EVOKE: STRUCTURED CONSTRUCTION AND SEARCH FOR LLM-BASED NATIVE KERNEL EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Platform-native kernels are central to accelerator performance, yet developing them for an emerging programming stack remains heavily dependent on expert knowledge. LLM-based systems broaden source-level search, but typically treat an entire kernel project as the unit of generation and optimization. On a stack sparsely represented in pretraining data, this forces the model to coordinate platform knowledge with tiling, memory movement, pipelining, computation, aggregation, and host–device integration in a single context. It also entangles optimization decisions: when an assembled candidate regresses, several potentially useful changes may be discarded together, even when a structural redesign would become effective only after further refinement. We present EVOKE, a framework for structured construction and structured search of Ascend C kernels. It builds a task-conditioned knowledge workspace and progressively generates dependency-ordered semantic genes, each pairing a component design with executable code. During search, every action declares its intent, target genes, and required adaptations, so redesign, component refinement, recombination, and parameter tuning can be evaluated and retained separately. A non-stationary contextual bandit selects the next action from the current search state and experience accumulated across tasks, and assigns delayed credit to actions whose benefits emerge only after later refinement. Across 13 representative Ascend C operators on Ascend 910B, EVOKE produces correct implementations for all 13 tasks versus 9 for the whole-project baseline and achieves a $1.34\times$ geometric-mean performance ratio on the nine tasks solved by both.

1 INTRODUCTION

High-performance kernels determine how effectively applications use an AI accelerator's compute and memory system. Their implementation remains hardware specific, requiring coordinated choices over tiling, data movement, pipelines, computation, aggregation, and host-side launch logic. Search-based compilers explore predefined transformations (Chen et al., 2018; Zheng et al., 2020), whereas large language models (LLMs) can propose broader source-level changes and have shown substantial progress on GPU kernels (Ouyang et al., 2025; Li et al., 2025; Guo et al., 2025).

Native kernel synthesis is harder because the target is a coupled host–device project rather than an isolated function. Tile geometry propagates into metadata, loop bounds, and storage, while queue order constrains both movement and computation. AscendC exemplifies this regime: examples and optimization idioms are sparse, and recent systems rely on DSL structure or explicit platform knowledge to improve generation feasibility (Cao et al., 2026; Wen et al., 2026; Qiu et al., 2026b). This raises a broader question: can the structure that makes initial generation manageable also preserve useful search directions during optimization?

Whole-project evolution leaves both the optimization intent and its required scope implicit. Local ideas must be realized against a long context, cross-component changes must recover several contracts at once, and multiple decisions may be discarded together when their assembled candidate regresses. The current context, immediate feedback, and incumbent code can therefore bias which directions are attempted and retained. Explicit intents, target components, and adaptation scopes instead turn these directions into separately executable and persistent trajectories.

We present EVOKE, which preserves component structure across construction and optimization. It retrieves task-relevant platform knowledge and generates dependency-ordered semantic genes that

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.

pair designs with executable implementations. Search operators then specify a redesign, refinement, recombination, parameter, or coordinated intent together with its target genes and required adapters. A non-stationary contextual bandit selects among these actions from the current state and cross-task experience, including delayed credit for actions that enable later gains.

On 13 Ascend C operators under matched generation and hardware-evaluation budgets, EVOKE solves all 13 tasks versus 9 for the whole-project baseline and wins 8 of the 9 common successes, with a $1.34\times$ geometric-mean performance ratio. Controlled ablations isolate progressive construction, gene-level search, and contextual scheduling.

Our contributions are:

- We show that a shared component structure can improve both initial generation and subsequent optimization on a native programming stack with limited training coverage.
- We introduce EVOKE, combining progressive semantic-gene construction, intent- and scope-controlled search, and context-aware operator scheduling with delayed credit.
- Comparisons with one-shot generation, whole-project optimization, and random scheduling isolate these components and demonstrate their combined effect on feasibility and performance.

2 RELATED WORK

Learning-based kernel generation. TVM and Ansor search schedules within predefined transformation spaces (Chen et al., 2018; Zheng et al., 2020), whereas LLM systems synthesize source-level changes. KernelBench and TritonBench provide executable evaluation, and EvoEngineer introduced population-based kernel evolution (Ouyang et al., 2025; Li et al., 2025; Guo et al., 2025). For native NPU stacks, AscendKernelGen studies direct AscendC generation, AscendCraft uses DSL-guided transcompilation, and AgenticCANN combines platform knowledge with stage-adaptive evolution (Cao et al., 2026; Wen et al., 2026; Qiu et al., 2026b). EVOKE instead asks whether the representation used for construction can remain editable during optimization.

LLM-guided program evolution and repair. FunSearch, Evolution of Heuristics, and AlphaEvolve use LLMs to mutate executable programs (Romera-Paredes et al., 2024; Liu et al., 2024; Novikov et al., 2025); E2OC co-evolves descriptions and code for interdependent heuristic operators (Qiu et al., 2026a). In a coupled native project, however, the optimization intent and jointly modified components remain implicit. Building on evidence that repair can preserve useful kernel structure better than regeneration (Sun et al., 2026), EVOKE makes semantic components persistent loci for multiple explicit search intents.

Adaptive operator selection. Bandit controllers have long allocated evolutionary operators from observed rewards (Fialho et al., 2010), while contextual Thompson sampling conditions this choice on state (Agrawal & Goyal, 2013). EVOKE adapts this formulation to changing operator utility, action preconditions, and delayed gains in kernel evolution.

3 PROBLEM FORMULATION

Let s specify an operator's semantics and input domain, h the target accelerator stack, and p a complete host–device implementation. Evaluation over tests $\mathcal{T}$ yields validity $V(p; s, h, \mathcal{T}) \in \{0, 1\}$ and a performance score $q(p)$ to maximize under generation and hardware-evaluation budget B.

Kernel components share contracts over tile shapes, buffers, queues, data types, addresses, and launch arguments. At step t, an action transforms a retained parent p_t into p'_t. We characterize the action by its optimization *intent*, its primary components, and the *constraint scope* of adapters that must change with them. The modified components must also remain persistent loci that later actions can inherit. Structured search makes these properties explicit so that heterogeneous directions can be executed, evaluated, and retained separately over the same valid program state.

This action space introduces a state-dependent selection problem. A controller observes x_t, chooses a feasible $a_t \in \mathcal{A}_t$, and receives reward r_t after evaluation, with objective $\max_\pi \mathbb{E}[\sum_{t=1}^{B} r_t]$. Because

action utility changes as the implementation matures and earlier actions may enable delayed gains, selection must use both the current state and accumulated experience.

4 METHOD

4.1 OVERVIEW

EVOKE connects *structured construction* and *structured search* through a shared semantic genome. Given an operator specification s, hardware stack h, and tests $\mathcal{T}$, it retrieves task-specific platform evidence and generates an initial genome G_0 in dependency order. During search, a scheduler selects action a_t; an Analysis agent turns the action and hardware feedback into an atomic plan π_t; and a Generation agent modifies only the declared genes and adapters. The resulting genome is assembled into a complete project and evaluated for correctness and performance. This evidence determines candidate retention, bounded repair, and the next scheduling decision.

The shared representation is central to the design. Progressive construction narrows the knowledge and cross-component decisions that must be resolved at each generation stage. The same components then become persistent, executable search loci: different optimization intents can target them separately, and useful changes can be retained without regenerating the project. Figure 1 summarizes the resulting loop.

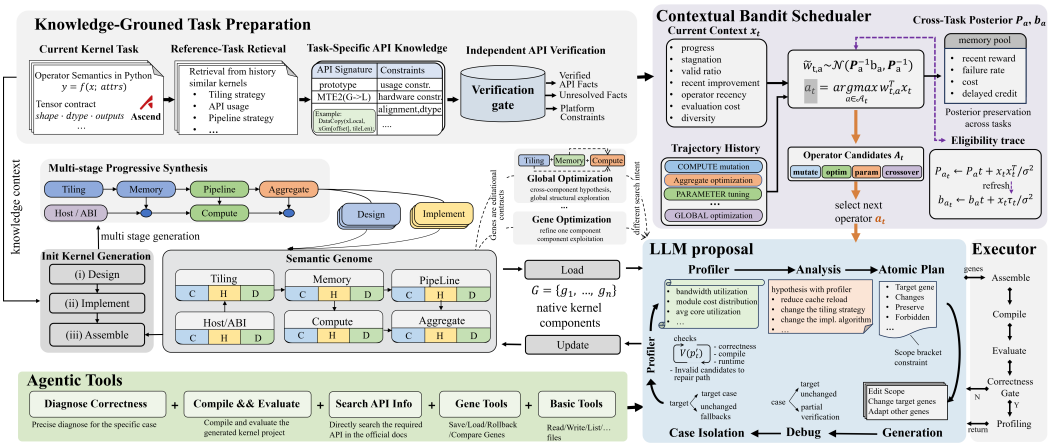


Figure 1: Overview of EVOKE. Task evidence supports dependency-ordered semantic-gene construction; intent-specific actions modify declared genes under an atomic plan; and contextual scheduling uses external evaluation feedback to select the next attempt.

4.2 PROGRESSIVE SEMANTIC-GENE CONSTRUCTION

Task-conditioned evidence. Ascend C APIs, memory hierarchies, and explicit pipeline constraints are sparsely represented in general pretraining corpora. For each task, EVOKE therefore retrieves relevant mapping strategies from official documentation and related kernels, follows them to the required API specifications, and verifies critical facts against their source locations. The resulting workspace K_s is cached across initialization branches and disclosed component by component. Each stage receives the shared platform evidence, its local objective, and only the upstream designs needed to resolve that objective; Appendix A details the retrieval fields and disclosure protocol.

Semantic genome. Source files are poor optimization boundaries because one design responsibility may span several files. We instead represent a kernel as $G = (g_1, \ldots, g_m)$, with each gene $g_i = (d_i, c_i, h_i)$ coupling a component design d_i, executable implementation c_i, and optimization history h_i. Principal genes encode memory organization, data movement, computation, and optional aggregation; structural genes encode tiling, kernel entry, shape inference, allocation, headers, and the physical ABI. Their contracts record tile extents, buffer capacity, queue order, data types, addresses, and launch

arguments. Thus a gene exposes both the intent to be changed and the executable state to be inherited and evaluated.

Dependency-ordered synthesis. Initialization proceeds in two passes. A design pass first specifies the tiling and host–device interface, followed by pipeline, memory, computation, and aggregation decisions in dependency order. An implementation pass lowers those decisions into the kernel class, data-movement and compute routines, entry points, shape inference, and glue code. Each stage repairs missing fields locally while preserving the same evidence checkpoint. A template assembler then checks required components and ABI relations before compilation. The output is both an executable candidate and the structured state on which subsequent search operates.

4.3 Structured Multi-Operator Search

Semantic genes provide persistent modification sites. To direct how they are explored, every action specifies an optimization intent, target genes, and the adaptation scope needed to preserve their contracts. This structure reduces search bias induced by the current context, immediate performance feedback, and incumbent code shape, thereby expanding the set of effective trajectories that can be explored and retained. Table 1 gives the action space used in our experiments.

Intent and action	Target and controlled change
Component redesign (`mutation_module`)	Replace one gene's design and implementation, plus required adapters
Component refinement (`module_optim`)	Improve a profiled bottleneck while preserving its role and contracts
Recombination (`crossover`)	Import one compatible donor strategy into the stronger parent
Parameter tuning (`param_tune`)	Adjust authorized constants without structural rewriting
Coordinated optimization (`1round_optima`)	Apply a declared cross-component change and its adaptations

Table 1: Search actions instantiated in EVOKE.

Each invocation follows an Analysis–Plan–Generation protocol. The orchestrator freezes the retained parent and provides the Analysis agent with profiling evidence, gene histories, and previous attempts. The resulting atomic plan names one optimization hypothesis, its target genes, required adapters, preserved behavior, forbidden changes, and authorized parameters. The Generation agent then realizes only that plan; the orchestrator rejects out-of-scope edits and confines repair to the same scope. Crossover additionally separates the frozen baseline parent from the declared donor contribution. This correspondence between one action, one plan, and one candidate makes hardware feedback attributable while allowing successful changes to persist at their semantic loci. Appendix A gives the complete realization and validation protocol.

4.4 Contextual Operator Scheduling

The action space broadens exploration, but random selection can repeatedly invoke an action that is poorly matched to the current genome. Moreover, action utility changes as components mature and the search stagnates or improves. We formulate operator choice as a non-stationary contextual bandit. Let $\mathcal{A}$ be the action space and $\mathcal{A}_t \subseteq \mathcal{A}$ the actions whose parent requirements are satisfied at decision t. The context x_t combines search progress, recent improvement, stagnation, valid-candidate ratio, genome diversity, and each action's recent reward and recency. For every feasible action, Thompson sampling (Agrawal & Goyal, 2013) draws

$$\widetilde{w}_{t,a} \sim \mathcal{N}(P_a^{-1} b_a, P_a^{-1}), \qquad a_t = \arg\max_{a \in \mathcal{A}_t} \widetilde{w}_{t,a}^{\top} x_t, \tag{1}$$

where P_a and b_a are the precision-form posterior statistics of action a. Discounting these statistics before each update lets preferences change with the search state, while posterior uncertainty preserves exploration of less frequently selected actions.

The immediate reward separates action quality from parent maturity:

$$r_t = r_t^{\mathrm{par}} + \gamma r_t^{\mathrm{best}} + \delta \mathbb{1}[d_t^{+} > d_t^{-}]. \tag{2}$$

Here, r_t^{par} is clipped improvement over the retained parent (or $-\kappa$ for an invalid offspring), r_t^{best} is improvement in the running best, and the final term rewards increased valid-population diversity. Crossover uses its stronger parent as the reference.

Structural changes may become useful only after later refinement. An eligibility trace therefore assigns a bounded fraction of each positive best refresh to earlier non-current actions:

$$e_{t,a} = \gamma_e \lambda_e e_{t-1,a} + \mathbb{1}[a = a_t],$$
$$r^{\text{delay}}_{t,a} = \beta \min(1, g_t) \frac{e_{t,a}}{\sum_{b \in \mathcal{A} \setminus \{a_t\}} e_{t,b}}, \quad a \neq a_t, \tag{3}$$

where $g_t > 0$ is the relative gain when the running best improves; the bonus is zero when the denominator vanishes. Delayed rewards update an action using the context saved at its latest invocation. The learned posteriors may initialize later tasks with the same action space, after which new contexts and continued discounting adapt the transferred experience. Full context features, posterior updates, reward definitions, and parameter settings appear in Appendix A.

5 EXPERIMENTS

Our evaluation isolates the three design choices in EVOKE: progressive construction, gene-level search, and contextual scheduling. We then compare the complete system with CANNBot under matched model-call and hardware-evaluation budgets.

5.1 EXPERIMENTAL SETUP

We evaluate 13 Ascend C operators spanning six computational patterns and L1–L3 difficulty on an Ascend 910B NPU. A disjoint eight-operator L4 set stress-tests initialization on kernels that coordinate multiple algorithmic stages and memory hierarchies. Appendix B lists all tasks and the evaluation protocol.

For an operator with M correctness cases, we define its score as the geometric-mean speedup relative to the official implementation,

$$S = \exp\left(\frac{1}{M} \sum_{i=1}^{M} \log \frac{t_i^{\text{ref}}}{t_i^{\text{gen}}}\right). \tag{4}$$

We report Solve@Budget, Fast@Budget, and paired score ratios on tasks solved by both methods. Failed tasks are not assigned zero performance.

CANNBot (Huawei CANN Team, 2026) is the whole-project baseline. Controlled variants compare end-to-end and progressive initialization (E2E-Init/MG-Init), global and gene-level search (Global-Single/Modular-Single), and random and contextual scheduling (MS-random/MS-bandit); E2E+1R and E2E+MO provide additional whole-project optimization baselines. Non-ablated components and evaluation protocols are held fixed (Appendix B).

5.2 MAIN RESULT: STRUCTURED GENERATION IMPROVES THE FEASIBILITY–QUALITY FRONTIER

Table 2 compares the complete systems and stratifies the aggregate by difficulty, following the capability-by-stratum presentation used in prior kernel-agent evaluations. EVOKE passes correctness on all 13 operators, whereas CANNBot does so on 9. Solve@Budget therefore rises from 69.2% to 100%, and Fast@Budget rises from 30.8% to 53.8%. On the nine operators solved by both systems, EVOKE wins eight; the geometric mean and median of the paired performance ratio are 1.34× and 1.42×, respectively. The L3 row exposes the only common-success loss rather than allowing it to disappear inside the overall result.

The per-method Fast@Budget counts are not disjoint. Four operators are fast under both systems, three are fast only under EVOKE, none is fast only under CANNBot, and six are fast under neither; hence the union contains seven operators rather than the sum $4 + 7$. Of the four operators not solved by CANNBot, EVOKE makes all four correct, but only *EngramGateFusion* is also faster than the reference.

Level	n	Solve C/E	Fast C/E	Paired wins E/C	E/C geo. mean
L1	4	3/4	2/3	3/0	1.260×
L2	4	3/4	0/1	3/0	2.346×
L3	5	3/5	2/3	2/1	0.818×
All	13	9/13	4/7	8/1	**1.342**×

Table 2: Complete-system results by difficulty. C/E denotes CANNBot/EVOKE; E/C denotes EVOKE/CANNBot. Paired columns include only operators solved by both methods. Per-method Fast@Budget counts overlap and must not be added.

The gain has two distinct components: a wider feasibility boundary and higher quality on the common-success subset. The former accounts for the increase in Solve@Budget, while the latter is reflected in eight wins out of nine paired tasks and a 1.34× geometric-mean ratio. Exact task-level scores, including the single common-success exception, are reported in Appendix B.

5.3 Why Structure Helps: Better Starts and Smaller Search Scopes

Progressive initialization. With the same specification, knowledge budget, and correctness checks, MG-Init reduces the average number of debugging rounds from 3.31 to 2.56, a 22.7% reduction, while both variants reach a final case pass rate of 100%. At the first fully correct implementation, the geometric-mean initial score is 0.557 for MG-Init and 0.179 for E2E-Init, a 3.12× improvement. Progressive disclosure thus reaches the same final correctness with less repair and a stronger starting point, because interface, layout, tiling, pipeline, and compute constraints are established in smaller decision scopes.

Modular search. To isolate the representation effect, we fix the initial program, optimization action set, and candidate budget, and disable crossover. Global-Single edits the complete program, whereas Modular-Single edits one semantic gene and its required dependencies. The modular variant raises the positive-improvement rate from 49.0% to 62.1%, the mean positive absolute gain from 0.090 to 0.158, and the median positive relative gain from 4.71% to 10.53%. Modular-Single has a higher positive-improvement rate on 9 of 12 operators. Since both variants use the same actions and budget, the result is consistent with matching the scope of an optimization action to the semantic structure, rather than simply sampling more candidates.

(a) Progressive initialization

Method	Debug rounds ↓	Case pass rate	Initial score ↑
E2E-Init	3.31	100%	0.179
MG-Init	**2.56**	100%	**0.557**

(b) Matched single-individual optimization

Scope	Positive rate ↑	Mean positive gain ↑	Median relative gain ↑
Global-Single	49.0%	0.090	4.71%
Modular-Single	**62.1%**	**0.158**	**10.53%**

Table 3: Controlled mechanism ablations. Panel (a) compares aggregate initialization outcomes; panel (b) fixes the initial candidate, action set, and trial budget while disabling crossover. Per-operator outcomes are deferred to the appendix.

L4 stress set: quality can require more repair. The disjoint L4 set exposes a different regime from the L1–L3 initialization study. Across all eight tasks, E2E produces 6/8 valid initializations and multi-stage generation produces 7/8. Six tasks have correctness-valid scores from both methods. On this strict paired subset, multi-stage generation produces the faster kernel on 4/6 tasks, with a 1.116× median and 1.769× geometric-mean ratio. The geometric mean is driven by a 59.16× gain on *LSTM*; excluding that task gives 0.877×, so the aggregate does not imply uniform dominance. The more stable observation is a shifted quality–repair frontier: multi-stage generation uses more repair rounds on 5/6 pairs (3.50 versus 2.17 rounds on average) while preserving a more ambitious implementation

on several tasks. Appendix B.4 reports the complete eight-task matrix and the *LSTM* implementation audit.

Aggregate evidence	E2E	Multi-stage	Paired effect
Valid initializations (all 8)	6/8	**7/8**	+1 valid task
Performance wins (6 strict pairs)	2/6	**4/6**	1.769× geo.; 1.116× med.
Mean debug rounds (6 strict pairs)	**2.17**	3.50	+1.33 rounds
Runs requiring more repair	1/6	5/6	favors E2E

Table 4: Aggregate L4 initialization evidence. Coverage uses all eight tasks; ratios and repair statistics use only the six correctness-valid pairs. The full operator matrix is reported in Appendix B.4.

5.4 Adaptive Scheduling and End-to-End Attribution

Bandit versus random scheduling. We fix the multi-gene representation, progressive initial program, action set, and optimization budget, and replace only random action selection with the contextual bandit. After excluding the only archive pair with unequal sample counts, the comparison contains 13 matched-budget pairs across 12 unique operators; one operator contributes two independent pairs from different batches. Bandit wins 9/13 comparisons, with a $1.435\times$ geometric-mean final-score ratio and a median improvement of 7.47%. The pair-level win rates are 3/5, 3/4, and 3/4 for L1, L2, and L3, respectively. Gains vary across tasks, with the largest improvements occurring when the search state provides useful information for action selection. Appendix B reports every matched pair.

End-to-end attribution. Table 5 separates scheduler behavior by difficulty and then summarizes how the complete construction path accumulates. Relative to E2E+1R, EVOKE wins on 11/13 operators, with a 1.59× geometric-mean ratio and a 1.36× median ratio. Against E2E+MO, which already has multiple whole-program actions, EVOKE wins on 11 of 12 strictly paired operators, with geometric-mean and median ratios of 2.75× and 1.59×. Adding more whole-program actions is therefore not sufficient to replace progressive initialization and gene-level scope. Appendix B reports every pair and the per-family construction-path matrix.

(a) Equal-budget contextual scheduling

Level	Pairs	Bandit wins	Geo. mean ratio	Median ratio
L1	5	3/5	1.017×	1.075×
L2	4	3/4	1.601×	1.298×
L3	4	3/4	1.977×	1.090×
All	13	9/13	**1.435**×	**1.075**×

(b) End-to-end construction paths

Comparison	Pairs	EVOKE wins	Geo. mean ratio	Median ratio
EVOKE vs. E2E+1R	13	11/13	1.59×	1.36×
EVOKE vs. E2E+MO	12	11/12	2.75×	1.59×

Table 5: Aggregate optimization ablations. Panel (a) stratifies the 13 equal-budget Bandit–Random pairs; panel (b) compares complete construction paths on strict pairs. Task-level values remain in the appendix.

Taken together, progressive synthesis, modular scope, and adaptive scheduling make a fixed search budget more productive across the tested Ascend C workloads; the appendix records the task-level exceptions and cost fields.

6 Discussion

EVOKE targets native accelerator projects whose optimization responsibilities can be named and whose assembled candidates can be tested on hardware. The concrete genes follow the host/device

organization of the evaluated stack; a different native stack requires a corresponding schema and assembler. Gene boundaries need not make components independent: contracts and adapter scopes carry dependencies across them. Planned-versus-realized gene changes provide a direct diagnostic when a boundary repeatedly fails to isolate a useful optimization responsibility.

The contextual controller learns which operator should be selected next rather than fitting a full long-horizon policy from a short online run. Posterior discounting tracks changes in immediate operator utility, while eligibility traces capture short enabling effects and persisted posteriors accumulate evidence across tasks. Logged contexts, reward components, and behavior propensities make its decisions auditable and support future offline evaluation of richer trajectory policies.

7 Conclusion

EVOKE organizes native-kernel evolution around a shared structured program state. Dependency-ordered semantic genes reduce the simultaneous decisions required to construct a complete Ascend C project and provide persistent loci for inheritance and reconstruction. Explicit search intents and atomic adaptation scopes then expose distinct algorithmic, component-level, recombinative, parametric, and cross-component trajectories, while a non-stationary contextual scheduler uses the current state and accumulated experience to select among them and credit actions whose benefits emerge later. Across 13 representative operators, EVOKE solves 13 tasks versus 9 for the whole-project baseline and attains a $1.34\times$ geometric-mean performance ratio on their common-success set. Together, the results support structured construction, structured search, and adaptive operator selection as a coherent approach to evolving coupled native programs.

Reproducibility Statement

Sections 3 and 4 provide the algorithmic definition of EVOKE. Section 5 specifies the evaluation tasks, baselines, budgets, correctness criteria, and performance protocol, while Appendices A and B document the implementation and additional experimental details. The supplementary material contains the configurations and scripts used to run the experiments, aggregate measurements, and produce the reported statistics. We also report the hardware and software environments, evaluation budgets, and scheduler parameters required to reproduce the results.

References

Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, pp. 127–135, 2013.

Xiang Cao, Jidong Zhai, Pan Li, Zhen Hu, Chao Yan, Baolei Mu, Guanqun Fang, Biao She, Jiahao Li, Yu Su, Dacheng Tao, Xun Huang, Fan Xu, Fan Yang, Yichao Lu, Chun-Dong Wang, Yutong Lu, Wei Xue, Bowen Zhou, and Yonghong Tian. Ascendkernelgen: A systematic study of LLM-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*, 2026.

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

Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60: 25–64, 2010.

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

Huawei CANN Team. CANNBot: Ascend C operator generation and precision-debug toolchain. Official repository, 2026. URL `https://gitcode.com/cann/cannbot-skills.git`. Accessed 2026-09-26.

Jian Li, Shizhe Li, Zhen Gao, Qinyang Shi, Yihong Li, Zilong Wang, Jian Huang, Haojie Wang, Jing 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.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 32201–32223, 2024.

Alexander Novikov, Nguyễn Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Anne Ouyang, Shunyu 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.

Junhao Qiu, Xin Chen, Liang Ge, Liyong Lin, Zhichao Lu, and Qingfu Zhang. Evolving interdependent operators with large language models for multi-objective combinatorial optimization. *arXiv preprint arXiv:2601.17899*, 2026a.

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

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Yansong Sun, Shenxiu Wu, Siyuan Chen, Runlin Hou, Junhao Qiu, Junming Cao, Shudi Shao, Zhichao Lu, and Qingfu Zhang. Don't regenerate, debug: A domain-specific agent for repairing near-miss hardware operators. *arXiv preprint arXiv:2608.02712*, 2026.

Zhen Wen, Shudi Shao, Ziheng Li, Yiheng Ge, Tianyi Xu, Yujun Lin, and Tianqi Zhang. Ascendcraft: Automatic ascend NPU kernel generation via DSL-guided transcompilation. *arXiv preprint arXiv:2601.22760*, 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 *14th USENIX Symposium on Operating Systems Design and Implementation*, pp. 863–879, 2020.

A Implementation and Reproducibility Details

Task knowledge and progressive construction. For each operator, the retrieval stage searches an index of official documentation and kernels with related computational patterns. It first collects candidate hardware-mapping strategies, including block partitioning, GM–UB movement, tail handling, reduction, and host–kernel tiling protocols. For every API implicated by those strategies, the workspace records its prototype, parameter constraints, required headers, applicability conditions, and source location. Critical facts are checked against the recorded documentation before the evidence is compressed into K_s. The workspace is cached or checkpointed so that initialization branches restore the same platform baseline rather than rebuilding it.

The design pass specifies the tiling interface and host–device metadata before pipeline, memory, computation, and optional aggregation strategies. The implementation pass follows the same dependency order, producing the kernel class, `CopyIn`/`Compute`/`CopyOut` routines, aggregation code, entry points, shape inference, output allocation, and binding code. Each stage receives K_s, the necessary upstream designs, and its target-gene schema. The assembler checks required fields, dynamic-output

conventions, and host–kernel signatures before compilation and execution resolve the remaining constraints.

Genome fields. The implementation stores each gene as a dictionary with a stable name, design text, executable implementation, and an optimization-history list. The current schema includes MEM, PIPELINE, COMPUTE, and optional AGGREGATE genes; tiling fields, tiling function body, includes, kernel entry, shape inference, output allocation, physical ABI, and binding code are represented as structural genes when applicable.

Atomic-plan realization. For every invocation, the orchestrator freezes the retained parent and profiles it once. The Analysis agent receives the selected action, profiler evidence, component histories, and previously attempted directions, and returns a schema-validated plan containing the optimization hypothesis, target genes, required adapters, intended and forbidden changes, preserved behavior, and authorized parameters. For crossover, the plan separately identifies the stronger baseline parent and the donor strategy to import. The Generation agent may edit only the declared genes and adapters. The realized diff is checked against that scope; bounded repair and micro-tuning remain within the same plan. Out-of-scope candidates are rejected and the frozen parent artifacts are restored.

Action instances and preconditions. The five actions in Table 1 are separate scheduler arms. Crossover requires two valid parents; every other action requires one. These requirements define the feasible set $\mathcal{A}_t$ at each decision.

Action	Valid parents
`mutation_module`	1
`crossover`	2
`module_optim`	1
`param_tune`	1
`1round_optima`	1

Table 6: Parent requirements for the action instances used in the experiments.

Context representation. For $K = |\mathcal{A}|$ actions, the scheduler uses

$$x_t = [1, \rho_t^{\text{gen}}, \rho_t^{\text{sample}}, \Delta_t^{(3)}, \zeta_t, v_t, d_t, \{\bar{r}_{t,a}^{(5)}\}_{a\in\mathcal{A}}, \{u_{t,a}\}_{a\in\mathcal{A}}] \in \mathbb{R}^{7+2K}. \tag{5}$$

Here, ρ_t^{gen} and ρ_t^{sample} are generation and sampling progress, $\Delta_t^{(3)}$ is the relative best-score change over three recorded steps, ζ_t is normalized stagnation, v_t is the valid-candidate ratio, and d_t is genome diversity. The action-specific terms are the mean parent-relative reward over the five most recent invocations and normalized recency $u_{t,a}$. Progress, validity, diversity, stagnation, and recency are clipped to $[0, 1]$; best-score change and recent reward are clipped to $[-1, 1]$. Stagnation and the number of decisions since an action's last invocation are divided by 20 before clipping. With five actions, x_t has 17 dimensions.

Posterior update. After each selection, the scheduler discounts every action model as $P_a \leftarrow \eta P_a$ and $b_a \leftarrow \eta b_a$. When feedback for the selected action becomes available, its sufficient statistics are updated by

$$P_{a_t} \leftarrow P_{a_t} + \frac{x_t x_t^\top}{\sigma^2}, \qquad b_{a_t} \leftarrow b_{a_t} + \frac{x_t r_t}{\sigma^2}. \tag{6}$$

The implementation retains the context of each pending decision until its evaluation returns, allowing generation and hardware evaluation to proceed asynchronously while preserving the decision–feedback correspondence.

Reward components. The immediate terms in Equation 2 are

$$r_t^{\mathrm{par}} = \begin{cases} -\kappa, & V(p'_t) = 0, \\ \mathrm{clip}\left(\dfrac{q(p'_t) - q(p_t^{\mathrm{par}})}{|q(p_t^{\mathrm{par}})| + \epsilon}, -1, 1\right), & V(p'_t) = 1, \end{cases}$$
$$r_t^{\mathrm{best}} = \mathrm{clip}\left(\frac{q_t^\star - q_{t-1}^\star}{|q_{t-1}^\star| + \epsilon}, 0, 1\right). \tag{7}$$

For crossover, p_t^{par} is the stronger parent. The best-refresh term is zero until a running best exists. When Equation 3 assigns a delayed reward to action a, the scheduler updates that action with the context $x_{t_a^-}$ saved at its most recent invocation t_a^-:

$$P_a \leftarrow P_a + \frac{x_{t_a^-} x_{t_a^-}^\top}{\sigma^2}, \qquad b_a \leftarrow b_a + \frac{x_{t_a^-} r_{t,a}^{\mathrm{delay}}}{\sigma^2}. \tag{8}$$

Scheduler constants. The implementation uses observation variance $\sigma^2 = 0.25$ and posterior discount $\eta = 0.99$. Eligibility uses $\gamma_e = 0.9$, $\lambda_e = 0.8$, and delayed-credit scale $\beta = 0.5$. The immediate reward uses best-refresh coefficient $\gamma = 0.9$, diversity bonus $\delta = 0.1$, and invalid-offspring penalty $\kappa = 0.5$. The resource-penalty coefficient is set to $\lambda_c = 0$ in the reported configuration. Parent-relative and best-refresh improvements are clipped to unit magnitude.

Persistence and logging. Without transferred experience, each arm starts from $P_a = I$ and $b_a = 0$. Each run saves precision matrices, response vectors, and update counts as a JSON posterior that can initialize later runs. A configurable prior scale controls transferred confidence. The behavior propensity of the selected action is estimated with 64 additional posterior draws; every trajectory entry records this propensity, the context, proposed and executed action, parent identifiers and scores, reward components, tokens, and wall time.

End-to-end procedure. The procedure first builds, assembles, and evaluates initial genomes with dependency-ordered synthesis. At each subsequent step, it constructs x_t and the valid-action mask $\mathcal{A}_t$, samples an operator, selects and profiles its preservation parent, and freezes one atomic plan. It then implements the plan over the target genes and required adapters; assembles, compiles, executes, and validates the offspring; and updates the population, reward model, eligibility traces, and trajectory log. Search returns the highest-scoring valid program when its generation or evaluation budget is exhausted. Every attempt produces a trajectory record, including invalid or interrupted attempts. An offspring that modifies frozen plan artifacts is rejected and the parent artifacts are restored. If generation stops after changing the genome, the concrete partial candidate is evaluated rather than silently replaced by its parent, preserving the correspondence between the executed action and its observed outcome.

B Additional Experimental Results

This appendix provides the per-operator values, controlled ablations, a representative optimization trajectory, and the cost fields underlying the main-text claims. Unless stated otherwise, every score is the geometric-mean speedup relative to the official implementation. We use *invalid* for an archived run that did not pass the complete correctness suite, *NR* when no result was archived, and *n/a* when a score or comparison is not applicable; none is treated as zero.

B.1 Experimental Protocol and Variants

The 13-operator primary set contains *Sigmoid*, *Exp*, *Dilation2D*, *AddRmsNormDynamicQuant*, *Gather*, *MaskedScale*, *EngramGateFusion*, *Transpose*, *ArgMax*, *CrossEntropyLoss*, *Gcd*, *Foreach-Norm*, and *Scatter*. It covers elementwise computation, reduction, layout transformation, indirect memory access, fused computation, and numerically stable reduction across L1–L3 difficulty. The disjoint L4 initialization set contains *GQA*, *GroupedMatmulSwigluQuant*, *GRU*, *LSTM*, *MHA*, *MLA*, *MLA-Prolog*, and *SparseFlashAttention*. Every method uses the same per-operator correctness cases

and device-side performance tests. Compilation failures, runtime errors, timeouts, and incorrect candidates consume the search budget but do not enter performance averages. Each operator contributes at most once to a win-rate statistic.

CANNBot treats the complete operator project as the generation, debugging, and optimization unit. The controlled variants are: **E2E-Init**, one-shot end-to-end initialization; **MG-Init**, dependency-ordered multi-gene initialization; **Global-Single**, one whole-program optimization step; **Modular-Single**, one gene-level optimization step; **E2E+1R**, end-to-end initialization followed by one optimization round; **E2E+MO**, end-to-end initialization with multiple whole-project optimizers; **MS-random**, multi-gene construction followed by random action scheduling; and **MS-bandit**, the complete method. Each isolated comparison holds the non-ablated components, correctness checks, and performance protocol fixed; complete-system comparisons match model-call and hardware-evaluation budgets.

B.2 Per-Operator Comparison with CANNBot

Family	Operator	L	CANNBot corr.	CANNBot	EVOKE corr.	EVOKE
Elementwise	Sigmoid	1	20/20	0.7834	20/20	**1.1410**
Elementwise	Exp	1	20/20	4.2117	20/20	**4.5179**
Contraction	Dilation2D	3	20/20	4.5703	20/20	**8.7306**
usedComposite	AddRmsNormDynamicQuant	3	20/20	7.0633	20/20	**9.9949**
IndexGather	Gather	2	20/20	0.1523	20/20	**1.1323**
MaskPredicate	MaskedScale	1	20/20	1.9087	20/20	**2.4431**
VVFusion	EngramGateFusion	3	0/20	n/a	20/20	**3.1198**
LayoutTransform	Transpose	3	20/20	**0.2867**	20/20	0.0581
SortSelect	ArgMax	2	20/20	0.7673	20/20	**0.8603**
NumericalStable	CrossEntropyLoss	2	17/20	n/a	20/20	0.0509
Broadcast	Gcd	2	20/20	0.1941	20/20	**0.3006**
Reduction	ForeachNorm	1	0/20	n/a	20/20	**0.1302**
ScatterUpdate	Scatter	2	invalid	n/a	20/20	**0.1981**

Table 7: Per-operator comparison. CANNBot's Transpose run reached the wall-clock limit after producing a valid 20/20 candidate; it is counted as solved while the timeout remains part of the cost accounting.

B.3 Initialization and Modular Search

Initialization	Debug rounds	Final case pass rate	Initial score
E2E-Init	3.31	100%	0.179
MG-Init	**2.56**	100%	**0.557**

Table 8: Matched initialization comparison. The score is measured at the first fully correct implementation.

Search scope	Positive rate	Mean positive gain	Median relative gain
Global-Single	49.0%	0.090	4.71%
Modular-Single	**62.1%**	**0.158**	**10.53%**

Table 9: Single-individual modular-search ablation with matched budgets and crossover disabled.

B.4 Level-4 Initialization Stress Set

Table 10 keeps the L4 stress set separate from the 13-operator primary evaluation. E2E yields valid initializations on 6/8 tasks and multi-stage generation on 7/8. Six operators have correctness-valid scores from both methods. For *GroupedMatmulSwigluQuant*, the original E2E run ends at 0/20 after ten repair rounds, whereas the archived multi-stage repair reaches 20/20 with score 0.02501 after one replay repair round. The available *GRU* run is multi-stage but remains invalid, and no E2E result is archived. Neither task enters the strict paired performance aggregate. For preloaded initializations,

the reported count is the number of replay repair rounds; original generation rounds are retained separately to prevent double counting.

Operator	E2E corr.	E2E score	E2E dbg.	MS corr.	MS score	MS dbg.
GQA	20/20	0.01004	3	20/20	0.00579	4
GroupedMatmulSwigluQuant	0/20	n/a	10	20/20	**0.02501**	1
GRU	NR	n/a	NR	invalid	n/a	6
LSTM	20/20	0.00549	2	20/20	**0.32450**	3
MHA	20/20	0.00519	2	20/20	**0.00614**	4
MLA	20/20	0.00316	2	20/20	**0.00635**	3
MLA-Prolog	20/20	0.04627	3	20/20	**0.04846**	2
SparseFlashAttention	20/20	**0.15603**	1	20/20	0.05614	5

Table 10: All eight L4 stress-set operators. Bold marks the higher score for strict valid pairs and the sole valid score for an unmatched task. Invalid and NR entries never enter performance aggregates.

Across the six strict pairs, multi-stage initialization wins 4/6 and has a 1.116× median score ratio. Its 1.769× geometric mean is dominated by *LSTM*: the geometric mean over the other five pairs is 0.877×. Multi-stage initialization nevertheless spends more repair rounds on five of the six pairs, with a mean of 3.50 rounds versus 2.17 for E2E. These results refine the easier-set finding rather than simply extending it: on L4 kernels, decomposing the implementation can preserve an ambitious hardware-aware plan, but coordinating that plan introduces more failure surfaces during initial repair.

The *LSTM* artifacts provide a directly inspectable example. The valid E2E implementation contains nested scalar loops with global-memory `GetValue`/`SetValue` accesses for gate and recurrent products and does not instantiate the matrix-multiplication primitive. The multi-gene implementation instead materializes aligned tiles and uses matrix-multiplication plus explicit data movement. This difference explains why fewer E2E repair rounds do not by themselves imply a better initialization: a simpler implementation can cross the correctness boundary sooner while leaving substantially more performance on the table. We restrict this implementation-path explanation to *LSTM*; the aggregate table alone cannot establish that every E2E kernel follows the same pattern.

B.5 Bandit Stratification and End-to-End Matrix

Level	Bandit wins	Matched pairs
L1	3/5 (60.0%)	5
L2	3/4 (75.0%)	4
L3	3/4 (75.0%)	4
All	9/13 (69.2%)	13

Table 11: Bandit wins over random scheduling among equal-budget archive pairs. The 13 pairs cover 12 unique operators.

Operator	Level	Random	Bandit	Bandit/Random
MaskedScale	L1	2.1886	2.4431	1.116×
Maximum (A)	L2	0.7208	1.1041	1.532×
Sigmoid	L1	1.0617	1.1410	1.075×
Maximum (B)	L2	0.5749	0.6119	1.064×
Mish	L1	1.4299	1.8849	1.318×
AddRmsNormDynamicQuant	L3	5.2647	5.9315	1.127×
CrossEntropyLoss	L2	0.0509	0.0452	0.888×
Dilation2D	L3	7.2487	7.6321	1.053×
EngramGateFusion	L3	0.0997	3.1198	31.279×
Exp	L1	4.5179	3.8961	0.862×
ForeachNorm	L1	0.1302	0.1041	0.799×
Gather	L2	0.2493	1.1323	4.542×
Transpose	L3	0.0581	0.0239	0.411×

Table 12: Equal-budget Bandit–Random pairs retained in the scheduler comparison. Maximum (A) and (B) are independent runs from two experiment batches.

Operator family	E2E+1R	E2E+MO	MS-bandit
usedComposite	12.6543	5.4341	9.9949
Contraction	4.8100	n/a	8.7306
Elementwise	3.8553	4.0648	4.5179
LayoutTransform	0.0141	0.0081	0.0581
SortSelect	0.1354	0.0517	0.8603
Normalization	1.2744	1.6439	1.7307
Reduction	0.1078	0.1090	0.1302
Broadcast	0.9075	0.5334	1.1041
ScatterUpdate	0.1330	0.0033	0.1981
VVFusion	1.6088	0.7052	3.1198
IndexGather	0.3477	0.8893	1.1323
MaskPredicate	2.0956	1.8260	2.4431
NumericalStable	0.1208	0.0780	0.0509

Table 13: Per-family construction-path results. The equal-budget MS-random comparison is reported separately in Table 12; the rows here must not be used as an unpaired aggregate average.

B.6 Representative Optimization Trajectory

Table 14 gives a Dilation2D best-so-far trajectory. The trace is used to make the mechanism concrete, not as an additional independent replicate. Local and global actions can both enter the same evolutionary path, while unsuccessful candidates leave the running best unchanged.

Step	Action	Scope	Outcome	Best score
0	Initialization	Multi-gene	Initial candidate	6.783395
1	Compute mutation	Compute gene	Accepted	6.850952
2	Aggregate optimization	Aggregate gene	Accepted	6.881831
3	Parameter tuning	Parameter gene	Accepted	7.043839
4–7	Compute/parameter actions	Local gene	No refresh	7.043839
8	Global optimization	Whole program	Accepted	7.232278
9	Parameter tuning	Parameter gene	Accepted	7.272470

Table 14: A representative multi-component search trajectory for Dilation2D.

B.7 Cost and Scope of the Claim

Structured search exchanges additional staged interaction for a higher fraction of useful candidates. We therefore retain failed and timed-out runs when reporting model tokens, compilation attempts, device evaluations, and wall-clock time. In deployment, the stopping rule can be fixed either by hardware trials or by model tokens, depending on whether search efficiency or model-call cost is the binding resource.

The observed boundary cases matter. The CANNBot win on *Transpose* indicates that locally scoped actions can hinder a coordinated layout rewrite. Bandit losses on several operators show that short-horizon reward estimates remain sensitive to early noise. Finally, the newly solved *CrossEntropyLoss*, *ForeachNorm*, and *Scatter* implementations are correct but slower than the reference, so expanding the feasibility boundary and reaching high performance should be treated as sequential rather than equivalent objectives.

B.8 Cost and Reproducibility

The audit record retains, for every operator and method, model calls and tokens, compilation attempts, correctness evaluations, performance evaluations, wall-clock time, and terminal state. Failed and timed-out runs remain in the cost accounting. A complete release should additionally report random seeds, model and version, decoding parameters, CANN version, device model, warm-up count, and timing repetitions, together with bootstrap confidence intervals and multiplicity-adjusted paired tests.