ASCENDAGENT: EVOLVING CONTEXT TO FIT THE HARNESS IN CANN KERNEL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

High-performance accelerator kernels are fundamental to efficient large language model (LLM) inference, yet optimizing them still requires substantial systems expertise. Existing approaches often optimize workloads independently, making it difficult to reuse experience across input patterns and hardware. We present ASCENDAGENT, a kernel optimization framework that evolves reusable knowledge across workloads while keeping the optimization harness fixed. It organizes API documentation, community implementations, and technical discussions into conditional, composable transformations with expected effects and supporting evidence. ASCENDAGENT operates through two mechanisms. **(1)** Within each workload, a frozen knowledge snapshot guides evolutionary search over a program graph. The harness proposes changes, checks correctness, and measures latency. Compatible programs share evaluation evidence and explored continuations, while LLM-guided mutations explore changes beyond existing knowledge. **(2)** After each source search, workflow replay uses recorded proposals and execution outcomes to revise transformation preferences, application conditions, and reusable combinations. Independent confirmation determines which updates guide later workloads. Model weights, the workflow, and the evaluator remain fixed, and each new workload starts with fresh local search state. We evaluate ASCENDAGENT with five language models on four kernel benchmarks spanning Ascend NPUs and NVIDIA GPUs under a budget of 40 candidate compilations per task. With Kimi K3, ASCENDAGENT achieves mean best speedups of **1.40**× on CANN-Bench and **1.75**× on TritonBench-G, compared with **1.21**× and **1.36**× for the same harness with frozen initial knowledge.

1 INTRODUCTION

Kernel optimization seeks a fast implementation of a computation under hardware and numerical constraints. Performance has to be measured on the target device. AutoTVM and Ansor use measurements to guide search and reduce expensive trials (Chen et al., 2018; Zheng et al., 2020). LLMs broaden the programs that can be proposed, including structural edits that lie outside a predefined schedule space (Lange et al., 2026; Cao et al., 2026). Every proposal still needs to be checked for correctness and performance.

Repeated optimization provides experience about which changes are worth trying. Reusing a finished kernel is often too restrictive because the benefit of a change depends on the workload and the current program. Vectorization can improve aligned accesses while adding work for irregular tails. A buffering configuration may fit one tile and exceed the storage budget of another. Reusable knowledge therefore needs to describe a transformation together with the conditions under which it can be applied. The search must also decide how often to try it relative to other changes.

API documentation, community implementations, and technical discussions provide platform-specific advice. A transformation must retain its application conditions and expected effect. Source relevance alone does not establish that it will help a new workload. Section 2 motivates this problem through a platform-local comparison against community implementations.

We organize this advice into a knowledge base used by a fixed optimization harness (Figure 2). The harness selects a parent, retrieves relevant entries, proposes a change, and evaluates the resulting program. Each proposal records an expectation before execution so that later analysis can distinguish

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.

a useful program change from evidence for its predicted mechanism. Identical source packages share graph nodes and evaluation evidence. Only validated candidates enter the program archive.

After a source workload, workflow replay identifies incomplete conditions, unsupported expectations, and missing advice. Proposed knowledge updates are frozen before independent confirmation. Accepted updates guide later workloads. Program archives and local search statistics are reset. This is a restricted form of recursive self-improvement. The reusable knowledge changes while the model weights, workflow, evaluator, and update procedure remain fixed. AFlow evolves agentic workflows through execution feedback (Zhang et al., 2025). AscendAgent instead holds the workflow fixed and updates its knowledge. STOP, the Darwin Gödel Machine, and Hyperagents instead revise optimizer or agent programs (Zelikman et al., 2023; Zhang et al., 2026a;b).

We combine a knowledge-guided program graph with linked execution records and confirmed knowledge updates across workloads. To assess these updates, we separate benchmark comparison, workload-local controls, and cross-workload knowledge transfer. In particular, Frozen-KB Harness retains the initial knowledge base, while ASCENDAGENT can inherit confirmed source-side updates.

2 BACKGROUND AND MOTIVATING STUDY

2.1 AI ACCELERATORS AND KERNEL OPTIMIZATION

AI accelerators execute tensor computations through *kernels*, which map mathematical operations onto device execution units and memory hierarchies. We represent a workload as $x = (f, \mathcal{Z}_x, \varepsilon_x)$, where f is a reference computation, $\mathcal{Z}_x$ specifies the tested inputs, including their shapes, data types, and layouts, and ε_x specifies numerical tolerances. Given a target hardware/software platform d and a compilation budget B, a kernel optimizer $\mathcal{A}$ aims to produce an executable implementation

$$\widehat{P} = \mathcal{A}(x, d, B). \tag{1}$$

The implementation must satisfy $\widehat{P}(z) \approx_{\varepsilon_x} f(z)$ for every $z \in \mathcal{Z}_x$. The objective is to reduce measured device latency under this correctness contract, without assuming global optimality.

Kernel optimization searches over interacting choices such as tiling, fusion, vectorization, and buffering. AutoTVM and Ansor guide this search using hardware measurements and learned performance estimates (Chen et al., 2018; Zheng et al., 2020), while LLM-based approaches such as K-Search explore program-level transformations (Cao et al., 2026). These choices depend on the platform. NVIDIA GPUs use the CUDA ecosystem, whereas Huawei Ascend NPUs use CANN and Ascend C. Their different programming interfaces and execution constraints mean that an effective optimization strategy on one platform need not transfer directly to another.

2.2 PILOT STUDY OF THE MODEL-TO-COMMUNITY GAP

We design the pilot comparison to examine whether LLM-generated kernels approach established implementations equally closely on NVIDIA and Ascend. The comparison uses the same base LLM, matched workload semantics and input contracts, a common generation workflow, and $B = 40$ candidate compilations per target, including failed compilations. Each workload–platform pair (x, d) has a framework reference $P^{\mathrm{fw}}_{x,d}$ and a reproducible community implementation $P^{\mathrm{com}}_{x,d}$. It demonstrates performance already attainable on that device without being assumed globally optimal. This comparison measures the distance from generated code to existing engineering solutions, rather than comparing absolute hardware speeds.

Let $\widehat{P}_{x,d}$ denote the best independently validated *generated* kernel, excluding reference fallback. We express its latency relative to each reference as

$$g^{(r)}_{x,d} = \frac{T_d(\widehat{P}_{x,d}, x)}{T_d(P^{(r)}_{x,d}, x)}, \qquad r \in \{\mathrm{fw}, \mathrm{com}\}, \tag{2}$$

where $T_d(P, x)$ is the geometric-mean latency of P over $\mathcal{Z}_x$. Thus, $g = 1$ denotes parity and $g > 1$ indicates a slower generated implementation. Table 1 separates correctness from performance to examine the hypothesis that generated kernels remain further from community performance on Ascend. A larger Ascend-side gap would identify optimization headroom already demonstrated on that platform. Reference quality and toolchain differences can also contribute to this gap.

Table 1. Pilot study on MultiKernelBench. Kimi K3 is evaluated on four representative operator tasks across NVIDIA and Ascend with $B = 40$ candidate compilations per target, including failures. (a) Correctness rate. (b) Relative latency against the framework reference. (c) Relative latency against the community reference. Lower latency is better, and 1.0 denotes parity.

	Correctness (%)		NVIDIA Rel. Latency		Ascend Rel. Latency	
Operator	NVIDIA	Ascend	+Comm	Base	+Comm	Base
RMSNorm	90	62	0.84	1.34	2.04	3.35
Softmax	86	68	0.85	1.29	1.93	3.05
MatMul	82	38	0.86	1.42	2.35	3.65
Gather	80	62	0.82	1.17	1.67	2.78

2.3 From Expert Priors to the Optimization Harness

We hypothesize that insufficient platform-specific optimization priors contribute to the gap examined in Section 2.2. This raises a question: *how can expert priors be incorporated into an agent harness for automated kernel optimization?* We distinguish two complementary types: procedural priors on *how to optimize* and knowledge priors on *what to try and when*.

- **Procedural priors.** These priors govern workload decomposition, bottleneck diagnosis, transformation proposals, and feedback-guided refinement. We encode them in an optimization harness that coordinates generation, compilation, correctness checking, and hardware measurement. The harness structures the search but still requires platform-specific knowledge to guide its decisions.
- **Knowledge priors.** We organize advice from API documentation, community code, and technical discussions into *persistent expert context*. Each policy seed $\rho = (c, a, h, e)$ records an applicability condition c, a transformation a, a hypothesized performance effect h, and supporting sources e. These seeds suggest search directions rather than guarantee improvements; the resulting programs still require validation and measurement.

Our approach: The proposed AscendAgent combines the knowledge-guided harness in Section 3.1 with replay-driven knowledge evolution in Section 4. A static knowledge base can provide uneven coverage, bias proposals toward overrepresented strategies, and leave useful knowledge poorly organized. Replay links retrieved entries, proposed transformations, expected effects, and execution outcomes to refine applicability conditions, recover missing knowledge, and reorganize entries. The expert context remains fixed within each workload; only independently confirmed revisions are transferred to subsequent workloads. Thus, procedural priors specify *how to optimize*, while replay refines *what to try and when*, adapting expert context to the harness's decision needs without changing the harness, model weights, or evaluator.

3 AscendAgent Harness: Context-Guided Evolutionary Search

Following Section 2.3, AscendAgent encodes procedural priors in a fixed optimization harness and obtains knowledge priors from external context. Given workload and hardware constraints, the harness repeatedly generates multiple kernel candidates, executes them on the target device, and uses measured profiles to refine the current kernel. This section describes this workload-local optimization loop. Section 4 describes how the resulting execution experience is further used to improve the persistent context across workloads.

3.1 Problem Formulation

The task takes a constraint specification C and external context K as input and returns an optimized kernel program P:

$$C, K \to P. \tag{3}$$

The three components are defined as follows:

- **Constraints C.** The workload and platform requirements, including the target computation, input shapes and data types, numerical tolerances, programming interfaces, and hardware resource limits.
- **External Context K.** Optimization knowledge available to the harness, such as vendor documentation, community implementations, technical discussions, and previously organized expert knowledge. It guides what implementations and transformations to try, but does not replace compilation, correctness checking, or hardware measurement.
- **Kernel P.** Executable kernel code, together with its required launch configuration, that satisfies C.

Given a compilation budget B, we write the optimization process as

$$P = H(C, K, B), \tag{4}$$

where H denotes the fixed optimization harness. Among all generated kernels that pass correctness validation and independent timing, the harness returns the one with the lowest measured latency. Every actual compilation counts toward B, including failed candidates and confirmation recompilations. Thus, P is the best validated kernel found within the search budget rather than a globally optimal program. Figure 1 illustrates this process. The important distinction is that C specifies *what the kernel must satisfy*, while K provides expert guidance on *how it may be optimized*. Actual performance is always determined by execution on the target device.

3.2 Optimization Workflow

AscendAgent performs evolutionary search over kernel programs. Instead of independently sampling implementations, it maintains validated kernels and repeatedly generates, executes, and refines candidates using external context and measured feedback.

Candidate generation. The harness selects a current or historical validated kernel and retrieves context according to its code, constraints, and execution profile. The LLM generates multiple candidates by instantiating applicable transformations or proposing free-form modifications when existing knowledge is insufficient. The best kernel serves as the performance anchor, while earlier kernels remain available as alternative parents. Their versions and transformations form a program graph, where progressive widening and exploration control which branches are expanded (Appendix A.2).

Parallel execution and profiling. Candidates are compiled, checked for correctness, and measured on the target device—Ascend NPUs for CANN and NVIDIA GPUs for CUDA. Evaluations run in parallel when resources permit, while timed executions remain isolated. The harness records latency, available hardware profiles such as compute, memory, and synchronization behavior, as well as compilation and correctness diagnostics. Failed candidates are excluded from the validated kernels but still consume the compilation budget and may provide feedback for bounded repair.

Profile-guided kernel update. Validated candidates are compared with the current kernel, and the fastest one becomes the new incumbent. Its code and execution profile, together with outcomes from other candidates, guide the next round of retrieval and generation. Thus, later candidates depend on both *what was changed* and *what happened after execution*, rather than being independent Best-of-N samples.

Search history and evidence reuse. The program graph retains validated historical kernels and their measured outcomes. Identical source packages share compatible evaluation evidence even when reached through different paths; measurements are not counted twice, and cycle-forming links are rejected. This graph represents workload-local search experience, whereas K represents reusable expert knowledge. Execution updates the former but leaves the latter fixed within the workload.

Final confirmation and trajectory. The harness reserves budget for independent confirmation and returns the fastest confirmed generated kernel. If later refinements fail, an earlier validated kernel may remain the output; if no candidate passes validation, the run reports failure rather than using a reference fallback. For every attempt, the harness retains the parent, proposed change, generated code, execution profile, validation result, supporting context, and actual resource cost. This trajectory provides the evidence for context evolution in Section 4.

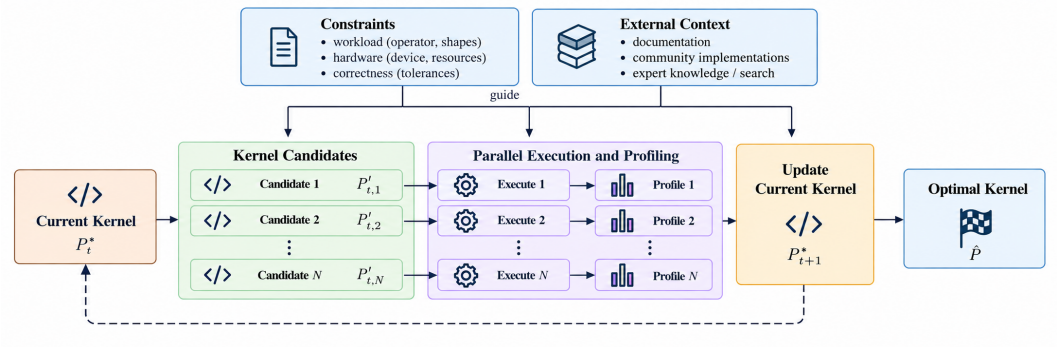


Figure 1. The AscendAgent optimization harness. Constraints and external context guide candidate generation. Candidate kernels are executed and profiled in parallel, and validated feedback updates the current kernel for the next round.

3.3 External Context for the Harness

The external context K provides platform-specific optimization knowledge that is difficult to recover from the workload specification alone. We organize this persistent expert context through three stages: construction, retrieval, and execution-grounded use.

Context construction. We collect knowledge from vendor documentation, community implementations, public repositories, and technical discussions. An LLM organizes these sources into reusable optimization knowledge describing *when* an optimization applies, *what* transformation to consider, *what effect* is expected, and *where* the knowledge comes from. Applicability conditions capture factors such as tensor shapes, memory layouts, hardware generations, and programming interfaces, since the same transformation may be effective in one setting but ineffective or invalid in another.

Context retrieval. During optimization, the harness retrieves compatible knowledge according to the workload, target platform, current kernel, and execution profile. Retrieval therefore adapts with the search state: early iterations may use general implementation strategies, while later iterations can focus on bottlenecks revealed by profiling. The retrieved context guides which transformations to try, but their actual utility is determined through compilation, correctness checking, and hardware measurement.

Motivation for evolving context. The persistent context remains frozen within each workload, while execution records which context guided a transformation, what effect was expected, and what was actually observed. These trajectories are valuable because the context is initially organized *before* observing how the harness uses it: even relevant knowledge may have missing conditions, poor priorities, incomplete coverage, or organization that does not fit the decisions encountered during search. Rather than changing the optimization procedure, AscendAgent replays these *context–transformation–expectation–outcome* relationships in Section 4 to refine existing knowledge, recover missing knowledge, and reorganize useful strategies. Only validated revisions are transferred to later workloads, while the harness, model weights, and evaluator remain fixed.

4 Evolving Context to Fit the Harness via Replay

Section 3 keeps the external context K fixed within each workload, while recording how retrieved knowledge leads to kernel changes and what execution actually shows. These trajectories reveal whether the supplied context fits the harness: a strategy may be useful only under certain conditions, consistently unproductive, effective only in combination with others, or missing entirely. AscendAgent therefore replays this execution evidence to improve the context while keeping the harness, model weights, and evaluator fixed. Figure 2 summarizes the process: external sources initialize the expert context, the fixed harness produces execution trajectories, and replay turns these trajectories into validated context updates for later workloads.

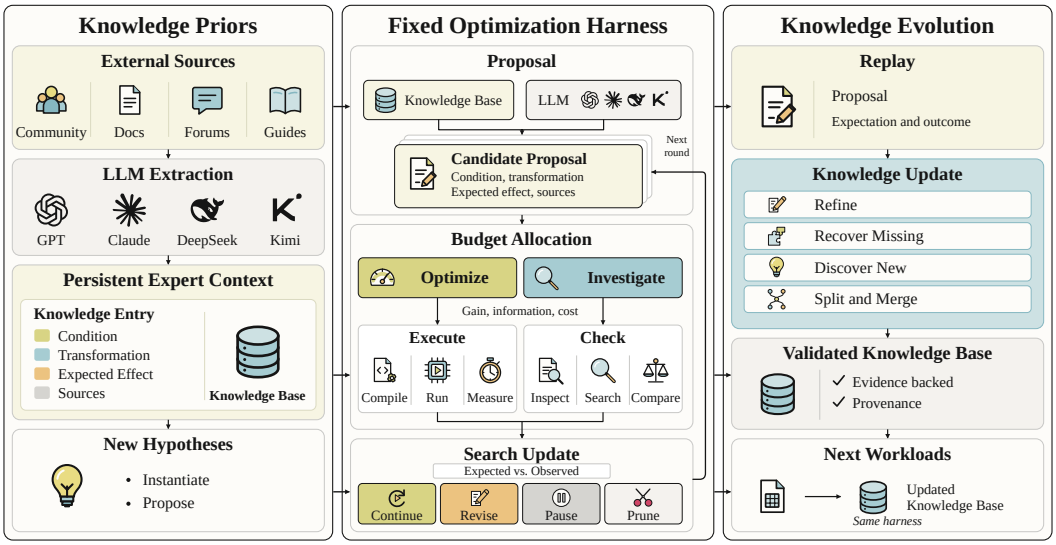


Figure 2. Overview of ASCENDAGENT. Left, external sources are organized into a knowledge base. Center, a fixed harness proposes changes, allocates budget, and uses execution or investigation to update the local search. Right, replay validates knowledge updates for later workloads. Only validated knowledge is transferred to later searches.

4.1 LEARNING FROM EXECUTION

Replay execution trajectories. Replay connects the retrieved context and proposed transformation with its expected and observed effects. Successful and failed attempts provide evidence about applicability and reliability, while valid parent–child comparisons reveal performance effects. Duplicate evaluations provide no additional evidence.

Compare expectation with observation. A transformation is assessed relative to the parent on which it was applied, so a useful change need not immediately beat the incumbent. Performance improvement alone also does not confirm its expected mechanism: for example, a claim about reduced memory traffic requires corresponding profile evidence. When multiple transformations are applied together, their effect is attributed to the combination unless controlled evidence separates them.

Identify context mismatches. Disagreement between expectation and execution can expose missing conditions, poor transformation priorities, or missing knowledge. Replay first checks the original knowledge and its source, and can investigate additional evidence when needed. Verified incompatibility can narrow an applicability condition, whereas slow or failed trials alone only provide evidence for lowering priority rather than declaring a transformation invalid.

4.2 EVOLVING THE CONTEXT

Context revision. Replay updates the context by adjusting transformation preferences, refining applicability conditions, recovering missing knowledge, and composing transformations that repeatedly work together. Each revision retains its supporting evidence. These changes modify *what to try and when*, while the optimization procedure itself remains unchanged.

Validation and commitment. A replayed revision is not immediately made persistent. Each proposed update is frozen and independently confirmed on source workloads separate from those used to construct it. Validation follows the claim: conditions require evidence for their applicability, while performance claims require repeated measurements. Unsupported revisions remain pending. We summarize the update as

$$K_{m+1} = U(K_m, \tau_m), \tag{5}$$

where τ_m is the execution trajectory from source workload m. If no revision passes confirmation, $K_{m+1} = K_m$.

Transfer to later workloads. The confirmed context is transferred to the next workload, while the program graph and workload-local statistics are reset. Thus, experience transfers through improved

context rather than previous kernel states or changes to the harness. On held-out targets, the context remains frozen and target results cannot revise or select an update.

Experiments

4.3 Experimental Setup

Benchmarks. The evaluation protocol covers CANN-Bench (Gao et al., 2026), KernelBench (Ouyang et al., 2025), MultiKernelBench's CUDA configuration (Wen et al., 2025), and TritonBench-G (Li et al., 2025). The task counts are 53, 250, 285, and 183, respectively. Each comparison uses the same task IDs and input suites.

Hardware and models. The hardware specifications are eight Huawei Ascend NPUs with SoC identifier `Ascend910_9382` and eight NVIDIA A100 GPUs with 80 GB memory per GPU. Device counts describe the systems rather than per-kernel parallelism. Table 2 covers five model configurations, with Kimi K3 as the primary configuration. Table 4 lists the required execution records.

Comparison methods. The main matrix retains KernelBand, K-Search, AB-MCTS-M, KernelFoundry, and AscendAgent. KernelBand uses hardware profiling and runtime clustering (Ran et al., 2025). Its full $K = 3$ configuration is separate from the local $K = 1$ control. K-Search couples program search with an intrinsic world model (Cao et al., 2026). AB-MCTS-M uses feedback to select between wider sampling and deeper refinement (Inoue et al., 2026). KernelFoundry combines a MAP-Elites archive, evolving prompts, and template-parameter search (Wiedemann et al., 2026).

Budget and validation. Each run permits 40 actual candidate compilations per task, including failures and recompilation. Newly compiled parameter instantiations also count. Each source and target search uses this cap. Final-confirmation recompilations count within it. Knowledge replay and independent source-side confirmation contribute to the separate learning cost. Reuse avoids a compilation but retains its lookup and controller cost. The repeat identifiers are 17, 42, and 101.

Metrics. A target score is the geometric mean of paired input speedups for one validated candidate. S_{40} averages the best target scores at attempt 40, with reference fallback at $1\times$. C_{40} averages the first attempt reaching $1.05\times$, capped at 40. R_{40} is the percentage reaching that threshold. We report means $\pm$ sample SD across three runs. Bold and underline mark the best and second distinct displayed means within each comparison group. Appendix B.2 describes the aggregation.

4.4 Main Results

In Table 2, AscendAgent with Kimi K3 (Moonshot AI, 2026) has the highest CANN-Bench speedup (1.40), but AB-MCTS-M reaches the threshold sooner (15.08 versus 18.05 compilations) and on more targets (80.50% versus 76.10%). Larger terminal gains therefore need not imply broader or earlier success. Different methods lead the CUDA benchmarks, and AscendAgent does not lead CANN-Bench with GPT-5.6 Sol. This variation motivates workload-dependent transformation choices.

Overall averages benchmarks within each run. Framework-relative gains do not establish closure of the community gap in Section 2.2, which requires matching anchors and workloads.

4.5 Detailed Analysis

Early progress and terminal quality. In Figure 3(d), AscendAgent trails some alternatives early but finishes highest in this local group. This motivates retaining multiple validated parents rather than judging a direction only by its first improvement. Panels (b,e) include overhead, so compilation counts alone do not establish cheaper search. These curves do not isolate the benefit of graph sharing or transformation replay. KernelBand here uses $K = 1$, distinct from the main table's $K = 3$ configuration.

Updating knowledge rather than accumulating it. In panels (c,f), repeated updates raise the endpoint success rate from 43.75% to 65.28% on Ascend and from 47.22% to 60.42% on CUDA. The repeated-update curves also contain regressions as more source workloads become available. Source-side confirmation therefore cannot guarantee improvement on every held-out target. One-

Table 2. Kernel search performance across five models and four benchmarks.

	Ascend			CUDA						Overall		
	CANN-Bench $N^\star = 53$			KernelBench $N^\star = 250$		MultiKernel Bench $N^\star = 285$		TritonBench-G $N^\star = 183$			Benchmark macro		
Method	$C_{40}\downarrow$	$S_{40}\uparrow$	$R_{40}\uparrow$	$C_{40}\downarrow$	$S_{40}\uparrow$	$C_{40}\downarrow$	$S_{40}\uparrow$	$C_{40}\downarrow$	$S_{40}\uparrow$	$R_{40}\uparrow$	$C_{40}\downarrow$	$S_{40}\uparrow$	$R_{40}\uparrow$
Kimi K3 (Moonshot AI, 2026)													
KernelBand	20.7±0.4	1.2±0.0	58.5±1.9	14.8±1.1	1.7±0.0	16.4±0.7	1.5±0.0	13.8±0.4	**1.8**±0.0	78.0±1.3	16.4±0.3	1.6±0.0	70.3±0.4
K-Search	18.8±2.1	1.3±0.1	69.2±7.1	14.9±0.9	1.9±0.0	13.3±0.7	1.7±0.1	16.4±0.9	1.7±0.1	74.7±2.3	15.8±0.8	1.6±0.0	76.7±2.3
AB-MCTS-M	**15.1**±1.3	1.3±0.1	**80.5**±5.8	12.4±1.2	1.8±0.0	13.7±0.4	**1.7**±0.0	**12.6**±0.3	1.6±0.1	**83.8**±0.8	13.4±0.5	1.6±0.0	82.4±2.2
KernelFoundry	20.7±2.7	1.3±0.0	66.7±7.6	15.1±0.5	**2.1**±0.0	16.1±0.7	1.6±0.0	18.0±1.2	1.6±0.0	74.3±3.4	17.5±1.1	1.6±0.0	75.1±2.8
AscendAgent	18.1±2.4	**1.4**±0.1	76.1±7.6	**10.1**±0.7	1.9±0.0	**10.1**±0.3	1.7±0.0	15.2±0.1	1.8±0.0	79.6±2.3	**13.4**±0.7	**1.7**±0.0	**83.5**±2.9
DeepSeek V4.1 Flash (DeepSeek, 2026)													
KernelBand	20.1±2.2	1.2±0.1	61.0±7.6	14.7±0.5	1.7±0.1	16.7±0.5	1.5±0.0	12.4±0.7	**1.8**±0.1	82.0±1.9	16.0±0.8	1.6±0.0	71.6±2.5
K-Search	20.0±1.8	1.3±0.0	66.0±5.0	14.8±0.8	1.9±0.0	13.2±0.4	1.7±0.0	14.8±1.3	1.6±0.0	79.6±2.8	15.7±0.9	1.6±0.0	77.1±2.5
AB-MCTS-M	**14.7**±0.3	1.3±0.0	**79.9**±1.1	11.7±0.1	1.8±0.0	13.7±0.2	**1.7**±0.1	**11.7**±0.4	1.6±0.0	**86.9**±0.0	13.0±0.1	1.6±0.0	**83.1**±0.2
KernelFoundry	21.0±1.3	1.3±0.0	65.4±5.8	15.4±0.9	**2.0**±0.0	16.9±0.3	1.6±0.0	16.3±1.0	1.6±0.0	78.3±2.3	17.4±0.5	1.6±0.0	75.5±1.8
AscendAgent	17.7±1.4	**1.4**±0.1	73.6±3.3	**9.8**±0.1	1.9±0.0	**9.7**±0.1	1.7±0.0	14.0±0.1	1.7±0.0	82.2±0.8	**12.8**±0.4	**1.7**±0.0	79.5±1.1
GPT-5.6 Sol (OpenAI, 2026)													
KernelBand	21.2±2.8	1.2±0.0	59.1±7.1	14.7±0.6	1.7±0.0	16.9±0.5	1.5±0.0	11.8±0.9	**1.8**±0.1	83.2±3.2	16.2±0.4	1.6±0.0	71.2±0.7
K-Search	19.7±1.5	**1.3**±0.0	68.6±6.6	14.3±1.4	**2.1**±0.1	12.7±0.7	**1.7**±0.0	13.5±0.7	1.8±0.0	83.6±3.0	15.1±0.4	**1.7**±0.0	79.6±0.8
AB-MCTS-M	**14.6**±2.5	1.3±0.1	**80.5**±8.9	11.7±1.3	1.9±0.1	13.9±1.2	1.7±0.1	**10.4**±0.6	1.6±0.0	**90.0**±1.6	**12.7**±0.6	1.6±0.0	**84.3**±1.7
KernelFoundry	21.8±1.2	1.3±0.0	64.2±3.3	15.7±0.9	2.1±0.1	16.8±0.8	1.6±0.0	15.9±0.8	1.6±0.1	80.3±2.7	17.5±0.6	1.6±0.1	75.8±1.8
AscendAgent	19.6±2.9	1.3±0.1	70.4±9.5	**10.4**±1.2	1.9±0.1	**10.4**±0.5	1.6±0.1	15.4±1.2	1.7±0.0	78.8±3.6	13.9±0.4	1.7±0.0	82.1±1.6
Claude Opus 4.8 (Anthropic, 2026)													
KernelBand	21.6±2.0	1.2±0.0	56.6±6.5	14.3±1.4	1.7±0.0	17.3±1.4	1.5±0.0	11.9±0.8	**1.8**±0.1	83.6±3.0	16.3±1.0	1.6±0.0	70.8±2.9
K-Search	19.7±1.4	1.3±0.0	67.3±4.8	14.5±1.1	2.0±0.1	**13.7**±0.9	1.7±0.0	13.6±0.5	1.7±0.0	83.2±1.4	15.4±0.5	1.7±0.0	78.3±1.7
AB-MCTS-M	**14.7**±1.0	1.3±0.0	**80.5**±2.2	**11.4**±1.2	1.9±0.1	13.8±1.0	**1.8**±0.1	**11.1**±0.3	1.7±0.0	**89.4**±1.1	**12.7**±0.5	1.7±0.0	**84.3**±1.0
KernelFoundry	22.8±2.0	1.3±0.0	62.3±6.8	15.0±1.2	**2.2**±0.1	16.6±1.1	1.7±0.0	15.4±0.1	1.7±0.0	81.1±2.2	17.4±0.8	1.7±0.0	75.6±2.7
AscendAgent	18.4±0.9	**1.4**±0.1	72.3±4.8	14.6±1.1	2.0±0.0	15.0±1.4	1.7±0.0	13.6±0.3	1.8±0.0	85.1±0.8	15.4±0.3	**1.7**±0.0	79.4±1.9
GLM-5.3 (Z.ai, 2026)													
KernelBand	22.7±1.9	1.2±0.0	52.8±5.7	15.4±0.4	1.7±0.0	17.7±1.0	1.5±0.0	13.1±1.3	**1.8**±0.1	79.8±2.4	17.2±0.6	1.5±0.0	67.8±1.5
K-Search	22.4±2.2	1.3±0.0	58.5±5.0	15.6±0.7	1.9±0.0	14.0±0.7	1.7±0.0	15.7±1.4	1.6±0.0	76.3±4.4	16.9±0.5	1.6±0.0	73.3±1.1
AB-MCTS-M	**17.0**±0.8	1.3±0.1	**73.6**±3.3	13.4±1.0	1.8±0.1	14.3±0.2	**1.7**±0.0	**12.2**±1.2	1.6±0.1	**84.2**±3.0	14.2±0.6	1.6±0.0	79.6±1.6
KernelFoundry	24.4±1.8	1.3±0.1	53.5±5.5	15.8±0.6	**2.0**±0.0	18.1±0.5	1.6±0.0	17.4±1.7	1.6±0.0	74.9±5.9	18.9±0.6	1.6±0.0	70.0±2.1
AscendAgent	18.9±0.6	**1.4**±0.1	73.0±2.9	**10.9**±0.6	1.9±0.0	**10.9**±0.5	1.6±0.1	15.7±1.1	1.7±0.1	78.2±2.8	**14.1**±0.5	**1.6**±0.0	**81.2**±1.1

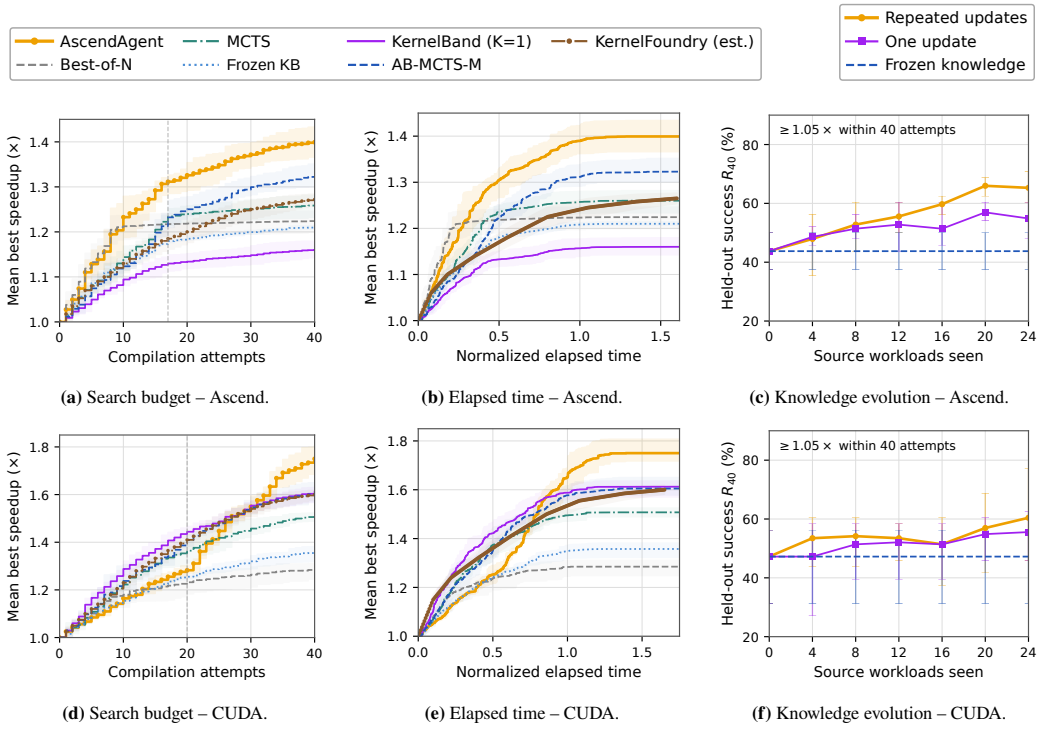
Means ± sample SD across three runs. $N^\star$ gives task counts.C_{40} is capped first-hit cost, S_{40} is mean best speedup, and R_{40} is the percentage reaching $1.05\times$. Overall averages four benchmark aggregates within each run before computing SD. KernelBand uses the full $K = 3$ configuration. The local $K = 1$ control is separate.

update checkpoints independently revise the initial knowledge using each prefix. Repeated updates inherit previous revisions. The target list and harness remain fixed.

Learning must repay its cost. At 100 target uses, Table 12 gives lower amortized Ascend cost to one update than repeated updates (26.24 versus 27.23 minutes), despite the latter's higher success rate. On CUDA, frozen knowledge remains cheapest (10.40 minutes), compared with 13.51 for repeated updates. Better target performance therefore need not repay acquisition, replay, and confirmation costs.

4.6 Ablation Study

Table 3 separates procedural and knowledge priors (see Section 2.3). Best-of-N and MCTS vary search behavior. Frozen-KB Harness and AscendAgent instead share the program graph, evaluation reuse, and controller. Only the inherited knowledge differs in this comparison. The table reports S_{40} and R_{40} for all four methods. AscendAgent has paired S_{40} gains of 0.19 ± 0.06 on CANN-Bench and 0.39 ± 0.02 on TritonBench-G relative to Frozen-KB Harness. Their R_{40} values increase from 72.33% to 76.10% and from 76.50% to 79.60%, respectively. Appendix Table 8 gives the corresponding success counts.



(a) Search budget – Ascend. **(b)** Elapsed time – Ascend. **(c)** Knowledge evolution – Ascend.

(d) Search budget – CUDA. **(e)** Elapsed time – CUDA. **(f)** Knowledge evolution – CUDA.

Figure 3. Search efficiency and knowledge evolution. The top row is CANN-Bench and the bottom row is TritonBench-G. (a,d) Best speedup against the compilation budget. (b,e) Completion time, normalized by Best-of-N. (c,f) Held-out success rate for frozen, single-update, and repeated-update knowledge schedules. Each one-update checkpoint uses its source prefix in an independent update from the initial knowledge. Frozen KB abbreviates Frozen-KB Harness.

Table 3. Knowledge evolution and search baselines at 40 compilations.

	CANN-Bench		TritonBench-G	
Arm	$S_{40}\uparrow$	R_{40} (%) $\uparrow$	$S_{40}\uparrow$	R_{40} (%) $\uparrow$
Best-of-N	1.22±0.03	58.49±1.89	1.29±0.04	66.85±2.28
MCTS	1.26±0.03	72.96±7.14	1.51±0.03	73.41±3.11
Frozen-KB Harness	1.21±0.00	72.33±7.63	1.36±0.01	76.50±2.84
AscendAgent	**1.40**±0.06	**76.10**±7.63	**1.75**±0.03	**79.60**±2.28

Means ± sample SD across three runs with Kimi K3. R_{40} is the percentage reaching $1.05\times$ within 40 compilations.

The frozen knowledge variant scores below MCTS on both platforms. This pattern shows how relevant advice can still misdirect a finite search budget, so retrieval alone is insufficient. The updated score does not identify which conditions, preferences, or compositions mattered. That attribution requires linked execution records. The four rows compare distinct search strategies and knowledge variants. A causal analysis requires the matched controls in Appendix B.1.

5 Conclusion

We presented AscendAgent, which connects a fixed kernel-optimization harness to an evolving knowledge base. Workflow replay links advice to execution evidence, and independent confirmation controls what later workloads inherit. The evaluation separates local search quality from cross-workload knowledge evolution.

Reproducibility Statement

The appendix specifies the 40-attempt search protocol and its configuration controls. The source includes frozen Figure 3 records, editable plots, and a script that recomputes the local tables from those records. The original budget-specific tables are archived without being relabelled. Hardware and model identifiers, implementation revisions, and matched empirical traces remain required for the main benchmark matrix. The current package is a manuscript draft, not a validated benchmark archive.

Use of AI Assistance

AI assistance was used to revise prose, tables, and figure layouts, integrate the tables and figures, and check the compiled manuscript. Unmeasured 40-attempt cells remain NM. No new hardware experiments were performed during this revision. The authors are responsible for the method, implementation, and empirical claims.

References

Anthropic. Introducing Claude Opus 4.8, 2026. URL https://www.anthropic.com/news/claude-opus-4-8.

Shiyi Cao, Ziming Mao, Joseph E Gonzalez, and Ion Stoica. K-Search: LLM kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

DeepSeek. DeepSeek API change log and thinking-mode documentation, 2026. URL https://api-docs.deepseek.com/updates/.

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

Yuichi Inoue, Kou Misaki, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. Wider or deeper? scaling LLM inference-time compute with adaptive branching tree search. *Advances in Neural Information Processing Systems*, 38:35448–35484, 2026.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, 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*, pp. 23053–23066, 2025.

Moonshot AI. Kimi K3 API documentation, 2026. URL https://platform.kimi.ai/docs/guide/kimi-k3-quickstart.

OpenAI. GPT-5.6 Sol model documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol.

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.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, et al. KernelBand: Steering LLM-based kernel optimization via hardware-aware multi-armed bandits. *arXiv preprint arXiv:2511.18868*, 2025.

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*, 2025.

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

Z.ai. GLM-5.3 model documentation, 2026. URL https://docs.z.ai/guides/llm/glm-5.3.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026a.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879, 2020.

A Harness and Replay Specification

A.1 Execution Records and Budget Accounting

A proposal $q_t = (P_t, a_t, h_t, E_t)$ is recorded before execution. Its expectation h_t names the observable quantity and an expected direction or range. An initial proposal uses $P_t = \perp$ and constructs a program rather than modifying a validated parent. A failed program can be passed back to the model for repair, but is not admitted to $\mathcal{P}_t$. The record retains the original proposal, generated code, completed checks, measurements, source identifiers, and actual resource costs. Unavailable observations remain unknown.

All attempted candidate compilations, including initial candidates and failures, consume B. A comparison that compiles a new candidate also consumes this budget. Inspection, retrieval, and static rejection consume their own resource allowances and elapsed time, but do not count as compilations that never occurred. Confirmation measurements and controller time remain in the reported device and wall-clock costs. Common proposal and elapsed-time caps prevent unlimited work between compilations.

A reused evaluation requires matching code, input suite, hardware, compiler settings, precision, and measurement protocol. A cache reference is not a new observation. Compatible evaluations can support a shared program node within a workload. New workloads start with empty program graphs and local statistics.

A.2 Graph Selection and Transformation Allocation

Let P_0 be the first validated generated program and $P_t^\star$ the incumbent before an expansion. Using the latency definition in Section 2.2, we define the root-relative log speedup $\ell(P) = \log[T_d(P_0, x)/T_d(P, x)]$ for valid programs. This quantity is distinct from the reference gap $g_{x,d}^{(r)}$ in Eq. (2). Each independent valid evaluation j provides

$$r_j = \tanh\big(\ell(P_j)\big). \tag{6}$$

Since $\tanh$ is increasing, this transformation preserves the latency ordering and bounds valid rewards in $(-1, 1)$. Substituting the latency ratio gives

$$r_j = \frac{T_d(P_0, x)^2 - T_d(P_j, x)^2}{T_d(P_0, x)^2 + T_d(P_j, x)^2}. \tag{7}$$

A program with the root latency has reward zero. Faster programs have positive rewards, and slower valid programs have negative rewards. The reward identity and value are retained in an evidence map at the evaluated node and its ancestors. Multiple paths never duplicate one evaluation. Invalid candidates provide no performance evidence. New parents inherit the existing evidence of a merged node, including its descendants.

At a node P visited $N(P)$ times, progressive widening permits at most $\lceil wN(P)^\alpha \rceil$ proposals before traversal, with both distinct children and attempted requests checked against this allowance. This is the progressive-widening restriction on branching. Otherwise, the selected outgoing edge maximizes

$$V(Q) + c_{\text{uct}} \sqrt{\frac{\log(N(P)+1)}{N(P \to Q)}}, \qquad V(Q) = \max_{j \in \mathcal{E}(Q)} r_j, \tag{8}$$

where $\mathcal{E}(Q)$ is the set of independent evidence identities reachable from Q. Edge counts record traversals separately even when children are shared. An unvisited edge is selected before evaluating the finite exploration score. A child without valid evidence receives no performance value until it passes validation. The exploration term follows the visit-count form of UCT, while the exploitation value uses the best reachable evaluation. The default values are $w = 1.5$, $\alpha = 0.5$, and $c_{\text{uct}} = 0.35$, with maximum traversal depth eight. This best-reachable value is a search heuristic, not an estimate of the next expansion's expected return or a standard mean-return UCT guarantee.

For valid parent and child, the observed transformation effect is

$$\Delta_t = \log \frac{T_d(P_t, x)}{T_d(P_t', x)}. \tag{9}$$

It is undefined if either timing is invalid. A positive Δ_t can improve an earlier parent without beating the incumbent. For any valid child, log ratios decompose as

$$\ell(P') = \ell(P) + \Delta. \tag{10}$$

Let $b_t(P) = \ell(P_t^\star) - \ell(P)$ be the improvement needed to match the incumbent. The amount by which the child exceeds it is therefore

$$[\ell(P') - \ell(P_t^\star)]_+ = [\Delta - b_t(P)]_+. \tag{11}$$

Conditioning on validity and assigning no improvement to an invalid candidate gives expected improvement multiplied by the probability of validity. Dividing by expected elapsed cost yields the action score

$$A_t(P, a) = \frac{\widehat{p}_{\text{valid}}(P, a)\widehat{\mathbb{E}}[(\Delta - b_t(P))_+ \mid \text{valid}]}{\widehat{C}_t(P, a)}. \tag{12}$$

Here $[v]_+ = \max(0, v)$ and $\widehat{C}_t$ estimates elapsed generation and evaluation time. The validity and gain estimates use independent local outcomes and compatible frozen source statistics. For the normal approximation $\Delta \mid \text{valid} \sim \mathcal{N}(\mu, \sigma^2)$, let $z = (\mu - b_t(P))/\sigma$ for $\sigma > 0$. Integrating the positive tail gives

$$\begin{aligned} \text{EI}_t(P, a) &= \int_{b_t(P)}^{\infty} (u - b_t(P)) \frac{1}{\sigma} \phi\left(\frac{u - \mu}{\sigma}\right) du \\ &= (\mu - b_t(P))\Phi(z) + \sigma\phi(z), \end{aligned} \tag{13}$$

where ϕ and Φ are the standard normal density and distribution functions. The limit at $\sigma = 0$ is $[\mu - b_t(P)]_+$. Cost estimates must be positive. Reuse is handled by the compatibility checks rather than assigning a new independent reward to a zero-compilation lookup. Missing hardware or structural facts do not establish compatibility. Deterministic replay avoids estimated generation cost. With probability 0.15, selection explores an available transformation uniformly instead of taking the largest score. Free mutation has a separate positive probability, initially 0.25, which increases under stagnation or recent failures. When no transformation is available, mutation is used. The score estimates immediate improvement per unit cost. It does not establish finite-budget convergence or an optimal search policy. All allocation settings and optional extensions must match across compared knowledge variants.

Ordinary expansions start from valid programs. Compilation failures enter a bounded repair queue. Repairs consume normal resource allowances. Repeated proposals and cache hits incur their actual generation and lookup cost but create no independent timing evidence. The controller reserves resources to confirm the selected program and initial valid program. A regressing or invalid candidate falls back to that generated initial program. Incomplete confirmation is reported as unconfirmed. If no valid initial program is obtained, the run reports failure. Information retrieval can supply proposal context within its declared allowances, but is not a separate bandit action in this search algorithm.

Algorithm 1 separates parent selection, program expansion, evaluation, and feedback. AFlow uses a similar staged presentation for workflow search (Zhang et al., 2025). Here the searched objects are kernel programs and the workflow is fixed. SELECTPARENT applies progressive widening and Equation 8, with bounded repair requests handled by the repair queue. PROPOSE uses Equation 12, uniform exploration, or free mutation. PROPOSE and APPLY debit generation and transformation costs. APPLY retains application checks for replayed and generated edits. EVALUATE charges every attempted compilation. REUSE debits lookup cost and links existing evidence without adding a sample. CONFIRM checks the incumbent and the initial generated program within the reserved resources. It returns the confirmed faster choice, or reports unconfirmed status if the required checks cannot finish.

Algorithm 1 Knowledge-guided program search with frozen knowledge

Input Workload x, platform d, knowledge $\mathcal{K}_m$, compilation budget B
Output Confirmed program or failure/unconfirmed status, execution record τ_m

Initialize resource ledger $\mathbf{b}$ with a final-confirmation reserve
$(P_0, \tau_m) \leftarrow \textsc{Initialize}(x, d, \mathcal{K}_m, \mathbf{b})$
if P_0 is invalid **then**
    return $(\text{failure}, \tau_m)$
end if
Initialize graph $\mathcal{G}$ and validated archive $\mathcal{P}$ with P_0
$P^\star \leftarrow P_0$
while the ledger permits another proposal **do**
    $P \leftarrow \textsc{SelectParent}(\mathcal{G}, \mathbf{b})$
    $q \leftarrow \textsc{Propose}(P, \mathcal{K}_m, \mathcal{G}, \mathbf{b})$
    $P' \leftarrow \textsc{Apply}(P, q, \mathbf{b})$
    Charge this proposal to the widening allowance of P
    if $P' = \perp$ because an application check failed **then**
        Record the rejection and its charged cost in τ_m
        continue
    end if
    if P' has compatible cached evidence **then**
        Link its existing node if the link preserves acyclicity
        $(o, c) \leftarrow \textsc{Reuse}(P', \mathbf{b})$ ▷ Retain evaluation identities
        Propagate stored evidence to new ancestors by evaluation identity
    else
        $(o, c) \leftarrow \textsc{Evaluate}(P', x, d, \mathbf{b})$
        Register P' and its outcome in $\mathcal{G}$
        if P' passes correctness checks and timing **then**
            Update $\mathcal{P}$ and $P^\star$ using $T_d(P', x)$
            Propagate each new evidence identity to the ancestors of P'
        else
            Queue an eligible repair within the remaining allowances
        end if
    end if
    Append (q, P', o, c) and its total charged cost to τ_m
    Update action statistics from independent outcomes
end while
$\widehat{P} \leftarrow \textsc{Confirm}(P^\star, P_0, x, d, \mathbf{b})$
return $(\widehat{P}, \tau_m)$

A.3 Knowledge Update Validation

Each proposed update is fixed before independent confirmation. Related workloads and duplicate sources must be grouped before constructing the revision and confirmation partitions. Let $\mathcal{D}$ be a confirmation set of diagnostic decisions. Each decision z has a fixed set $\mathcal{C}_z$ of validated, measured programs, including an unchanged parent. The performance score $S_z(P)$ is speedup over that parent under the same input and timing protocol. Let $\widehat{P}_{\mathcal{K},z}$ be the program selected using the frozen harness and knowledge $\mathcal{K}$. The recorded performance scores are withheld from this selection. If the advice rejects every proposed change, the choice is the parent. The diagnostic decision loss is

$$\mathcal{L}(\mathcal{K}, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{z \in \mathcal{D}} \left[\max_{P \in \mathcal{C}_z} S_z(P) - S_z(\widehat{P}_{\mathcal{K},z}) \right]. \tag{14}$$

The same $\mathcal{C}_z$ is used for the current and proposed knowledge bases. This evaluates their choices among already measured alternatives, not the prospective performance of a complete future search.

The unchanged parent is included in every candidate set. Rejecting all edits therefore still produces an evaluable choice. For $|\mathcal{K}|$ knowledge entries and $n_c(\mathcal{K})$ explicit application conditions, we define

$$J(\mathcal{K}, \mathcal{D}) = \mathcal{L}(\mathcal{K}, \mathcal{D}) + \lambda_e |\mathcal{K}| + \lambda_c n_c(\mathcal{K}), \tag{15}$$

where the nonnegative weights are fixed before confirmation. A proposal $\mathcal{K}'$ is accepted only if its required claim checks pass and

$$J(\mathcal{K}_m, \mathcal{D}) - J(\mathcal{K}', \mathcal{D}) \geq \delta, \qquad \delta > 0. \tag{16}$$

The margin δ is also fixed in advance. Expanding this inequality shows that the reduction in decision loss must cover both the acceptance margin and the change in entry and condition penalties. The same candidate sets are used for both knowledge snapshots, so narrowing applicability cannot remove a difficult decision from the comparison.

Algorithm 2 applies this rule after source-workload replay. Failure retains the current knowledge base. A revised proposal requires fresh confirmation evidence. Rejected updates remain in the learning-cost record. Held-out targets do not influence revision generation, confirmation, or checkpoint selection.

Algorithm 2 Source-workload replay and confirmed knowledge updates

Input Initial knowledge $\mathcal{K}_0$, ordered source workloads $x_0, \ldots, x_{M-1}$
Input Platform d, budget B, disjoint confirmation data $\mathcal{D}_m$
Output Knowledge $\mathcal{K}_M$ frozen before held-out target evaluation

for $m \leftarrow 0, \ldots, M-1$ **do**
  $(\widehat{P}_m, \tau_m) \leftarrow \mathcal{H}(x_m, d, \mathcal{K}_m, B)$
  $\mathcal{K}' \leftarrow \textsc{Replay}(\mathcal{K}_m, \tau_m)$
  Freeze $\mathcal{K}'$ before checking it on $\mathcal{D}_m$
  $v_m \leftarrow \textsc{ValidateClaims}(\mathcal{K}', \mathcal{D}_m)$
  Compare $J(\mathcal{K}', \mathcal{D}_m)$ and $J(\mathcal{K}_m, \mathcal{D}_m)$
  if v_m and the acceptance condition in Eq. (16) holds **then**
    $\mathcal{K}_{m+1} \leftarrow \mathcal{K}'$
  else
    $\mathcal{K}_{m+1} \leftarrow \mathcal{K}_m$
  end if
  Record replay and confirmation costs, including rejected updates
  Discard program graphs and workload-local statistics
end for
return $\mathcal{K}_M$

B Evaluation Protocol

B.1 Configuration and Controlled Comparison

Table 4 lists the configuration records. Table 5 specifies the matched controls for knowledge evolution. Frozen-KB Harness keeps $\mathcal{K}_0$, whereas AscendAgent uses source-confirmed $\mathcal{K}_m$. The target-time workflow is otherwise identical. Both variants start each target with a fresh program archive and local search statistics.

Table 4. Evaluation configuration and required run records.

Item	Specification
Platforms and comparison groups	
Ascend system	8× Ascend910_9382 NPUs for CANN-Bench
CUDA system	8× NVIDIA A100 80 GB for three CUDA benchmarks
Main methods and models	Five methods and five models (Table 2)
Local controls	Search baselines and knowledge variants (Tables 3, 6)
KernelBand configurations	Main $K = 3$, local control $K = 1$
AB-MCTS-M adapter	Official TreeQuest ABMCTSM revision
Search and measurement	
Compilation cap	$B = 40$, failures and recompilation included
Success threshold	$1.05\times$ after full validation within B
Repeat seeds	17, 42, 101
Timing contract	Warmup, launches, allocation, synchronization, confirmation
Elapsed-time normalization	Median Best-of-N time at B, separately by platform
Records required for measured runs	
Search settings	Widening, exploration, proposal probabilities
Evaluation identity	Source-package hashes and independent evidence IDs
Adapter settings	Batching, reward mapping, implementation changes
Knowledge settings	Initial hash, source order, update and confirmation budgets
Update acceptance	Fixed confirmation rule and supporting evidence

Table 5. Matched controls for the knowledge-update comparison.

Control	AscendAgent	
Shared across both variants		
Tasks and references	Same task IDs, inputs, references, and tolerances	
Model and repeats	Same served model, generation settings, and run IDs	
Initial knowledge	Same $\mathcal{K}_0$ and source-workload order	
Search controller	Same proposal, graph-search, and evidence-reuse rules	
Evaluation	Same compilation, correctness, and timing protocol	
Target budget	$B = 40$, including final recompilation	
Other search limits	Same proposal and wall-time caps	
Learning cost	Source learning and update confirmation charged separately	
Within each workload	Knowledge snapshot and harness remain fixed	
Between workloads	Program graph and local statistics reset	
Varied factor		
Inherited knowledge	Initial $\mathcal{K}_0$	Source-confirmed $\mathcal{K}_m$

B.2 Metric Definitions

For a fixed input suite $\mathcal{Z}_x$, the paired speedup is

$$s(P, x) = \exp\left(\frac{1}{|\mathcal{Z}_x|} \sum_{z \in \mathcal{Z}_x} \log \frac{t_{\text{ref}}(z)}{t_P(z)}\right). \tag{17}$$

Each ratio uses the same launch and allocation contract. The original experimental summaries use $S_{x,r}(b)$ for the best validated speedup on target x and repeat r after b compilations, with reference fallback at $1\times$. This fallback is not a newly generated correct program. Set $B = 40$ and $\theta = 1.05$. The first successful attempt is

$$T_{x,r} = \inf\{b \in \{1, \ldots, B\} : S_{x,r}(b) \geq \theta\}. \tag{18}$$

An empty set gives $T_{x,r} = \infty$. With N targets and n repeats, the retained aggregate metrics are

$$S_B = \frac{1}{Nn} \sum_{x=1}^{N} \sum_{r=1}^{n} S_{x,r}(B). \tag{19}$$

$$C_B = \frac{1}{Nn} \sum_{x=1}^{N} \sum_{r=1}^{n} \min\{T_{x,r}, B\}. \tag{20}$$

$$R_B = \frac{100}{Nn} \sum_{x=1}^{N} \sum_{r=1}^{n} \mathbf{1}\{T_{x,r} \le B\}. \tag{21}$$

Failures remain in the cohort. A failure and a success at the final attempt both contribute B to C_B, but differ in R_B. The cap is applied before averaging. A threshold crossing of a mean curve does not determine the mean first-hit time or the target success rate.

B.3 Transfer Schedule and Inclusive Cost

Frozen, single-update, and repeated-update schedules use disjoint source, confirmation, and held-out target groups. The frozen schedule retains the initial knowledge throughout. Each single-update checkpoint starts independently from $\mathcal{K}_0$ and performs one update using the available source prefix. Its curve compares source-data amounts, not successive updates to one knowledge snapshot. The repeated-update schedule instead carries confirmed knowledge forward across successive source workloads. Every checkpoint is evaluated on the same target list. Target results cannot select an update or checkpoint, and related tasks must be grouped before splitting.

For source-learning time L and complete target-search times T_h, amortized cost after H uses is

$$A(H) = \frac{L + \sum_{h=1}^{H} T_h}{H}. \tag{22}$$

Learning includes acquisition, extraction, replay, source searches, and accepted and rejected confirmations. Each target search includes retrieval, cache lookup, controller work, compilation, validation, and timing. The per-target compilation cap does not absorb source-learning cost. Report L, target-search time, and $A(H)$ separately for each backend and schedule. Table 12 reports these cost components. Lower target-search time alone does not establish lower inclusive cost.

C Supplementary Results

The following tables follow the protocol in Section 4.2. Local search uses 53 CANN-Bench and 183 TritonBench-G tasks. Knowledge transfer uses 48 targets per platform. These groups are aggregated separately.

C.1 Local Search and Run Variation

Table 6 reports the TritonBench-G 40-attempt summaries for Kimi K3. KernelBand uses the local $K = 1$ control, distinct from the full $K = 3$ configuration in Table 2. Table 7 reports the ranges over the three runs at the main benchmark denominators.

Table 6. TritonBench-G local controls at 40 compilations with Kimi K3.

Method	$C_{40}\downarrow$	$S_{40}\uparrow$	$R_{40}\uparrow$
Best-of-N	18.24±0.82	1.29±0.04	66.85±2.28
MCTS	16.21±1.10	1.51±0.03	73.41±3.11
Frozen-KB Harness	<u>14.40</u>±1.31	1.36±0.01	76.50±2.84
KernelBand (K=1)	16.97±0.89	<u>1.61</u>±0.02	72.50±2.80
AB-MCTS-M	**12.63**±0.28	<u>1.61</u>±0.06	**83.79**±0.83
KernelFoundry	17.95±1.24	1.60±0.04	74.32±3.41
AscendAgent	15.18±0.14	**1.75**±0.03	<u>79.60</u>±2.28

Table 7. Variation across three runs at 40 compilations.

Method	C_{40} range	S_{40} range	R_{40} range (%)
CANN-Bench, 53 targets			
Best-of-N	[21.38, 23.02]	[1.19, 1.24]	[56.60, 60.38]
MCTS	[16.00, 19.02]	[1.24, 1.29]	[67.92, 81.13]
Frozen-KB Harness	[14.28, 17.92]	[1.20, 1.21]	[67.92, 81.13]
KernelBand (K=1)	[18.66, 20.87]	[1.14, 1.17]	[62.26, 69.81]
AB-MCTS-M	[13.89, 16.45]	[1.28, 1.40]	[75.47, 86.79]
KernelFoundry	[17.85, 23.09]	[1.26, 1.29]	[62.26, 75.47]
AscendAgent	[15.30, 19.43]	[1.35, 1.47]	[71.70, 84.91]
TritonBench-G, 183 targets			
Best-of-N	[17.55, 19.15]	[1.25, 1.34]	[65.03, 69.40]
MCTS	[15.21, 17.38]	[1.47, 1.54]	[69.95, 75.96]
Frozen-KB Harness	[13.45, 15.89]	[1.35, 1.36]	[73.22, 78.14]
KernelBand (K=1)	[16.32, 17.98]	[1.59, 1.64]	[69.40, 74.86]
AB-MCTS-M	[12.31, 12.82]	[1.56, 1.67]	[83.06, 84.70]
KernelFoundry	[16.58, 19.00]	[1.58, 1.65]	[71.58, 78.14]
AscendAgent	[15.07, 15.33]	[1.72, 1.78]	[77.05, 81.42]

C.2 Success Counts and Threshold Sensitivity

Table 8 exposes the integer counts behind the success rates. Table 9 varies the threshold while retaining each task group.

Table 8. Integer success counts and corresponding rates.

Method	Run 17	Run 42	Run 101	R_{40} (%)
CANN-Bench, 53 targets				
Best-of-N	31/53	30/53	32/53	58.49±1.89
MCTS	37/53	36/53	43/53	72.96±7.14
Frozen-KB Harness	36/53	36/53	43/53	72.33±7.63
KernelBand (K=1)	33/53	33/53	37/53	64.78±4.36
AB-MCTS-M	42/53	40/53	46/53	80.50±5.76
KernelFoundry	33/53	33/53	40/53	66.67±7.63
AscendAgent	38/53	38/53	45/53	76.10±7.63
TritonBench-G, 183 targets				
Best-of-N	121/183	119/183	127/183	66.85±2.28
MCTS	136/183	128/183	139/183	73.41±3.11
Frozen-KB Harness	143/183	134/183	143/183	76.50±2.84
KernelBand (K=1)	134/183	127/183	137/183	72.50±2.80
AB-MCTS-M	153/183	152/183	155/183	83.79±0.83
KernelFoundry	134/183	131/183	143/183	74.32±3.41
AscendAgent	149/183	141/183	147/183	79.60±2.28

Table 9. Success-rate sensitivity to the speedup threshold.

Method	$\tau = 1.05$	$\tau = 1.10$	$\tau = 1.25$	$\tau = 1.50$
CANN-Bench, 53 targets				
Best-of-N	58.49±1.89	57.86±2.18	29.56±2.18	13.21±5.66
MCTS	72.96±7.14	69.81±5.66	40.88±2.18	15.09±1.89
Frozen-KB Harness	72.33±7.63	66.67±6.07	30.82±2.88	9.43±3.27
KernelBand (K=1)	64.78±4.36	58.49±4.99	22.01±7.63	3.77±1.89
AB-MCTS-M	**80.50**±5.76	**77.36**±9.43	49.06±9.98	20.13±2.88
KernelFoundry	66.67±7.63	64.78±6.07	42.77±5.76	19.50±2.88
AscendAgent	76.10±7.63	74.84±5.45	**58.49**±8.22	**30.19**±3.77
TritonBench-G, 183 targets				
Best-of-N	66.85±2.28	65.57±2.84	43.53±5.39	19.49±4.96
MCTS	73.41±3.11	72.50±4.14	61.20±4.47	38.62±0.63
Frozen-KB Harness	76.50±2.84	73.04±4.24	52.46±4.27	23.86±2.58
KernelBand (K=1)	72.50±2.80	71.58±3.58	64.66±3.29	47.54±3.32
AB-MCTS-M	**83.79**±0.83	**83.24**±1.26	71.22±0.32	44.81±0.55
KernelFoundry	74.32±3.41	73.41±2.80	64.12±3.01	44.99±0.63
AscendAgent	79.60±2.28	79.24±2.89	**72.13**±3.04	**54.28**±0.63

C.3 Knowledge Evolution and Inclusive Cost

Tables 10 and 11 report the 48-target comparison, including the shared initialization and unchanged frozen schedule. Each one-update checkpoint is an independent update from $\mathcal{K}_0$ using that source prefix, as specified in Appendix B.3. Table 12 combines those success rates with the cost components. Its amortization follows Equation 22. The cost columns are not recoverable from binary target outcomes alone.

Table 10. Target success after 24 source workloads under three knowledge schedules.

Schedule	R_{40} (%)	**Min–max (%)**	ΔR_{40} (pp)
Ascend			
Frozen knowledge	43.75±6.25	[37.50, 50.00]	0.00±0.00
One update	54.86±6.36	[47.92, 60.42]	↑ +11.11±1.20
Repeated updates	**65.28**±5.24	[60.42, 70.83]	↑ +21.53±11.47
CUDA			
Frozen knowledge	47.22±13.87	[31.25, 56.25]	0.00±0.00
One update	55.56±8.67	[45.83, 62.50]	↑ +8.33±5.51
Repeated updates	**60.42**±15.73	[45.83, 77.08]	↑ +13.19±8.42

Table 11. Target success rates as source knowledge accumulates.

	Ascend			CUDA		
K	**Frozen**	**One update**	**Repeated**	**Frozen**	**One update**	**Repeated**
0	**43.75**±6.25	**43.75**±6.25	**43.75**±6.25	**47.22**±13.87	**47.22**±13.87	**47.22**±13.87
4	43.75±6.25	**48.61**±3.18	47.92±11.02	47.22±13.87	47.22±17.47	**53.47**±8.67
8	43.75±6.25	51.39±4.34	**52.78**±8.42	47.22±13.87	51.39±10.28	**54.17**±9.08
12	43.75±6.25	52.78±8.42	**55.56**±5.24	47.22±13.87	52.08±10.83	**53.47**±6.70
16	43.75±6.25	51.39±7.89	**59.72**±3.18	47.22±13.87	**51.39**±10.28	**51.39**±12.21
20	43.75±6.25	56.94±3.18	**65.97**±3.18	47.22±13.87	54.86±7.89	**56.94**±13.87
24	43.75±6.25	54.86±6.36	**65.28**±5.24	47.22±13.87	55.56±8.67	**60.42**±15.73

Table 12. Target performance and inclusive cost after 24 source workloads.

Schedule	$C_{40}\downarrow$	$S_{40}\uparrow$	$R_{40}\uparrow$	L (min)	T (min)	$A(100)$ (min)
Ascend						
Frozen knowledge	27.41±1.47	1.21±0.01	43.75±6.25	**22.01**±0.90	28.65±0.41	28.87±0.42
One update	23.15±1.75	1.30±0.01	54.86±6.36	550.37±9.04	20.73±0.39	**26.24**±0.45
Repeated updates	**19.08**±1.52	**1.38**±0.03	**65.28**±5.24	931.93±12.61	**17.91**±1.03	27.23±1.01
CUDA						
Frozen knowledge	25.80±4.22	1.36±0.02	47.22±13.87	**18.98**±0.58	10.21±0.44	**10.40**±0.43
One update	21.90±2.73	1.57±0.06	55.56±8.67	341.64±9.04	8.63±0.45	12.05±0.39
Repeated updates	**19.77**±6.02	**1.71**±0.01	**60.42**±15.73	507.94±30.14	**8.43**±0.39	13.51±0.68