ASMEVO: AGENTIC ASSEMBLY-LEVEL OPTIMIZATION OF AMD GPU KERNELS WITH FUNCTIONAL EQUIVALENCE VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

High-performance ML systems increasingly rely on GPU kernels whose editable source is unavailable, generated, or too distant from final machine code to expose remaining optimizations. Existing LLM kernel optimizers and autotuners mainly operate on CUDA, Triton, HIP, or tensor-program source and validate against reference implementations. We study a stricter setting: optimizing an already compiled AMDGPU code object, using K_0 as the behavioral reference under a captured launch contract. We present AsmEvo, an agentic assembly-level optimizer for AMD GPU kernels. Given an AMDGPU code object K_0, AsmEvo reconstructs a reassemblable representation, proposes profile-guided low-level edits with a long-horizon agent, rebuilds an ABI-preserving object with regenerated metadata or a conservative in-place patch, and accepts candidates only after capture-scoped equivalence and paired validation against K_0 in a generated persistent evaluator. On MI308X, AsmEvo improves all 30 selected KernelBench kernels, reaching $1.41\times$ geometric-mean and $3.53\times$ maximum speedup. On MI300X production workloads, it improves all evaluated AITER binaries and vLLM/SGLang Triton assembly kernels, reaching $1.12\times$/$1.33\times$ and $1.49\times$/$2.08\times$ geometric-mean/maximum speedups, respectively, while preserving functional equivalence.

1 INTRODUCTION

Modern machine learning workloads depend on GPU kernels whose final performance is decided not only by high-level schedules, but also by low-level decisions made after lowering: wait-counter placement, instruction selection, register allocation, kernel descriptors, and ABI-visible metadata (Tillet et al., 2019; Chen et al., 2018a; Zheng et al., 2020). Compilers (Chen et al., 2018a), tensor-program autotuners (Zheng et al., 2020; Chen et al., 2018b), DSLs such as Triton, and vendor libraries such as AITER (AMD ROCm, 2025), rocBLAS (AMD ROCm, 2016), and Composable Kernel (AMD ROCm, 2019) are highly effective, yet they also create a practical boundary: once a kernel has become an AMDGPU code object (AMD ROCm, 2023), the remaining optimization opportunities may no longer be exposed through the source program—and in many deployments the source is not available at all.

This source-free setting is common in deployed ML systems: serving stacks may expose only a compiled HSACO (AMD ROCm, 2023), a JIT cache artifact (Tillet et al., 2019), or a vendor binary such as an AITER code object or a Triton kernel embedded in vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024). Re-optimizing the deployed artifact targets the exact code that runs on the GPU, but it is brittle: a local instruction edit can corrupt memory, violate the kernarg ABI, desynchronize metadata, or appear faster only by changing semantics.

Recent LLM-based kernel systems (Wang et al., 2025; Zhang et al., 2025; Dai et al., 2026; Chen et al., 2026) generate and refine CUDA, Triton, HIP, or tensor-program kernels from high-level specifications, including KernelBench (Ouyang et al., 2025), assuming an editable source artifact and an independent reference implementation. Compiled code-object kernels provide neither: the original binary is both the deployed artifact and the only behavioral reference, scoped to a captured launch.

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.

This paper asks whether an already compiled GPU kernel can be optimized using the original binary as its behavioral reference. We present AsmEvo, a post-compilation optimizer that recovers a round-trip-faithful AMDGCN representation while preserving the code-object ABI. Kerncap (Ramos & Lowery, 2026) is used once to capture a production dispatch—launch geometry, kernarg, referenced memory and module state, and validation policy. From this immutable capture, AsmEvo generates a kernel-specific persistent evaluator that restores identical state, runs the original and candidate objects under the same launch contract, checks policy-aware output equivalence, and measures paired latency.

An LLM agent proposes profile-guided assembly edits, while a deterministic controller owns build routing, verification, commit decisions, and lineage. A candidate advances only after static ABI/resource checks, capture-scoped correctness, and independent paired validation with A/A calibration and confidence bounds. Verified lineage, multi-start search, composition, and stall-triggered redirection support long-horizon exploration, but the agent cannot declare correctness or speedup. The output is a lower-latency, drop-in-compatible code object for the captured launch contract.

On MI308X, AsmEvo improves all 30 selected KernelBench kernels ($1.41\times$ geometric-mean and $3.53\times$ maximum speedup). On MI300X production workloads, it improves all evaluated AITER binaries ($1.12\times$/$1.33\times$ geometric-mean/maximum) and all vLLM/SGLang Triton assembly kernels ($1.49\times$/$2.08\times$), while preserving functional equivalence. AsmEvo thus complements source-level compilers, JIT systems, autotuners, and production inference libraries.

Our contributions are as follows:

- We formulate **source-free, ABI-preserving GPU kernel optimization** as a correctness-gated post-compilation problem over AMDGPU code objects, where K_0 is the capture-scoped reference and every reported speedup must pass external verification.
- We build a **binary recovery, rebuild, and differential-evaluation pipeline**: round-trip recovery of reassemblable AMDGCN assembly, metadata-aware ABI-preserving rebuild, conservative patch fallback, real-dispatch acquisition with Kerncap (Ramos & Lowery, 2026), and a generated persistent harness that evaluates original and candidate code objects under the same restored launch contract.
- We instantiate a **long-horizon agentic search driver under external verification**: the agent plans architecture-specific edits, uses failed attempts as memory, explores diverse verified start points, composes compatible improvements, and recovers from stalls, while deterministic gates own correctness, screening, and formal commits; evaluations on KernelBench, AITER, and vLLM/SGLang Triton JIT kernels show consistent post-compilation gains.

2 Related Work

LLM Kernel Optimization and Tensor Compilers. KernelBench (Ouyang et al., 2025) established an execution-guided setting in which generated kernels are iteratively refined using correctness and performance feedback. Subsequent work explores reinforcement learning with measured rewards (Baronio et al., 2026; Li et al., 2026; Liu et al., 2026; Dai et al., 2026), agentic and evolutionary search over candidate implementations (Liao et al., 2026; Lange et al., 2025), and specialized kernel-generation models (Fisches et al., 2025), while related benchmarks (Li et al., 2025; Lange et al., 2025) show that performance optimization remains difficult and that insufficient validation can reward fast but semantically incorrect kernels. In parallel, tensor compilers and DSLs such as Halide, TVM, AutoTVM, Ansor, MetaSchedule, and Triton optimize schedules, tiling, memory movement, and code generation (Ragan-Kelley et al., 2013; Chen et al., 2018a;b; Zheng et al., 2020; Shao et al., 2022; Tillet et al., 2019). These approaches operate before or during compilation and assume editable source or IR together with a high-level correctness reference. AsmEvo targets the complementary post-compilation setting: its input is the emitted AMDGPU code object, and K_0 is the capture-scoped behavioral reference.

Assembly Optimization and Binary Rewriting. Superoptimizers search for faster programs at the instruction or IR level: STOKE (Schkufza et al., 2013) applies stochastic search to x86 assembly, Souper (Sasnauskas et al., 2017) discovers missing LLVM optimizations, and Al-

phaDev (Mankowitz et al., 2023) learns low-level sorting routines. Recent learning-based systems optimize assembly directly: Wei et al. (2025b) apply reinforcement learning to CPU assembly, while CuAsmRL (He & Yoneki, 2025) formulates NVIDIA SASS instruction scheduling as an assembly game and learns to reorder instructions for higher throughput. These methods assume CPU-oriented specifications or constrained SASS rescheduling. AsmEvo instead targets source-free AMDGPU code objects and jointly addresses assembly recovery, ABI- and metadata-preserving rebuild, broader agent-proposed edits, and capture-scoped evaluation against the original binary.

GPU binary frameworks primarily target instrumentation: NVBit (Villa et al., 2019) modifies NVIDIA SASS and Luthier (Raayai-Ardakani et al., 2025) operates on loaded ROCm code objects, while Kerncap (Ramos & Lowery, 2026) isolates AMD HIP and Triton kernels by capturing an HSA dispatch and a virtual-address-faithful device-memory snapshot for standalone replay. AsmEvo uses this capture once to generate a persistent evaluator, then produces faster, verified drop-in objects preserving launcher semantics, resource declarations, descriptors, and metadata.

Correctness of Optimized Code. Translation-validation systems such as Alive2 (Lopes et al., 2021) check whether compiler optimizations preserve semantics, and program-equivalence reasoning remains difficult even for strong language models (Wei et al., 2025a), motivating execution-based verification rather than model judgment. The source-level or framework references used by existing GPU optimizers are unavailable for HSACO-only artifacts, so AsmEvo compares candidates against K_0 in a capture-derived harness and checks equivalence before timing; its guarantees are empirical and limited to the captured launch and state.

Positioning of AsmEvo. To our knowledge, AsmEvo is the first system to combine agentic search, ABI- and metadata-preserving AMDGPU code-object rebuilding, and capture-scoped evaluation against the original deployed binary. It recovers HSACO-only artifacts and admits candidates only after drop-in ABI checks and equivalence to K_0 under a captured launch. AsmEvo complements rather than replaces source-level compilers, autotuners, and inference libraries.

3 METHOD

3.1 OVERVIEW

AsmEvo optimizes compiled AMDGPU code objects without source code or an independent reference implementation (Figure 1). It uses Kerncap once to acquire an immutable real-dispatch capture $\mathcal{C}$, then generates a persistent evaluator that restores the captured state and compares K_0 with each candidate under the same launch contract. An optimized object K' is admitted only after ABI-preserving rebuild, capture-scoped equivalence, and paired performance validation, all owned by a deterministic controller; a long-horizon LLM only proposes edits and replans from structured feedback. We denote recovered assembly by $s(K)$ and report final speedup as $\hat{s}(K) = T(K_0)/T(K)$; each acceptance round instead compares the candidate with the current historical best $K^\star$.

3.2 PROBLEM FORMULATION

Given K_0, AsmEvo seeks an optimized code object K' satisfying three requirements.

R1 (ABI preservation). K' preserves symbols, kernarg layout, launch semantics, and externally visible metadata, allowing the original host launcher to invoke it unchanged.

R2 (Functional equivalence). Let $\mathcal{C} = (\lambda, a, M_{\mathrm{pre}}, Y_0, \mathcal{P})$ contain the launch λ, raw kernarg a, pre-state M_{pre}, reference post-state Y_0, and per-region policies $\mathcal{P} = \{P_j\}$. The generated harness produces $Y(K;\mathcal{C})$ from the same restored state. AsmEvo requires:

$$\mathrm{eq}_{\mathcal{C}}(K') \equiv \bigwedge_j \mathrm{cmp}_{P_j}(Y_j(K';\mathcal{C}), Y_{0,j}) \wedge \iota(K';\mathcal{C}),$$

$$\mathrm{cmp}_{P_j}(y,\hat{y}) = \begin{cases} [\,y = \hat{y}\,], & P_j = \textsc{ByteExact}, \\ [\,\forall i:\ |y_i - \hat{y}_i| \le a_j + r_j|\hat{y}_i|\,], & P_j = \textsc{AllClose}(a_j, r_j), \\ [\,\cos(y,\hat{y}) \ge \theta_j\,], & P_j = \textsc{Cosine}(\theta_j). \end{cases} \tag{1}$$

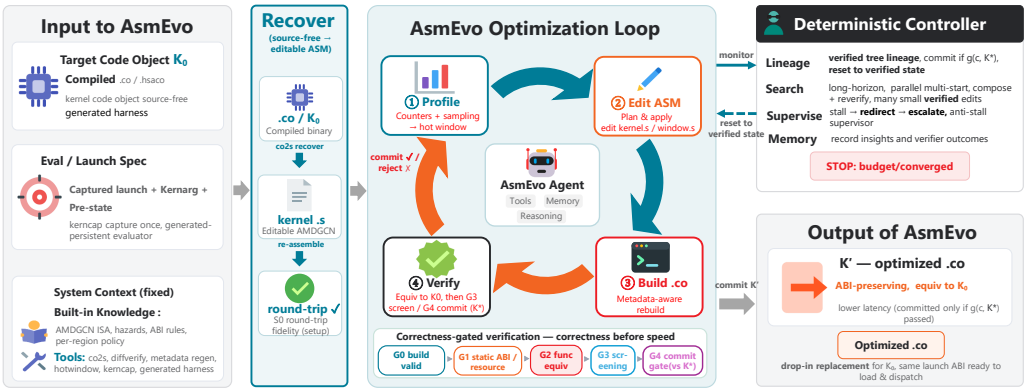


Figure 1: AsmEvo combines faithful recovery, ABI-preserving patch or rebuild, and correctness-before-performance evaluation in a persistent harness generated from a Kerncap capture. The LLM proposes edits through multi-start search, verified composition, and anti-stall redirection; only independently validated improvements enter the lineage.

Unspecified ranges are byte-exact; ι also requires opaque and untouched captured ranges to remain unchanged.

R3 (Latency improvement). K' reduces measured latency. Each acceptance round requires a verified gain over the current historical best $K^\star$; reported speedup is measured separately as $\hat{s}(K') = T(K_0)/T(K')$ against the original object.

Equivalence is empirical and scoped to $\mathcal{C}$ rather than unseen shapes or states.

3.3 Code-Object Recovery and Round-Trip Fidelity

AsmEvo reconstructs editable assembly $s(K_0)$ from the AMDGPU ELF container. Because raw disassembly omits the declaration layer, it recovers sections, symbols, notes, kernel descriptors, AMDGPU metadata, and the AMDGCN instruction body, reconstructs `.amdhsa_kernel` declarations, and re-symbolizes PC-relative control flow so that branches remain valid after instruction insertion, deletion, or reordering.

A one-time round-trip gate reassembles and relinks $s(K_0)$ and compares the regenerated instruction body, kernel descriptors, and metadata with K_0, masking only link-determined fields; passing it establishes that later edits operate on a faithful, rebuildable representation. Rare recovery repairs must pass the same byte-level round-trip check.

3.4 Real-Dispatch Acquisition and Generated Evaluation

Kerncap (Ramos & Lowery, 2026) intercepts one selected application dispatch and records $\mathcal{C}$: pointer-rich kernargs, launch geometry, referenced memory, module state, and validation policy. It thereby acquires a real input without a source-level test, but it is not part of the candidate-evaluation loop.

AsmEvo materializes from $\mathcal{C}$ a kernel-specific persistent evaluator that freezes the symbol, kernarg and launch geometry, memory mapping, module-state restoration, and policy. It restores M_{pre}, loads K_0 and a candidate under the same contract, and checks Eq. 1: exact virtual-address relationships keep nested pointers valid, while untouched captured ranges expose out-of-footprint writes. For timing, the process stays alive across observations and restores mutable state before each one. The agent may invoke this evaluator but cannot alter its capture or decision.

3.5 Metadata-Aware Rebuild

AsmEvo converts each edited assembly into a loadable code object by rescanning it for VGPR/SGPR/AGPR, LDS, and scratch usage, recomputing resource fields in both the kernel de-

scriptor and AMDGPU metadata, and relinking. The kernarg layout and segment size, symbol interface, and launch semantics remain frozen, preserving compatibility with the original launcher.

If rebuilding fails because an object contains unsupported or incompletely recovered constructs, AsmEvo may fall back to a conservative in-place byte patch, restricted to resource-neutral and size-non-increasing edits and retaining the original descriptors and metadata. Patched candidates still pass static consistency, capture-scoped equivalence, and the formal performance gate, so the fallback changes only the rebuild mechanism, not the acceptance criterion.

3.6 Profiling and Hot-Window Localization

Hardware counters, instruction sampling, and static analysis identify stall-dominant instruction windows in large kernels. The resulting profile summarizes instruction mix, dependency chains, memory behavior, occupancy constraints, and resource pressure for the search driver. Instead of placing the entire assembly in every optimization context, AsmEvo selects a hot window together with the surrounding data dependencies and relevant metadata, then splices the proposed local edit back into the complete assembly before whole-kernel rebuild and verification. Localization therefore reduces search context and discourages unrelated modifications without weakening the full-kernel acceptance criterion.

3.7 Gated Verification Harness

After one-time round-trip validation and harness generation, each candidate passes Algorithm 1: assembly, static ABI/resource checks, capture-scoped correctness, and inexpensive paired screening. Incorrect candidates are never timed, and screening only guides search—it cannot commit a candidate.

A promising candidate is rebuilt independently and enters a formal persistent session: warmup removes cold effects, captured state is restored before each observation, paired ABBA interleaving with reciprocal role placement reduces temporal and placement bias, and A/A blocks calibrate a session-specific margin. Per-GPU serialization and fail-closed health checks reject contaminated or unstable observations, and a nonparametric block bootstrap over paired speedups yields $\hat{r}$ and a percentile 95% confidence interval; the commit gate uses the lower bound (constants in Appendix B).

3.8 Long-Horizon Agentic Search

The verification harness accepts any candidate generator; AsmEvo instantiates it with a long-horizon LLM search driver. A deterministic controller owns the wall-clock budget, workspace state, rebuild and verification calls, commit decisions, candidate lineage, and termination conditions. The LLM analyzes profiling evidence, selects bottlenecks, proposes localized assembly transformations, interprets structured failures, and replans after unproductive attempts, but it cannot declare either correctness or performance.

Commit gate and verified lineage. Conditioned on a healthy, A/A-calibrated formal session, candidate c is committed only if it beats the current historical best $p = K^\star$:

$$\begin{aligned} g(c,p) \equiv \mathrm{eq}_{\mathcal{C}}(c) \wedge \mathrm{MajorityPositive}(c,p) \\ \wedge\, L(c,p) > 1 + m(c,p) \wedge \hat{r}(c,p) > s_{\mathrm{floor}}, \end{aligned} \tag{2}$$

where $\hat{r}(c,p)$ is the log-median of placement-corrected paired-block speedups, $L(c,p)$ is the lower endpoint of its percentile bootstrap 95% CI, $\mathrm{MajorityPositive}$ requires at least 80% of those blocks to be faster, $s_{\mathrm{floor}} > 1$, and $m(c,p) = \max(m_{\mathrm{sig}}, m_{\mathrm{abs}}, m_{\mathrm{slot}})$ (Appendix B). Screening may compare a proposal with its edit parent; acceptance does not. The confidence bound, not a point estimate, advances search. After commit, $K^\star \leftarrow c$ and $\hat{s}(c) = \hat{s}(p)\hat{r}(c,p)$; reported final speedup is remeasured directly against K_0.

Accepted candidates form a tree-structured lineage rather than a single destructive chain. Each node records its parent, parent-relative verified speedup, changed instruction windows, resource usage, and optimization rationale. Workers may therefore branch from the current best or return to an earlier verified version when a direction reaches a local optimum, but a commit still requires $g(c, K^\star)$, so the reported global best never regresses.

Multi-start exploration. In single mode, the driver proposes one edit from the current best. In team mode, a planner assigns orthogonal optimization directions—such as latency hiding, dependency reduction, register-pressure control, memory-access restructuring, or instruction simplification—to parallel workers. Start points come from diverse verified lineage nodes rather than a single shared candidate, and every proposal is independently rebuilt and passed through GATEDEVAL.

Algorithm 1 GATEDEVAL: correctness and screening.

Require: edited assembly s'; generated harness $H_{\mathcal{C}}$; parent p

if $\neg$ASSEMBLES(s') **then**
  return $\langle$ASM_INVALID$\rangle$
end if
$c \leftarrow$ BUILD(s')
if $\neg$STATICCONSISTENT(c) **then**
  return $\langle$ABI_INVALID$\rangle$
end if
if $\neg\, \mathrm{eq}_{\mathcal{C}}(c)$ **then**
  return $\langle$DIVERGENT$\rangle$
end if
$\hat{r}_{\mathrm{screen}} \leftarrow$ PAIREDSCREEN$(H_{\mathcal{C}}, p, c)$
return $\langle$SCREENED$, \hat{r}_{\mathrm{screen}}\rangle$

Verified composition. Individually successful edits are not assumed to remain correct or beneficial when combined. The integrator takes candidates that share a base and modify disjoint or complementary instruction regions, applies their edits incrementally, and rebuilds and verifies after each step. A composition is forwarded only if it passes all correctness gates and improves over both parents; it is committed only if it also beats $K^{\star}$. Otherwise the best individual candidate is retained.

Memory and anti-stall supervision. A bounded run memory stores successful optimization insights, structured gate outcomes, failure signatures, and hashes of repeated candidates. On prolonged no-improvement, repeated proposals, or infrastructure failures, the controller requests a redirect that summarizes attempted directions and marks exhausted hypotheses; the driver then changes the targeted bottleneck or backtracks to an earlier verified node, and persistent stalls escalate to wider multi-start exploration. Redirected candidates remain subject to the same deterministic gates.

Algorithm 2 in Appendix A summarizes the complete loop; the controller resets each worker to a verified state after evaluation, preventing unverified modifications from accumulating across iterations.

3.9 SCOPE AND LIMITATIONS

AsmEvo provides empirical differential equivalence rather than formal verification; its guarantee is limited to the captured launch and state. Repeated execution of $\mathcal{C}$ may expose nondeterminism but does not broaden shape coverage, and additional shapes require a new capture and generated harness. The harness is architecture-agnostic, but the current recovery and metadata-aware rebuild implementation is calibrated for CDNA-class AMD GPUs, and achievable gains depend on the headroom remaining in the compiled kernel.

4 EXPERIMENTS

We evaluate whether AsmEvo (1) improves compiled benchmark kernels, (2) transfers to source-free production artifacts, and (3) benefits from its search-driver components under a shared capture-derived evaluator, using KernelBench Level 1 and Level 2, AITER code objects, and Triton JIT HSACOs emitted by vLLM and SGLang. The unmodified binary is always the $1.00\times$ reference.

4.1 EXPERIMENTAL SETUP

KernelBench runs on MI308X (ROCm 7.2.4) and production AITER and vLLM/SGLang objects on MI300X (ROCm 7.2.0); both stacks are in Appendix B.1. Unless noted, AsmEvo uses Claude Opus 4.8, four GPU workers, four team directions, diverse lineage starts, and a 12-hour budget per kernel. Early stopping is disabled, so the controller spends the full budget unless a safety or infrastructure condition intervenes.

Table 1: Kernel sources, counts, and operator coverage.

Source	Size	Coverage
KernelBench L1	15	activations, losses, reductions/norms, depthwise conv, matmul, pooling
KernelBench L2	15	fused GEMM/matmul and conv pipelines, normalization, activations, pooling, reductions
AITER	3	code-object FP8 MoE, BF16 FMHA, BF16 FMHA backward
vLLM/SGLang	5	Triton JIT A16W16 GEMM, A8W8 GEMM, GPTQ/AWQ MoE, MoE, unified attention

For every kernel, K_0 is immutable. Setup performs round-trip recovery and instantiates a generated-persistent evaluator from one real-dispatch capture that fixes the symbol, launch, kernarg, memory and module state, and per-region policy; all candidates for that kernel share this capture identity. Evaluation is fail-closed: build and static checks, capture-scoped correctness, paired screening that cannot commit, then independent formal validation with paired ABBA, A/A calibration, health checks, and a bootstrap confidence interval (Section 3.7). Reported speedups are direct original-versus-final measurements against K_0, and protocol constants and generated-persistent margin definitions appear in Appendix B. Production per-region comparators are in Table 8.

Full is the team configuration above. Single is a single-path bundle: one GPU, one direction, the current best as start, and one-worker rescue. No Sup. keeps the team operator but disables stall-triggered re-planning, backtracking, and supervisor escalation. Opus 5 changes only the search model. Code is an external raw-binary Claude Code baseline that receives a prompt, the original HSACO, and the capture, generates its own search-time harness, and is checked post hoc with the common evaluator, without AsmEvo recovery, lineage, supervisor, or commit gates.

4.2 Artifact Sources and Coverage

Table 1 lists the four sources, covering the operator types most common in large language models: 15 validated KernelBench Level 1 operators, 15 Level 2 operators, and 8 production kernels (3 AITER code objects and 5 vLLM/SGLang Triton JIT HSACOs).

The sources differ in how the input binary is obtained. For KernelBench, we use an AMD GEAK-style agent (Wang et al., 2025) with GPT-5.0 to refine HIP versions of the PyTorch benchmark kernels for five rounds, compile them, and give AsmEvo only the compiled object as K_0; the selected Level 1 and Level 2 HIP inputs are all faster than their PyTorch operators in the source-level benchmark, so K_0 is not a weak baseline. The 30 kernels are exactly this HIP-faster-than-PyTorch subset (Appendix C); reported speedups compare the optimized object with that compiled HIP binary, not with the KernelBench PyTorch baseline. Production K_0 are unmodified deployed binaries: three AITER library objects (AMD ROCm, 2025) and five vLLM/SGLang Triton JIT HSACOs from the AgentKernelArena GEAK head-kernel baselines (Younesian et al., 2026) (Appendix B.2). Every binary is verified by a generated persistent harness instantiated from a Kerncap (Ramos & Lowery, 2026) real-dispatch capture, so evaluation uses the application's actual shapes, dtypes, and device state rather than synthesized arguments. Per-operator HIP/PyTorch measurements, KernelBench selection, search constants, per-kernel AsmEvo results, capture details, and case studies appear in Appendices B–F.

4.3 Metrics

For an optimized object K', verified speedup is $\hat{s}(K') = T(K_0)/T(K')$, where $T(\cdot)$ is the formal paired-session latency estimate from a direct original-versus-final measurement. A kernel counts as improved only if it passes capture-scoped correctness and the commit gate in Eq. 2; geometric means include all evaluated kernels in a family, assigning $1.00\times$ when there is no accepted gain. HIP/PyTorch ratios in Appendix C describe input quality, not AsmEvo speedups.

4.4 RESULTS

4.4.1 KERNELBENCH KERNELS

Table 2 reports the default AsmEvo results on MI308X. AsmEvo improves all 15 Level 1 kernels ($1.62\times$ geometric mean, $3.53\times$ maximum on FrobeniusNorm) and all 15 Level 2 fused-operator kernels ($1.23\times$ geometric mean, $1.94\times$ maximum on Matmul+Swish+Sum+GroupNorm), for a $1.41\times$ geometric mean over all 30 selected kernels. Figure 2 ranks these verified speedups and marks the operator class of each kernel.

Table 2: KernelBench performance under the default AsmEvo configuration. Speedups are measured against the original compiled code object after capture-scoped equivalence and the formal paired gate.

Split	Kernels	Improved	Geo.	Max	GPU	Best case
KernelBench L1	15	15 (100%)	1.62×	3.53×	MI308X	FrobeniusNorm
KernelBench L2	15	15 (100%)	1.23×	1.94×	MI308X	Matmul+Swish+Sum+GroupNorm
All KernelBench	30	30 (100%)	1.41×	3.53×	MI308X	FrobeniusNorm

The two splits exhibit different improvement profiles. Level 1 has a median speedup of $1.49\times$, and seven kernels exceed $1.50\times$, spanning depthwise convolution, normalization, activation, pooling, and reduction operators: besides the $3.53\times$ FrobeniusNorm result, MaxPool2D reaches $3.28\times$, RMSNorm $2.68\times$, and depthwise Conv2D with asymmetric input and a square kernel $2.19\times$. This operator diversity indicates that the gains are not confined to one instruction pattern or workload class.

Level 2 is more concentrated near modest gains: its median is $1.19\times$ and only two fused kernels exceed $1.50\times$. The next-largest gains after Matmul+Swish+Sum+GroupNorm are $1.52\times$ on ConvTranspose2d+Softmax+BiasAdd+Scaling+Sigmoid and $1.33\times$ on GEMM+GroupNorm+Swish+Multiply+Swish. The narrower distribution is consistent with fused kernels exposing less local post-compilation headroom, while the $1.94\times$ best case shows that substantial opportunities can still remain in individual fused artifacts.

4.4.2 PRODUCTION CODE OBJECTS

Table 3 reports production kernels individually rather than as a single aggregate, because production binaries differ in launch semantics and capture contracts. On MI300X, AsmEvo improves all three AITER code objects ($1.12\times$ geometric mean, $1.33\times$ maximum on the FP8 blockscale MoE kernel) and all five Triton JIT HSACOs from vLLM and SGLang ($1.49\times$ geometric mean, $2.08\times$ maximum on the GPTQ/AWQ fused MoE kernel), for a $1.34\times$ geometric mean over all eight production kernels. The same post-compilation harness therefore applies beyond benchmark-generated kernels.

Table 3: Verified per-kernel speedups over the original production binaries on MI300X, covering binary-level AITER objects and vLLM/SGLang Triton JIT assembly evaluated with the capture-derived persistent harness. Family geometric means are $1.12\times$ (AITER) and $1.49\times$ (Triton JIT), and $1.34\times$ over all eight kernels.

Source	Kernel	Speedup
AITER	aiter_fmoe_fp8_blockscale_subgu256	1.33×
AITER	aiter_fmha_bwd_hd128_bf16_causal_a32	1.05×
AITER	aiter_fmha_hd192x128_bf16_causal	1.03×
vLLM/SGLang	fused_moe_kernel_gptq_awq	2.08×
vLLM/SGLang	kernel_unified_attention_2d	1.96×
vLLM/SGLang	_gemm_a8w8_blockscale_kernel	1.29×
vLLM/SGLang	_gemm_a16_w16_kernel	1.25×
vLLM/SGLang	fused_moe_kernel	1.10×

Headroom also depends strongly on the upstream artifact: the three AITER binaries contain one large gain and two comparatively small attention-kernel gains, whereas two of the five vLLM/SGLang Triton results exceed $1.90\times$. Because these kernels implement different operators

and launch configurations, the comparison does not isolate the upstream compiler as the cause; it shows that AsmEvo recovers useful opportunities from both vendor code objects and runtime-generated HSACOs. We report only assembly-level optimization for Triton JIT kernels and binary-level optimization for AITER; IR-level variants are excluded from all aggregates and claims.

4.4.3 Ablation Study

Table 4 is a search-driver comparison under a common post-hoc evaluator, not a single-factor causal ablation. Existing source-level kernel agents lack recovery, ABI-preserving rebuild, and capture-derived gates for HSACO-only artifacts, so the relevant baseline is search variation inside this harness—except Code, which searches outside it.

Table 4: Production-kernel search-driver comparison over the eight kernels in Table 3. All variants use the same originals, captures, generated-persistent evaluator, and formal gate (Section 4.1); Single and No Sup. are bundled search-policy changes rather than single-factor ablations, and Code is an external raw-binary Claude Code baseline evaluated post hoc. Geometric means use all eight kernels.

Kernel	Full	Single	No Sup.	Opus 5	Code
aiter_fmoe_fp8_blockscale_subgu256	1.33×	1.28×	1.31×	1.32×	1.00×
aiter_fmha_bwd_hd128_bf16_causal_a32	1.05×	1.00×	1.05×	1.09×	1.00×
aiter_fmha_hd192x128_bf16_causal	1.03×	1.00×	1.00×	1.03×	1.00×
fused_moe_kernel_gptq_awq	2.08×	2.09×	1.51×	2.48×	1.00×
kernel_unified_attention_2d	1.96×	1.05×	1.82×	1.18×	1.00×
_gemm_a8w8_blockscale_kernel	1.29×	1.20×	1.24×	1.39×	1.18×
_gemm_a16_w16_kernel	1.25×	1.03×	1.31×	1.34×	1.24×
fused_moe_kernel	1.10×	1.09×	1.05×	1.52×	1.00×
Geo. speedup	1.34×	1.18×	1.26×	1.37×	1.05×
Improved rate	8/8	6/8	7/8	8/8	2/8

Single jointly removes team fan-out and diverse starts: it retains the GPTQ/AWQ MoE gain but improves only six of eight kernels, dropping the geometric mean from $1.34\times$ to $1.18\times$, with unified attention falling from $1.96\times$ to $1.05\times$. No Sup. removes the stall-rescue path, lowering the geometric mean to $1.26\times$ and losing the forward FMHA gain. Opus 5 keeps the AsmEvo evaluator and raises the geometric mean to $1.37\times$, its largest additional gain being GPTQ/AWQ MoE ($2.48\times$). Code, given a prompt, the compiled binary, and the capture, improves only A16W16 GEMM ($1.24\times$) and A8W8 GEMM ($1.18\times$) and does not yield a verified gain on the remaining six under the post-hoc formal gate, for a geometric mean of $1.05\times$. Per-kernel narratives for the largest KernelBench, vLLM/SGLang Triton, and AITER gains, together with drop-in dispatch validation, appear in Appendix F.

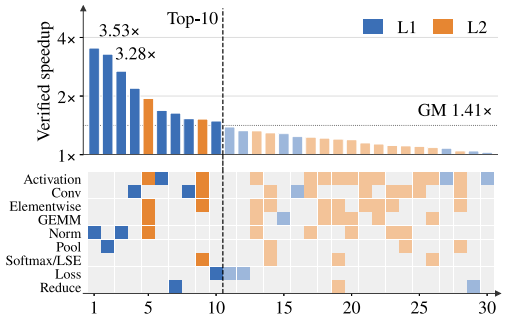


Figure 2: Verified AsmEvo speedups over the original code objects ($1.00\times$) for the 30 selected KernelBench kernels, ranked by speedup, with Level 1 and Level 2 marked separately.

5 Conclusion

AsmEvo extends GPU kernel optimization to source-free AMDGPU code objects, using a capture-derived persistent evaluator against K_0 and admitting candidates only after ABI-preserving rebuild and capture-scoped paired validation. It improves all 30 selected KernelBench kernels ($1.41\times$ geometric mean, $3.53\times$ maximum) and all eight production kernels from AITER, vLLM, and SGLang ($1.34\times$ geometric mean, $2.08\times$ maximum), showing that deployed binaries retain recoverable low-level headroom that agentic search can exploit under the deterministic gates of Section 3.9.

AI USE STATEMENT

In this work, we used generative AI for research ideation and execution, to retrieve and identify related work, to draft sections of the manuscript, and to edit the language and organization of the paper. For execution, Claude Opus 4.8 was the default AsmEvo search model, Claude Opus 5 and Claude Code with Claude Opus 4.8 were alternative search drivers in the ablation experiments, and GPT-5.0 was used in a GEAK-style agent to generate the KernelBench input implementations. We did not use generative AI to prove mathematical claims, generate or edit figures, or determine kernel correctness or performance; correctness and performance were established exclusively through deterministic verification and hardware measurements. We reviewed all AI-assisted content and take responsibility for the final text, claims, measurements, and artifacts.

REPRODUCIBILITY STATEMENT

Search and evaluation constants, the generated-persistent margin, and the bootstrap procedure are in Appendix B. KernelBench selection and HIP/PyTorch input checks are in Appendix C. Capture and harness construction are in Appendix E. Each reported speedup is one 12-hour search per kernel; the paper does not report variation across search seeds.

REFERENCES

AMD ROCm. rocBLAS: ROCm basic linear algebra subprograms library. `https://github.com/ROCm/rocBLAS`, 2016. Accessed: 2026-07-28.

AMD ROCm. Composable Kernel: Performance portable programming model for machine learning tensor operators. `https://github.com/ROCm/composable_kernel`, 2019. Accessed: 2026-07-28.

AMD ROCm. User Guide for AMDGPU Backend. `https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/AMDGPUUsage.html`, 2023. Accessed: 2026-07-28.

AMD ROCm. AITER: AI tensor engine for ROCm. `https://github.com/ROCm/aiter`, 2025. Accessed: 2026-07-28.

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

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, 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, and Humphrey Shi. AVO: Agentic variation operators for autonomous evolutionary search. *arXiv preprint arXiv:2603.24517*, 2026.

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

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS 2018*, volume 31, 2018b.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA Agent: Large-scale agentic RL for high-performance CUDA kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. KernelLLM: Making kernel development more accessible. `https://huggingface.co/facebook/KernelLLM`, 2025.

Guoliang He and Eiko Yoneki. CuAsmRL: Optimizing GPU SASS schedules via deep reinforcement learning. In *CGO 2025*, pp. 493–506, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *SOSP 2023*, pp. 611–626, 2023.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of ACL 2025*, pp. 23053–23066, 2025.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. In *ICLR 2026*, 2026.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Ruichao Xiao, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Roman Levenstein, Dianshi Li, Liyuan Li, Ajit Mathews, Varna Puvvada, Feng Shi, Nathan Yan, Xiayu Yu, Uladzimir Pashkevich, Matt Steiner, Carole-Jean Wu, and Gaoxiang Liu. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at Meta. In *ISCA 2026*, pp. 733–747, 2026.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. Kernel: Reinforcement learning done right for Triton kernel generations. In *ICML 2026*, 2026.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for LLVM. In *PLDI 2021*, pp. 65–79, 2021.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257–263, 2023.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *ICML 2025*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415, 2025.

Matin Raayai-Ardakani, Andrew Nguyen, Ivan Rosales, Daoxuan Xu, Yuwei Sun, Yifan Sun, David Kaeli, and Norman Rubin. Luthier: A dynamic binary instrumentation framework targeting AMD GPUs. In *ISPASS 2025*, pp. 137–149, 2025.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI 2013*, pp. 519–530, 2013.

Cole Ramos and Keith Lowery. Kerncap: Automated kernel extraction and isolation for AMD GPUs. *arXiv preprint arXiv:2605.03208*, 2026.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS 2013*, pp. 305–316, 2013.

Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In *NeurIPS 2022*, volume 35, pp. 35783–35796, 2022.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *ACM SIGPLAN MAPL 2019*, pp. 10–19, 2019.

Oreste Villa, Mark Stephenson, David W. Nellans, and Stephen W. Keckler. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *MICRO 2019*, pp. 372–383, 2019.

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

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago S. F. X. Teixeira, Diyi Yang, Ke Wang, and Alex Aiken. EquiBench: Benchmarking large language models' reasoning about program semantics via equivalence checking. In *EMNLP 2025*, 2025a.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. Improving assembly code performance with large language models via reinforcement learning. *arXiv preprint arXiv:2505.11480*, 2025b.

Sharareh Younesian, Wenwen Ouyang, Sina Rafati, Mehdi Rezagholizadeh, Sharon Zhou, Ji Liu, Yue Liu, Yuchen Yang, Hao Li, Ziqiong Liu, Dong Li, Vikram Appia, Zhenyu Gu, and Emad Barsoum. AgentKernelArena: Generalization-aware benchmarking of GPU kernel optimization agents. *arXiv preprint arXiv:2605.16819*, 2026.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI 2020*, pp. 863–879, 2020.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *NeurIPS 2024*, volume 37, pp. 62557–62583, 2024.

Appendix

This appendix collects material omitted from the main paper. Appendix A gives the long-horizon search loop. Appendix B records search and evaluation constants, the generated-persistent margin and bootstrap, the MI308X and MI300X nodes and software stacks, production-kernel sources, and per-kernel validation policies. Appendix C describes how the 30 KernelBench operators were selected and reports source-level HIP/PyTorch input quality. Appendix D lists per-operator AsmEvo speedups. Appendix E details real-dispatch capture and generated-persistent evaluation. Appendix F provides per-kernel narratives and drop-in dispatch checks.

A Long-Horizon Verified Search Loop

Algorithm 2 gives the complete search loop summarized in Section 3.8. The controller owns the wall-clock budget, start-point selection, rebuild and verification calls, commit decisions, and lineage updates. Screening proceeds through GatedEval (Algorithm 1); a commit additionally requires the formal gate $g(\tilde{c}, K^\star)$ in Eq. 2 against the current historical best.

Algorithm 2 AsmEvo long-horizon verified search.

Require: original object K_0; capture $\mathcal{C}$; budget B; direction count M; GPUs G

assert RoundTrip$(s(K_0), K_0)$
$H_{\mathcal{C}} \leftarrow$ GenerateHarness$(K_0, \mathcal{C})$
$K^\star \leftarrow K_0$; initialize lineage $\mathcal{L}$ and memory $\mathcal{M}$
while elapsed time $< B$ **do**
  $D \leftarrow$ Plan$(K^\star, \text{profile}, \mathcal{M}, M)$
  $P \leftarrow$ StartPoints$(\mathcal{L}, D)$; $C \leftarrow \emptyset$
  for each $(d, b) \in P$ assigned to GPU $g \in G$ **do**
    $s' \leftarrow$ Edit$(s(b), d, \mathcal{M})$
    $r \leftarrow$ GatedEval$(s', H_{\mathcal{C}}, b)$
    if $r = \langle$screened$, \hat{r}\rangle$ **then**
      $C \leftarrow C \cup \{(s', b, \hat{r})\}$
    end if
  end for
  $\tilde{c} \leftarrow$ Integrate(C)
  if $\tilde{c} \neq \emptyset$ **then**
    $r^\star \leftarrow$ DirectorPaired$(H_{\mathcal{C}}, K^\star, \tilde{c})$
    if $r^\star$ healthy $\wedge\ g(\tilde{c}, K^\star)$ **then**
      $K^\star \leftarrow \tilde{c}$; append to $\mathcal{L}$; re-profile
    end if
  end if
  $\mathcal{M} \leftarrow$ UpdateMemory$(\mathcal{M}, C)$
  if stalled **then**
    RedirectOrEscalate$(\mathcal{L}, \mathcal{M})$
  end if
  reset worker workspaces to verified $K^\star$
end while
return $K^\star$ and lineage $\mathcal{L}$

B Experiment Configuration

We distinguish two speedup definitions. Source-level input quality is measured as

$$q_{\text{HIP}} = \frac{T(\text{PyTorch})}{T(\text{generated HIP})},$$

whereas AsmEvo's post-compilation speedup is $\hat{s}(K') = T(K_0)/T(K')$. Here K_0 is the compiled object supplied to AsmEvo and K' is the final verified optimized code object. Thus the HIP/PyTorch values in Appendix C qualify the input baseline and are not AsmEvo speedups.

KernelBench and production kernels share the generated-persistent backend. From each Kerncap capture the harness materializes a private replay view, restores captured memory and module variables, and loads reference and candidate objects in one persistent process. Correctness uses Eq. 1. Timing uses HIP-event launch-to-completion measurement under reciprocal-role-swap paired ABBA. Formal acceptance uses Eq. 2; Table 5 lists the constants. A non-formal fallback retains $k = 0.85$ and controller epsilon 0.001, but it is not the commit authority for the reported runs. Kerncap's legacy HSA replay path remains an infrastructure fallback and is not the timing backend for the numbers in this paper; we therefore do not use the legacy A/A-interval half-width as $m(c, p)$.

Paired-block estimate. Each formal block runs both module placements. Writing $r_{f,b}$ and $r_{r,b}$ for the forward and reverse speedups, the placement-corrected block ratio and the slot bias are $r_b = \sqrt{r_{f,b}/r_{r,b}}$ and $b_b = \sqrt{r_{f,b}\, r_{r,b}}$. The point estimate is the log-median $\hat{r}(c, p) = \exp(\mathrm{median}_b \log r_b)$. With B the number of paired blocks, MajorityPositive is $q_+(c, p) = |\{b : r_b > 1\}|/B \geq 0.8$, and s_{floor} requires $\hat{r}(c, p) > 1$.

Bootstrap interval. We bootstrap the B block ratios in log-speedup space: each of $N_{\mathrm{boot}} = 2000$ deterministic resamples draws the observed blocks with replacement and computes $\exp(\mathrm{median}(\log r))$, and the 2.5th and 97.5th percentiles of the sorted samples (indices $\lfloor 0.025(N_{\mathrm{boot}} - 1) \rfloor$ and $\lfloor 0.975(N_{\mathrm{boot}} - 1) \rfloor$) form the 95% interval. $L(c, p)$ is the lower endpoint. A/B and A/A bootstraps use PRNG seeds 0 and 1. No parametric normality assumption is used.

Generated-persistent margin. After orienting alternating ABBA/BAAB blocks into one direction, $b_{\mathrm{AA}} = \exp(\mathrm{median} \log r^{\mathrm{AA}})$ is the log-median of the oriented A/A speedups and $b_{\mathrm{AB}} = \exp(\mathrm{median} \log b_b)$ is the log-median of the oriented A/B slot biases. Define $d_{\mathrm{slot}} = |b_{\mathrm{AB}}/b_{\mathrm{AA}} - 1|$ and charge $m_{\mathrm{slot}} = d_{\mathrm{slot}}$ only when $d_{\mathrm{slot}} > 0.003$ (otherwise 0). With baseline median latency t_0, the absolute timing floor is $m_{\mathrm{abs}} = (0.2\,\mu\mathrm{s})/t_0$. The formal margin is $m(c, p) = \max(m_{\mathrm{sig}}, m_{\mathrm{abs}}, m_{\mathrm{slot}})$, with $m_{\mathrm{sig}} = 0$ in reported runs. A/A calibration additionally requires $d_{\mathrm{slot}} \leq 0.10$ and a coefficient of variation of the oriented A/A speedups of at most 0.02; the A/A bootstrap interval is diagnostic on this path and does not set m. The commit gate uses the strict test $L(c, p) > 1 + m(c, p)$. The fallback constants k and the controller epsilon do not appear in Eq. 2.

Team search plans $M = 4$ directions per round. A redirect is triggered after five consecutive non-improving steps or three repeated identical candidate hashes. Three unsuccessful supervisor redirect cycles trigger diversified escalation.

B.1 Evaluation Nodes and Software Stacks

KernelBench experiments run on the MI308X node; production AITER binaries and Triton JIT HSACOs are recovered, rebuilt, and timed on the MI300X node. Table 6 gives both nodes and Table 7 both container stacks.

B.2 Production Kernel Sources

The three AITER kernels in Table 3 are compiled code objects from the AITER library (AMD ROCm, 2025) (container package 0.1.12.post1). The five Triton JIT kernels originate in vLLM and SGLang (Kwon et al., 2023; Zheng et al., 2024) and are taken from the AgentKernelArena (Younesian et al., 2026) `geak_benchmark` head-kernel baselines (`https://github.com/AMD-AGI/AgentKernelArena/tree/geak_benchmark/head_kernels/tasks/campaign20/baseline`): `fused_moe_kernel_gptq_awq`, `kernel_unified_attention_2d`, `_gemm_a16_w16_kernel`, `_gemm_a8w8_blockscale_kernel`, and `fused_moe_kernel`. AsmEvo receives only the compiled HSACO as K_0.

Production validation policies. Table 8 records the per-region comparators used in Eq. 1 for the eight production captures. Absent an explicit range, every captured output region is byte-exact. The two tolerance policies apply to one typed range each and remain byte-exact outside that range.

Table 5: Search and evaluation constants. Formal acceptance uses the generated-persistent margin and percentile bootstrap in this section; the fallback noise multiplier applies only when calibrated formal evidence is unavailable.

Setting	Reported ASM value
Search budget	12 hours per kernel
Team width	4 workers / 4 directions
Start policy	diverse verified lineage nodes
Early stop	disabled
Performance backend	generated-persistent
Screening repetitions	2
Independent verification repetitions	7
Formal paired blocks	at least 5
A/A calibration blocks	at least 3
Launches per observation	300
Warmup launches per module	20
Within-observation estimator	5%-trimmed mean
Block-bootstrap resamples	2000 (A/B seed 0, A/A seed 1)
Percentile CI	2.5th / 97.5th of sorted resamples
Slot-bias deadband / max drift	0.003 / 0.10
Max A/A coefficient of variation	0.02
Absolute timing floor	$0.2\,\mu$s over baseline median
Fixed significance floor m_{sig}	0
Formal margin $m(c,p)$	$\max(m_{\mathrm{sig}}, m_{\mathrm{abs}}, m_{\mathrm{slot}})$
Positive-block requirement $q_{\min}$	at least 0.8
Anti-lazy floor s_{floor}	strictly greater than $1.00\times$
Commit test on $L(c,p)$	$L(c,p) > 1 + m(c,p)$
Fallback noise multiplier k	0.85
Controller epsilon (fallback only)	0.001

Table 6: Evaluation nodes. "–" marks an item not recorded for that node.

Item	MI308X (KernelBench)	MI300X (production)
GPU	AMD Instinct MI308X (gfx942)	AMD Instinct MI300X (gfx942)
GPUs / node	8	8
Compute units / GPU	80	304
HBM / GPU	–	≈192 GiB
CPU	Intel Xeon Platinum 8480C (112 cores)	2× Intel Xeon Platinum 8480C (224 logical CPUs)
Host OS	Red Hat Enterprise Linux 8.6	Ubuntu 22.04.5 LTS
GPU driver	6.12.12	6.16.13
Container image	`rocm/atom:latest`	`lmsysorg/sglang:v0.5.11-rocm720-mi30x`

C KernelBench Selection and Input Quality

We select 15 KernelBench Level 1 operators and 15 Level 2 fused pipelines that cover the classes most common in large-language-model serving: activation and loss, reduction and normalization, pooling, convolution (including depthwise), matmul/GEMM, and fused GEMM–normalization–activation or convolution pipelines (Table 1). Within each class we keep operators for which an AMD GEAK-style agent (Wang et al., 2025) with GPT-5.0 can produce a compilable HIP implementation after five refinement rounds, and we discard candidates whose compiled HIP object is not faster than the corresponding PyTorch operator. AsmEvo then receives only that compiled object as K_0. This filter avoids evaluating post-compilation search on deliberately weak source-level inputs; the HIP/PyTorch ratios below are input-quality checks, not AsmEvo speedups.

C.1 Level 1 HIP/PyTorch Summary

All 15 selected Level 1 HIP implementations outperform their PyTorch baselines (Table 9). Their geometric-mean speedup is $1.65\times$, with a minimum of $1.02\times$ and a maximum of $4.95\times$. The arithmetic mean is $1.81\times$, the median is $1.42\times$, and all 15 kernels exceed $1.00\times$.

Table 7: ROCm, compiler, and Python packages on the two evaluation containers. "–" marks a package not recorded in that container manifest.

Item	MI308X (KernelBench)	MI300X (production)
Container OS	Ubuntu 24.04.4 LTS	Ubuntu 22.04.5 LTS
glibc / gcc	2.39 / 13.3.0	2.35 / 11.4.0
Python	3.12.3	3.10.12
ROCm	7.2.4	7.2.0
HIP runtime / hipcc	7.2.53211	7.2.26015
Device compiler	AMD clang 22.0.0git	AMD clang 22.0.0git
Assembler / linker	AMD LLVM `llvm-mc` / LLD 22.0.0git	AMD LLVM `llvm-mc` / LLD 22.0.0git
PyTorch	2.10.0+rocm7.2.4	2.9.1+rocm7.2.0
Triton	3.6.0	3.6.0
SGLang	–	0.5.11
AITER	–	0.1.12.post1
rocprofiler-sdk	–	1.1.0
kerncap	–	0.1.0
NumPy / SciPy / pandas	2.4.6 / 1.17.1 / 3.0.3	2.2.6 / 1.15.3 / 2.3.3
transformers	5.2.0	5.6.0
Claude Code	2.1.170	2.1.202

Table 8: Per-region validation policies for the eight production kernels. ByteExact applies to every captured output region unless a single AllClose or Cosine range is listed; that range remains byte-exact outside.

Kernel	Policy
`_gemm_a16_w16_kernel`	ByteExact, all outputs
`aiter_fmha_bwd_hd128_bf16_causal_a32`	AllClose $a_j{=}r_j{=}10^{-3}$ on float32 `dQ` (offset 6,291,456 B, length 6,291,456 B)
`aiter_fmha_hd192x128_bf16_causal`	ByteExact, all outputs
`aiter_fmoe_fp8_blockscale_subgu256`	Cosine $\theta_j{=}0.9999$ on BF16 `O` (offset 1,376,256 B, length 458,752 B)
`fused_moe_kernel_gptq_awq`	ByteExact, all outputs
`fused_moe_kernel`	ByteExact, all outputs
`_gemm_a8w8_blockscale_kernel`	ByteExact, all outputs
`kernel_unified_attention_2d`	ByteExact, all outputs

Table 9: Per-operator source-level HIP/PyTorch speedups for the selected KernelBench Level 1 inputs, sorted by speedup.

Kernel	HIP/PyTorch
88_MinGPTNewGelu	4.95×
99_TripletMarginLoss	2.96×
30_Softsign	2.27×
82_conv_depthwise_2D_square_input_square_kernel	1.87×
98_KLDivLoss	1.87×
42_Max_Pooling_2D	1.78×
14_Matmul_for_upper_triangular_matrices	1.50×
36_RMSNorm_	1.42×
83_conv_depthwise_2D_square_input_asymmetric_kernel	1.41×
95_CrossEntropyLoss	1.28×
29_Softplus	1.26×
48_Mean_reduction_over_a_dimension	1.23×
37_FrobeniusNorm_	1.22×
47_Sum_reduction_over_a_dimension	1.17×
84_conv_depthwise_2D_asymmetric_input_square_kernel	1.02×

C.2 Level 2 HIP/PyTorch Summary

All 15 selected Level 2 HIP implementations also outperform their PyTorch baselines (Table 10). Their geometric-mean speedup is 1.19×, with a minimum of 1.01× and a maximum of 2.93×. The distribution is concentrated near parity: 9 of the 15 kernels fall between 1.01× and 1.08×, inclusive. The largest source-level gain is on `51_Gemm_Subtract_GlobalAvgPool_LogSumExp_GELU_ResidualAdd`. The arithmetic mean is 1.25×, the median is 1.08×, and all 15 kernels exceed 1.00×.

Table 10: Per-operator source-level HIP/PyTorch speedups for the selected KernelBench Level 2 inputs, sorted by speedup.

Kernel	HIP/PyTorch
51_Gemm_Subtract_GlobalAvgPool_LogSumExp_GELU_ResidualAdd	2.93×
57_Conv2d_ReLU_HardSwish	1.45×
91_ConvTranspose2d_Softmax_BiasAdd_Scaling_Sigmoid	1.42×
37_Matmul_Swish_Sum_GroupNorm	1.23×
82_Conv2d_Tanh_Scaling_BiasAdd_Max	1.19×
7_Conv3d_ReLU_LeakyReLU_GELU_Sigmoid_BiasAdd	1.18×
72_ConvTranspose3d_BatchNorm_AvgPool_AvgPool	1.08×
17_Conv2d_InstanceNorm_Divide	1.08×
30_Gemm_GroupNorm_Hardtanh	1.04×
88_Gemm_GroupNorm_Swish_Multiply_Swish	1.03×
95_Matmul_Add_Swish_Tanh_GELU_Hardtanh	1.02×
38_ConvTranspose3d_AvgPool_Clamp_Softmax_Multiply	1.02×
81_Gemm_Swish_Divide_Clamp_Tanh_Clamp	1.01×
64_Gemm_LogSumExp_LeakyReLU_LeakyReLU_GELU_GELU	1.01×
19_ConvTranspose2d_GELU_GroupNorm	1.01×

D Per-Kernel AsmEvo Results

This section reports the per-kernel values underlying the aggregate results in the main paper. Every listed final candidate passes capture-scoped equivalence and the formal paired gate. "Byte-exact" denotes regions checked by exact comparison. For KernelBench, "Tolerance" denotes the synthetic generated-harness comparator: cosine at least 0.9999 on floating outputs, exact comparison on integer outputs and guard canaries, and a reported maximum absolute error that is diagnostic only (it is not part of the pass predicate). This is not the production AllClose or Cosine policy in Table 8.

Level 1. AsmEvo improves all 15 kernels, with a $1.62\times$ geometric mean and a $3.53\times$ maximum speedup on FrobeniusNorm.

Level 2. AsmEvo improves all 15 kernels, with a $1.23\times$ geometric mean and a $1.94\times$ maximum speedup on Matmul+Swish+Sum+GroupNorm. Table 11 reports both splits with the equivalence check and the corresponding search-attempt and commit counts.

E Real-Dispatch Capture and Generated-Persistent Evaluation

Capture acquisition. The application is launched once with `libkerncap.so` in `LD_PRELOAD`. The library registers a rocprofiler-sdk HSA table callback, replaces queue-submission and allocation API entries, and intercepts the selected HSA kernel-dispatch packet. It records the dispatch packet, raw kernarg, launch geometry, loaded code object, tracked allocations, module variables, and pre-dispatch state. The Python entry point is `run_capture`; it selects the kernel and optional dispatch index through `KERNCAP_KERNEL` and `KERNCAP_DISPATCH`. This offline step produces $\mathcal{C}$; it does not evaluate search candidates.

Memory-region capture. The interceptor tracks the base and size returned by HSA pool-allocation, region-allocation, and virtual-memory reserve/map APIs and removes entries on free or unmap. It serializes these tracked allocations in `memory_regions.json`; each region is copied device-to-host with `hsa_memory_copy` into a file named by its original virtual address. This is allocation tracking rather than recursive pointer discovery; memory not observed through the hooked HSA allocation APIs is not captured.

Generated evaluator. AsmEvo converts $\mathcal{C}$ into a kernel-specific persistent harness. The harness restores required virtual-address relationships, remaps captured memory as required by the capture contract, reconstructs the launch, loads original and candidate HSACOs, and applies the per-region validation policy in Eq. 1. Production policies are those in Table 8; KernelBench uses the synthetic cosine/exact comparator in Appendix D. Candidate evaluation therefore depends on the immutable generated harness and capture identity, not on agent-written tests.

Table 11: Per-kernel AsmEvo speedups and search counts for KernelBench Level 1 and Level 2, measured against the original compiled code object after capture-scoped equivalence and the formal paired gate. Attempts count candidates that entered GATEDEVAL; Commits count candidates that passed the formal paired gate in Eq. 2.

Kernel	Equiv.	Speedup	Attempts	Commits
KernelBench Level 1				
37_FrobeniusNorm_	Tolerance	3.53×	127	4
42_Max_Pooling_2D	Byte-exact	3.28×	120	5
36_RMSNorm_	Byte-exact	2.68×	103	8
84_conv_depthwise_2D_asymmetric_input_square_kernel	Byte-exact	2.19×	107	8
88_MinGPTNewGelu	Tolerance	1.69×	105	6
48_Mean_reduction_over_a_dimension	Byte-exact	1.63×	104	5
83_conv_depthwise_2D_square_input_asymmetric_kernel	Byte-exact	1.53×	130	11
95_CrossEntropyLoss	Byte-exact	1.49×	86	11
98_KLDivLoss	Byte-exact	1.39×	121	10
99_TripletMarginLoss	Byte-exact	1.33×	96	5
14_Matmul_for_upper_triangular_matrices	Byte-exact	1.28×	74	7
82_conv_depthwise_2D_square_input_square_kernel	Byte-exact	1.24×	74	7
30_Softsign	Byte-exact	1.08×	97	2
47_Sum_reduction_over_a_dimension	Byte-exact	1.04×	101	2
29_Softplus	Byte-exact	1.03×	81	2
KernelBench Level 2				
37_Matmul_Swish_Sum_GroupNorm	Tolerance	1.94×	103	12
91_ConvTranspose2d_Softmax_BiasAdd_Scaling_Sigmoid	Byte-exact	1.52×	116	5
88_Gemm_GroupNorm_Swish_Multiply_Swish	Tolerance	1.33×	100	9
38_ConvTranspose3d_AvgPool_Clamp_Softmax_Multiply	Tolerance	1.29×	99	4
19_ConvTranspose2d_GELU_GroupNorm	Byte-exact	1.23×	91	4
81_Gemm_Swish_Divide_Clamp_Tanh_Clamp	Byte-exact	1.22×	119	12
51_Gemm_Subtract_GlobalAvgPool_LogSumExp_GELU_ResidualAdd	Tolerance	1.20×	93	2
30_Gemm_GroupNorm_Hardtanh	Tolerance	1.19×	117	10
7_Conv3d_ReLU_LeakyReLU_GELU_Sigmoid_BiasAdd	Byte-exact	1.15×	122	3
95_Matmul_Add_Swish_Tanh_GELU_Hardtanh	Byte-exact	1.13×	113	6
17_Conv2d_InstanceNorm_Divide	Byte-exact	1.11×	64	5
72_ConvTranspose3d_BatchNorm_AvgPool_AvgPool	Byte-exact	1.11×	42	3
57_Conv2d_ReLU_HardSwish	Byte-exact	1.10×	129	2
64_Gemm_LogSumExp_LeakyReLU_LeakyReLU_GELU_GELU	Byte-exact	1.09×	86	11
82_Conv2d_Tanh_Scaling_BiasAdd_Max	Byte-exact	1.05×	118	3

Nested pointers and fixed addresses. Kernarg values and pointer-containing device buffers are preserved as opaque bytes. The evaluator does not traverse or rewrite nested pointers. Exact virtual-address restoration—via page-aligned `MAP_FIXED_NOREPLACE` reserves and `hsa_amd_vmem_address_reserve`/`hsa_amd_vmem_map` when the capture contract requires it—keeps embedded pointers valid. Replay fails if a requested address is relocated.

Paired timing and calibration. Reported KernelBench and production measurements use the generated-persistent HIP backend. Each formal session uses a private capture view, restores captured mutable state before observations, and pins hashes for the original, candidate, capture, policy, and harness source. The harness performs warmup, paired ABBA interleaving,

reciprocal role placement, and A/A calibration, and reports a percentile bootstrap 95% confidence interval (Appendix B) plus device-health and contamination evidence. Per-GPU locks serialize formal sessions. Kerncap's legacy HSA replay path remains an infrastructure fallback: it copies saved memory-region blobs, submits the captured packet, and reads timestamps from `hsa_amd_profiling_get_dispatch_time`. It is not the timing backend for the reported speedups.

F Case Studies and Drop-in Validation

F.1 Representative KernelBench Results

The largest KernelBench gain occurs on `37_FrobeniusNorm_`, which improves by $3.53\times$. Two other Level 1 kernels show similarly substantial but structurally different gains: `42_Max_Pooling_2D` reaches $3.28\times$, while `36_RMSNorm_` reaches $2.68\times$. Depthwise convolution `84_conv_depthwise_2D_asymmetric_input_square_kernel` improves by $2.19\times$. Together, these cases cover normalization, pooling, and convolution, illustrating why a single fixed peephole rule would be insufficient for the evaluated set.

The strongest Level 2 case, `37_Matmul_Swish_Sum_GroupNorm`, reaches $1.94\times$ despite combining several operations in one compiled artifact. AsmEvo treats the complete kernel as the acceptance unit: local assembly changes are rebuilt into the full code object and retained only when the fused output passes the same capture-scoped evaluator. This is important for fused pipelines, where an apparently independent instruction sequence may affect a later reduction, normalization, or activation stage.

F.2 vLLM/SGLang Triton Production Kernels

The strongest vLLM/SGLang Triton assembly result is `fused_moe_kernel_gptq_awq` at $2.08\times$. The `kernel_unified_attention_2d` kernel reaches $1.96\times$, and `_gemm_a8w8_blockscale_kernel` reaches $1.29\times$, showing that measurable headroom remains even in quantized block-scale GEMM and attention kernels emitted for a production serving stack. The A16W16 GEMM kernel reaches $1.25\times$, while `fused_moe_kernel` obtains a smaller but verified $1.10\times$ gain. The range from $1.10\times$ to $2.08\times$ illustrates why AsmEvo measures each rebuilt HSACO rather than assuming that a syntactically plausible assembly transformation will improve every JIT-generated kernel.

For these kernels, the generated persistent harness is instantiated from Kerncap (Ramos & Lowery, 2026) captures of the AgentKernelArena campaign20 baseline launches (Appendix B.2). The original kernarg layout, launch geometry, and referenced device memory are restored for both K_0 and the optimized object. This keeps the reported assembly gains tied to those baseline captures, not to live serving traffic.

F.3 AITER Code Objects

AITER kernels provide a useful stress test because they are product-grade inference artifacts. Although difficult to express at the source level after compilation, the accepted edits we inspected are narrow changes visible in recovered AMDGCN assembly: scheduling around long-latency memory operations, wait-counter placement, cache-hint and buffer-load variants, and reduced conversion or address-generation work. These are observations about those edits, not a causal breakdown of every reported gain.

Across the AITER cases, the most promising edits are localized rather than structural. The FP8 blockscale MoE kernel reaches $1.33\times$, and the two BF16 causal FMHA variants reach $1.05\times$ and $1.03\times$. These attention and MoE kernels expose opportunities around routing cleanup, pointer-rich access patterns, causal attention paths, and resource-pressure-sensitive scheduling. Hoisting, interleaving, or tighter wait counters can reduce stalls without changing the launcher ABI. Every accepted edit is evaluated in the capture-derived harness before its speedup is reported.

F.4 Dispatch and End-to-End Validation

AsmEvo rebuilds each accepted edit into a loadable AMDGPU code object that preserves the original symbol, kernarg layout, descriptor-visible resources, and metadata contract. The optimized artifact therefore follows the same host-side dispatch path as the original. For KernelBench-derived HIP kernels, we replace the original MI308X object and run the same benchmark harness.

For production libraries, we additionally validate the native runtime path. AITER replacements pass its unit tests under the original launch configurations. For vLLM and SGLang, we replace the cached Triton HSACO, disable recompilation, and run framework tests against the replacement. These checks confirm that the replacements load through the original dispatch path. They are not end-to-end serving-latency measurements.