AVO: Agentic Variation Operators for Autonomous Evolutionary Search

Terry Chen*, **Zhifan Ye***, **Bing Xu***, **Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen**
Andrew Kerr, Haicheng Wu, Vedaanta Agarwalla, Fengzhe Zhou, Harry Li, Yang Xu
Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu
Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu
Fung Xie, Michael Lightstone, Humphrey Shi
NVIDIA

Abstract

Agentic Variation Operators (AVO) are a new family of evolutionary variation operators that replace the fixed mutation, crossover, and hand-designed heuristics of classical evolutionary search with autonomous coding agents. Rather than confining a language model to **candidate generation within a prescribed pipeline**, AVO instantiates **variation as a self-directed agent loop** that can consult the current lineage, a domain-specific knowledge base, and execution feedback to propose, repair, critique, and verify implementation edits. We evaluate AVO on **attention**, among the most aggressively optimized kernel targets in AI, on NVIDIA Blackwell (B200) GPUs. Over 7 days of continuous autonomous evolution on multi-head attention, AVO discovers kernels that outperform **cuDNN by up to 3.5%** and **FlashAttention-4 by up to 10.5%** across the evaluated configurations. The discovered optimizations transfer readily to grouped-query attention, requiring only **30 minutes** of additional autonomous adaptation and yielding gains of up to **7.0% over cuDNN** and **9.3% over FlashAttention-4**. Together, these results show that agentic variation operators move beyond prior LLM-in-the-loop evolutionary pipelines by elevating the agent from candidate generator to variation operator, and can discover performance-critical micro-architectural optimizations that produce kernels surpassing state-of-the-art expert-engineered attention implementations on today's most advanced GPU hardware.

1 Introduction

Large language models have emerged as powerful components in evolutionary search, replacing hand-crafted mutation operators [1] with learned code generation [2–5]. In these systems, an LLM generates candidate solutions conditioned on selected parents, while a surrounding framework, which is usually heuristic-based, handles parent sampling, evaluation, and population management. This combination has produced notable results in mathematical optimization and algorithm discovery, including flagship systems such as FunSearch and AlphaEvolve [3, 4]. However, confining the LLM to candidate generation within a prescribed pipeline fundamentally limits what the LLM can discover: it produces a single output per invocation, with no ability to proactively consult reference materials, test its changes, interpret feedback, or revise its approach before committing a candidate. For the most aggressively hand-tuned implementations, where further improvement requires deep, iterative engineering, this constraint is especially limiting.

We study this problem in the context of attention [6], the central operation in Transformer architectures, and one of the most heavily optimized GPU kernels. The FlashAttention lineage [7–10] and NVIDIA's

*Equal Contribution

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

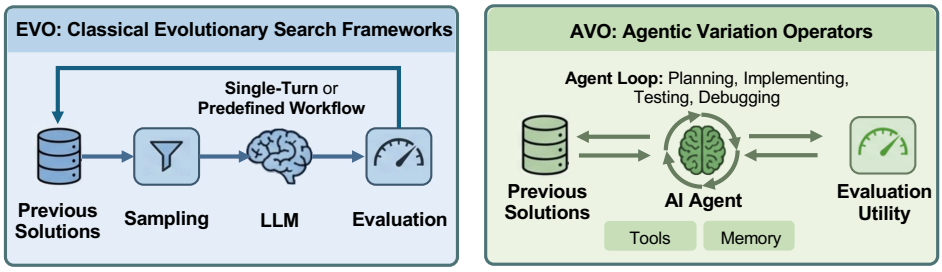

Figure 1: **EVO vs AVO**: Comparison between prior evolutionary search frameworks (e.g. FunSearch, AlphaEvolve, and related LLM-augmented evolutionary approaches) and the proposed Agentic Variation Operator. **Left:** Prior approaches follow a fixed pipeline where the LLM is confined to a single-turn generation step or a predefined workflow, with sampling and evaluation controlled by the framework. **Right:** AVO replaces this pipeline with an autonomous AI agent that iteratively plans, implements, tests, and debugs across long-running sessions, with direct access to previous solutions, evaluation utilities, tools, and persistent memory.

cuDNN library [11] have pushed attention throughput progressively closer to hardware limits across successive GPU generations, with both FlashAttention-4 (FA4) and cuDNN requiring months of manual optimization on the latest Blackwell architecture. Surpassing these implementations demands sustained, iterative interaction with the development environment: studying hardware documentation, analyzing profiler output to identify bottlenecks, implementing and testing candidate optimizations, diagnosing correctness failures, and revising strategy based on accumulated experience.

Recent progress in *deep agents* [12–16] demonstrates that LLMs augmented with planning, persistent memory, and tool use can autonomously navigate such multi-step engineering workflows, with applications ranging from resolving complex GitHub issues to generating key deep learning software [17]. This motivates a fundamentally different role for LLMs in evolutionary search: rather than confining them within a fixed pipeline, we can elevate a deep agent to serve as the variation operator itself. To this end, we propose **Agentic Variation Operators (AVO)**, in which a self-directed coding agent replaces the mutation and crossover process in previous works based on single-turn LLMs [3–5] or fixed workflows [18]. The AVO agent has access to all prior solutions, a domain-specific knowledge base, and the evaluation utility. It autonomously decides what to consult, what to edit, and when to evaluate, enabling continuous improvements over extended time horizons.

To demonstrate its effectiveness, we apply AVO to multi-head attention (MHA) kernels on the Blackwell B200 GPU, and directly compare against the expert-optimized cuDNN and FlashAttention-4 kernels. Over 7 days of continuous evolution without human intervention, the agent explored over 500 optimization directions and evolved 40 kernel versions, producing MHA kernels achieving up to 1668 TFLOPS at BF16 precision, outperforming cuDNN by up to 3.5% and FlashAttention-4 by up to 10.5%. Our analysis of agent-discovered optimizations reveals that they span multiple levels of kernel design, including register allocation, instruction pipeline scheduling, and workload distribution, reflecting genuine hardware-level reasoning. Empirically, we find that the optimization techniques discovered on MHA transfer effectively to grouped-query attention (GQA): adapting the evolved MHA kernel to support GQA requires only 30 minutes of additional autonomous agent effort, yielding up to 7.0% performance improvement over cuDNN and 9.3% over FlashAttention-4. In addition, we further test AVO on two structurally different kernels: GatedDeltaNet, a linear-attention variant, and Triangle Attention, a key operator in AlphaFold; in both cases AVO produces kernels that consistently outperform expert-tuned baselines (Appendix A), demonstrating the generalizability of AVO across diverse kernel optimization tasks. Finally, to isolate the contribution of the agentic variation operator itself, we benchmark AVO against three prior LLM-driven evolutionary search (EVO) baselines under a matched wall-clock budget on the same hardware, where AVO achieves an approximately 31% improvement over the strongest EVO baseline (Appendix B).

Our contributions are as follows:

- We introduce Agentic Variation Operators (AVO), a new family of evolutionary variation operators that elevate the agent from candidate generator to variation operator, autonomously

exploring domain knowledge, implementing edits, and validating results through iterative interaction with the environment.

- We achieve state-of-the-art MHA throughput on NVIDIA B200 GPUs across the benchmarked configurations, reaching up to 1668 TFLOPS and outperforming cuDNN by up to 3.5% and FlashAttention-4 by up to 10.5%. Furthermore, we show that the discovered optimizations readily transfer to GQA, requiring only 30 minutes of autonomous adaptation and yielding gains of up to 7.0% over cuDNN and 9.3% over FlashAttention-4.
- We provide a detailed analysis of the micro-architectural optimizations discovered by the agent under the benchmarked settings, showing the agent performs genuine hardware-level reasoning rather than superficial code transformations.
- We further demonstrate that AVO generalizes beyond attention to other widely-used GPU kernels, achieving up to $4.1\times$ speedup over Flash Linear Attention on GatedDeltaNet and consistent gains over NVIDIA's cuEquivariance on Triangle Attention.

2 Background

2.1 Evolutionary Search and Variation Operators

Evolutionary search optimizes over a space of candidates by maintaining a population $\mathcal{P}$ and iteratively expanding it with new solutions [19]. A population is a set of solution-score pairs $\mathcal{P} = \{(x_i, \mathbf{f}(x_i))\}$, where $\mathbf{f}$ is a scoring function that evaluates each candidate solution. Each iteration produces a new candidate x_{t+1} and updates the population:

$$\mathcal{P}_{t+1} = \texttt{Update}\big(\mathcal{P}_t,\ (x_{t+1},\ \mathbf{f}(x_{t+1}))\big), \quad x_{t+1} = \texttt{Vary}(\mathcal{P}_t), \tag{1}$$

where `Update` adds the new solution to the population, possibly pruning low-score members to maintain a bounded archive. We call `Vary` the *variation operator*: the mechanism by which new candidates are produced from existing ones. In works such as FunSearch [3], AlphaEvolve [4], and related LLM-augmented evolutionary methods [2, 20, 5], the variation operator decomposes into two stages:

$$\texttt{Vary}(\mathcal{P}_t) = \texttt{Generate}\big(\texttt{Sample}(\mathcal{P}_t)\big), \tag{2}$$

where `Sample` selects one or more parent solutions from $\mathcal{P}_t$ (typically guided by score-based and diversity-based heuristics), and `Generate` produces a new candidate conditioned on the sampled parents.

LLM-augmented variation. In these approaches, `Generate` is implemented by an LLM that is prompted with the sampled parents and asked to produce a more optimized solution. The `Sample` step, however, remains a fixed algorithmic procedure: AlphaEvolve maintains an island-based evolutionary database inspired by MAP-Elites [21], where a prompt sampler selects parent and inspiration programs using predefined fitness-based and diversity-based heuristics. LoongFlow [18] similarly relies on a MAP-Elites archive with Boltzmann selection for `Sample`, while structuring `Generate` as a fixed Plan-Execute-Summarize pipeline where the LLM sequentially generates a modification plan, produces the code, and summarizes insights. In all these approaches, the LLM only participates in `Generate`: the sampling strategy, evaluation protocol, population management, and the order of operations are all determined by the framework, not by the LLM.

Learned variation. TTT-Discover [22] goes further by updating the LLM policy itself through test-time gradient updates, enabling the model to learn an improved `Generate` during the search. Nevertheless, `Sample` remains a fixed algorithm: a PUCT-based selection rule [23] determines which states to expand, and a buffer manages the population with predetermined update rules. Even with a learned `Generate`, the LLM's role is still confined to candidate generation within a rigid algorithmic structure that prescribes when and how it is invoked.

In contrast, the agentic variation operator we introduce in Section 3 replaces the entire `Vary` with a self-directed agent that subsumes `Sample`, `Generate`, and evaluation into a single autonomous loop. The agent has full agency over when to consult reference materials and past solutions $\mathcal{P}_t$, what diagnostic tests to run, and how to revise its optimization strategy.

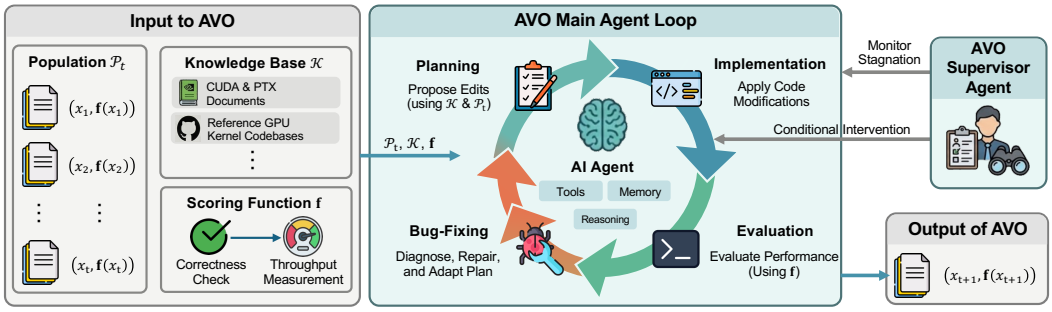

Figure 2: Illustration of the Agentic Variation Operator (AVO).

AVO is orthogonal to the choice of population structure: the agentic operator can in principle be used within archive-based, island-based, or single-lineage evolutionary regimes. In this paper we study the single-lineage setting to isolate the effect of the operator itself.

2.2 Attention Kernels on Modern GPUs

Attention computation. Given query, key, and value matrices Q, K, V, attention computes $O = \mathrm{softmax}(QK^\top/\sqrt{d})\,V$, where d is the head dimension. A naive implementation materializes the full $N \times N$ score matrix $S = QK^\top$, making the operation memory-bound for large sequence lengths N. The FlashAttention algorithm [7] avoids this by computing attention in tiles: it processes key blocks sequentially, maintaining a running softmax (with running row-maximum and row-sum) and accumulating the output O incrementally. This tiling eliminates the need to store the full score matrix, shifting the bottleneck from memory bandwidth to compute throughput on modern GPUs.

Attention kernel on Blackwell hardware. On NVIDIA's Blackwell architecture, state-of-the-art attention kernels such as FA4 [10] employ *warp specialization*: different warp groups within a thread block are assigned distinct roles in the attention pipeline. *MMA warps* execute the two core matrix multiplications via Blackwell's tensor core instructions: the QK GEMM (producing scores S) and the PV GEMM (multiplying the softmax output $P = \mathrm{softmax}(S)$ by V to accumulate the output O). *Softmax warps* compute attention weights P from the scores S, applying the online softmax algorithm with a running row-maximum. *Correction warps* rescale the output accumulator O when the running maximum changes across K-block iterations (a requirement of the online softmax algorithm). *Load and epilogue warps* handle data movement via the Tensor Memory Accelerator (TMA). In FA4's pipeline, these groups operate concurrently across two Q-tiles (a *dual Q-stage* design), with barrier-based signaling to coordinate handoffs. For causal attention, some K-block iterations are fully masked (no valid attention entries) and others are fully unmasked, leading to different execution paths within the same kernel. With FA4 already representing a highly optimized design, further improvements demand deep hardware expertise, broad exploration across diverse optimization strategies, and repetitive debugging and profiling.

3 Agentic Variation Operators

AVO consolidates the sampling, generation, and evaluation stages of evolutionary search into a single autonomous agent run, eliminating the rigid pipeline that constrains existing approaches. Below we formalize this operator, detail what occurs within a single variation step, and describe the mechanism that enables multi-day autonomous exploration.

3.1 Formulation

Previous evolutionary search approaches [3, 4] decompose the variation operator as:

$$\texttt{Vary}(\mathcal{P}_t) = \texttt{Generate}(\texttt{Sample}(\mathcal{P}_t)), \tag{3}$$

confining the LLM to the `Generate` step within a fixed pipeline. As illustrated in Figure 2, AVO replaces this decomposition with a single autonomous agent run:

$$\texttt{Vary}(\mathcal{P}_t) = \texttt{Agent}(\mathcal{P}_t, \mathcal{K}, \mathbf{f}), \tag{4}$$

where $\mathcal{P}_t = \{(x_1, \mathbf{f}(x_1)), \ldots, (x_t, \mathbf{f}(x_t))\}$ is the full lineage of solutions and their scores, $\mathcal{K}$ is a domain-specific knowledge base, and $\mathbf{f}$ is the scoring function.

In our setting, each x_i is a CUDA kernel implementation (source code with inline PTX), and $\mathbf{f}$ evaluates a candidate along two dimensions: numerical correctness against a reference implementation, and throughput in TFLOPS on the target hardware. In practice, $\mathbf{f}(x_i) = (f_1(x_i), f_2(x_i), \ldots, f_n(x_i))$ is an n-dimensional vector and f_j represents the score for test configuration j. A candidate x_i that fails correctness is assigned zero score (i.e., $f_j(x_i) = 0$) regardless of throughput. The knowledge base $\mathcal{K}$ contains CUDA programming guides, PTX ISA documentation, Blackwell architecture specifications, and existing kernel implementations including FlashAttention-4 source code.

AVO defines a family of agentic variation operators for evolutionary search. In this work, we instantiate AVO in a single-lineage autonomous run starting from an initial program x_0, producing a sequence of committed improvements $x_1, x_2, \ldots, x_t$. The accumulated lineage $\mathcal{P}_t$ serves as context for subsequent variation steps.

3.2 Anatomy of a Variation Step

A single variation step in AVO, producing x_{t+1} from the current lineage $\mathcal{P}_t$, is an autonomous agent loop. The agent is a general-purpose coding agent with planning, tool use, and persistent memory (details in Section 4), and a single step may involve numerous internal actions.

We observe that the agent frequently examines multiple prior implementations in $\mathcal{P}_t$ within a single variation step, comparing their profiling characteristics to identify bottlenecks and opportunities, and consulting documentation in $\mathcal{K}$ to understand the relevant hardware constraints before implementing a candidate optimization. The agent then invokes $\mathbf{f}$ to test the result. When a candidate fails correctness checks or fails to improve on the current benchmark suite, the agent diagnoses the issue and revises its approach, repeating this edit-evaluate-diagnose cycle until it commits a satisfactory x_{t+1}. This design allows the agent to adapt its optimization strategy as the search progresses: early steps may focus on structural changes informed by reference implementations in $\mathcal{K}$, while later steps can shift toward micro-architectural tuning guided by profiling feedback from $\mathbf{f}$ and patterns observed across the accumulated lineage $\mathcal{P}_t$.

In our current implementation, we persist a new committed version only when it passes correctness checks and matches or improves the benchmark score relative to the best committed version so far; unsuccessful intermediate attempts remain part of the agent's internal search trajectory but are not added to the committed lineage.

3.3 Continuous Evolution

Although AVO is defined at the level of variation operators for evolutionary search, the present study evaluates a single-lineage continuous instantiation, leaving a full treatment of population-level branching and archive management to future work; we report a preliminary cross-lineage study in Appendix F. The AVO agent operates as a continuous loop that periodically produces new solutions without human intervention. Each committed version x_i is persisted as a git commit along with its score, maintaining full state continuity across the entire evolutionary process.

In long-running autonomous optimization, two failure modes can impede progress: the agent may *stall* when it exhausts its current line of exploration, or it may enter *unproductive cycles* of edits that repeatedly fail to improve scores. To mitigate both, AVO incorporates a self-supervision mechanism that detects these scenarios and intervenes. Once triggered, the mechanism reviews the overall evolutionary trajectory and steers the search toward several candidate optimization directions. This conditional intervention effectively redirects exploration with fresh perspective when the current strategy has plateaued.

Concretely, each run maintains the archive $\mathcal{P}_t$ of committed implementations and their measured fitness on the local filesystem, which the agent may consult freely when producing a new candidate. The 7-day run that produced our final multi-head attention kernel spanned 40 successive versions.

Throughout this process, the main agent autonomously decided when to attempt new optimizations, when to revisit earlier approaches in $\mathcal{P}_t$, and when to shift strategy, while the supervisor maintained forward progress by intervening during periods of stagnation.

4 Experiments

This section reports our results on multi-head attention and grouped-query attention kernels. Additional applications of AVO to other kernel optimization tasks and comparisons against existing evolutionary search (EVO) baselines are provided in Appendix A and Appendix B, respectively.

4.1 Setup

Agent. We use an internally-developed general-purpose coding agent powered by Claude Opus 4.6 as the AVO variation operator. The agent has access to standard software engineering tools, including autonomous code editing, shell command execution, file system navigation, and documentation retrieval. It maintains persistent memory through its conversation history, which accumulates the full context of prior edits, compiler outputs, profiling results, and reasoning across the evolutionary process. No task-specific modifications are made to the agent for kernel optimization; the same agent used for general software engineering tasks is deployed here, with the domain-specific knowledge base $\mathcal{K}$ and scoring function $\mathbf{f}$ provided to the agent as described in Section 3.1.

Hardware and software. Following the setup of FA4 [10], all experiments in this section are conducted on NVIDIA B200 GPUs with CUDA 13.1 and PyTorch 2.10.0.

Baselines. We compare against two state-of-the-art baselines: (1) **cuDNN**: NVIDIA's closed-source attention kernel, measured using cuDNN version 9.19.1, which includes custom optimizations for Blackwell; and (2) **FlashAttention-4** (FA4) [10]: the latest open-source attention kernel optimized for Blackwell, measured using the official implementation (commit `71bf77c`).

Benchmark Configurations. We evaluate the forward prefilling throughput with head dimension 128 and BF16 precision across sequence lengths $\{4096, 8192, 16384, 32768\}$. Following FlashAttention-4 [10], we control the total number of tokens to 32768 by adjusting the batch size for each sequence length (e.g., batch size 8 at sequence length 4096, batch size 1 at sequence length 32768). For multi-head attention (MHA), we use 16 heads under both causal and non-causal masking. For grouped-query attention (GQA), we evaluate two configurations drawn from the Qwen3 model family [24]: 32 query heads with 4 KV heads (group size 8, as in Qwen3-30B-A3B) and 32 query heads with 8 KV heads (group size 4, as in Qwen3-8B). For throughput measurement, we use the same timing script from the FA4 repository[2] and the same number of warm-up and repeat rounds as the FA4 paper. In addition, we run each experiment 10 times to obtain the average performance and the standard deviation. The same setup is used both for agent evolution and for benchmarking the final evolved kernels against the baselines.

4.2 Multi-Head Attention

Figure 3 presents the benchmarking results for MHA. On causal attention, AVO outperforms both baselines across all tested configurations, with gains ranging from $+0.4\%$ to $+3.5\%$ over cuDNN and $+5.0\%$ to $+10.5\%$ over FA4. On non-causal attention, AVO achieves modest gains at longer sequences ($+1.8\%$ to $+2.4\%$ over cuDNN at sequence lengths larger than 16384) but is within measurement noise of both baselines at shorter sequences. In Section 4.4, we show how the agent obtains the performance gains through continuous evolution.

4.3 Grouped-Query Attention

To evaluate whether agent-discovered optimizations transfer beyond the benchmark settings used in evolution, we prompted the AVO agent to adapt the evolved MHA kernel to support GQA. The agent

[2] `https://github.com/Dao-AILab/flash-attention/blob/main/benchmarks/benchmark_attn.py`, BSD-3-Clause license.

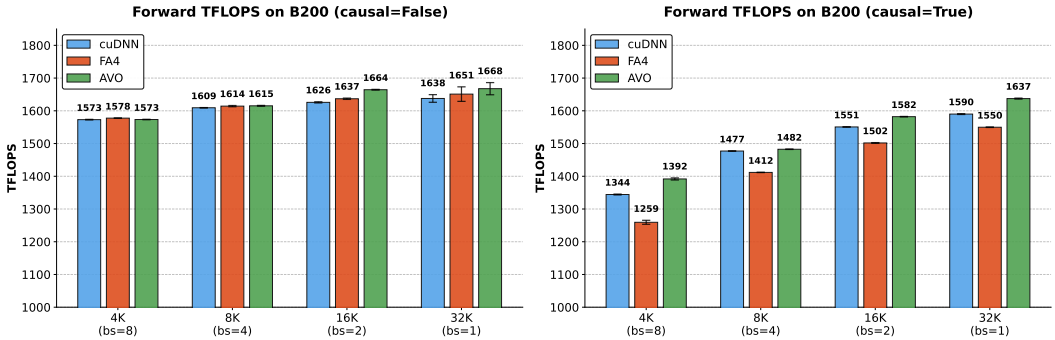

Figure 3: Multi-head attention forward-pass prefilling throughput (TFLOPS) on NVIDIA B200 with head dimension 128, 16 heads, and BF16 precision. Batch size and sequence length are varied with a fixed total of 32k tokens.

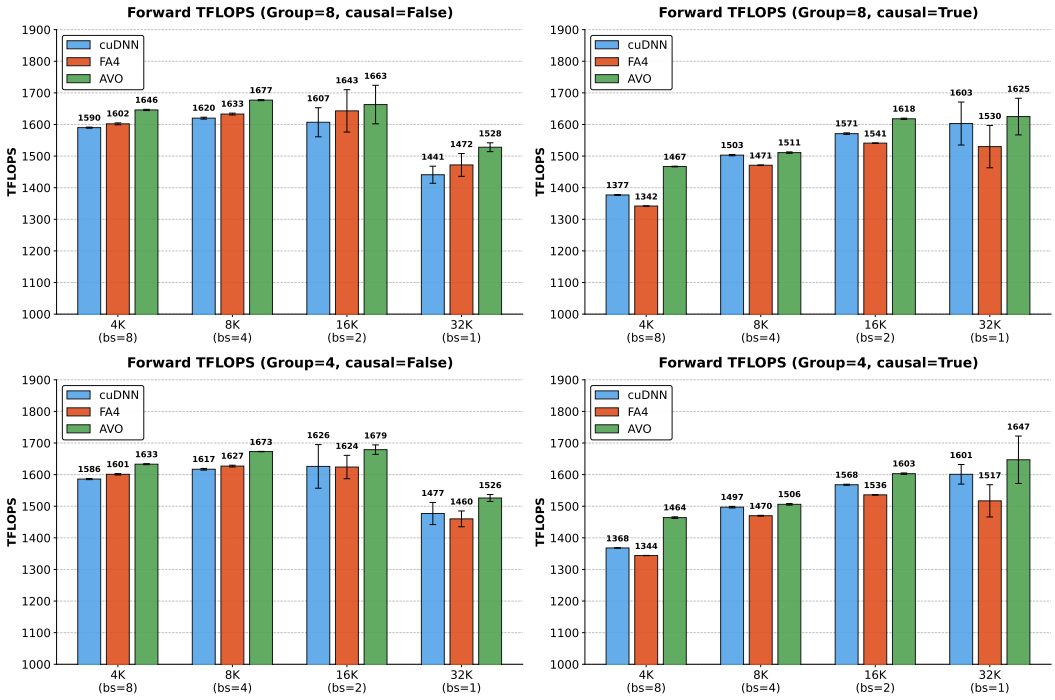

Figure 4: Grouped-query attention forward-pass prefilling throughput (TFLOPS) on NVIDIA B200 with 32 query heads, head dimension 128 and BF16 precision. Results are shown for two GQA configurations (group sizes 8 and 4) under both causal and non-causal masking. The GQA kernel was produced by prompting the AVO agent to adapt the evolved MHA kernel, requiring approximately 30 minutes of autonomous effort.

completed this adaptation autonomously in approximately 30 minutes, producing a GQA-capable kernel without any human guidance on the required changes.

Figure 4 presents the results across two GQA configurations. AVO outperforms both baselines across all configurations. On causal GQA, AVO achieves up to $+7.0\%$ over cuDNN and $+9.3\%$ over FA4. On non-causal GQA, gains reach up to $+6.0\%$ over cuDNN and $+4.5\%$ over FA4. The strong GQA performance demonstrates that the optimizations discovered by the agent during MHA evolution are not specific to the MHA configurations used during evolution, but generalize to the distinct compute and memory access patterns of GQA.

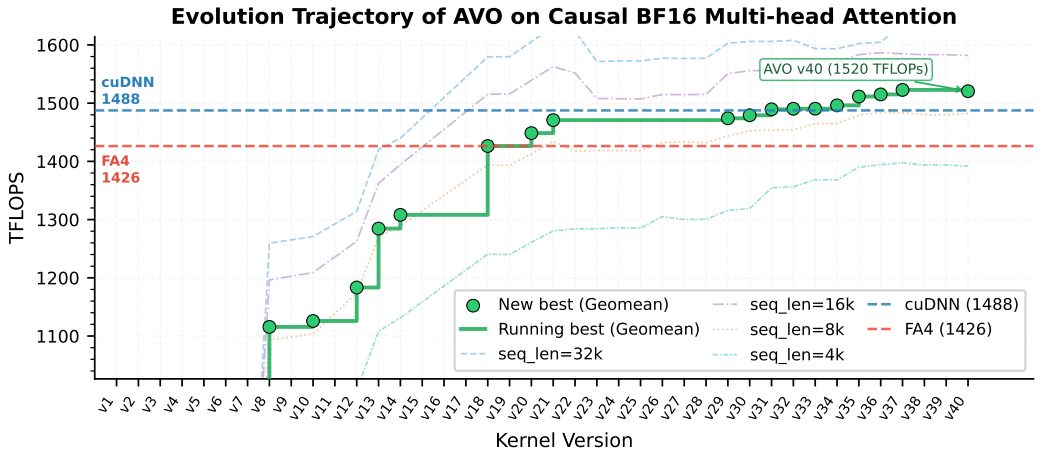

Figure 5: Evolution trajectory of AVO across 40 kernel versions over 7 days on **causal** MHA. The solid green line tracks the running-best geometric mean throughput across all configurations; green circles mark versions that set a new best. Dashed colored lines show per-configuration throughput (seq_len = 4k, 8k, 16k, 32k). Horizontal dashed lines indicate the geometric mean throughput of cuDNN and FA4.

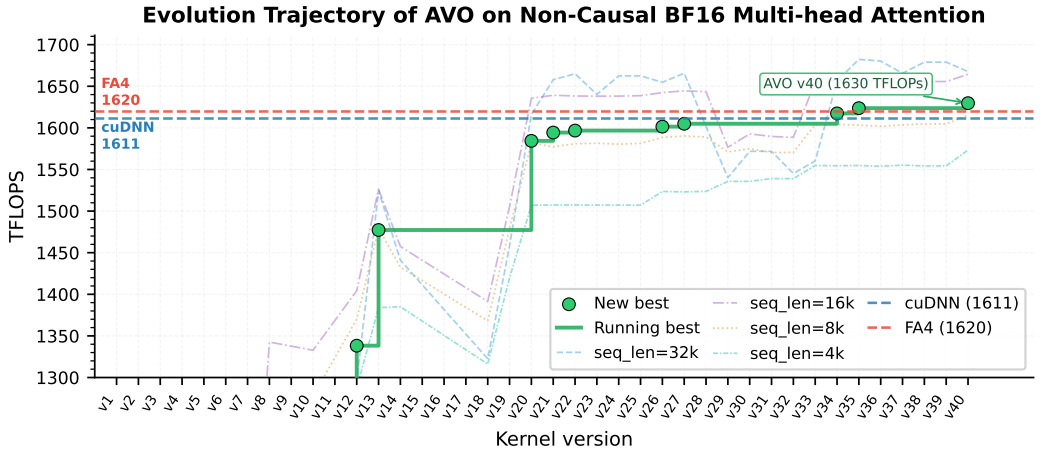

Figure 6: Evolution trajectory of AVO across 40 kernel versions over 7 days on **non-causal** MHA. The solid green line tracks the running-best geometric mean throughput across all configurations; green circles mark versions that set a new best. Dashed colored lines show per-configuration throughput (seq_len = 4k, 8k, 16k, 32k). Horizontal dashed lines indicate the geometric mean throughput of cuDNN and FA4.

4.4 Evolution Trajectory

In Figure 5 and Figure 6, we show the evolution trajectory of AVO across the 40 committed kernel versions produced during the 7-day evolution. Note that these trajectories visualize the committed sequence, rather than the full internal search tree explored between the commits. We observe the following patterns:

Scale of exploration. The 40 committed versions shown in the trajectory represent only the successful outcomes of a much larger search. Over the 7-day evolution, the agent explored over 500 candidate optimization directions internally, including attempts that failed correctness checks, regressed throughput, or were abandoned after profiling. This volume of systematic exploration, each direction requiring reading documentation, implementing changes, compiling, testing, and profiling, far exceeds what a human engineer could accomplish in the same timeframe.

Table 1: Summary of the agent-discovered optimizations examined in this section and their measured ablation gains (geomean throughput improvement over the immediately preceding version, across all benchmark configurations).

Optimization	Versions	Non-causal	Causal
Branchless accumulator rescaling	v19 $\rightarrow$ v20	$+8.1\%$	$+1.6\%$
Correction/MMA pipeline overlap	v29 $\rightarrow$ v30	$+1.1\%$	$+0.4\%$
Register rebalancing across warp groups	v32 $\rightarrow$ v33	$+2.1\%$	$\sim 0\%$

Discrete jumps rather than gradual improvement. Throughput improves in distinct steps separated by plateaus where successive versions refine implementation details without measurably changing performance. The five largest gains correspond to architectural inflection points: the introduction of QK-PV interleaving with bitmask causal masking (version 8), a restructured single-pass softmax computation (version 13), the branchless accumulator rescaling with a lighter memory fence for unmasked iterations (version 20), the correction/MMA pipeline overlap (version 30), and register rebalancing across warp groups (version 33). We discuss some of the representative optimizations in Section 5. The remaining versions contribute individually smaller but collectively substantial micro-architectural refinements.

5 Analysis of Agent-Discovered Optimizations

The 40-version AVO evolution produced multi-level optimizations that individually yield measurable throughput gains and collectively account for the improvements reported in Section 4. We examine three representative optimizations to illustrate the nature and depth of the agent's hardware reasoning. For each, we describe the bottleneck the agent identified in its own kernel, the change it made, and its measured impact (ablation between the version immediately before and after). Table 1 summarizes their measured impact.

5.1 Branchless Accumulator Rescaling

Bottleneck. The online softmax algorithm requires the output accumulator O to be rescaled whenever the running row-maximum changes across key blocks (Section 2.2). In version 19 of the AVO kernel, this rescaling was guarded by a conditional branch that checked whether any thread in the warp required it and skipped the operation when the maximum was unchanged. While this avoids unnecessary computation, the branch introduces warp synchronization overhead on every iteration of the key-block loop, and the conditional control flow prevents the use of lighter memory fences in the correction path.

AVO's approach. In version 20, the agent replaced the conditional branch with a branchless speculative path: the rescale factor is always computed, and a predicated select substitutes 1.0 when rescaling is unnecessary, at a cost negligible compared to the synchronization overhead it replaces. Eliminating the branch also removed warp divergence in the correction path, allowing a blocking memory fence to be replaced with a lighter non-blocking fence that merely enforces ordering; this is safe because uniform control flow guarantees reconvergence before the next synchronization point. Together these changes yield $+8.1\%$ geomean throughput on non-causal and $+1.6\%$ on causal attention, the largest single optimization in the evolution. The asymmetry arises because the branchless path applies only to fully unmasked iterations: non-causal attention processes all key blocks without masking, while causal attention retains the branched logic for masked blocks.

5.2 Correction/MMA Pipeline Overlap

Bottleneck. The attention pipeline processes two Q-tiles concurrently (dual Q-stage; see Section 2.2), each requiring a PV GEMM followed by output normalization by the correction warp. In version 29 of the AVO kernel, the two stages were serialized at the MMA-to-correction boundary: the correction warp had to wait for both PV GEMMs to complete before it could begin normalizing either stage's output, leaving it idle throughout the second GEMM.

AVO's approach. In version 30, the agent restructured the pipeline to allow the correction warp to begin normalizing the first stage's output as soon as its PV GEMM completes, overlapping this work with the second stage's PV GEMM. This converts a sequential dependency into a pipelined execution, reducing the idle time on the correction warp and yielding $+1.1\%$ geomean throughput on non-causal and $+0.4\%$ on causal attention.

5.3 Register Rebalancing Across Warp Groups

Bottleneck. Blackwell partitions a fixed budget of 2048 warp-registers per SM across warp groups. In version 32 of the AVO kernel, the allocation followed the pattern of FlashAttention-4 [10]: 192 registers for the 8 softmax warps, 80 for the 4 correction warps, and 48 for the remaining 4 warps. Profiling revealed that the correction warp group was spilling values to slower local memory due to its limited 80-register budget, while the softmax group had substantial headroom.

AVO's approach. In version 33, the agent redistributed 8 registers from the softmax group to each of the other two groups, arriving at a 184/88/56 allocation. This redistribution is viable because the AVO kernel's softmax implementation processes score values in small fragments with packed arithmetic, resulting in a low peak register usage that leaves ample headroom even at 184 registers. The correction warp group benefits from the additional registers because, following the pipeline overlap optimization (Section 5.2), it runs concurrently with the second PV GEMM and is on the execution critical path. With 88 rather than 80 registers, fewer output values spill to local memory, reducing stalls and yielding $+2.1\%$ geomean throughput on non-causal attention (within noise on causal).

5.4 Discussion

What is notable about these optimizations is that each requires jointly reasoning about multiple hardware subsystems, including synchronization and memory ordering, pipeline scheduling, and register allocation, rather than tuning any single parameter in isolation. This depth of reasoning, carried out autonomously through iterative interaction with documentation and profiling feedback, suggests that agentic variation operators can serve as an effective mechanism for expert-level kernel optimization.

6 Conclusion

We introduced Agentic Variation Operators (AVO), a new family of evolutionary variation operators that elevate the agent from candidate generator to variation operator. Applied to forward-pass attention on NVIDIA Blackwell GPUs, AVO produces kernels surpassing cuDNN by up to 3.5% and FlashAttention-4 by up to 10.5% over 7 days of continuous autonomous evolution. Furthermore, we show that the discovered optimizations transfer readily to grouped-query attention, requiring only 30 minutes of additional autonomous adaptation. Beyond attention, AVO proves effective across a broad range of GPU kernels, outperforming the expert-tuned reference on every additional kernel class we evaluate (Appendix A). Together, these results demonstrate that AVO can discover performance-critical micro-architectural optimizations that produce kernels surpassing state-of-the-art expert-engineered implementations. Because AVO operates at the level of variation operators rather than being tied to a specific domain, it points toward a broader path for autonomous optimization beyond attention kernels, including other performance-critical software systems on diverse hardware platforms, and engineering or scientific domains that demand extended autonomous exploration.

Acknowledgement

We thank the NVIDIA Cutlass, cuDNN, TensorRT-LLM, FlashInfer, DevTech, IPP, and Compiler teams for valuable feedback and support. We also thank the FlashAttention-4 authors for open-sourcing their implementation and benchmark scripts, which served as a baseline and a reference for this work. This work was funded and supported by NVIDIA.

References

[1] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3):339–363, 2010.

[2] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. 2022. URL https://arxiv.org/abs/2206.08896.

[3] 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, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

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

[5] Angelica Chen, David M. Dohan, and David R. So. Evoprompting: Language models for code-level neural architecture search. 2023. URL https://arxiv.org/abs/2302.14838.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023. URL https://arxiv.org/abs/1706.03762.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. 2022. URL https://arxiv.org/abs/2205.14135.

[8] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. 2023. URL https://arxiv.org/abs/2307.08691.

[9] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. 2024. URL https://arxiv.org/abs/2407.08608.

[10] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. 2026. URL https://arxiv.org/abs/2603.05451.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. 2014. URL https://arxiv.org/abs/1410.0759.

[12] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? 2024. URL https://arxiv.org/abs/2310.06770.

[13] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. 2024. URL https://arxiv.org/abs/2405.15793.

[14] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. 2025. URL https://arxiv.org/abs/2407.16741.

[15] Anthropic. Claude 3.7 sonnet and claude code. https://www.anthropic.com/news/claude-3-7-sonnet, February 2025. Accessed: 2026-03-25.

[16] OpenAI. https://openai.com/index/introducing-codex/. https://openai.com/index/introducing-codex/, May 2025. Accessed: 2026-03-25.

[17] Bing Xu, Terry Chen, Fengzhe Zhou, Tianqi Chen, Yangqing Jia, Vinod Grover, Haicheng Wu, Wei Liu, Craig Wittenbrink, Wen mei Hwu, Roger Bringmann, Ming-Yu Liu, Luis Ceze, Michael Lightstone, and Humphrey Shi. VibeTensor: System Software for Deep Learning, Fully Generated by AI Agents. 2026.

[18] Chunhui Wan, Xunan Dai, Zhuo Wang, Minglei Li, Yanpeng Wang, Yinan Mao, Yu Lan, and Zhiwen Xiao. Loongflow: Directed evolutionary search via a cognitive plan-execute-summarize paradigm. 2025. URL https://arxiv.org/abs/2512.24077.

[19] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. Handbook of evolutionary computation. *Release*, 97(1):B1, 1997.

[20] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. 2024. URL https://arxiv.org/abs/2402.01145.

[21] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. 2015.

[22] Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. 2026. URL https://arxiv.org/abs/2601.16175.

[23] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

[25] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025. URL https://arxiv.org/abs/2412.06464.

[26] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

[27] Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL https://github.com/fla-org/flash-linear-attention.

[28] NVIDIA. cuequivariance, 2024. URL https://github.com/NVIDIA/cuEquivariance. GitHub repository.

[29] Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-coder-next technical report, 2026. URL https://arxiv.org/abs/2603.00729.

[30] Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems, 2026. URL https://arxiv.org/abs/2601.00227.

[31] 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.

[32] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL `https://github.com/algorithmicsuperintelligence/openevolve`.

[33] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

A Generalization to Additional Kernel Optimization Tasks

Beyond the dense MHA and GQA prefill kernels of Section 4, we evaluate AVO on two further groups of kernels: linear and structured attention operators (Appendix A.1), and sparse attention and non-attention kernels (Appendix A.2).

A.1 Linear and Structured Attention

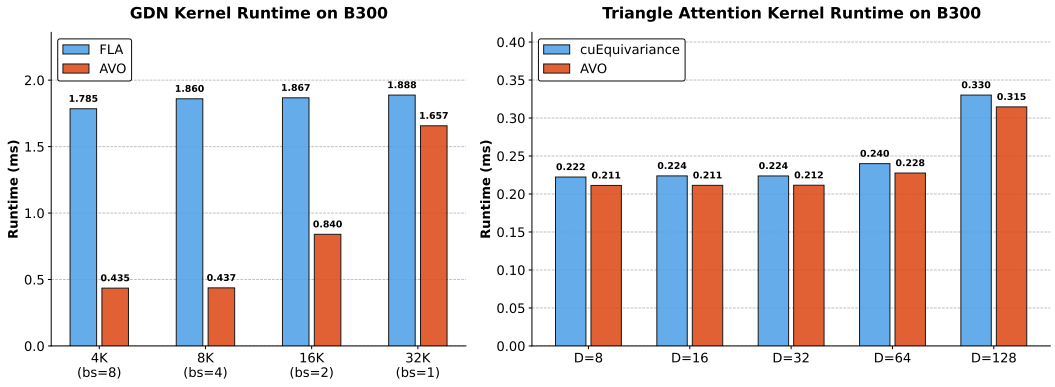

Figure 7: Kernel runtime (ms) on NVIDIA Blackwell B300 GPUs, comparing AVO against Flash Linear Attention (FLA) for GatedDeltaNet and against NVIDIA's cuEquivariance for Triangle Attention. **Left:** GatedDeltaNet (GDN) forward-pass runtime under varying sequence lengths and batch sizes (maintaining a constant 32K total tokens). **Right:** Triangle Attention runtime across different hidden dimensions ($D = 8$ to 128) with a fixed sequence length of 512 and 4 heads.

Tasks and Baselines. We further apply AVO to two widely used GPU kernels: (1) GatedDeltaNet [25], a recent linear attention operator serving as an efficient alternative to traditional softmax attention computation; and (2) Triangle Attention, introduced in AlphaFold2 [26] to model the geometric relationships between interacting amino acid pairs. We use expert-optimized libraries as our performance baselines: Flash Linear Attention (FLA) [27] for GatedDeltaNet, and NVIDIA's cuEquivariance [28] for Triangle Attention.

Agent and Hardware. We use the same agent setup, time budget (i.e., 7 days per task), and benchmarking protocol as described in Section 4. All evaluations are conducted on NVIDIA Blackwell B300 GPUs (rather than the B200 used in the main experiments).

Benchmark Configurations. For GatedDeltaNet, we benchmark using the configuration of Qwen3-Next-80B-A3B [29] (matching the head count and head dimension), and test performance under the same sequence length and batch size combinations as in Section 4. For Triangle Attention, we fix the sequence length at 512 and the head count at 4 to mirror realistic biological workloads and the AlphaFold2 [26] architecture, respectively. We sweep the head dimension (D) from 8 to 128 to systematically evaluate the kernel's performance scaling from memory-bound to compute-bound regimes.

Results. As shown in Figure 7, AVO generalizes to both kernels and yields substantial gains over the expert baselines. For GatedDeltaNet (Figure 7, left), the AVO-generated kernel substantially reduces runtime compared to FLA across all sequence lengths. Most notably, at shorter sequence lengths with larger batch sizes (e.g., 4K sequence length, batch size 8), AVO achieves a $4.1\times$ speedup, reducing latency from 1.785 ms to 0.435 ms. For Triangle Attention (Figure 7, right), AVO-generated kernels consistently outperform NVIDIA's cuEquivariance library across all tested hidden dimensions, with a stable runtime reduction of roughly 4.7% to 5.5% across the $D = 8$ to 128 sweep. These results demonstrate AVO's capability to discover performance optimizations even when competing against vendor-engineered libraries.

A.2 Sparse Attention and Non-Attention Kernels

Tasks and Baselines. We apply AVO to three further kernel classes: (1) sparse multiple-sequence-alignment (MSA) attention, benchmarked against the official MiniMax implementation; (2) fused FP8 mixture-of-experts (MoE), benchmarked against the FlashInfer-Bench [30] reference; and (3) MXFP4 grouped GEMM, benchmarked against the OpenAI Triton reference. Sparse MSA extends the evaluation from dense to sparse attention, while fused MoE and grouped GEMM fall outside attention entirely.

Agent and Hardware. Each task is evolved for 24 hours on a single NVIDIA B200, using the same agent setup and benchmarking protocol as Section 4. No task-specific changes are made to the search procedure; adapting AVO to a new task requires only supplying the corresponding correctness and performance benchmark.

Results. As shown in Table 2, AVO outperforms the expert-engineered reference on all three kernel classes. Together with the linear and structured attention results above, this indicates that the agentic variation operator transfers across kernel classes rather than being specialized to attention.

Table 2: AVO on sparse attention and non-attention kernels. Speedup is the geometric mean over the listed workloads relative to the expert-engineered reference.

Kernel class	Reference	Workloads	Speedup
Fused FP8 MoE	FlashInfer-Bench [30]	19 traces; len. 1–14,107	$1.604\times$
MXFP4 grouped GEMM	OpenAI Triton	5 layers $\times$ 4 token counts	$1.102\times$
Sparse MSA	MiniMax MSA	batch 1–4; len. 4,096–32,768	$1.492\times$

B Comparison with EVO Baselines

To compare AVO against prior LLM-driven evolutionary search frameworks, we evaluate on three kernel optimization tasks adopted from K-Search [31], drawn from the FlashInfer-Bench [30] dataset: **GQA decode**, **MLA decode**, and **MLA prefill**. All experiments are conducted on a server with a single NVIDIA H100 GPU.

Baselines. We compare AVO against three representative LLM-based evolutionary search (EVO) baselines: K-Search [31], OpenEvolve [32] (an open-source implementation of AlphaEvolve [4]), and ShinkaEvolve [33]. We follow the same benchmarking script and evaluation protocol as K-Search; the speedup of each generated kernel is measured against the FlashInfer-Bench [30] reference implementation.

AVO setup. We use the same agentic setup as described in Section 4.1. For a fair comparison with the EVO baselines, we cap the AVO evolution time budget per task at the wall-clock time required by K-Search, the most time-consuming baseline, on the same server, which averages 11.22 hours per task across the three benchmarks.

Results. Figure 8 compares AVO-generated kernels against those produced by the EVO baselines. On all three tasks, AVO outperforms the strongest EVO baseline (K-Search) by 33.0%, 38.6%, and 22.0% on GQA decode, MLA decode, and MLA prefill respectively, corresponding to a geometric-mean improvement of approximately 31%. Beyond the comparison with EVO baselines, AVO

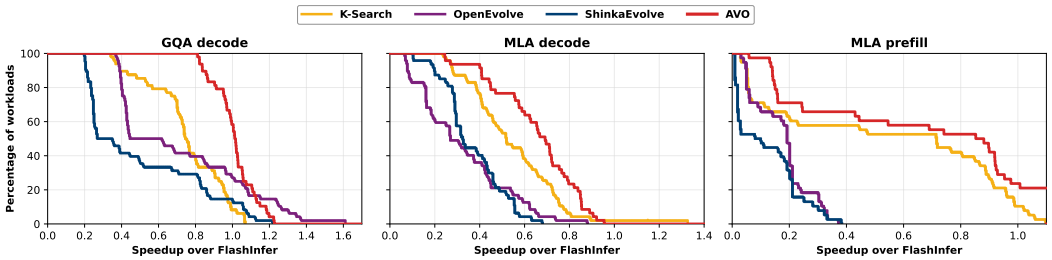

Figure 8: Time-matched comparison between AVO and prior LLM-driven evolutionary search frameworks (K-Search, OpenEvolve, ShinkaEvolve) on three FlashInfer-Bench tasks. Each curve shows the percentage of workloads (y-axis) achieving at least the corresponding speedup over the FlashInfer-Bench reference implementation (x-axis); curves further to the upper right indicate stronger and more uniform improvements. All methods are run for 11.22 hours per task on a single NVIDIA H100 GPU.

further identifies optimization opportunities that surpass the FlashInfer-Bench reference: for example, AVO is faster than the reference on 60.4% of GQA decode workloads and 10.5% of MLA prefill workloads. Together, these results confirm AVO's consistent advantage over prior EVO baselines under a matched wall-clock budget on the same hardware, isolating the variation operator itself as the source of the gains, and further show that AVO can discover non-trivial optimizations that match or exceed expert-engineered references on a meaningful fraction of workloads.

C Comparison Using FA4-Reported Baseline Performance

Section 4 reports cuDNN and FA4 throughput measured on our hardware. In practice, minor system-level differences (driver versions, thermal conditions, clock frequencies) can affect absolute TFLOPS. Therefore, we additionally compare AVO against the cuDNN and FA4 numbers published in the FA4 paper [10]. Figure 9 presents this comparison.

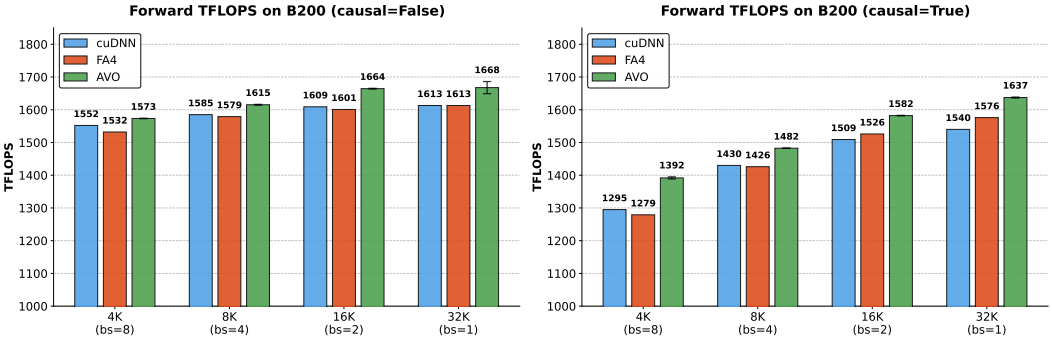

Figure 9: Multi-head attention forward-pass throughput (TFLOPS) on NVIDIA B200, comparing AVO (measured on our hardware) against cuDNN and FA4 baseline numbers as reported in the FA4 paper [10]. Head dimension 128, 16 heads, BF16. Left: non-causal. Right: causal.

On non-causal attention, AVO outperforms the FA4-reported baselines across all configurations, with gains of $+1.4\%$ to $+3.4\%$ over cuDNN and $+2.3\%$ to $+3.9\%$ over FA4. On causal attention, AVO achieves $+3.6\%$ to $+7.5\%$ over cuDNN and $+3.7\%$ to $+8.8\%$ over FA4, with the largest gains observed at shorter sequences (bs=8, seq=4096). These results are broadly consistent with the comparisons in Section 4.

D Generalization to Unseen Workload Shapes

Appendix A asks whether AVO transfers to new kernels; here we ask whether the evolved kernels transfer to workload shapes they were never tuned on. The benchmark suite of Section 4—

four sequence lengths under both causal and non-causal masking—also defined the scoring function during evolution, so we expand the sweep to batch sizes $\{1, 2, 4, 8, 16\}$, sequence lengths $\{1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072\}$, and both causal and non-causal masking, yielding 80 configurations per kernel family. Both kernels are evaluated on an NVIDIA B200 using the same correctness and timing protocol as Section 4.1. Improvements are reported as the geometric mean of baseline latency divided by AVO latency, and "wins" counts the configurations on which AVO attains lower latency.

Table 3: AVO across 80 configurations per kernel family (batch size $\times$ sequence length $\times$ mask), well beyond the benchmark suite used during evolution.

Kernel	vs. cuDNN	Wins	vs. FA4	Wins
MHA	$+9.97\%$	68/80	$+3.03\%$	74/80
GQA	$+7.07\%$	75/80	$+2.45\%$	68/80

Within this grid, the most demanding subset is long-context inference. Table 4 reports the geometric mean over sequence lengths $\{65536, 131072\}$ and both attention masks at each batch size. Gains over cuDNN are substantially larger at the higher batch sizes, while gains over FA4 remain positive throughout.

Table 4: Long-context subset of the 80-configuration grid. Each entry is the geometric mean over sequence lengths $\{65536, 131072\}$ and both causal and non-causal masking.

Kernel	Batch	vs. cuDNN	vs. FA4
MHA	2	$+11.94\%$	$+1.77\%$
	4	$+21.10\%$	$+1.35\%$
	8	$+23.92\%$	$+1.89\%$
	16	$+22.70\%$	$+3.42\%$
GQA	2	$+7.68\%$	$+1.07\%$
	4	$+8.59\%$	$+1.15\%$
	8	$+9.04\%$	$+1.24\%$
	16	$+9.13\%$	$+1.29\%$

The speedups are therefore not an artifact of the benchmark suite that defined the search objective: they hold across two orders of magnitude in sequence length, and at batch–length combinations the scoring function never covered, since evolution tied each sequence length to a single batch size.

E Ablation Study

To quantify how much each element of the AVO loop contributes, we disable one component of the full system at a time and re-run the search with everything else unchanged. We remove the self-supervision mechanism of Section 3.3 and, separately, the knowledge base $\mathcal{K}$ of Section 3.1. All conditions share the same backbone model, tools, compiler, correctness tests, benchmark harness, GPU, and 24-hour budget per task; only the ablated component differs. We evaluate on GQA decode (H100) and fused FP8 MoE (B200), following the task setup of FlashInfer-Bench [30].

Table 5: Ablation of AVO under a matched 24-hour budget per task. Each row disables one component of the full system. Lower latency is better.

Configuration	GQA decode latency	FP8 MoE latency
Full AVO	$\mathbf{18.39}\ \mu s$	$\mathbf{327.16}\ \mu s$
AVO without self-supervision	$34.57\ \mu s$	$484.01\ \mu s$
AVO without the knowledge base $\mathcal{K}$	$18.84\ \mu s$	$373.69\ \mu s$

Disabling self-supervision costs the most: full AVO attains 46.8% lower latency on GQA decode and 32.4% on FP8 MoE, even though the variation operator retains the knowledge base, the archive, and the evaluator. Across the search trajectories, the stagnation episodes that trigger self-supervision

recur in three forms: repairing one compilation error only to expose another; oscillating between faster-but-incorrect and correct-but-slower candidates; and regressing throughput through increased register pressure or reduced parallelism. Sustained progress over a long horizon therefore depends on monitoring and redirecting the trajectory, not only on implementing and testing individual candidates. Removing the knowledge base costs 2.4% and 12.5% respectively, a consistent but substantially smaller effect.

F Toward Population-Level AVO

The main experiments instantiate AVO in a single-lineage regime (Section 3.3). Because AVO replaces the variation operator rather than the population structure, it is compatible with both unary mutation and multi-parent recombination.

Lineage-internal semantic crossover. Committed solutions, their scores, and associated artifacts remain in the archive $\mathcal{P}_t$ and are directly accessible to the agent, which can inspect several historical kernels and integrate compatible optimizations into a single candidate. We call this *lineage-internal semantic crossover*: it recombines mechanisms rather than source text, but all parents share one trajectory.

Cross-lineage population sharing. We compare eight isolated AVO runs, each with a private population, against eight lineages whose populations are synchronized every four hours, exposing each lineage to kernels and measurements found by the others. Both settings use eight four-hour windows (32-hour wall clock, up to 256 aggregate GPU-hours per workload), with model, initialization, prompts, hardware, and evaluator unchanged. Tasks are taken from FlashInfer-Bench [30] and run on H100 GPUs.

Table 6: Preliminary cross-lineage study under a matched aggregate budget. Lower latency is better; the isolated baseline is the *best* of eight independent replicas.

Evolutionary regime	GQA decode	MLA decode	MLA prefill
Best isolated AVO run (8 replicas)	$15.54\ \mu s$	$20.58\ \mu s$	$32.90\ \mu s$
Synchronized population (8 lineages)	$\mathbf{14.65}\ \mu s$	$\mathbf{17.98}\ \mu s$	$\mathbf{31.57}\ \mu s$
Latency reduction	5.7%	12.6%	4.0%

Even against the best of eight isolated runs, simple periodic sharing reduces geomean latency by 4.0% to 12.6%. We report this as preliminary evidence that AVO benefits from cross-lineage solutions, not as a tuned population-based system; lifting the agent itself to the population level, so that it reconciles parents carrying complementary optimizations, is a natural next step that we leave to future work.

G Limitations and Future Work

Broader hardware coverage. Our main evaluation targets server-grade NVIDIA Blackwell (B200) GPUs, where cuDNN and FlashAttention-4 have already absorbed months of expert tuning and therefore provide the most stringent test of autonomous kernel optimization. We additionally evaluate AVO on Blackwell B300 (Appendix A.1) and Hopper H100 (Appendices B, E, and F). Extending AVO to other accelerator families and to resource-constrained client and edge devices, and characterizing how the discovered optimization patterns transfer across architectures, remain future work.

No convergence guarantee. AVO provides no formal guarantee of converging to a global optimum: the trajectories in Figures 5 and 6 show diminishing returns rather than optimality, and the quality of the final kernel depends on the search budget and on the capability of the underlying model. The evolved kernels also remain open to further refinement, both by human experts and by complementary automated optimization tools.

Co-evolving the knowledge base. In our current setup, the knowledge base $\mathcal{K}$ is fixed throughout the evolutionary run. While this is sufficient to reach state-of-the-art performance on the benchmarked configurations, it leaves the agent reliant on a static body of external material that does not capture

the lessons accumulated during the search itself. A natural extension is to co-evolve $\mathcal{K}$ alongside the candidate solutions, allowing the agent to distill profiling observations, failed optimization attempts, and recurring hardware-level patterns into the knowledge base as the run progresses.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims are supported in Sec. 4, Sec. 5, and Appendix A and B.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in Appendix G.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: This paper has no theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The AVO method is described in Sec. 3, and experimental setups are detailed in Sec. 4.1, Appendix A, and Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: Code is not provided.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper has no training. Testing setups are disclosed in Sec. 4.1, Appendix A, and Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Fig. 3 and Fig. 4 report mean and one standard deviation over 10 independent runs, as described in Sec. 4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Agent running time and GPU configurations are detailed in Sec. 4.1, Appendix A, and Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We confirm that the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This work is fundamental research on general agentic-driven search methods.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This paper does not release new data or model.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All third-party assets are cited; the license of the FlashAttention-4 benchmarking script (BSD-3-Clause) is stated in Sec. 4.1.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: We do not release new assets in this paper.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are a core component of the proposed method: the Agentic Variation Operator (Section 3) is instantiated as an autonomous coding agent powered by Claude Opus 4.6, as detailed in Section 4.1.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.