ASCENDOPTIMIZER: EPISODIC AGENT FOR ASCEND NPU OPERATOR OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimizing AscendC (Ascend C) operators for Ascend NPUs is difficult for two reasons. First, unlike CUDA, the ecosystem offers few public kernels to learn from. Second, performance depends on a coupled two-part implementation: a host-side tiling program that controls data movement and a kernel program that schedules and pipelines computation. We present **AscendOptimizer**, an episodic agent that builds missing optimization knowledge from execution itself. For kernel optimization, AscendOptimizer *rewinds* strong implementations by removing optimizations in a controlled way, then keeps the changes whose removal measurably hurts performance as reusable experience for later rewriting. For host-side optimization, it runs profiling-in-the-loop evolutionary search to find valid, fast tiling and data-movement configurations directly from hardware feedback. On a transductive benchmark of 101 real AscendC operators, AscendOptimizer reaches a $1.21\times$ geometric-mean speedup over the open-source references, ahead of best-of-N sampling, evolutionary search, and agent baselines given the same number of hardware evaluations per operator.

1 INTRODUCTION

LLM-based kernel optimization has recently shown strong results on platforms with rich open-source ecosystems, especially CUDA and Triton (Ouyang et al., 2025; Li et al., 2025b). Modern code LMs are trained on large public-code corpora and can memorize or reproduce fragments of that data (Chen et al., 2021; Kocetkov et al., 2022; Yang et al., 2024), and CUDA/Triton supply many public optimized implementations and reusable performance idioms such as tiling, hierarchical data movement, and memory-efficient fused kernels (Tillet et al., 2019; NVIDIA Corporation, 2017; Dao et al., 2022). Optimization agents can therefore reason about programs while adapting implementation patterns already visible in the ecosystem. On emerging or restricted-access hardware, this support is much weaker. Prior studies report sharply lower LLM kernel-generation success on AscendC than on CUDA, and attribute this gap in part to limited representation of AscendC-specific syntax, execution models, and optimization principles in public or pretraining corpora (Wen et al., 2025b; Cao et al., 2026). Platform-specific optimization patterns are therefore less accessible to current models, and compiler or profiler feedback often identifies symptoms without directly specifying the structural rewrite needed to fix them. In this setting, relying solely on pretrained code knowledge or one-shot generation is insufficient, which raises a central question: *when external references are limited, how can an optimization agent build the missing expertise for itself?*

A broader family of methods answers limited supervision by constructing learning signals from the artifact itself or from interaction: self-supervised program repair synthesizes buggy programs from correct code and learns to recover them under compiler or test feedback (Allamanis et al., 2021; Ye et al., 2022), agents externalize trial-and-error outcomes into episodic memory or a growing skill library (Shinn et al., 2023; Wang et al., 2023), and hindsight-style reinforcement learning makes sparse feedback useful by reinterpreting trajectories after the fact (Andrychowicz et al., 2017). Following this principle, we treat optimized kernels as latent sources of optimization knowledge: a code motif whose removal measurably hurts hardware performance becomes evidence of a reusable optimization pattern.

We instantiate this perspective as **Optimization Rewind**: instead of assuming exact reversibility, we construct semantically valid de-optimized neighbors of strong kernels by removing recognizable

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.

optimization motifs, and retain only removals that pass validation but measurably hurt hardware performance. Each validated degradation becomes evidence that the removed motif was useful in its original context, and is distilled into reusable forward-optimization experience for later code rewriting. This mechanism is related to prior de-optimization (Hines et al., 2005) and rewind-style self-supervision (Zhang et al., 2025a), but is used here to mine practical optimization knowledge under scarce external demonstrations.

We study this problem on Ascend NPUs, where the lack of public optimization knowledge is pronounced. Recent MultiKernelBench results (Wen et al., 2025b) show a generalization gap of more than an order of magnitude between one-shot CUDA and AscendC operator generation (Table 1). The issue goes beyond syntax. AscendC exposes an explicitly managed memory hierarchy and requires developers to orchestrate data movement and synchronization through the on-chip Unified Buffer (UB) (Zhou et al., 2025a). An AscendC operator has two parts: a host-side *tiling* program decides how data are partitioned and moved, and a device-side *kernel* program decides how computation is scheduled and pipelined. Performance depends on both *how data are fed* and *how the kernel runs*, making Ascend a useful setting for studying coupled optimization under knowledge scarcity.

Table 1: One-shot operator generation pass rate (Pass@1), reported by MultiKernelBench (Wen et al., 2025b). The large CUDA–AscendC generation gap is consistent with limited AscendC-specific knowledge available to current models. We study whether execution-grounded experience can compensate for this limitation in post-optimization.

Model	**CUDA (Pass@1)**	**AscendC (Pass@1)**
DeepSeek-R1	$52.6\,\%$	$1.4\,\%$
Claude-Sonnet-4	$47.0\,\%$	$2.1\,\%$
Qwen3-235B (think)	$44.2\,\%$	$0.7\,\%$

Our empirical task is post-optimization, not generation: **AscendOptimizer** starts from an existing AscendC implementation that compiles and passes the correctness harness, and a generated kernel could serve as that starting point $\mathcal{K}^{(0)}$ without changing the method. It optimizes the two handles separately because they fail differently. Stage I refines the kernel: benchmark-wide Optimization Rewind builds the shared experience bank, which then guides retrieval-augmented rewriting under compiler and profiling feedback, supplying what sparse feedback alone struggles to teach—*how the code should be rewritten*. Stage II handles tiling, where small changes in tile sizes or data-movement schedules can turn a valid fast configuration into an invalid one; rather than encoding those rules in advance, it uses **evolution-guided program search** with hardware-in-the-loop feedback. The stages are asymmetric: Stage I produces the main structural gains, and Stage II adds resource-policy improvements once the refined kernel exposes better execution opportunities.

Our contributions are threefold: **1)** We introduce **Optimization Rewind**, a self-supervised way to bootstrap kernel optimization experience under knowledge scarcity by degrading strong implementations in a controlled manner and distilling the removals that verifiably matter for performance. **2)** We formulate AscendC optimization as a coupled problem of *host-side tiling* and *device-side kernel refinement*, and instantiate this view in **AscendOptimizer**, which combines retrieval-guided kernel rewriting with hardware-guided tiling search. **3)** On a transductive benchmark of **101** real AscendC operators, AscendOptimizer achieves a $1.21\times$ geometric-mean speedup over the open-source references under 230 hardware evaluations per operator.

2 Related Work

Automated Kernel Optimization. Compiler stacks and schedule search reduce manual tuning effort through DSLs, IRs, cost models, and automated exploration (Chen et al., 2018; Ragan-Kelley et al., 2013; Tillet et al., 2019; Wang et al., 2025b; Bondhugula et al., 2008; Baghdadi et al., 2019; Zhao et al., 2021; Zheng et al., 2020; 2021; Wu et al., 2025), yet high-end kernels such as FlashAttention (Dao et al., 2022; Shah et al., 2024) still require architecture-specific design. The gap is sharper on Ascend NPUs, whose memory hierarchy and instruction model make optimization heavily dependent on hardware expertise (Zhou et al., 2025a; Ai et al., 2025), and available Ascend analyses and profilers still rely on human interpretation (Zhou et al., 2025b; Moustafa, 2023). A recent line of work instead

lets LLM agents generate and refine CUDA or Triton kernels under compiler, profiler, multi-agent, and hardware-in-the-loop feedback (Ouyang et al., 2025; Li et al., 2025b; Wei et al., 2025; Lei et al., 2025; Zhang et al., 2025b; Liao et al., 2025; Wang et al., 2025a; Guo et al., 2025; Li et al., 2025a; Andrews & Witteveen, 2025; Li et al., 2026). Their success is tied to ecosystems where public kernels and optimization idioms are abundant; direct transfer to DSAs is difficult because aligned corpora and accessible architectural knowledge are scarce (Wen et al., 2025b; Cao et al., 2026).

Optimization Memory on Emerging Accelerators. AccelOpt (Zhang et al., 2026) records slow–fast pairs that arise during forward search for Trainium/NKI kernels, EvoKernel (Zheng et al., 2026) shares value-driven memory across NPU synthesis trajectories, and KernelEvolve (Liao et al., 2025) combines hardware documentation, code examples, persistent memory, and graph-based search across heterogeneous accelerators. Rewind differs in how it obtains the stored signal. It starts from a verified strong implementation, fixes host-side tiling, deliberately removes one recognizable motif, and retains the inverse edit only when the candidate remains correct and becomes slower on hardware. The slow endpoint is thus a controlled local probe rather than a candidate encountered during forward search. Earlier de-optimization work (Hines et al., 2005) instead reopened compiler phase-ordering choices for the same program and did not export the resulting edit as cross-operator knowledge. Our claim is this intervention-based construction and its use in the two-stage system, not de-optimization or episodic memory in isolation.

Internalized Experience and NPU Code Generation. A separate line of work stores optimization experience in model parameters through RL, SFT, or large-scale contrastive sampling (Baronio et al., 2025; Woo et al., 2025; Li et al., 2025c;d; Su et al., 2025; Seed et al., 2025). These approaches can be effective, but they require many domain-specific code-performance pairs and substantial training cost, and in immature hardware ecosystems that data dependency is often the bottleneck (Wen et al., 2025b); AscendOptimizer therefore targets a training-free setting. Closest to our platform, AscendKernelGen (Cao et al., 2026), AscendCraft (Wen et al., 2025a), and MultiKernelBench (Wen et al., 2025b) study Ascend or NPU code generation from high-level specifications or DSLs. AscendOptimizer addresses the complementary problem of improving *existing* functionally correct AscendC implementations, jointly optimizing host-side tiling and device-side kernels while building missing domain knowledge from execution feedback; Appendix A details why the MultiKernelBench evaluation artifacts do not overlap with ours.

3 The AscendOptimizer Agent

3.1 Method Overview

Knowledge scarcity does not affect every part of AscendC optimization in the same way. Kernel code contains structured motifs such as pipelining, vectorization, buffer management, tail handling, and synchronization elimination. Because these motifs are compositional, they can be mined from strong existing implementations. Host-side tiling is different. It is tightly constrained by input shapes, alignment rules, buffer capacity, and resource-allocation decisions, so its feasible region is discrete and highly discontinuous. Because these constraints are strongly configuration-dependent, we do not rely primarily on cross-operator retrieval for tiling. Instead, we optimize it directly with hardware feedback. We therefore treat the two parts differently instead of forcing them into a single optimizer.

Concretely, **AscendOptimizer** has a benchmark-wide offline bootstrapping phase and a per-operator online optimization phase, shown in Figure 1. Offline, each operator is explored with **Optimization Rewind** under a fixed rewind budget. After rewind finishes across the benchmark, the hardware-validated de-optimization pairs are distilled into a shared experience bank. Online, Stage I uses this bank to refine the target kernel, and Stage II then searches host-side tiling configurations under the refined kernel.

3.2 Problem Formulation

Under the Ascend NPU heterogeneous computing architecture, we represent the operator to be optimized as a tuple $\mathcal{O} = \langle \mathcal{T}, \mathcal{K}, \mathcal{S} \rangle$: the host-side tiling function $\mathcal{T} \in \mathbb{C}_{\text{tiling}}$ determines data partitioning, movement, and resource policies; the device-side kernel code $\mathcal{K} \in \mathbb{C}_{\text{kernel}}$ determines instruction scheduling, buffering, and synchronization structure; and $\mathcal{S}$ collects the static operator attributes (e.g., input shape, data type, layout). Given hardware constraints H (e.g., buffer capacity,

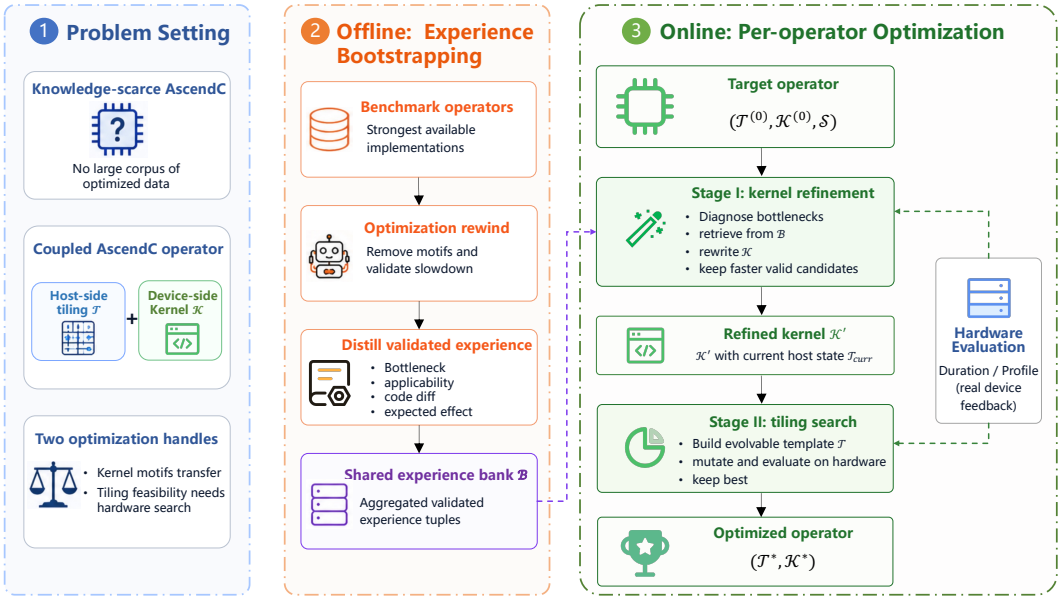

Figure 1: Overview of **AscendOptimizer**. Offline, we rewind benchmark operators and distill validated optimization motifs into a shared experience bank. Online, the bank guides Stage I kernel refinement, after which Stage II searches host-side tiling configurations under the refined kernel.

pipeline stages, and alignment rules), our objective is to identify the tiling function $\mathcal{T}^*$ and kernel implementation $\mathcal{K}^*$ that minimize end-to-end latency on real hardware:

$$(\mathcal{T}^*, \mathcal{K}^*) = \arg\min_{\mathcal{T},\mathcal{K}} \mathcal{L}\left(\text{Exec}(\mathcal{T}, \mathcal{K}, \mathcal{S}) \mid H\right). \tag{1}$$

Here, $\text{Exec}(\cdot)$ denotes hardware compilation and execution, and $\mathcal{L}$ denotes the measured latency. For brevity, when H and $\mathcal{S}$ are fixed, we write $\mathcal{L}(\mathcal{K} \mid \mathcal{T}_{\text{curr}})$ as shorthand for $\mathcal{L}(\text{Exec}(\mathcal{T}_{\text{curr}}, \mathcal{K}, \mathcal{S}) \mid H)$ and omit $\text{Exec}(\cdot)$ and H.

Although $\mathcal{T}$ and $\mathcal{K}$ are coupled, they play different roles. Kernel rewrites change instruction scheduling, buffer allocation, and synchronization topology, which in turn changes which tiling strategies are feasible or worthwhile. Tiling, by contrast, searches for better data partitioning and resource policies under a fixed kernel structure. In our implementation, kernel refinement is evaluated under the current tiling function $\mathcal{T}_{\text{curr}}$, yielding $\mathcal{K}_{\text{curr}}$, and tiling search is then run under that refined kernel. Since H contains many non-differentiable black-box constraints (e.g., bank conflicts, alignment failures, and cache thrashing), and the code space $\mathbb{C}_{\text{tiling}} \times \mathbb{C}_{\text{kernel}}$ is highly discrete and non-convex, direct gradient-based optimization is not applicable.

3.3 Offline Experience Bootstrapping via Optimization Rewind

Why Rewind Works. High-performance kernels contain layered optimization motifs such as pipelining, vectorization, buffer management, tail handling, and barrier elimination. These motifs interact, so Optimization Rewind does not assume exact reversibility. Instead, many motifs leave recognizable local signatures in code or profiles. We therefore remove one targeted motif at a time, keep only semantic-preserving candidates that become measurably slower on hardware, and distill the removed motif as reusable forward-optimization experience. We formalize this view as a local one-factor-at-a-time screening on the discrete motif space in Appendix B.1.

Definition 3.1 (Hardware-validated Rewind acceptance)**.** Let $\mathcal{K}_A$ be a kernel whose active motif set is $A \subseteq \Omega$, and let r_m denote the rewind action for motif $m \in A$ under the fixed tiling $\mathcal{T}$. Define the local one-factor-at-a-time effect $\widehat{\Delta}_m(\mathcal{K}_A; \mathcal{T}) = \mathcal{L}(\mathcal{K}_{A\setminus\{m\}} \mid \mathcal{T}) - \mathcal{L}(\mathcal{K}_A \mid \mathcal{T})$. We accept the inverse rewrite $\mathcal{K}_{A\setminus\{m\}} \to \mathcal{K}_A$ as experience only if $r_m(\mathcal{K}_A) \neq \perp$ (the candidate compiles and

remains semantically equivalent) and $\widehat{\Delta}_m(\mathcal{K}_A; \mathcal{T}) > \tau_{\text{noise}}$. The criterion identifies a positive local contribution at the measured basepoint; it is not a convergence guarantee or a global causal claim.

Per-Operator Rewind Trajectory Construction. Before any online optimization, we run a benchmark-wide offline bootstrapping phase. For each operator o in the benchmark, we start from its strongest available implementation and allocate a rewind budget B_r. In each round, an inverse agent proposes a *semantically meaningful* de-optimization, such as removing double buffering, breaking vectorized data paths, or reintroducing unnecessary synchronization, rather than corrupting the code at random. This produces a validated de-optimization trajectory

$$\mathbb{T}_o = \left(\mathcal{K}_o^{(0)}, \mathcal{K}_o^{(1)}, \ldots, \mathcal{K}_o^{(T_o)}\right), \quad T_o \leq B_r. \tag{2}$$

Hardware-Validated Experience Distillation. Each rewound candidate is then subjected to compilation, correctness, and profiling checks on Ascend hardware. We only keep pairs whose latency meaningfully degrades after the semantic de-optimization, which tells us that the removed motif was actually helping performance rather than merely changing style. For each validated pair, the agent distills a structured experience tuple

$$\mathcal{M} = \langle \text{Title, Bottleneck, Applicability, Expected Effect, Code Diff, Source} \rangle. \tag{3}$$

Here, *Bottleneck* and *Applicability* describe when the pattern should be retrieved, while *Expected Effect* and *Code Diff* describe what change should be made and why it should help. *Source* records the contributing operator and its category. After every operator finishes Rewind, we aggregate the validated tuples once into a shared hardware-validated experience bank $\mathcal{B}$. The online phase treats this bank as immutable: accepted online candidates are not written back. Source provenance also lets us exclude held-out operators and audit self-source retrieval without changing the retrieval key. Algorithm 1 states the full bank-construction procedure.

3.4 Online Retrieval-Guided Kernel Refinement

Once the shared bank has been built, each target operator enters online optimization. Stage I uses the frozen Rewind-derived bank to refine the target kernel, followed by Stage II tiling search. Algorithm 2 gives the full online optimization loop.

Bottleneck Diagnosis. Given the current operator state $(\mathcal{T}_{\text{curr}}, \mathcal{K}_{\text{curr}}, \mathcal{S})$, the agent analyzes compiler diagnostics and profiling traces, then summarizes the main structural bottleneck as a query q.

Experience Retrieval. Using q, a dense retriever fetches the Top-k tuples $\{\mathcal{M}_i\}_{i=1}^{k}$ from the frozen bank $\mathcal{B}$. Ranking uses bottleneck symptoms and structural applicability rather than operator identity; the stored source field is retained for auditing and exclusion rules.

Experience-Guided Rewriting and Selection. A refiner LLM rewrites $\mathcal{K}_{\text{curr}}$ using the retrieved tuples as structured guidance. Each rewritten candidate is compiled, checked for numerical correctness, and profiled under the current tiling function $\mathcal{T}_{\text{curr}}$. We only accept candidates that both pass validation and improve measured latency, updating $\mathcal{K}_{\text{curr}}$ accordingly. Stage I is therefore more than plain RAG and more than blind trial-and-error: it combines *bootstrapped shared experience* with *hardware-grounded candidate selection*.

3.5 Hardware-in-the-Loop Tiling Search

Stage II targets a different optimization object. Unlike kernel motifs, tiling validity depends much more heavily on shape-specific constraints, alignment, buffer capacity, and hardware resource interactions. A small edit can turn a valid fast candidate into an invalid one. For that reason, retrieval is not the main mechanism here. Instead, we rely on hardware-in-the-loop search under the current refined kernel.

Search Space Construction. Starting from the original host-side implementation and attributes $\mathcal{S}$, the agent synthesizes an evolvable template $\mathcal{T}_{\text{base}}$ containing mutation markers (Appendix B.4, Figure 4). The template exposes the locations where tile size, core allocation, buffer policy, and lightweight control logic can be changed.

Mutation. At each search step, the LLM mutates the current tiling candidate. Mutations include changes to tile sizes, core counts, buffer policies, and lightweight branch logic for data movement and launch configuration.

Hardware Evaluation and Survivor Update. Each mutated tiling candidate is compiled and executed with the current refined kernel $\mathcal{K}_{\text{curr}}$. Candidates that fail compilation or correctness checks are discarded immediately. Among the survivors, the fastest one becomes the parent for the next search step. This keeps the search inside the implicit feasible region defined by hardware constraints while exploiting the execution structure already established by Stage I.

4 EXPERIMENTS

The following subsections ask, in turn, how ASCENDOPTIMIZER compares with search and agent baselines under a fixed hardware-evaluation budget, whether Rewind improves over documentation-derived and forward-search experience, which code changes explain the gains, what each component contributes, and whether a frozen bank transfers beyond the original benchmark and survives a second LLM. Appendix A gives additional benchmark details and the full operator list.

4.1 EXPERIMENTAL SETUP

Setup and correctness. Experiments run on Ascend 910B2 NPUs with CANN 8.3. The 101-operator main experiments use DeepSeek-V3.2 for Rewind, kernel rewriting, and tiling mutation. The held-out and second-model studies use DeepSeek-V4-Pro and are reported separately. The embedding model is `text-embedding-3-small`. All methods within a comparison use the same toolchain, stream settings, synchronization, inputs, and correctness harness. We compare NPU outputs with CPU references using absolute and relative tolerance checks following official CANN examples.

Benchmark and stratification. We construct the main benchmark from the official AscendC `cann-ops`[1] repository and use its implementations as references. We retain operators that compile, execute, pass CPU-reference checks, and show stable baseline latency under repeated profiling, yielding 101 operators for the main experiments. The operator implementations used in our evaluation are publicly available in an anonymized repository.[2] We stratify them by *engineering complexity*, not measured speed or presumed optimization potential: Level 1 contains 25 regular elementary operators; Level 2 contains `foreach`, activation, normalization, fused, pooling, and upsampling operators; and Level 3 contains 13 operators with complex parallelization, irregular access, or elaborate data orchestration. Appendix A gives the complete assignment and the shape-adjustment protocol.

Baselines. We compare against BoN, OpenEvolve, an AscendC-adapted implementation of CudaForge, and EvoKernel. All methods use the same 101 operators and receive 230 hardware evaluations per operator.

Metrics and budget. We report latency, the geometric-mean speedup over `cann-ops` of $\text{speedup}(op) = T_{\text{baseline}}(op)/T_{\text{gen}}(op)$, and $\text{fast}_p = |\{op \in \mathcal{O} \mid \text{speedup}(op) > p\}| \,/\, |\mathcal{O}|$, the fraction of operators exceeding a speedup of p. Each hardware evaluation is one compile–correctness–profiling attempt taking roughly two minutes of accelerator time; failures and timeouts count toward the budget but do not update the incumbent. All main-comparison methods receive 230 evaluations per operator, which ASCENDOPTIMIZER splits into 20 for offline Rewind and 210 for online optimization. The 101-operator experiment therefore costs about 774 accelerator-hours, of which bank construction accounts for 67. That cost is paid once and amortizes to about 40 minutes per benchmark operator, and a new target can reuse the frozen bank.

4.2 OVERALL PERFORMANCE

Table 2 answers the first question by comparing ASCENDOPTIMIZER with Best-of-N sampling (BoN), OpenEvolve (Sharma, 2025), and an AscendC-adapted implementation of CudaForge (Zhang et al., 2025b), and EvoKernel (Zheng et al., 2026) under the same 230-evaluation budget per operator.

[1] `https://gitee.com/ascend/cann-ops`

[2] `https://anonymous.4open.science/r/ascendOptimizer-bench-7461/README.md`

For each engineering-complexity level defined in Section 4.1, we report geometric-mean speedup (GM) and the fast$_x$ ratios for $x \in \{1.2, 1.4, 1.8, 2.0\}$.

Across all three levels, **AscendOptimizer** has the highest GM and the best or tied-best tail ratios. On Level 3, the gap is largest: AscendOptimizer reaches 1.89 GM, compared with 1.55 for EvoKernel, 1.47 for CudaForge, 1.45 for OpenEvolve, and 1.38 for BoN. Level 2 is grouped by engineering complexity rather than expected headroom: 33 of its 63 operators belong to the structurally regular `foreach` family, and the official references already contain common vectorization, DMA, buffering, and tiling optimizations. Even so, 9/63 Level 2 operators exceed $1.2\times$ and 4/63 exceed $2.0\times$.

Table 2: **Main performance.** GM is geometric-mean speedup over `cann-ops`; fast$_p$ is the fraction of operators exceeding $p\times$ speedup.

Level	Tasks	Method	GM ↑	fast$_{1.2}$ ↑	fast$_{1.4}$ ↑	fast$_{1.8}$ ↑	fast$_{2.0}$ ↑
level1	25	BoN	1.04	8.00%	4.00%	4.00%	0.00%
		OpenEvolve	1.11	8.00%	8.00%	4.00%	4.00%
		CudaForge	1.03	4.00%	4.00%	4.00%	0.00%
		EvoKernel	1.05	8.00%	8.00%	4.00%	0.00%
		AscendOptimizer	**1.24**	**12.00%**	**12.00%**	**12.00%**	**8.00%**
level2	63	BoN	1.05	6.35%	3.17%	3.17%	1.59%
		OpenEvolve	1.03	4.76%	0.00%	0.00%	0.00%
		CudaForge	1.03	3.17%	1.59%	1.59%	1.59%
		EvoKernel	1.05	1.59%	1.59%	1.59%	1.59%
		AscendOptimizer	**1.10**	**14.29%**	**6.35%**	**6.35%**	**6.35%**
level3	13	BoN	1.38	46.15%	38.46%	23.08%	15.38%
		OpenEvolve	1.45	38.46%	30.77%	23.08%	23.08%
		CudaForge	1.47	30.77%	23.08%	23.08%	15.38%
		EvoKernel	1.55	30.77%	30.77%	30.77%	30.77%
		AscendOptimizer	**1.89**	**53.85%**	**46.15%**	**30.77%**	**30.77%**

Table 3 aggregates the same runs over all 101 operators and places them next to the AscendOptimizer variants. With DeepSeek-V3.2 and the hardware-evaluation budget fixed, AscendOptimizer reaches 1.21 GM, compared with 1.10 for EvoKernel (Zheng et al., 2026).

Table 3: All-operator comparison and component ablation on the 101-operator main benchmark. Every method receives 230 hardware evaluations per operator. The lower block lists independently run AscendOptimizer variants; they are not sequential stages that inherit one another's intermediate results.

Method	Experience	GM	fast$_{1.2}$	fast$_{1.4}$	fast$_{1.8}$	fast$_{2.0}$
BoN	None	1.09	11.88%	7.92%	5.94%	2.97%
OpenEvolve	None	1.10	9.90%	5.94%	3.96%	3.96%
CudaForge	None	1.08	6.93%	4.95%	4.95%	2.97%
EvoKernel	Forward memory	1.10	6.93%	6.93%	5.94%	4.95%
Stage II only (tiling)	None	1.02	2.97%	1.98%	0.00%	0.00%
Stage I w/o Experience	None	1.09	12.87%	8.91%	5.94%	3.96%
Stage I	Rewind	1.14	**18.81%**	**12.87%**	9.90%	7.92%
AscendOptimizer	Rewind + tiling	**1.21**	**18.81%**	**12.87%**	**10.89%**	**9.90%**

4.3 Matched Experience-Source Comparison

To examine the source of retrieved knowledge, we replace Rewind records with a documentation-derived bank (Doc-KB) while keeping the retriever, prompt, profiler, correctness harness, and 230-evaluation budget fixed. Doc-KB is distilled into the same schema from the official AscendC Best Practices documentation. The pre-specified 25-operator subset contains five Level 1, seven Level 2, and all thirteen Level 3 operators.

Table 4: Matched experience-source comparison on the same stratified 25-operator subset.

Source	GM	$\text{fast}_{1.0}$	$\text{fast}_{1.2}$	$\text{fast}_{1.4}$	$\text{fast}_{1.8}$	$\text{fast}_{2.0}$
Doc-KB	1.12	48%	16%	12%	12%	8%
Rewind	**1.24**	**60%**	**28%**	**24%**	**16%**	**12%**

Rewind outperforms Doc-KB at every reported speedup threshold on this subset.

4.4 Rewind Mines Model-Scarce Optimization Knowledge

A cross-operator example shows that Rewind extracts knowledge the model rarely produces from bottleneck text alone (Figure 2).

Mining and transfer. Rewinding `clip_by_value_v2` identifies a loop-consolidation motif by removing it: the strong implementation folds separate tail processing into the main loop, turning `for(i<loopCount-1){...}; tail: CopyIn; Compute; CopyOut;` into a unified `for(i<loopCount)` with `(i==last) ? tailNum : partNum`. Removing it cuts pipeline stalls by 28.6% and raises latency from 56 µs to 75 µs, so the motif is stored with its mechanism, applicability conditions, and rewrite template. The structurally unrelated `upsample_nearest_exact3d` later profiles at 156 µs with its bottleneck localized to the `GatherData` hot path, where separate main-slide and tail-slide handling stalls the vector pipeline. That diagnosis says where the problem is but not how to rewrite it; used as the retrieval query, it returns the `clip_by_value_v2` motif, and the analogous rewrite—one `for(i<totalSlides)` loop with internal conditional dispatch—produces the first major drop, to 145 µs.

Counterfactual: the rewrite is rarely recovered without retrieval. Given the same oracle bottleneck description but no retrieved experience, only 1 of 100 independent attempts reaches 145 µs; adding the official Ascend C Best Practices documentation leaves the hit rate at 1/100. Without retrieval the optimizer usually proposes broader but ineffective edits such as *Vector Pipeline Barrier Reduction* or generic scalar cleanup.

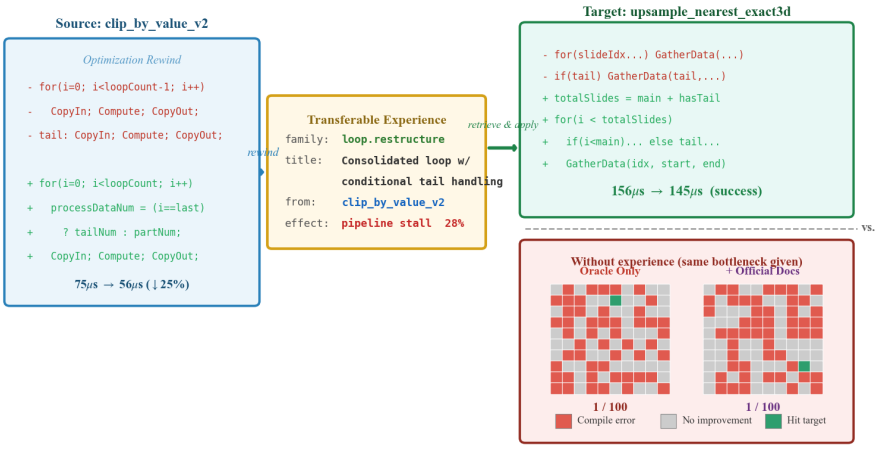

Figure 2: Cross-operator transfer for `upsample_nearest_exact3d`. A loop-consolidation pattern mined from `clip_by_value_v2` transfers to the target operator; without retrieval, oracle bottleneck guidance rarely finds the same rewrite.

Both the full system and the no-experience ablation identify related bottlenecks, including scalarized gather or offset processing and poor vector utilization. Rewind-derived retrieval supplies the missing step from diagnosis to an *actionable structural transformation*: the loop-consolidation rewrite that unguided LLM generation and static vendor documentation rarely produce. Optimization Rewind thereby turns latent structure in existing kernels into explicit, retrievable knowledge that direct optimization rarely recovers in this case.

4.5 Component Analysis

The lower block of Table 3 isolates each component. Stage I is the strongest single component at 1.14 GM. Removing experience lowers GM to 1.09 and $\text{fast}_{2.0}$ from 7.92% to 3.96%, putting the variant on par with the experience-free baselines, while Stage II alone reaches 1.02 GM on the original kernels. The full pipeline reaches 1.21 GM and 9.90% $\text{fast}_{2.0}$.

A single-operator example illustrates the sequential optimization process: for `matmul_all_reduce`, kernel refinement achieves 1.35× speedup, and subsequent host-side tiling optimization raises it to 2.00× over the same reference while keeping the kernel fixed (Appendix C).

4.6 Transfer Beyond the Original Bank and Second-Model Sensitivity

Frozen-bank transfer. The 101-operator experiment is transductive, so we separately test a frozen bank on eight loss operators from the official `cann/ops-nn/loss` repository. None appears in the original benchmark or contributes to the bank, and no held-out result is written back. Selection uses only pre-optimization compilability, correctness, and successful profiling. Both arms use DeepSeek-V4-Pro, the same inputs and harness, and the same total evaluation budget. The Plain arm runs the same optimization workflow with Rewind and experience retrieval disabled. Its GM is 1.02, compared with 1.06 for AscendOptimizer; the paired GM ratio is 1.04. Figure 3 shows the per-operator results.

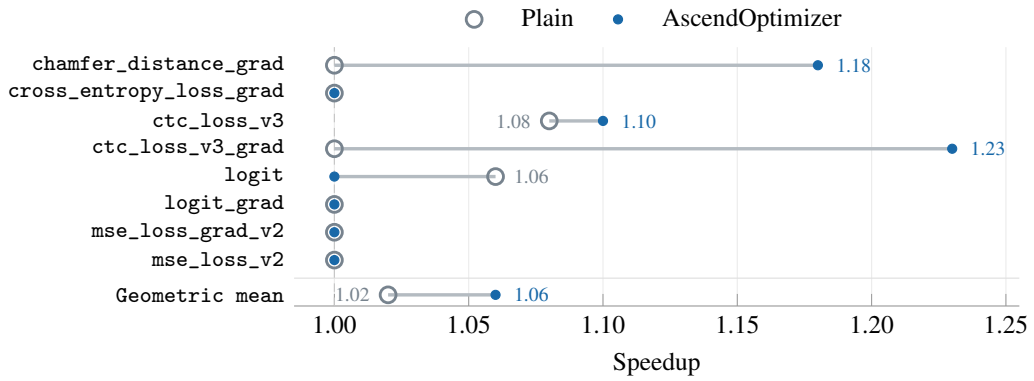

Figure 3: Frozen-bank transfer on eight held-out operators. Gray circles (Plain) and blue dots (AscendOptimizer) show speedups over the same reference implementation; overlapping markers indicate ties. The bottom row reports the geometric mean.

Second-model sensitivity. We also repeat the with/without-Rewind comparison using DeepSeek-V4-Pro on the pre-specified 25-operator subset from Table 4. Both methods receive 90 hardware evaluations: Optimization-only uses all 90 online, while AscendOptimizer uses 15 for Rewind and 75 online. AscendOptimizer raises $\text{fast}_{1.0}$ from 36% to 68%, $\text{fast}_{1.2}$ from 20% to 28%, and $\text{fast}_{1.4}$ from 16% to 24%; Table 5 reports all thresholds.

5 Conclusion and Limitations

Conclusion. Optimization Rewind extracts reusable optimization knowledge from existing high-performance kernels through controlled de-optimization and hardware validation. Integrated into AscendOptimizer, it helps achieve a 1.21× geometric-mean speedup on Ascend. Cross-operator results demonstrate that mined experience supplies structural rewrites rarely generated from bottleneck descriptions or vendor documentation alone, supporting optimized code as a source of transferable optimization knowledge.

Limitations. AscendOptimizer requires a correct, compilable initial implementation, and Rewind's knowledge coverage depends on the quality and diversity of available optimized kernels. Transferred patterns require validation under new code, inputs, or hardware; generalization beyond Ascend remains unverified.

Reproducibility Statement

The operator implementations used in our evaluation are publicly available in an anonymized repository.[3] Section 4.1 documents the hardware platform (Ascend 910B2, CANN 8.3), the LLMs used (DeepSeek-V3.2 for main experiments, DeepSeek-V4-Pro for sensitivity), the embedding model (`text-embedding-3-small`), the 230-evaluation-per-operator budget, and the correctness harness.

AI Use Statement

We used DeepSeek-V3.2 and DeepSeek-V4-Pro as core components of AscendOptimizer for Optimization Rewind candidate generation, experience distillation, retrieval-guided kernel rewriting, and host-side tiling mutation. LLM-based tools were also used for research ideation and execution, literature retrieval, checking mathematical derivations and claims, and polishing the manuscript. All LLM-assisted outputs, citations, mathematical claims, and manuscript content were reviewed and verified by the authors, who take full responsibility for the final paper.

References

Xin Ai, bing zhang, Qiange Wang, Yanfeng Zhang, Hao Yuan, Shufeng Gong, and Ge Yu. NeutronAscend: Optimizing GNN training with Ascend AI processors. *ACM Transactions on Architecture and Code Optimization*, 22(4):1–26, 2025.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

Martin Andrews and Sam Witteveen. GPU Kernel Scientist: An LLM-driven framework for iterative kernel optimization. *arXiv preprint arXiv:2506.20807*, 2025.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI 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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness. *the 36th International Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[3] `https://anonymous.4open.science/r/ascendOptimizer-bench-7461/README.md`

Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. EvoEngineer: Mastering automated CUDA kernel code evolution with large language models. *arXiv preprint arXiv:2510.03760*, 2025.

Stephen Hines, Prasad Kulkarni, David Whalley, and Jack Davidson. Using de-optimization to re-optimize code. In *Proceedings of the 5th ACM international conference on Embedded software*, pp. 114–123, 2005.

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.

Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. PRAGMA: A profiling-reasoned multi-agent framework for automatic kernel optimization. *arXiv preprint arXiv:2511.06345*, 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, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025b.

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*, 2025c.

Shiyang Li, Zijian Zhang, Winson Chen, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Stitchcuda: An automated multi-agents end-to-end gpu programing framework with rubric-based agentic reinforcement learning. *arXiv preprint arXiv:2603.02637*, 2026.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025d.

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.

Salli Moustafa. Accelerating sparse matrix-matrix multiplication with the Ascend AI core. In *The 5th Workshop on Accelerated Machine Learning (AccML)*, 2023.

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.

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.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. *The 38th Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

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.

Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L2: Surpassing cuBLAS performance for matrix multiplication through reinforcement learning. *arXiv preprint arXiv:2512.02551*, 2025.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *The 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

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.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel AI agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194*, 2025a.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, et al. TileLang: A composable tiled programming model for AI systems. *arXiv preprint arXiv:2504.17577*, 2025b.

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.

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*, 2025a.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025b.

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.

Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. In *The 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–13, 2022.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In *Proceedings of the 9th Conference on Machine Learning and Systems (MLSys)*, 2026. URL https://arxiv.org/abs/2511.15915. arXiv:2511.15915.

Jiahui Zhang, Yusen Luo, Abrar Anwar, Sumedh Anand Sontakke, Joseph J Lim, Jesse Thomason, Erdem Biyik, and Jesse Zhang. ReWiND: Language-guided rewards teach robot policies without new demonstrations. *arXiv preprint arXiv:2505.10911*, 2025a.

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*, 2025b.

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, and Muning Wen. Towards cold-start drafting and continual refining: A value-driven memory approach with application to NPU kernel synthesis. *arXiv preprint arXiv:2603.10846*, 2026. URL https://arxiv.org/abs/2603.10846.

Yuhang Zhou, Zhibin Wang, Guyue Liu, Shipeng Li, Xi Lin, Zibo Wang, Yongzhong Wang, Fuchun Wei, Jingyi Zhang, Zhiheng Hu, et al. Squeezing operator performance potential for the ascend architecture. In *The 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025a.

Yuhang Zhou, Zibo Wang, Zhibin Wang, Ruyi Zhang, Chen Tian, Xiaoliang Wang, Wanchun Dou, Guihai Chen, Bingqiang Wang, Yonghong Tian, et al. Accelerating model training on Ascend chips: An industrial system for profiling, analysis and optimization. In *2025 USENIX Annual Technical Conference (USENIX ATC)*, 2025b.

A Additional Experimental Details

Final benchmark size. After benchmark filtering and shape adjustments, we retain 101 operators for the main benchmark experiments. The held-out study uses eight operators outside the main benchmark and its experience bank. The second-model study uses a pre-specified 25-operator subset of the main benchmark.

Benchmark construction and filtering. We start from the `cann-ops` AscendC repository and use the provided implementations as references. We retain operators only when they compile, execute, pass numerical checks against CPU references, and have stable baseline latency under repeated profiling. The released harness uses `gen_data.py` to generate inputs and PyTorch/NumPy golden outputs, `main.cpp` to invoke ACLNN, `verify_result.py` for absolute and relative error checks, and `run.sh` to build, execute, verify, and profile each candidate.

Input-shape adjustment. For workloads whose timing is unstable at the original size, we increase the input shape while preserving operator semantics. The reference and every method use the same adjusted shape, inputs, and correctness oracle. We accept a benchmark configuration only after repeated profiling shows stable latency.

Why AOE/OPAT is not included in the matched-budget comparison. AOE/OPAT tunes supported operators through CANN's compilation and execution workflow, with selected strategies stored in an operator knowledge base.[4] Our benchmark instead evaluates edits to existing native AscendC `op_kernel` and `op_host` implementations under a fixed correctness harness and hardware-evaluation budget. A paired comparison would require mapping these workloads to supported OPAT execution paths while matching input shapes, numerical checks, and evaluation accounting. We have not established this mapping for the present benchmark, so we do not include AOE/OPAT in the matched-budget comparison.

Evaluation paradigm and data overlap. The offline Rewind phase is benchmark-wide: each of the 101 main operators contributes 20 Rewind evaluations before the validated trajectories are aggregated into the shared bank. The main result is therefore transductive system autotuning on a fixed workload suite and hardware/software stack. No model weights are updated. MultiKernelBench supplies functional references for generation and does not supply the AscendC implementations or test artifacts used here; seven generic operator names match exactly, but the implementations and data do not overlap. The eight held-out operators in Figure 3 are outside both the 101-operator benchmark and its frozen bank.

Second-model sensitivity. DeepSeek-V4-Pro is used for Rewind generation, experience distillation, and online rewriting in the AscendOptimizer arm. Table 5 reports threshold coverage.

Table 5: Second-model sensitivity in threshold coverage: matched 90-evaluation comparison with DeepSeek-V4-Pro on the pre-specified 25-operator subset.

Method	R/O	$\text{fast}_{1.0}$	$\text{fast}_{1.2}$	$\text{fast}_{1.4}$	$\text{fast}_{1.8}$	$\text{fast}_{2.0}$
Optimization-only	0/90	36%	20%	16%	16%	16%
AscendOptimizer	15/75	**68%**	**28%**	**24%**	**20%**	**20%**

Operator list by level. The following table gives the full operator assignment under the engineering-complexity grading defined in Section 4.1.

[4] `https://www.hiascend.com/document/detail/zh/canncommercial/80RC2/devaids/auxiliarydevtool/aoe_16_001.html`

Level	Operator
Level 1 (25)	add_custom, angle_v2, arange, cast, cos, equal, eye, fill, greater_equal, heaviside, icamax, icamin, is_inf, isamin, less, less_equal, logical_not, logical_or, muls, neg, reciprocal, reshape, rsqrt, sqrt, trunc
Level 2 (63)	foreach_acos, foreach_add_list, foreach_add_scalar, foreach_add_scalar_list, foreach_addcdiv_scalar, foreach_addcmul_scalar, foreach_addcmul_scalar_list, foreach_asin, foreach_copy, foreach_div_list, foreach_div_scalar, foreach_erf, foreach_erfc, foreach_exp, foreach_lerp_scalar, foreach_log2, foreach_maximum_scalar, foreach_minimum_scalar, foreach_mul_list, foreach_mul_scalar, foreach_neg, foreach_pow_scalar, foreach_pow_scalar_and_tensor, foreach_reciprocal, foreach_round_off_number, foreach_sigmoid, foreach_sign, foreach_sinh, foreach_sqrt, foreach_sub_scalar, foreach_tan, foreach_tanh, foreach_zero_inplace, ge_glu_grad_v2, ge_glu_v2, ge_glu_v3, gelu_quant, swi_glu, swi_glu_grad, pad_v3_grad_replication, reflection_pad3d_grad, strided_slice_assign_v2, upsample_bilinear2d_grad, upsample_nearest_exact3d, addcdiv, addcmul, clip_by_value_v2, fast_gelu_grad, gcd, gelu, gelu_grad, lerp, lin_space, mul_sigmoid, swish, tril, triu, adaptive_avg_pool3d_grad, avg_pool3_d, add_layer_norm_grad, add_rms_norm_dynamic_quant, add_rms_norm_quant, group_norm_swish
Level 3 (13)	motion_compensation, upsample_bilinear2d_aa, upsample_bilinear2d_aa_backward, upsample_trilinear3d_backward, add_sigmoid_mul_reduce_sum_d, bev_pool, flash_attention_score_with_large_head_dim, matmul_all_reduce, matmul_api_constant, scatter_reduce, strideslice_neg_concat_v2, complex_mat_mul, foreach_non_finite_check_and_unscale

B Method Details and Experience Bank Analysis

This section provides additional analysis of the optimization experience bank together with detailed method snippets.

B.1 Rewind as Local One-Factor-at-a-Time Screening

Following the problem formulation in Section 3, an operator is represented as $\mathcal{O} = \langle \mathcal{T}, \mathcal{K}, \mathcal{S} \rangle$. During the offline rewind phase, we fix the host-side tiling function $\mathcal{T}$ and focus on the kernel-side implementation space.

Let $\mathbb{K}_{\mathcal{S}}$ denote the set of kernel implementations that realize the operator semantics under static attributes $\mathcal{S}$ and pass correctness validation under the current execution harness. For a fixed tiling function $\mathcal{T}$, we define the hardware utility of a kernel as

$$U_{\mathcal{T}}(\mathcal{K}) = -\mathcal{L}(\mathcal{K} \mid \mathcal{T}), \tag{4}$$

where $\mathcal{L}(\mathcal{K} \mid \mathcal{T})$ is the measured latency introduced in Eq. (1); higher utility corresponds to lower latency.

We view a kernel as activating a set of recognizable optimization motifs $A \subseteq \Omega$, where Ω is a finite catalog of named motifs such as double buffering, vectorized data movement, loop-tail consolidation, synchronization elimination, or redundant-write removal. Equivalently, A is represented by a binary motif vector $z \in \{0,1\}^{|\Omega|}$ with $z_m = 1$ iff motif m is active. Multiple kernels in $\mathbb{K}_{\mathcal{S}}$ may share the same active set A; we therefore write $\mathcal{K}_A \in \mathbb{K}_{\mathcal{S}}$ for any representative implementation whose active motif set equals A, with the choice of representative determined by the inverse agent and the validation harness.

A rewind action for motif m is a *partial* map

$$r_m : \mathbb{K}_{\mathcal{S}} \rightharpoonup \mathbb{K}_{\mathcal{S}}, \tag{5}$$

left undefined (written $r_m(\mathcal{K}) = \perp$) whenever the proposed de-optimized candidate fails compilation, fails correctness validation, or cannot be isolated as a semantic-preserving rewrite of motif m. When $\mathcal{K} = \mathcal{K}_A$ with $m \in A$, a successful rewind yields a neighboring representative

$$r_m(\mathcal{K}_A) \;=\; \mathcal{K}_{A\setminus\{m\}}. \tag{6}$$

The measured effect of this single-motif removal is

$$\widehat{\Delta}_m(\mathcal{K}_A;\mathcal{T}) \;=\; U_{\mathcal{T}}(\mathcal{K}_A) - U_{\mathcal{T}}(\mathcal{K}_{A\setminus\{m\}}) \;=\; \mathcal{L}(\mathcal{K}_{A\setminus\{m\}} \mid \mathcal{T}) - \mathcal{L}(\mathcal{K}_A \mid \mathcal{T}), \tag{7}$$

so $\widehat{\Delta}_m > 0$ indicates that motif m contributes positively to performance at the basepoint $\mathcal{K}_A$. Equation (7) is a local one-factor-at-a-time (OAT) finite difference on the discrete motif space, evaluated at a single strong-implementation basepoint per operator. This is in the spirit of elementary-effect screening in sensitivity analysis, with two practical differences: (i) the perturbation flips a binary motif variable rather than shifting a continuous input, and (ii) we use the local difference as a *validated screening signal*, not as an estimator of a global elementary-effect distribution.

We retain a rewind-derived experience only when

$$\widehat{\Delta}_m(\mathcal{K}_A;\mathcal{T}) > \tau_{\text{noise}}, \tag{8}$$

where τ_{noise} is calibrated above profiling noise on the target hardware. Candidates with $r_m(\mathcal{K}_A) = \perp$ or with effects below τ_{noise} are discarded. Each accepted pair $(\mathcal{K}_A, \mathcal{K}_{A\setminus\{m\}})$, together with its validated effect $\widehat{\Delta}_m$, is the raw material from which the structured experience tuple $\mathcal{M}$ in Eq. (3) is distilled. Thus the experience bank stores hardware-validated local screening signals rather than assumed optimization rules.

B.2 Optimization Algorithms

Algorithm 1 Shared Experience Bank Construction via Optimization Rewind

Require: Benchmark operator set $\mathcal{O}$, rewind budget B_r
Output: Shared experience bank $\mathcal{B}$

$\mathcal{B} \leftarrow \emptyset$
for each operator $o \in \mathcal{O}$ **do**
  $\mathcal{K}_{\text{curr}} \leftarrow$ strongest available implementation of o
  for $b = 1$ to B_r **do**
    propose semantically meaningful rewind candidate $\tilde{\mathcal{K}}$
    if $\tilde{\mathcal{K}}$ compiles, passes correctness, and is slower than $\mathcal{K}_{\text{curr}}$ **then**
      distill tuple $\mathcal{M}$ from $(\mathcal{K}_{\text{curr}}, \tilde{\mathcal{K}})$ and profiling deltas
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathcal{M}\}$; $\mathcal{K}_{\text{curr}} \leftarrow \tilde{\mathcal{K}}$
    end if
  end for
end for
return $\mathcal{B}$

Algorithm 2 Per-Operator Online Optimization with Kernel Refinement and Tiling Search

Require: Target operator $(\mathcal{T}^{(0)}, \mathcal{K}^{(0)}, \mathcal{S})$, shared bank $\mathcal{B}$, kernel steps S, tiling steps U
Output: Optimized pair $(\mathcal{T}^{\dagger}, \mathcal{K}^{\dagger})$

$\mathcal{T}_{\text{curr}} \leftarrow \mathcal{T}^{(0)}$, $\mathcal{K}_{\text{curr}} \leftarrow \mathcal{K}^{(0)}$
for $s = 1$ to S **do**
  $q \leftarrow \textsc{DiagnoseBottleneck}(\mathcal{T}_{\text{curr}}, \mathcal{K}_{\text{curr}})$
  $\{\mathcal{M}_i\}_{i=1}^{k} \leftarrow \textsc{RetrieveTopK}(\mathcal{B}, q)$
  $\tilde{\mathcal{K}} \leftarrow \textsc{RewriteKernel}(\mathcal{K}_{\text{curr}}, \{\mathcal{M}_i\}_{i=1}^{k})$
  if $\tilde{\mathcal{K}}$ compiles, passes correctness, and improves $\mathcal{L}(\tilde{\mathcal{K}} \mid \mathcal{T}_{\text{curr}})$ **then**
    $\mathcal{K}_{\text{curr}} \leftarrow \tilde{\mathcal{K}}$
  end if
end for
for $u = 1$ to U **do**
  $\tilde{\mathcal{T}} \leftarrow \textsc{MutateTiling}(\mathcal{T}_{\text{curr}}, \mathcal{S})$
  if $\tilde{\mathcal{T}}$ compiles, passes correctness, and improves $\mathcal{L}(\mathcal{K}_{\text{curr}} \mid \tilde{\mathcal{T}})$ **then**
    $\mathcal{T}_{\text{curr}} \leftarrow \tilde{\mathcal{T}}$
  end if
end for
return $(\mathcal{T}_{\text{curr}}, \mathcal{K}_{\text{curr}})$

B.3 Experience Bank Analysis

Compared with official Ascend C best practices, the experience bank differs mainly in representation and scope. Official documentation is principle-level and written for human developers; the experience bank stores concrete, reusable optimization episodes tied to actual bottlenecks and code rewrites, so it can be retrieved and applied directly during automated optimization. In other words, the documentation says *what is generally advisable*, while the experience bank records *what specific transformation worked under what condition*.

Retrieval predominantly draws on experience from other operators. Among 1,025 parseable experience-guided mutations, 67 (6.5%) retrieve records contributed by the current operator, while 958 (93.5%) retrieve records from other operators. Of these 1,025 mutations, 416 (40.6%) involve retrieval across operator categories.

Table 6 summarizes whether the bank stays within the scope of official guidance or extends beyond it. The bank certainly covers many motifs that are consistent with the documentation, such as tiling adjustment, DMA consolidation, double buffering, and vectorized reduction. However, it also goes beyond the documentation in several ways: it preserves operator-specific control-flow rewrites, scalar-index simplifications, early-exit or search-pruning tricks, and specialized SIMD paths that are useful in practice but are not usually stated as general vendor best practices.

Table 6: How the experience bank differs from and extends official Ascend C best practices

Aspect	Official Ascend C Best Practices	Experience Bank in This Work
Knowledge form	Principle-level guidance for human developers	Concrete optimization episodes with code-level rewrites and applicability conditions
Primary role	Explain common optimization principles	Support direct retrieval and transfer during automated operator optimization
Covered and aligned with docs	Tiling, DMA efficiency, double buffering, vectorized reduction	Recovers and operationalizes these documented principles in reusable optimization records
Beyond documented scope	Usually does not enumerate long-tail operator-specific rewrites	Includes loop restructuring, scalar index simplification, early-exit/search pruning, fine-grained synchronization tuning, and specialized SIMD paths
Granularity of reuse	Broad recommendations	Mechanism- and rewrite-level transferable patterns

B.4 Method Details

This subsection gives two implementation-level views of the optimizer. Figure 4 shows how Stage II turns an operator and its attributes into an evolvable host-side tiling template. The mutation interface is defined by the shape-derived features, block/core resource parameters, and output fields rather than by the surrounding C++ syntax. Figure 5 shows the complementary Stage I representation: a rewind-validated experience record that stores the retrieval key, causal mechanism, hardware evidence, and transfer constraints for a concrete code rewrite.

Method details: base tiling function $\mathcal{T}_{\text{base}}$

```
// FOCUS evolvable region exposed to Stage II mutation/search
// # evolve_tiling_block_start
std::map<std::string, int64_t> AdaptiveTilingStrategy(const
    std::vector<int64_t>& xShape,
                                                       const
    std::vector<int64_t>& srcShape,

    int64_t dim,
                                                       const
    std::string& reduceType,
                                                       bool
    includeSelf,

    int64_t inputBytes)
{
    constexpr int64_t CORENUM = 1;
    constexpr int64_t BLOCK_BYTES_SIZE = 32;

    std::map<std::string, int64_t> params;
    // Guard clauses keep invalid candidates outside the
    search space.
    if (xShape.empty() || srcShape.empty()) {
        return params;
    }

```

```
    if (dim < 0 || dim >= static_cast<int64_t>(xShape.size()) ||
        dim >= static_cast<int64_t>(srcShape.size())) {
        return params;
    }

    // 1) Shape-derived features: fixed facts used by candidate tiling rules.
    int64_t batchSize = 1;
    for (int64_t i = 0; i < dim; ++i) {
        batchSize *= xShape[i];
    }

    const int64_t dimSizeX = xShape[dim];
    const int64_t dimSizeSrc = srcShape[dim];

    int64_t strideSize = 1;
    for (int64_t i = dim + 1; i < static_cast<int64_t>(xShape.size()); ++i) {
        strideSize *= xShape[i];
    }

    int64_t reduction = 0;
    if (reduceType == "sum") {
        reduction = 0;
    } else if (reduceType == "prod") {
        reduction = 1;
    } else if (reduceType == "mean") {
        reduction = 2;
    } else if (reduceType == "amax") {
        reduction = 3;
    } else if (reduceType == "amin") {
        reduction = 4;
    }

    // 2) Resource-granularity knobs: formulas/constants can be mutated.
    int64_t blockSize = (inputBytes > 0) ? (BLOCK_BYTES_SIZE / inputBytes) : BLOCK_BYTES_SIZE;
    if (blockSize <= 0) {
        blockSize = 1;
    }

    const int64_t blockNum = (strideSize + blockSize - 1) / blockSize;
    const int64_t coreNum = std::min<int64_t>(CORENUM, blockNum == 0 ? CORENUM : blockNum);

    const bool specialCase = (batchSize == 1 && dimSizeX == dimSizeSrc && !includeSelf &&
                              reduction == 4 && inputBytes == 4);
    const int64_t blockDim = specialCase ? coreNum : 0;

    // 3) Search interface: returned fields are consumed by host-side tiling.
    params["batchSize"] = batchSize;
```

```
    params["dimSizeX"] = dimSizeX;
    params["dimSizeSrc"] = dimSizeSrc;
    params["strideSize"] = strideSize;
    params["reduction"] = reduction;
    params["includeSelf"] = includeSelf ? 1 : 0;
    params["blockSize"] = blockSize;
    params["blockNum"] = blockNum;
    params["blockDim"] = blockDim;
    params["specialCaseFlag"] = specialCase ? 1 : 0;

    return params;
}
// # evolve_tiling_block_end
```

Figure 4: Base tiling function $\mathcal{T}_{\text{base}}$ with evolution markers, synthesized from the operator code and attributes $\mathcal{S}$. The highlighted/comments indicate what readers should inspect: shape-derived facts, mutable block/core resource choices, and returned `params` fields that form the Stage II search interface.

Figure 4 is therefore a template boundary rather than a hand-written final tiling policy. The optimizer keeps the input validation and semantic features stable, then searches over the resource and dispatch choices exposed through `blockSize`, `blockNum`, `coreNum`, `blockDim`, and the returned parameter map. This makes the host-side search concrete enough to compile and profile, while still leaving the performance-sensitive policy choices open to mutation.

Method details: rewind-derived experience tuple

```
{
  "op": "eye",
  "experience_type": "performance_recovery",
  "explore_mode": "degrade",

  // FOCUS 1 Retrieval key: groups this rewrite with
    related cases.
  "canonical_family": "kernel.memory.scatter",
  "optimization_title": "Eliminate redundant scattered zero
    writes",
  "mechanism_understood": true,

  // FOCUS 2 Mechanism: why the fast version is faster.
  "root_mechanism": "Output is already zero-initialized;
    only diagonal values need explicit writes.",
  "causal_chain": "remove scattered zero writes -> fewer L2
    misses -> higher memory bandwidth -> shorter vector
    critical path",
  "slow_duration_us": 925024.56,
  "fast_duration_us": 1193.78,
  "improvement_direction": "degraded_is_slow",

  // FOCUS 3 Hardware evidence: the rewrite is validated by
    profiling.
  "profile_evidence": [
```

```
    "cache_l2.aiv_read_hit_rate_pct: 0.02 -> 87.5 (+437400.00%)",
    "critical_path.aiv_vector0_time_us_mean: 1193.17 -> 925006 (+77424.98%)",
    "latency.task_duration_us: 1193.78 -> 925025 (+77386.76%)"
  ],
  "affected_metric_groups": ["cache_l2", "critical_path", "latency"],

  // FOCUS 4 Transfer contract: when to reuse or avoid this rule.
  "retrieval_keywords": ["diagonal matrix", "scattered writes", "pre-initialized buffer"],
  "reusable_when": [
    "Eye-like diagonal initialization",
    "Output buffer is guaranteed to be zero-initialized"
  ],
  "avoid_when": [
    "Output buffer is not known to be zero-initialized"
  ],
  "reuse_guidance": [
    "Remove scattered SetValue calls that write non-diagonal zeros",
    "Keep only yGm.SetValue(index, 1) for diagonal positions"
  ],

  // Code evidence: rewind adds the slow behavior, so the inverse is the optimization.
  "code_diff_excerpt": [
    "  yGm.SetValue(index, 1);",
    "+ for (int32_t k = 0; k < num_columns; ++k) {",
    "+   if (k != i) {",
    "+     yGm.SetValue(i * num_columns + k, 0);",
    "+   }",
    "+ }  // redundant scattered zero writes"
  ]
}
```

Figure 5: Compact rewind-derived optimization record for `eye`. The highlighted fields show how a code difference becomes reusable knowledge: a retrieval family, a causal mechanism, hardware-profile evidence, transfer constraints, and a short diff excerpt identifying the redundant scattered zero writes.

Figure 5 also clarifies why rewind is useful for knowledge construction. The degraded version adds scattered writes to non-diagonal entries; profiling verifies that this edit destroys cache locality and dominates latency. Reversing the validated degradation gives a concrete optimization rule that can be retrieved for later Eye-like kernels, together with explicit conditions under which the rule should not be applied.

C Stage-wise Kernel and Tiling Optimization Case Study

We examine `matmul_all_reduce` to answer what kernel refinement and tiling search do in the optimized system. Stage I rewrites the kernel execution path itself, achieving $1.35\times$ speedup over the reference. The main edits are structural: the kernel separates full-block work from tail cases, restructures reduction-style loops, adjusts pipeline depth and buffering decisions, and exposes more vector-core parallelism with less synchronization overhead. In other words, Stage I changes how the hot path runs inside the kernel. Starting from that Stage I-optimized kernel, Stage II adjusts host-side tiling and resource policies, further increasing speedup over the same reference to $2.00\times$. At this stage, the focus is no longer kernel control flow but launch and resource selection: the tiler chooses `tileM` using matrix shape, compute intensity, transpose flags, and all-reduce communication cost; chooses `blockDim` under load-balance and communication penalties; sizes UB/L1 buffers and workspace hierarchically; and emits pipeline, prefetch, and double-buffer hints. Figure 6 connects this trajectory to representative diff hunks and shows the separation clearly: kernel refinement changes the internal execution structure, while tiling search tunes the surrounding tiling and resource policy for that refined kernel.

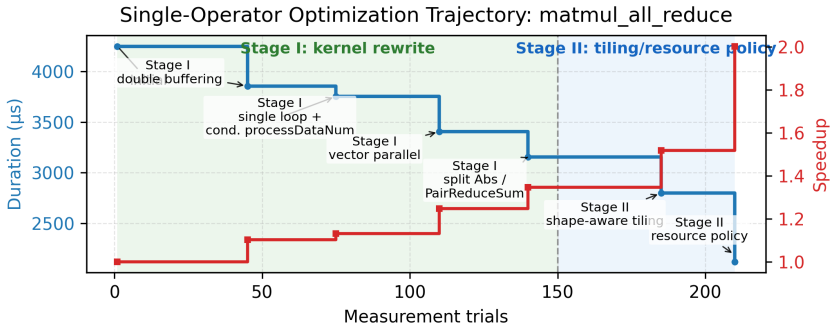

Figure 6: Trajectory for `matmul_all_reduce`: Stage I improves kernel structure, achieving $1.35\times$ speedup; Stage II improves tiling and resource policy, further increasing speedup to $2.00\times$. Both speedups are measured against the original reference implementation.

Representative code changes from the same case study are shown below.

Stage I: split loops

```
- for (i=0; i<blockCnt; i++) {
-  curM=isLastM?mTail:baseM;
-  curN=isLastN?nTail:baseN;
-  mm.SetSingleShape(curM,curN,K);
-  mm.Iterate(); }
+ while (full_block) {
+  mm.SetSingleShape(baseM,baseN,K);
+  mm.Iterate(); processed++; }
+ while (M_tail) {
+  mm.SetSingleShape(mTail,baseN,K);
+  mm.Iterate(); processed++; }
+ while (N_tail) {...}
+ while (MN_tail) {...}
```

Stage II: tiling

```
- tileM=ComputeTileM(m,cores,
-    defaultTileM,128);
- // M-only size thresholds
+ tileM=ComputeTileM(m,n,k,
+    cores,isTransA,isTransB,
+    dtypeSize);
+ intensity=2*m*n*k/totalBytes;
+ if (memory_bound)
+  baseTile=small_reuse(m);
+ else if (compute_bound)
+  baseTile=large_tile(m);
+ comm=m*n*dtypeSize/cores;
+ baseTile=adjust_allreduce(comm);
```

Stage II: resource policy

```
- blockDim=min(tiles,cores);
- if (matSize<10000)
-  blockDim=min(blockDim,4);
- // fixed thresholds only
+ for (d=1; d<=maxCores; d++) {
+  bal=1-extra/tiles;
+  util=tiles/d;
+  comm=1/(1+.05*log2(d+1));
+  score=bal*util*comm*cache;
+  if (score>best) bestDim=d; }
+ blockDim=bestDim;
```