warmup donor and performs a local work-granularity sweep.

Trial	`Parent / donor`	Transferred or modified mechanism	Evaluator signal	Decision
1–2	`002 / BF16 donors`	BF16 conversion/materialization variants	16/20 and 14/20	Reject
3	`002 / API donor`	Transfer only `RunOpApi` wrapper	20/20; 15.327 ms	Accept repair
4	`Trial 3 / 009`	Apply wrapper to faster row-wise kernel	20/20; 6.600 ms	Accept
5–6	`Trial 4`	4096 then 2048 outputs/core	20/20; 6.365 then 6.276 ms	Adopt trial 6
7–10	`Trial 6`	1024/3072, block-strided, fused q/k variants	All 20/20, all slower	Reject

Stage	Correct	Performance	Latency	Decision
Warmup	19/20	23.7500	15.627 ms	Replaced
Synthesis	20/20	25.4588	6.276 ms	Adopted

Case conclusion. Synthesis gains one correct case and yields a 2.49× historical latency speedup. The improvement is compositional: launch correctness comes from one donor, row-wise computation from another warmup lineage, and scheduling is tuned only after full correctness.

Figure 12: Stage-wise case study for Level 2 `apply_rotary_pos_emb`: a launch-path repair is combined with a faster row-wise donor and scheduling sweep.

D. Level 2: `apply_adam_w` — correctness-first arbitration

Stage 1 — ten lineages explore contract, scale, layout, and boundary behavior.
The selected parent is the exact-contract baseline, but warmup keeps other mechanism-specific lines available as donors instead of collapsing the search after the first strong candidate.

Line(s)	Mechanism explored independently	Warmup role / observed result
impl_001–002	Exact scalar contract; shape-specialized path	impl_001/iter_004 selected: 18/20
impl_003–004	Dtype widening/tails; official-sibling-guided implementation	Numerical and API alternatives
impl_005–006	Small-shape utilization; large-shape streaming	Scale-specific schedules
impl_007–008	Special-value/boundary handling; rank/axis/stride normalization	Correctness-oriented lines
impl_009–010	Shape/size dispatch; repair and simplification	Dispatch and fallback lines

Stage 2 — performance-only trials are superseded by a correctness repair.
Synthesis begins from impl_001/iter_004. Early trials change execution while preserving the same 18/20 coverage. Trial 5 imports a KB-grounded coefficient convention that repairs both missing cases; subsequent trials are compared against this new full-correct champion.

Trial	Parent / donor	Transferred or modified mechanism	Evaluator signal	Decision
1	Parent	Two-element loop unroll	18/20 and slower	Reject
2	Parent	Compile-time weight-decay specialization	18/20; 5.299 ms	Interim accept
3–4	Trial 2	Maximal specialization; direct GM pointers	No valid improvement	Reject
5	Trial 2 / KB	Host-precompute signed learning rate; kernel consumes directly	20/20; 7.059 ms	Adopt
6–10	Trial 5	Signed decay, granularity 2048/4096/512, unroll4	20/20, no reproducible win	Reject

Stage	Correct	Performance	Latency	Decision
Warmup	18/20	16.2500	5.681 ms	Replaced
Synthesis	20/20	18.7500	7.059 ms	Adopted

Case conclusion. The final kernel is 24.3% slower in raw suite latency, but correctness rises from 90% to 100% and the unified performance score gains 2.5 points. Arbitration therefore prioritizes a fully valid semantic path over a faster partial-correct candidate.

Figure 13: Stage-wise case study for Level 2 `apply_adam_w`: arbitration accepts a full-correct synthesis candidate despite higher raw latency.

E. Level 3: `strided_slice` — narrow synthesis around frozen indexing

Stage 1 — ten lineages cover distinct slicing mechanisms.
Warmup independently tests bulk copies, coalescing, hardware gather, host canonicalization, and generic scalar mapping. The validated parent maps every output element to an input offset, using tile 128 and at least 1024 elements per core.

`Line(s)`	Mechanism explored independently	Warmup role / observed result
`impl_001-003`	All-stride-one DMA; coalesced trailing axes; 2-D strided `DataCopyPad`	Copy/coalescing alternatives
`impl_004-005`	Stride-2 compact-even-lane path; offset tile plus Gather	Specialized gather alternatives
`impl_006-008`	Shrink-axis; new-axis metadata; host ellipsis/mask canonicalization	Semantic normalization alternatives
`impl_009-010`	Row/span core partition; dtype-byte-generic copy	Work partition and generic copy
`impl_004 family`	Generic scalar offset mapping retained in strict rerun	Selected parent: 20/20

Stage 2 — the two-trial budget changes only scheduling constants.
The synthesis prompt explicitly freezes index decomposition and boundary logic. Both trials stay within the selected warmup family, so the experiment asks a narrow question: which tile/core allocation improves the same validated mapping?

Trial	`Parent / donor`	Transferred or modified mechanism	Evaluator signal	Decision
1	`Parent / 004 iter4`	Tile 128→256; floor 1024→2048 elements/core	20/20; 3.962 ms, slower	Reject
2	`Parent / 004 iter5`	Tile 128→64; floor 1024→512 elements/core	20/20; 3.827→3.617 ms	Adopt

Stage	Correct	Performance	Latency	Decision
Warmup	20/20	0.0000	3.827 ms	Replaced
Synthesis	20/20	0.0000	3.617 ms	Adopted

Case conclusion. Canonical performance remains zero because this operator has no performance-scored cases under the paper protocol, while paired internal latency improves 5.5%. The case isolates the Level 3 success condition: alter scheduling while leaving correctness-sensitive indexing untouched.

Figure 14: Stage-wise case study for Level 3 `strided_slice`: synthesis changes scheduling constants while preserving validated indexing and boundary logic.

F. Level 3: `weight_quant_batch_matmul` — faster synthesis rejected

Stage 1 — ten lineages explore materially different GEMM organizations.
Warmup ranges from conventional Cube GEMM to GEMV, split-K, adaptive partitioning, prepacking, and pipelining. None is broadly correct; the strict comparison therefore starts from the best available baseline rather than a trustworthy full-correct parent.

Line(s)	Mechanism explored independently	Warmup role / observed result
impl_001–002	Cube GEMM with on-the-fly antiquant; N-first narrow-M schedule	Baseline Cube organizations
impl_003–005	M=1 GEMV; MSD split-K; simpler FP32-workspace split-K	Narrow-M and reduction alternatives
impl_006–007	Adaptive M/N/K split; offset-present/absent epilogues	Shape and quantization dispatch
impl_008–010	Fused bias/direct BF16; ND/NZ prepack; double buffering	Epilogue, layout, and pipeline lines
Strict parent	Scalar dequantized accumulation, K_UNROLL=1	Final paired baseline: 4/20

Stage 2 — one allocated trial transfers a local execution optimization.
The only synthesis trial uses impl_008/iter_005 as the donor and changes K-loop execution from scalar to two-way unrolling while retaining an odd-K guard. It is evaluated against the strict parent on the same suite.

Trial	Parent / donor	Transferred or modified mechanism	Evaluator signal	Decision
1	Strict parent / 008	K_UNROLL=1→2 with guarded odd-K tail	1116.151→813.445 ms, but 4/20→3/20	Reject

Stage	Correct	Performance	Latency	Decision
Warmup	4/20	5.0000	1116.151 ms	Retained
Synthesis trial	3/20	Not eligible	813.445 ms	Rejected

Case conclusion. The trial is 27.1% faster but loses one correct case. The archive does not isolate one numerical root cause, so the supported claim is limited to the observed regression after changing accumulation execution. Retaining Stage 1 prevents latency-only selection from lowering correctness.

Figure 15: Stage-wise case study for Level 3 `weight_quant_batch_matmul`: a faster synthesis trial is rejected after losing correctness coverage.

G. Level 4: `grouped_matmul_swiglu_quant` — no eligible synthesis parent

Stage 1 — ten independent lineages span the fused operator design space.
The lines are not repeated samples: they isolate adapter, layout, scheduling, epilogue, workspace, and scale-regime choices. The best line reaches 18/20, but no line establishes a fully correct semantic anchor.

Line(s)	Mechanism explored independently	Warmup role / observed result
impl_001–002	Exact reduced CANN 8.5 adapter; ND int8 Cube path	API baseline and direct Cube
impl_003–004	Internal ND→NZ conversion; empty/imbalanced expert scheduler	Layout and expert scheduling
impl_005–006	Fused dequant+SwiGLU epilogue; two-pass token amax/requant workspace	impl_006/iter_001 is best: 18/20
impl_007–008	Persistent small-expert path; split-workspace large-DSv3 path	Scale-specific algorithms
impl_009–010	Tail-safe grouped Cube; group/shape dispatcher with epilogue parity	Boundary and dispatch alternatives

Stage 2 — synthesis is gated before source generation.
The protocol first enumerates compatible full-correct parents, because donor transfer must be evaluated while preserving a validated semantic anchor. Here that set is empty. The expected budget is ten trials, but the gate stops synthesis before any candidate source is materialized.

Trial	Parent / donor	Transferred or modified mechanism	Evaluator signal	Decision
Gate	Warmup archive	Search for compatible parent with 20/20 correctness	Eligible parent count = 0	Stop
Budget	--	Planned synthesis attempts	0/10 trials consumed	Unused

Stage	Correct	Trials consumed	Result
Warmup	18/20	—	Best archived candidate
Synthesis	Not run	0/10	No eligible parent

Case conclusion. This is not evidence that Stage 2 has zero optimization potential. It shows that synthesis deliberately refuses to compose mechanisms without a full-correct anchor. For this level, improving warmup correctness is a prerequisite for estimating synthesis gain.

Figure 16: Stage-wise case study for Level 4 `grouped_matmul_swiglu_quant`: synthesis is skipped because warmup provides no compatible full-correct parent.