REPAIR: LEARNING WHAT TO OPTIMIZE FOR EFFICIENT ASCEND KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

High-performance kernels are essential for fully exploiting modern accelerators. Large language models (LLMs) offer a promising route to automated kernel generation and optimization, but struggle with the specialized requirements of Ascend kernels. Existing end-to-end approaches, including reinforcement-learning-based and agent-driven optimization, yield limited performance gains in this setting. We argue that a key challenge lies in coupling optimization decision-making with code implementation, requiring models to jointly determine what to optimize and how to implement the corresponding changes. To address this challenge, we propose REcursive kernel oPtimizAtion by Identification and Revision (REPAIR), a recursive optimization framework that explicitly decouples these two responsibilities. At each iteration, Identification selects an actionable optimization direction without modifying the kernel, while Revision takes this decision as input and implements the corresponding code changes. Building on this separation, we leverage publicly available agents to generate data for both stages and train a compact LLM with reinforcement learning specifically for Identification. This design focuses RL on learning which optimization to pursue, rather than jointly learning optimization decisions and complex code modifications, allowing the compact model to specialize in decision-making while delegating implementation to Revision. Experiments on 253 distinct Ascend kernels from MultiKernelBench show that REPAIR achieves up to 34.8× speedup over the original kernel code, outperforming the 9.5x for the baseline Codex agent.

1 INTRODUCTION

High-performance kernels are essential for translating accelerator capabilities into efficient deep learning workloads, with performance shaped by both compute throughput and data movement (Williams et al., 2009; Dao et al., 2022). Compiler and auto-scheduling systems such as TVM and Ansor automate hardware-specific tensor program optimization (Chen et al., 2018; Zheng et al., 2020). Large language models (LLMs) offer a promising way to reduce the expert effort required for kernel development (Chen et al., 2021; Ouyang et al., 2025), but their effectiveness depends on the target programming ecosystem (Wen et al., 2025). This challenge is particularly pronounced on Ascend neural processing units (NPUs), where Ascend C programming requires explicit coordination of data movement, on-chip buffers, heterogeneous compute units, and synchronization. These specialized requirements are compounded by limited publicly available code examples, which restrict opportunities for general-purpose models to acquire the relevant programming and optimization knowledge (Wen et al., 2026b). Consistent with these difficulties, MultiKernelBench reports substantially lower functional correctness for direct Ascend C generation than for CUDA under its evaluated model and prompting settings (Wen et al., 2025). These findings motivate domain-aware approaches to Ascend kernel optimization rather than assuming that general coding capabilities transfer directly to this specialized setting.

Reinforcement learning (RL) has been used to improve language-model instruction following with human feedback (Ouyang et al., 2022) and code generation with execution-based feedback (Le et al., 2022). Recent work has explored iterative feedback and RL to improve kernel generation and optimization. Kevin incorporates successive rounds of CUDA kernel generation, execution, and refinement into multi-turn RL training (Baronio et al., 2026). For Ascend, AscendKernelGen combines domain-specific data with supervised fine-tuning and RL using execution feedback to improve kernel

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.

synthesis (Cao et al., 2026). Agent-driven approaches provide another route: CudaForge separates a Coder from a Judge that analyzes hardware feedback and guides subsequent revisions (Zhang et al., 2025). Despite these advances, learning to optimize an existing Ascend kernel presents a challenge beyond generating a valid implementation. In end-to-end code-level optimization, the policy must both choose a useful transformation and implement it correctly. A promising optimization decision may therefore fail to produce a useful candidate because of implementation errors, obscuring the value of the decision itself. We suggest that this *coupling is an important obstacle to effective optimization*, particularly for compact models. Although agent role separation already offers a useful organizational mechanism, our focus is to make Ascend-specific optimization decision-making an explicit, separately trainable component rather than an implicit part of code generation.

In this work, we suggest that *identifying the valuable optimization directions could be viewed as decision-making problem*, and propose REcursive kernel oPtimizAtion by Identification and Revision (REPAIR) method. REPAIR consists of two key modules: (1) Identification modules examines the complete Host/Kernel source for a fixed input shape, reconstructs the computation, dataflow, and task ownership, and uses source-level estimates of launch-wide hardware work to formulate an actionable optimization plan. (2) Revision modules implements the selected plan while preserving its specified correctness constraints, rather than independently introducing new optimizations. A candidate becomes the starting point for the next iteration only after passing correctness checks and satisfying the measured Task Duration improvement criterion. Otherwise, the last best version is preserved. We implement both stages with publicly available agents to generate kernel optimization data and train an LLM with RL specifically for Identification. This design concentrates learning on determining optimization directions, rather than complex code modifications. To further improve the performance of REPAIR, we can *regard the identification process as a decision making problem* from the optimization plan set, which can be easily combined with reinforcement learning.

Our contributions are summarized as follow. First, we propose a novel idea of separation for recursive Ascend kernel optimization. While recent works (Wen et al., 2026b; Cao et al., 2026) focus on enabling LLMs to generate efficient Ascend kernels, we attempt to capture the decidable part in the process. Second, we provide a practical method, REPAIR, using publicly available agents for data generation and RL to specialize a compact model in Identification, without requiring that model to generate the revised kernel. Third, we evaluate REPAIR on more than 200 distinct Ascend kernels from MultiKernelBench, a public benchmark supporting kernel evaluation across multiple accelerator platforms (Wen et al., 2025). In this evaluation, REPAIR achieves up to 34.8× speedup over the original kernels, clearly outperforming the baseline Codex agent, demonstrating the effectiveness of the framework in the Ascend kernel optimization.

2 Related Works

LLM-based code generation. Transformer architectures (Vaswani et al., 2017) and large-scale autoregressive pretraining (Brown et al., 2020) provide important foundations for modern LLMs. Code-specialized language models, including Codex (Chen et al., 2021) and DeepSeek-Coder (Guo et al., 2024), have demonstrated the ability to generate programs from natural-language specifications and code context through training on large code corpora. Their evaluations establish functional program generation as an important capability, but do not by themselves establish hardware-specific execution efficiency. KernelBench (Ouyang et al., 2025) addresses this distinction by evaluating both correctness and performance for LLM-generated GPU kernels, including generation with execution and profiling feedback. Beyond direct code generation, QiMeng-Kernel (Zhu et al., 2026) separates high-level optimization strategies from low-level implementation through Macro Thinking and Micro Coding. It trains a lightweight model with reinforcement learning to propose optimization actions and uses general-purpose LLMs to implement them incrementally. This constitutes a close methodological precedent for separating optimization decision-making from code implementation. REPAIR shares this division of responsibilities and investigates its application to the recursive optimization of existing Ascend kernels.

LLM-based Ascend kernel generation and optimization. Research targeting Ascend addresses limited code availability through domain adaptation, intermediate representations, and hardware-guided search. AscendKernelGen (Cao et al., 2026) combines domain-specific reasoning data with supervised fine-tuning and execution-feedback RL, while AscendCraft (Wen et al., 2026b) gener-

ates kernels in a lightweight DSL before translating them into AscendC through constrained LLM passes. A complementary line accumulates reusable optimization experience. EvoKernel (Zheng et al., 2026) uses stage-specific Q-values to retrieve experiences for initial drafting and subsequent latency refinement, sharing memory across tasks without fine-tuning the generator. AscendOptimizer (Wu et al., 2026) mines optimization experience by selectively de-optimizing strong kernels and combines retrieval-guided rewriting with hardware-guided host-side tiling search. Hawk (Wen et al., 2026a) constructs executable hardware knowledge, retrieves it according to syntax and hardware bottlenecks, and updates the knowledge base using execution feedback. AgenticCANN (Qiu et al., 2026) combines structured domain knowledge with stage-adaptive agent interactions for candidate discovery and performance tuning. Evaluation is also expanding: alongside MultiKernelBench (Wen et al., 2025) and NPUKernelBench (Cao et al., 2026), CANN Bench (Gao et al., 2026) evaluates compilation, correctness, and performance against both PyTorch-on-Ascend baselines and analytical hardware limits. These studies already establish the value of staged workflows, reusable experience, and hardware feedback. Within this landscape, REPAIR studies a specific training allocation for optimizing existing Ascend kernels: a compact Identification model learns to select actionable optimization directions, and a separate Revision executor implements them. Its emphasis is on learning the decision stage through the downstream performance of revised kernels, complementing generator adaptation, memory-value learning, and knowledge-guided search.

LLM self-improvement and recursive refinement. Prior work improves generated outputs, trains models to revise their responses, or modifies the agent system itself. RLC (Pang et al., 2024) studies a fundamental problem about why LLM could self-improve. At inference time, Self-Refine (Madaan et al., 2023) alternates self-feedback and revision, while Reflexion (Shinn et al., 2023) retains verbal reflections in episodic memory; neither requires model-weight updates. Training-based approaches explicitly learn refinement behavior. RISE (Qu et al., 2024) formulates response improvement as a multi-turn decision process and iteratively fine-tunes models to correct unsuccessful attempts. SCoRe (Kumar et al., 2025) uses multi-turn RL on self-generated correction trajectories, and S^2R (Ma et al., 2025) combines supervised initialization of self-verification and self-correction with outcome- and process-level RL. For code generation, CoCoS (Cho et al., 2025) trains small models to preserve correct outputs and progressively repair incorrect ones using trajectory-level and fine-grained rewards. These methods motivate studying how training objectives shape the ability to improve outputs over successive attempts. Another line evolves the agent's implementation: SICA (Robeyns et al., 2025) edits its own codebase, and the Darwin Gödel Machine (Zhang et al., 2026a) maintains an archive of self-modifying coding agents evaluated on coding benchmarks. Hyperagents (Zhang et al., 2026b) further makes both the task agent and its meta-level modification procedure editable. REPAIR's recursive loop instead updates the target kernel through correctness- and latency-checked revisions. Public agents provide training data, RL specializes the Identification model, and the resulting optimizer is used to improve successive kernel implementations. This connects REPAIR most directly to learned refinement policies and feedback-guided program improvement; it does not entail self-modification of the optimizer during an optimization run.

3 PRELIMINARY

This section formalizes the Ascend kernel optimization task and introduces the reinforcement learning notation used to describe the Identification policy in REPAIR.

3.1 ASCEND KERNEL OPTIMIZATION TASK

We consider an operator specification F, an initial Ascend kernel K_0, a set of target workloads $\mathcal{W}$, and an execution environment h. The workloads specify the relevant input configurations, while h fixes the target hardware and software configuration. The optimization task is to revise K_0 to reduce execution latency while preserving the prescribed operator semantics.

Let $\mathcal{K}$ denote the space of permitted kernel implementations. We define a validity indicator $C_h(K; F, \mathcal{W}) \in \{0, 1\}$, which equals one when K compiles, executes successfully, and satisfies the prescribed correctness checks for F on $\mathcal{W}$, including any applicable numerical tolerances. We assume that K_0 satisfies these requirements. Validity is defined relative to the evaluation protocol: passing a finite set of correctness checks does not establish semantic equivalence on untested inputs. For a valid implementation K, let $T_h(K; \mathcal{W}) > 0$ denote its execution latency under a fixed mea-

surement protocol. When multiple workloads are evaluated, the latency aggregation rule is part of this protocol. The optimization objective is

$$\begin{aligned} \min_{K \in \mathcal{K}} \quad & T_h(K; \mathcal{W}) \\ \text{subject to} \quad & C_h(K; F, \mathcal{W}) = 1. \end{aligned} \tag{1}$$

This formulation specifies the optimization goal without assuming that a global optimum can be attained. The speedup of a valid candidate relative to the initial kernel is

$$S_h(K; K_0, \mathcal{W}) = \frac{T_h(K_0; \mathcal{W})}{T_h(K; \mathcal{W})}. \tag{2}$$

Thus, $S_h > 1$ indicates a latency reduction under the same workloads, execution environment, and measurement protocol. An invalid candidate is not considered a successful optimization, regardless of its observed execution time.

REPAIR addresses this task by separating the selection of an optimization direction from its implementation. *Identification* specifies what to optimize without modifying the kernel, whereas *Revision* implements the selected direction. The resulting code remains subject to the validity constraint and performance objective in Equation equation 1.

3.2 Reinforcement Learning for LLM Optimization

Following the policy view of language-model fine-tuning (Ouyang et al., 2022; Le et al., 2022), we represent an autoregressive LLM as a stochastic policy π_θ, parameterized by θ. Given a prompt x, the policy generates a response $y = (y_1, \ldots, y_m)$, where m denotes the response length, including the termination token. Its probability factorizes as

$$\pi_\theta(y \mid x) = \prod_{\ell=1}^{m} \pi_\theta(y_\ell \mid x, y_{<\ell}), \tag{3}$$

where $y_{<\ell}$ is the generated prefix. At the token level, the state is $s_\ell = (x, y_{<\ell})$, and the action is the next token $a_\ell = y_\ell$.

Let $\mathcal{D}$ denote the distribution of training prompts and $r(x, y)$ the scalar reward associated with a generated response. As in classical policy-gradient methods (Williams, 1992), the unregularized expected-reward objective is

$$\max_\theta J(\theta), \qquad J(\theta) = \mathbb{E}_{\substack{x \sim \mathcal{D} \\ y \sim \pi_\theta(\cdot|x)}} \big[r(x, y) \big]. \tag{4}$$

This prompt–response formulation provides the basic policy-learning notation. A particular training procedure may additionally specify regularization and a surrogate optimization objective; for example, proximal policy optimization (PPO) uses policy probability ratios to construct its surrogate objective (Schulman et al., 2017). When reward evaluation is stochastic, the expectation also includes that randomness. In REPAIR, π_θ denotes the model trained for *Identification*, rather than a policy that directly produces revised kernel code. At recursive iteration t, let x_t denote the prompt describing the current kernel K_t and the relevant task context. The response $y_t \sim \pi_\theta(\cdot \mid x_t)$ specifies an actionable optimization direction. *Revision* then uses this direction to produce a candidate implementation, denoted by K'_t. We use t for kernel optimization iterations and ℓ for token positions to distinguish these two levels of generation.

The training reward $r(x_t, y_t)$ and the kernel speedup $S_h(K'_t; K_0, \mathcal{W})$ are distinct quantities: the former supplies the learning signal for Identification, while the latter measures the performance of a valid implementation. Their relationship depends on the reward construction. This distinction allows the learning objective to be specified without assuming that every proposed direction is correctly implemented or yields a speedup.

4 Method

REPAIR separates optimization decision-making from code implementation, as shown in Figure 1. At each iteration, *Identification* takes the current kernel, its profiling data, and expert-derived optimization directions as input and produces an actionable optimization plan without modifying the

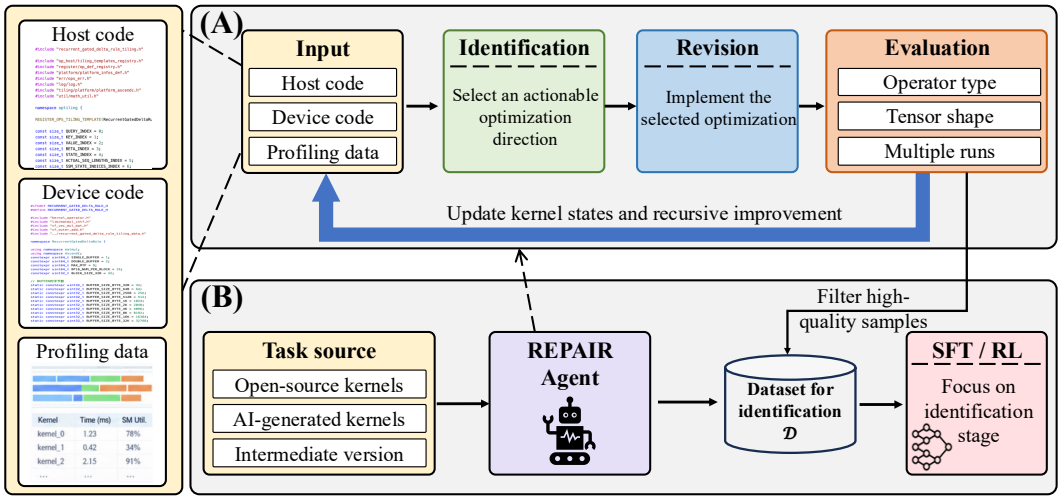


Figure 1: The overall framework of REPAIR: **(A)** The recursive kernel optimization process, which employs publicly available agents to identify optimization directions and revise the code. **(B)** The training procedure, which collects decision data and trains the model for the Identification stage.

code. *Revision* then implements this plan, subject to its stated constraints. A validated revision is evaluated on the target hardware and becomes the input to the next iteration only when it satisfies the performance-acceptance criterion. Building on this separation, we collect data for both stages using external agents and apply supervised fine-tuning / reinforcement learning (RL) only to a compact Identification model. The design aims to focus the trainable model on deciding what to optimize, while delegating implementation to Revision.

4.1 Identifying Optimization Plan as Decision Making Problem

Let $K_t \in \mathcal{K}$ denote the incumbent kernel implementation at optimization iteration t, including its complete Ascend C host and kernel source. Let P_t denote its profiling data for the workload under consideration, and $\mathcal{O}$ the supplied expert-derived optimization directions. With the task context $(F, \mathcal{W}, h)$ held fixed, the prompt and response of the Identification policy π_θ are written as

$$x_t = (K_t, P_t, \mathcal{O}), \qquad y_t \sim \pi_\theta(\cdot \mid x_t), \tag{5}$$

where the tuple defining x_t is shorthand for the prompt constructed from these inputs and the fixed task context, and y_t is a response specifying an optimization plan rather than revised source code. As in Section 3.2, t indexes kernel optimization iterations, while ℓ is reserved for token positions. Although profiling data are included in the input, a nonempty plan must be supported by source-level evidence and a comparison of full-launch work; profiler summaries alone do not establish an optimization direction.

Expert-derived optimization candidates. The direction catalog is based on practices summarized by human Ascend kernel experts. It covers seven categories: `reuse_onchip` reduces repeated global-memory (GM) accesses through on-chip residency; `batch_transfer` consolidates fragmented transfers and writebacks; `vectorize` replaces serial Scalar work or unnecessary reduction work with appropriate Vector structures; `cube` targets computation, reloads, and format conversions associated with Cube execution; `resize_tile` adjusts the work unit subject to capacity and execution constraints; `parallelize` improves effective physical-core coverage through valid output ownership; and `pipeline` targets directly evidenced waits through execution overlap. These categories organize the analysis rather than prescribe a fixed ranking or an independent parameter search.

Source-grounded reasoning. Identification first reconstructs the operator's mathematical semantics, task-to-output mapping, and dataflow from the complete source. The analysis must preserve numerical behavior, cross-tile state, and output ownership. For a fixed workload, it then accounts

for work over the entire kernel launch, represented as

$$\mathbf{w}(K) = (B_{\mathrm{GM}}, N_{\mathrm{DMA}}, N_{\mathrm{V}}, N_{\mathrm{C}}, N_{\mathrm{S}}, N_{\mathrm{task}}, N_{\mathrm{init}}, N_{\mathrm{sync}}, N_{\mathrm{wb}}). \tag{6}$$

The components denote GM bytes, DMA transactions, Vector repeat blocks, Cube operations, Scalar iterations, task lifecycles, state initializations, synchronization events, and writeback transactions, respectively. These heterogeneous quantities support structural comparisons; they are not interchangeable units of execution time. In particular, reducing work per task is insufficient when the proposed change increases the number of tasks and their aggregate transfers or reloads.

A proposed plan compares current work with the work implied by its target structure. It must identify a reduction in relevant hardware work, improved effective core coverage without increased work per output, or removal of a directly evidenced critical dependency. Changing loop syntax, increasing tile size, or adding queue slots is not sufficient evidence by itself. Reasoning proceeds from the mathematical structure and cross-tile state to data layout and residency, transfers, buffer lifetimes, tile size, core ownership, and finally pipelining. Consequently, interdependent changes are combined into one internally consistent plan rather than selected independently. The resulting cost argument is a prediction about the proposed structure, not a claim of measured speedup.

Generating optimization plan. The optimization plan records source facts and cost formulas (`evidence`); an explanation of the task, current implementation, problem, strategy, derivation, and boundaries (`reasoning`); the symbols to modify (`targets`); explicit current-to-target changes (`changes`); and the invariants that implementation must preserve (`guards`). These guards cover the relevant mathematical, addressing, tail-handling, synchronization, and application binary interface (ABI) constraints. If no supported and feasible plan can be established, Identification returns a no-plan response, denoted by $y_t = \varnothing$, instead of requesting speculative edits.

4.2 Revise the Kernel Codes with Coding Agent

Revision translates the selected plan into a candidate implementation:

$$K'_t = \mathcal{R}(K_t, y_t), \tag{7}$$

where $\mathcal{R}$ denotes the implementation executor. Its task is to realize the prescribed changes in the host and kernel code, not to reconsider the optimization direction. In particular, it must respect the plan's mathematical structure, tile choices, data layout, buffer lifetimes, output ownership, and synchronization constraints. It may not substitute a different algorithm or introduce an independent optimization. Repairs needed for compilation or execution must remain consistent with the same plan.

An implementation is eligible for performance evaluation only after compilation, execution, and numerical validation succeed. We use ValidatedRevision to denote Revision followed by these checks; a successful return satisfies $C_h(K'_t; F, \mathcal{W}) = 1$. Invalid code cannot replace the incumbent. This interface separates implementation correctness from optimization selection without assuming that every plan can be implemented successfully. Requiring faithful implementation also limits changes unrelated to the selected plan, although it does not make measured outcomes independent of executor reliability or timing variability.

4.3 Reinforcement Learning for Improved Identification

In previous section, we present that REPAIR can use public agent, e.g., Codex Yadav & Mondal (2025) and Claude Code Anthropic (2026) to perform recursive Ascend Kernel optimization from Identification and Revision stages. However, the optimization may early stop due to the lack of exploration. Here, we introduce how to combine REPAIR with RL to further improve its utilization.

Collecting golden Identification data. As illustrated in Figure 1(B), we first instantiate REPAIR with publicly available coding agents, such as Codex (Yadav & Mondal, 2025) and Claude Code (Anthropic, 2026), to collect optimization examples. For each operator, Identification proposes a plan, Revision implements it, and execution validates correctness and measures the resulting speedup. Among the collected candidates, we select the plan yielding the highest verified speedup as the *golden Identification* label, paired with its corresponding kernel context. This converts agent-driven optimization into training data for learning which changes to pursue.

Improving Identification through training. We train open-sourced LLMs on these examples, using golden labels as targets for privileged references for On-Policy Self-Distillation (OPSD) (Zhao et al., 2026). In OPSD, the student generates an optimization plan from the kernel context, while the same model, conditioned additionally on the golden label, provides teacher feedback. Token-level distribution matching on student-generated plans guides the Identification policy toward reference-informed predictions. Only Identification is updated; Revision remains fixed. The trained policy could be deployed in the recursive optimization loop in Figure 1(A), where its plans are implemented and validated before an improved kernel becomes the next input. This design focuses learning on optimization decisions while delegating code implementation to the coding agent.

5 EXPERIMENTS

In this section, we conduct extensive experiment to evaluate the effectiveness of REPAIR method for Ascend kernel optimization. We aim at answer the following key questions: (1) whether separating optimization decision-making from code implementation improves Ascend kernel performance? (Section 5.2) (2) Could RL enable further improve the system's ability for better identify kernel optimization direction? (Section 5.3) (3) How does RL compare to supervise fine-tuning for the identification task? (Section 5.4) We begin by introduce the experiment setting.

5.1 EXPERIMENTAL SETUP

Dataset and training data. We study Ascend kernel optimization on 253 operators from MultiKernelBench (Wen et al., 2025). For training data construction, we run REPAIR on 243 of these operators and collect four optimization examples per operator, yielding 972 candidate examples. For each operator, we select the example that achieves the highest measured kernel speedup after correctness validation. The corresponding Identification output serves as the *golden label* for subsequent SFT and on-policy training, producing 243 examples for training. Thus, each training operator contributes one selected optimization target from four candidates.

Model and training. Both training variants use Qwen3-8B (Yang et al., 2025) as the Identification backbone. SFT learns to reproduce the selected golden labels from the corresponding kernel inputs. For on-policy training, we adopt On-Policy Self-Distillation (OPSD) (Zhao et al., 2026), using the selected golden labels as privileged reference information. OPSD uses the same model as teacher and student under different contexts, and the teacher additionally receives the reference, while the student receives only the task input. Training minimizes token-level distributional divergence on student-generated responses. In our setting, these responses are optimization plans. Revision implements the plans as kernel changes, and training updates the Identification model.

Baselines. We compare REPAIR with three baselines: (1) **Codex-E2E**, which uses the Codex coding agent (Yadav & Mondal, 2025) to jointly choose and implement kernel optimizations; (2) **Identification-SFT**, which trains Qwen3-8B on the same 233 golden-label examples using supervised fine-tuning; and (3) **LLM-E2E**. Like Codex-E2E, LLM-E2E directly generates revised kernel code with different open-sourced LLMs[1],without a separate Identification stage.

Hardware and evaluation. All operator evaluations run on a single Ascend 910B4 NPU. For strategy-producing models, a constrained executor applies feasible proposed changes to an isolated copy of the fixed baseline and records skipped changes. Compilation or correctness failures terminate the attempt without automatic repair retries. We compare output counts and shapes with a PyTorch reference on identical fixed inputs at the manifest-specified shape. Finite-valued elements must satisfy $|y - \hat{y}| \leq 0.01 + 0.01|\hat{y}|$, where $\hat{y}$ is the reference output. After one synchronized warm-up in a separate process, we collect a single `msprof` PipeUtilization profile and extract the unique target AI Core task's positive, finite Task Duration. This measures device-task latency, excluding model inference, code modification, compilation, and reference execution. We define the *SpeedUp rate* as the execution time before code modification divided by that after modification. We provide more details about the experimental setup in Appendix A.

[1]In our exepriments, we use Qwen3-4b, Qwen3-8b (Yang et al., 2025) and Llama3.1-8b (Dubey et al., 2024).

Table 1: Performance of different methods on 243 Ascend kernels from MultiKernelBench. 'Comp.' represents the compile pass rate, 'Opti.' the rate for positive optimization and 'Speedup' measures the speed-up ratio (running duration of before/after).

	Training set ($n = 243$)			Validation set ($n = 10$)		
Method	Comp. ↑	Opt. ↑	Speedup ↑	Comp. ↑	Opt. ↑	Speedup ↑
End-to-end methods						
LLM-E2E	0.0	0.0	0.0	0.0	0.0	0.0
Agent-E2E	42.0	21.8	10.3	40.0	20.0	9.5
Separated Identification & Revision methods						
Base + Codex	0.0	0.0	0.0	0.0	0.0	0.0
SFT + Codex	100.0	**80.7**	**53.3**	10.0	0.0	0.0
OSPD + Codex	**100.0**	**79.0**	37.6	**100.0**	**70.0**	**34.8**

5.2 Main Results

Table 1 compares end-to-end kernel code generation with the separated Identification–Revision pipeline on 243 training operators and 10 evaluation operators. The separated variants use Codex for Revision and differ in how the Identification model is trained.

End-to-end generation. LLM-E2E produces no successfully compiled modifications on either split, whereas Agent-E2E achieves compilation rates of 42.0% and 40.0% on the training and evaluation splits, respectively. Agent-based execution therefore provides a viable end-to-end optimization route in this setting. However, its optimization success rates are lower, at 21.8% and 20.0%, showing that producing compilable code does not necessarily yield a correct, faster kernel. End-to-end generation combines optimization selection and implementation in a single workflow without requiring a separately trained Identification model, but its observed success rate remains limited.

Separating Identification from Revision. With OPSD-trained Identification, REPAIR achieves 100.0% compilation on both splits and optimization success rates of 79.0% and 70.0%. On the evaluation split, this corresponds to seven successfully optimized operators out of ten, compared with two for Agent-E2E: an improvement of 50 percentage points. The reported Speedup metric also increases from $9.5\times$ to $34.8\times$; on the training split, it increases from $10.3\times$ to $37.6\times$. These results support the effectiveness of a trained Identification model coupled with a capable coding agent. Explicit plans allow the trainable model to focus on optimization choices while Revision handles the corresponding source changes. The comparison evaluates this complete design, including task-specific training, rather than isolating stage separation as the sole cause of the gains.

5.3 Performance of REPAIR Combined with SFT/RL

Improving beyond supervised imitation. The Qwen3-8B results in Table 1 show that the training strategy substantially affects Identification performance. Although SFT achieves the highest training optimization success rate, it does not successfully optimize any evaluation operator. OPSD instead achieves 70.0% optimization success on the evaluation split while retaining strong training performance. This suggests that learning from golden plans is useful, but reproducing them through SFT alone does not ensure generalization. OPSD provides reference-conditioned feedback on the model's own generated plans, offering a way to improve decision-making beyond supervised imitation.

Comparison across Identification backbones. Figure 2 compares Qwen3-4B, Qwen3-8B, and Llama-3.1-8B within REPAIR. Across all three backbones, SFT produces compilable modifications for most evaluation operators but yields no reported speedup, whereas on-policy training achieves positive speedups for each backbone. Thus, the benefit of on-policy training extends beyond a single model, and compilation success alone does not imply effective optimization. Qwen3-8B achieves the highest evaluation Speedup in this comparison, while Qwen3-4B also benefits substantially. These results motivate exploring larger Identification models, although differences between model families cannot be attributed to parameter count alone.

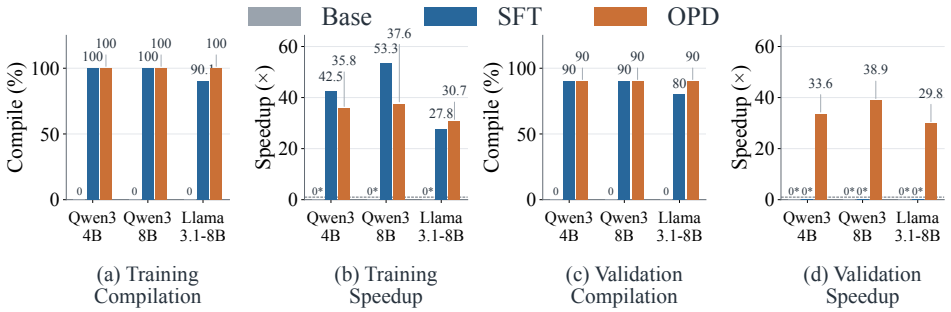


Figure 2: Comparison of different LLMs for Identification in REPAIR. (a,b) report compilation success and Speedup on the training set, and panels (c, d) report the corresponding evaluation results.

5.4 Performance of Supervised Fine-tuning on Identification Task

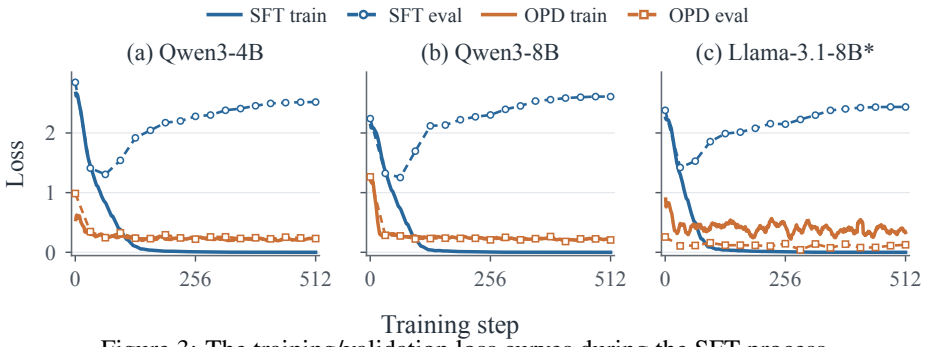


Figure 3: The training/validation loss curves during the SFT process.

Results in Table 1 presents that SFT fails to train an LLM to effectively identify valuable optimize plans. This section further investigates the training process of SFT. Figure 3 presents the training/validation loss during the SFT process. We observe that SFT training loss decreases, whereas validation loss initially falls and then rises as trainingcontinues. This divergence is consistent with overfitting to the available demonstrations. Further reducing training loss does not necessarily improve prediction on validation examples. The curves alone do not establish model capacity as the cause. Identification also requires more than reproducing the wording of a reference plan. A useful decision must match the current kernel's dataflow, memory constraints, and task ownership, and yield a faster implementation after Revision. Besides, it is important to noted that the loss curves do not directly measure kernel speedup or optimization success. Those outcomes must be evaluated through execution under the protocol of Table 1.

6 Conclusion

We present REPAIR, a recursive framework for Ascend kernel optimization that separates identifying what to optimize from implementing code changes. Identification generates actionable optimization plans from source code, profiling information, and expert knowledge, while Revision implements them under explicit correctness constraints. Correctness checks and hardware measurements determine which revisions are retained for subsequent iterations. This separation further enables the collection of verified optimization examples from external agents and the specialization of compact Identification models through supervised fine-tuning and on-policy training. Experiments on MultiKernelBench demonstrate this design on Ascend hardware and establish optimization decision-making as an explicit, separately trainable component of LLM-based kernel optimization. REPAIR still has several limitations. First, our learning experiments are limited to Qwen3-8B, leaving open whether larger Identification models or stronger training strategies can surpass the Codex agent that provides optimization examples. Second, Identification relies on a finite set of expert-defined optimization directions, which may miss opportunities from unfamiliar workloads, transformation interactions, or hardware features outside the catalog. Future work could explore scaling Identification models and learning to expand optimization directions from execution feedback, reducing reliance on fixed expert recommendations.

AI USE STATEMENT

We used generative AI tools solely for language editing and grammatical polishing of the manuscript. The tools were not used to generate research ideas of REPAIR, conduct experiments, analyze results, or formulate scientific conclusions. All AI-assisted edits were reviewed and verified by the authors.

REFERENCES

Anthropic. Claude code, 2026. URL https://claude.ai/.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. In *ICLR*, 2026.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, Dongyang Tao, Feidiao Yang, Chang-Dong Wang, Yutong Lu, Weicheng Xue, Bin Zhou, and Yonghong Tian. AscendKernelGen: LLM-driven kernel generation for NPUs. In *ACL Findings*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv*, abs/2107.03374, 2021.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

Jeonghun Cho, Deokhyung Kang, Hyounghun Kim, and Gary Geunbae Lee. Self-correcting code generation using small language models. In *EMNLP Findings*, 2025.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *NeurIPS*, 2022.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv*, abs/2407.21783, 2024.

Xue-Jian Gao, Deng Pan, Yueming Su, Jiasheng Li, Bin Du, Fengming Zhu, Chengdi Ma, Junyi Fan, Qichen Liao, Chengqiu Hu, Xinxian Chen, Lingchao Zheng, Jun Li, Jiwei Yang, and Yuwei Fan. CANN Bench: Benchmarking agent generated kernels against real NPU and algorithmic limits. *arXiv*, abs/2607.20518, 2026.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. *arXiv*, abs/2401.14196, 2024.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, JD Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. In *ICLR*, 2025.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, 2022.

Ruotian Ma, Peisong Wang, Cheng Liu, Xingyan Liu, Jiaqi Chen, Bang Zhang, Xin Zhou, Nan Du, and Jia Li. S^2R: Teaching LLMs to self-verify and self-correct via reinforcement learning. In *ACL*, 2025.

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. In *NeurIPS*, 2023.

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

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.

Jing-Cheng Pang, Pengyuan Wang, Kaiyuan Li, Xiong-Hui Chen, Jiacheng Xu, Zongzhang Zhang, and Yang Yu. Language model self-improvement by reinforcement learning contemplation. In *ICLR*, 2024.

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*, abs/2607.26661, 2026.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. In *NeurIPS*, 2024.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. In *ICLR Workshop on Scaling Self-Improving Foundation Models*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, abs/1707.06347, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

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*, abs/2607.01590, 2026a.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A multi-platform benchmark for kernel generation. *arXiv*, abs/2507.17773, 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*, abs/2601.22760, 2026b.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009.

Jiehao Wu, Zixiao Huang, Wenhao Li, Chuyun Shen, Junjie Sheng, and Xiangfeng Wang. AscendOptimizer: Episodic agent for Ascend NPU operator optimization. *arXiv*, abs/2603.23566, 2026.

Devansh Yadav and Shouvick Mondal. Evaluating pre-trained large language models on zero shot prompts for parallelization of source code. *Journal of Systems and Software*, 230:112543, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv*, abs/2505.09388, 2025.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. In *ICLR*, 2026a.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv*, abs/2603.19461, 2026b.

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*, abs/2511.01884, 2025.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv*, abs/2601.18734, 2026.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, 2020.

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*, abs/2603.10846, 2026.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. QiMeng-Kernel: Macro-thinking micro-coding paradigm for LLM-based high-performance GPU kernel generation. In *AAAI*, 2026.

A Experimental Details

In this section, we provide more details to supplement the main content with optimization-data construction, model-output processing, and device validation details. We distinguish recursive agent-based data collection from the fixed-baseline evaluation of trained Identification models. The two procedures share numerical and timing checks but differ in version selection, implementation permissions, and stopping rules.

A.1 Training Data Construction and Selection

Sampling and golden-label selection for SFT/RL. Our study uses 243 MultiKernelBench operators. For 233 operators, we collect four optimization examples per operator, giving 932 candidates, and retain the example with the highest measured speedup after correctness validation. Its Identification response provides the golden label, yielding one selected example per training operator. Each label must remain paired with the kernel state and task specification from which it was generated: a plan for an intermediate kernel is not interchangeable with a plan for the initial implementation. The selected responses are supervised targets for SFT and privileged references for OPSD; the latter trains on student-generated responses rather than simply reproducing reference tokens.

Recursive collection. A collection run starts from the complete Host/Kernel implementation at a fixed workload. Identification reconstructs its semantics, dataflow, and output ownership, then produces a unified optimization plan. Revision implements the plan, and compilation, numerical validation, and device profiling determine whether the candidate is accepted. Let T_{parent} and T_{child} denote their valid Task Durations. A candidate advances the trajectory only when

$$T_{\text{child}} < 0.95\, T_{\text{parent}}. \tag{8}$$

After acceptance, Identification re-analyzes the entire revised kernel. The documented collection procedure stops when no supported plan is available, the gain is insufficient, or four optimization rounds have been reached. The collection workflow permits limited implementation repair; the model-evaluation procedure below does not automatically retry failed builds or numerical checks.

Version validation and provenance. The collection workspace names initial implementations with the suffix `_0` and subsequent versions with numeric suffixes. A recorded latency is eligible for best-version selection only when numerical validation passes, the source fingerprint matches the evaluated code, and Task Duration is valid. The minimum-latency validated version and the parent admitted for further recursion need not coincide: a small improvement can be recorded without passing the strict 5% continuation gate. Records from the separate model-evaluation workspace are excluded from the collection summary.

Dated collection summary. Table 2 reports the collection summary dated September 1, 2026. It contains 83 retained operators with reported latency reductions exceeding 1% and 140 strategy records. These are counts in that dated collection record, not a benchmark-wide success rate or a replacement for the 233-example training specification above. The summary does not provide the full attempted-operator denominator.

Table 2: Dated agent-based collection summary. Strategy-record counts can exceed operator counts because the summary contains multiple records for some operators. These are not trained-model results.

Suite	Retained operators	Strategy records
KernelBench910B	45	73
Attention910B	29	53
MHC910B	9	14
Total	83	140

A.2 Generation Records and Baseline Alignment

Frozen model outputs. Evaluation inputs contain the system prompt, operator specification, and source code, together with an operator identifier used to locate the implementation. Generation records retain the model identifier, raw response, normalized response, JSON validity, parsing error, whether a nonempty strategy exists, input/output token counts, and termination reason. Normalization removes outer formatting such as code fences; it does not insert missing brackets, complete truncated responses, or invent strategy content. Valid JSON alone does not establish semantic validity or optimization success. Here, an untrained *Base* model means the starting checkpoint before task-specific Identification training, not necessarily a model without instruction tuning. Comparisons require the same generation rules across variants; alternative decoding configurations must be recorded separately.

Isolated parent and child projects. Each model–operator pair receives its own baseline parent and editable child project. The child does not inherit precision, performance, or strategy reports as evidence for its modified source. The current resolver removes the version suffix from the operator identifier and loads the corresponding `_0` implementation. Consequently, evaluation inputs must match this source; intermediate-state evaluation requires an explicit version-aware mapping. Baseline latency is read from the validated `_0` record rather than automatically remeasured. Reuse requires matching source, shape, dtype, and timing protocol, and the term "original kernel" refers to this aligned baseline.

A.3 Constrained Strategy Implementation

The evaluation executor processes proposed changes individually. It applies an explicit change when the required implementation follows from the current source and SDK. It skips no-ops, changes based on incorrect source facts, conflicting edits, and requests that require inventing an algorithm, API, or layout. An inapplicable change does not prevent an independent, feasible change from being implemented. The executor may edit only the child's Host/Kernel source; it must not modify the reference implementation or consult other models' results or historical best solutions to supply additional optimizations.

For each attempt, the implementation report records applied changes, skipped changes and their reasons, modified files, and a summary. The controller checks the frozen strategy hash, implementation record, and source fingerprint. No applied change or no actual source change produces a `no_actionable_change` outcome. Compilation or numerical failure terminates the attempt without an automatic repair loop. Source diffs and implementation records remain necessary for checking fidelity: hashes establish artifact consistency, not that every edit faithfully implements the proposed strategy.

Implementation can process up to four operators concurrently in separate agent contexts. Within the evaluation controller, OPP installation, correctness checks, and profiling are serialized on the single Ascend 910B4. Concurrent independent controllers would require additional coordination to maintain exclusive device access.

A.4 Numerical Validation and Device Timing

Build and reference inputs. Validation builds the Host/Kernel code, installs the corresponding vendor OPP, and prepares the extension used to invoke the operator. Build failures are distinct from numerical mismatches. The operator manifest and reference define the original shape, initialization parameters, and fixed random inputs. Identical inputs are supplied to the custom kernel and the PyTorch reference. Cached inputs and expected outputs may be reused when applicable, but each modified kernel is executed again. When an Attention or MHC operator replaces a subgraph, its inputs and reference outputs must correspond to the same subgraph boundary; a missing adapter is not evidence of numerical incorrectness.

Elementwise checks. The validator first checks output tensor counts and shapes, then compares outputs in FP32 chunks. For finite values, every element must satisfy

$$|y_j - \hat{y}_j| \leq 10^{-2} + 10^{-2}|\hat{y}_j|, \tag{9}$$

Table 3: Interpretation of evaluation states. A completed measurement is not by itself a successful optimization.

State	Interpretation
`generation_failed`	Output cannot enter implementation, e.g., invalid JSON or an empty strategy. The current controller may also assign this state to a missing baseline.
`no_actionable_change`	No model-proposed change was implemented as an actual source modification.
`implementation_failed`	Execution, implementation-record, or frozen-strategy checks failed.
`build_failed`	Host/Kernel compilation or build failed.
`precision_failed`	Validation failed; detailed logs distinguish numerical, runtime, reference, and environment causes.
`performance_failed`	No valid target-task latency was obtained.
`done`	Validation and timing completed; the result may be faster, unchanged, or slower than the baseline.

where $\hat{y}_j$ is the reference. A small mean error cannot compensate for an element outside tolerance. Reports retain maximum and mean absolute errors, tolerances, input metadata, and the source fingerprint. The fingerprint is checked again before performance collection, so a previous PASS cannot validate subsequently modified code.

These checks cover the manifest shape and fixed inputs rather than all shapes or boundary cases. The documented validator does not explicitly handle NaN/Inf comparisons, which can allow non-finite outputs to pass incorrectly. Therefore, its PASS status does not establish special-value correctness; explicit non-finite checks are needed to extend this claim.

Profiling protocol. After validation, the performance collector verifies the source fingerprint and installed vendor, executes one synchronized warm-up outside the profiler, and starts a separate runner under `msprof` with PipeUtilization CSV export. It identifies the target by operator name and AI Core task type, requires a unique matching task, and accepts only a finite, positive `Task Duration(us)`. The raw CSV, extracted latency and fingerprint, and profiling log are retained, including logs for failures.

The protocol uses one formal measurement rather than a mean or median of repeated trials. Warm-up and profiling occur in separate processes. Task Duration measures the target device task; it excludes model inference, code editing, compilation, and reference execution. Single measurements and reused historical baselines remain sensitive to measurement conditions. The 5% gate is an acceptance rule, not a statistical significance test.

A.5 Outcome Accounting and Evidence Retention

For a fixed evaluation baseline $T_{i,0}$ and measured candidate $T_{i,m}$, define fractional latency reduction $d_{i,m} = 1 - T_{i,m}/T_{i,0}$ and speedup $s_{i,m} = T_{i,0}/T_{i,m}$. Optimization success requires an implemented model-proposed change, an actual source modification, successful build and validation, valid latency, and $d_{i,m} > 0.05$. Equivalently, the speedup threshold is $s_{i,m} > 1/0.95$, not $s_{i,m} > 1.05$. A candidate exactly 5% faster in latency does not pass the strict threshold.

In model evaluation, the controller compares against the fixed `_0` parent. This comparison determines the reported success, even if a performance collector separately compares against the best historical version. In recursive collection, the comparison instead uses the parent admitted to the trajectory (Equation 8). The two reference points must not be mixed.

The success rate is

$$\text{SuccessRate} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\{\text{all success conditions hold for operator } i\}, \tag{10}$$

where N includes all evaluation tasks, including failures. Table 3 distinguishes the terminal states. Build and precision pass counts should be obtained from their respective stage reports, since a terminal-state summary can omit an earlier success followed by a later failure.

Table 4: Documented work estimates and measured latency for S2Attention operator revision. Static counts describe the full launch. Task Duration is obtained from device profiling.

Quantity	Before	After
DMA bursts	87,808	25,216
Vector repeat blocks	250,880	250,880
Scalar iterations	18,816	6,400
State initializations	12,544	128
Writeback bursts	12,544	6,272
Task Duration (μs)	173.647	51.902

For auditability, each attempt links the raw generation to its frozen strategy, applied/skipped-change record, source diff, precision report, latency record, raw profiling CSV, and baseline. This chain separates parsing, implementation, correctness, and performance outcomes, and allows failure causes to be inspected without discarding unsuccessful attempts from the denominator.

B Identification Constraints and an Optimization Example

B.1 Deriving a Consistent Optimization Plan

The expert directions in Section 4.1 organize a source-based analysis, not an independent search over tile sizes or core counts. Identification first establishes mathematical semantics, including accumulation types, reduction axes, prefix carry, and output ownership. It then determines data residency, transfers, buffer lifetimes, work units, core assignment, and finally pipeline overlap. Coupled changes must use a consistent layout and synchronization scheme.

Capacity calculations include all simultaneously live inputs, outputs, queue slots, partial results, masks, casts, padding, and API workspace. Transfer checks distinguish valid, transferred, computed, allocated, and written-back lengths, including tails and actual byte addresses. Parallelization must cover each output exactly once and account for aggregate reloads across the entire launch. Pipelining requires an identified dependency that can be overlapped; adding queue slots alone does not establish a benefit. A supported plan must reduce relevant hardware work, improve effective core coverage without increasing work per output, or remove a directly evidenced wait. Unresolved API, capacity, or synchronization requirements cannot be replaced by a speculative edit.

B.2 Case Study: Reusing Attention Weights

The documented `S2AttentionCustom` example uses $B = 128$, $K = 3$, $HW = 49$, and $C = 512$, computing

$$y_{b,h,c} = \sum_{k=1}^{3} x_{b,k,h,c}\, a_{b,k,c}. \tag{11}$$

The initial implementation assigns work to 6,272 spatial rows and splits each row into two 256-channel tiles. Although the attention weights $a_{b,:,:}$ are shared across 49 positions, they are reloaded for each row. The plan moves ownership to the batch level, retains 3×512 attention values in on-chip memory, and processes each spatial row using a full 512-channel transfer and writeback. It combines `reuse_onchip`, `batch_transfer`, and `parallelize` in one design.

Table 4 illustrates reduced data movement and control overhead with unchanged Vector repeat work. The revision passes numerical validation with zero reported maximum and mean absolute error, and reduces Task Duration by 70.11%. This is the specific revision described in the workflow example. The dated collection summary records a later best version at $34.142\,\mu$s; that value is not the outcome attributed to the single revision above. Neither record is a trained Identification model comparison.