# EVOK: EVIDENCE-CENTRIC MULTI-AGENT FRAMEWORK FOR GPU KERNEL GENERATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

High-performance GPU kernels are critical for improving the efficiency of modern deep-learning workloads, but their development still relies heavily on expert tuning. While recent LLM-based approaches show promise for automating kernel generation, they still struggle to achieve stable correctness and high performance across diverse operators and GPU platforms. This limitation stems from candidate-centric generation, which leaves reusable semantic patterns, implementation choices, and measured failures implicit in prior optimization histories rather than preserving them as reusable search evidence. To address this limitation, EVOK organizes prior and runtime evidence as persistent optimization state to guide LLM-based kernel search. It introduces two coupled designs: 1) evidence-guided staged optimization, which decomposes generation into route planning, candidate realization, and local admission so that reusable optimization opportunities can guide exploration under target-local validation; and 2) evidence-aware multi-agent collaboration, which enables specialized agents to accumulate, refine, and exchange validated optimization evidence across successive kernel searches. Experiments on KernelBench across Tesla T4, RTX 4090, and RTX PRO 6000 show that EVOK achieves 3.5–8.6$\times$ level-wise mean speedup over PyTorch eager, outperforming direct-LLM and agentic baselines.

## 1 INTRODUCTION

High-performance GPU kernels are a critical foundation for modern deep-learning systems. As model architectures and deployment workloads continue to grow, system efficiency increasingly depends on whether core tensor operators can fully utilize GPU hardware (Zhou et al., 2024; Sheng et al., 2023; Yu et al., 2022). However, developing such kernels remains labor-intensive, requiring expert knowledge of tensor semantics, memory hierarchy, parallel execution, and hardware-specific performance behavior. This motivates automatic kernel generation as an important systems problem.

Recent progress in large language models has made this problem increasingly practical. Given a high-level operator description, LLMs can synthesize CUDA (NVIDIA, 2026b) or Triton (Tillet et al., 2019) implementations and repair syntactic or semantic errors (Le et al., 2022; Fan et al., 2023; Ouyang et al., 2025). Yet GPU kernel generation is not ordinary code generation. Beyond syntactic validity, a useful kernel must preserve tensor semantics and induce an execution structure aligned with the target GPU's parallelism, memory hierarchy, and dispatch behavior. Correctness is therefore only a necessary condition: without structured optimization guidance, LLMs may produce valid but inefficient kernels, while useful optimization choices remain difficult to reuse across related operators or hardware.

Existing LLM-based GPU kernel generation methods mainly follow two directions. Training-based methods adapt models through fine-tuning or reinforcement learning (Baronio et al., 2025; Li et al., 2025; Woo et al., 2025). They improve the model's familiarity with CUDA or Triton syntax and optimization-oriented code patterns, but high-quality kernel data are scarce because each sample requires compilable code, numerical validation, and hardware-specific measurement. This data bottleneck limits the ability of training-based methods to learn stable semantic constraints and optimization strategies, often leading to low correctness or limited performance on unseen operators. Agentic methods instead wrap general-purpose LLMs with iterative generation, debugging, profiling, and judging loops (Zhang et al., 2025; Han et al., 2026). These systems exploit runtime feedback



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.

without additional training, but their optimization state is typically tied to the current candidate and prompt history, making reusable semantic patterns, implementation choices, and measured failures difficult to preserve.

EGG is a recent representative agentic system that introduces expert-guided staged optimization (Han et al., 2026). EGG-style systems improve over undirected candidate refinement by decomposing generation into algorithmic-structure design and hardware-specific tuning. This design injects expert optimization principles into the search process and provides clearer stage-wise objectives. However, the knowledge flow remains primarily task-local: optimization decisions are refined along the current candidate trajectory, while successful routes, implementation choices, and failed measurements are not explicitly shared across operators or hardware. Thus, expert guidance reduces local search disorder, but it does not by itself expose optimization opportunities whose evidence is distributed across related tasks and platforms.

These limitations suggest that fully exploiting LLMs for GPU kernel generation requires structuring not only optimization stages, but also the optimization knowledge that flows across stages, tasks, and hardware. Effective systems should preserve semantic routes, implementation choices, and measured outcomes as reusable evidence that constrains the design space and guides subsequent exploration.

To this end, we propose EVOK, an evidence-centric multi-agent framework for GPU kernel generation. As shown in Figure 1, EVOK consists of two coupled components. The first is *evidence-guided staged optimization*, which organizes generation into route planning, candidate realization, and local admission. Route planning retrieves positive and negative evidence and selects semantics-preserving optimization routes. Candidate realization turns selected routes into bounded implementations through semantic rewrite, evidence reuse, or LLM generation, followed by target-local tuning over backend path, execution layout, and runtime reuse. Local admission compiles, tests, benchmarks, and writes measured outcomes back as reusable evidence.

The second component is *evidence-aware multi-agent collaboration*. The Evidence Agent, Realization Agent, and Verification Agent communicate through structured evidence states rather than raw prompt histories or transient code candidates. The Evidence Agent authorizes semantic routes, the Realization Agent materializes implementation blueprints, and the Verification Agent attaches target-GPU validation. Positive evidence prioritizes promising routes and implementation forms, while scoped negative evidence prevents repeated exploration of invalid, unstable, or uncompetitive regions. This organization turns historical optimization outcomes into reusable search guidance under explicit semantic and validation constraints.

We evaluate EVOK on KernelBench Level 1–3 across Tesla T4, RTX 4090, and RTX PRO 6000. EVOK achieves 3.5–8.6$\times$ level-wise mean speedup over PyTorch eager and consistently outperforms direct-generation and agentic baselines. Mechanism coverage, component ablation, and ledger-size sensitivity further show that the gains come from reusable semantic routes, bounded realization, and verification-driven evidence accumulation.

Crucially, EVOK treats evidence as typed optimization state rather than a prompt cache, separating semantic legality, implementation blueprints, and hardware-local admission before any kernel is promoted.

The main contributions are summarized as follows:

- We propose an evidence-centric multi-agent framework for GPU kernel generation, which leverages prior and runtime evidence to guide LLM-based optimization.
- We introduce evidence-guided staged optimization, which decomposes kernel generation into route planning, candidate realization, and local admission to progressively constrain and refine the search process.
- We design *Evidence-Aware Multi-Agent Collaboration*, which enables specialized agents to accumulate, refine, and reuse validated optimization evidence across successive kernel searches.
- We evaluate EVOK on KernelBench across three GPU platforms, where it achieves 3.5–8.6$\times$ level-wise mean speedup over PyTorch eager and outperforms direct-LLM and agentic kernel-generation baselines.

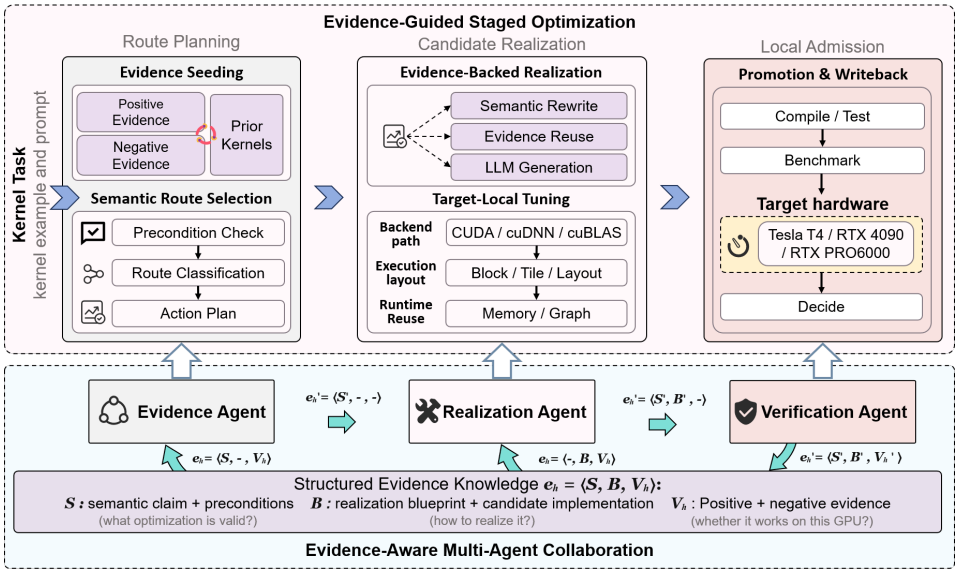



Figure 1: Overview of EVOK's staged optimization and evidence-aware agent collaboration.

## 2 RELATED WORK

**GPU Kernel Optimization Foundations.** Efficient GPU kernels arise from a close match between the mathematical structure of an operator and the execution model of the device. In practice, high performance is achieved not only by writing low-level code, but also by choosing an appropriate computation form, exposing parallel work at the right granularity, controlling data movement, and exploiting locality in registers, shared memory, and caches. These choices are reusable at the level of optimization principles, yet their concrete effect depends strongly on tensor shapes, data types, layouts, and hardware resources such as SM occupancy, memory bandwidth, and register pressure.

Existing optimization substrates embody this expertise in different forms. Vendor libraries and specialized kernels, such as cuBLAS (NVIDIA, 2026a) and cuDNN (NVIDIA, 2026d), provide carefully engineered implementations for common operator families, while compiler and programming systems, including TVM (Chen et al., 2018), Ansor (Zheng et al., 2020), PET (Wang et al., 2021), TensorIR (Feng et al., 2023), Triton (Tillet et al., 2019), TileLang (Wang et al., 2025), and PyTorch 2 (Ansel et al., 2024), expose higher-level interfaces for expressing transformations, schedules, or tile-level programs. These systems distribute optimization knowledge across libraries, schedulers, and code generators, but they still separate the choice of a valid transformation from the evidence that it works on a specific GPU. As a result, the central decision problem remains the same: for a new operator, one must determine which semantic transformation is valid, which realization form is promising, and which measured outcome is reliable on the target GPU. EVOK addresses this decision layer by using structured evidence to guide LLM-based selection, realization, and validation.

**Learning-based kernel generation.** Learning-based methods adapt LLMs to CUDA or Triton generation through supervised tuning, reinforcement learning, or reward-driven optimization. Kevin (Baronio et al., 2025) improves CUDA generation with multi-turn reinforcement learning, while AutoTriton (Li et al., 2025) and TritonRL (Woo et al., 2025) focus on learning Triton programming and optimization behavior. These methods can improve the model's familiarity with kernel syntax and local optimization patterns, but they depend on costly supervision or reward exploration. In practice, their learned policies often capture preferences at the level of local patterns rather than durable search state, and the dependence on expensive labels and rewards tends to bias them toward operator families with abundant traces. More importantly, the learned policy does not explicitly separate semantic validity, implementation choice, and hardware-local measurement, making it difficult to preserve successful and failed optimization outcomes as reusable decision state. EVOK is orthogonal to model training: it uses a general-purpose LLM inside an evidence-guided workflow where structured records constrain generation and admission.

**Agentic kernel generation.** Agentic methods reduce the need for model training by wrapping general-purpose LLMs with iterative coding, debugging, profiling, judging, or expert-guided stag-

ing. CudaForge (Zhang et al., 2025) uses hardware feedback to guide CUDA optimization, and EGG (Han et al., 2026) decomposes generation into algorithmic-structure design and hardware-specific tuning with stage-aware agent collaboration. These systems show that feedback and expert objectives can improve correctness and local performance. The optimization state is still mainly candidate-centric: semantic routes, implementation choices, measured wins, and failed cases are usually retained inside the current candidate trajectory or prompt context rather than as reusable evidence across operators or hardware. EVOK retains structured LLM decision-making but changes the optimization carrier from candidate trajectories to typed evidence records, enabling evidence-guided reuse, bounded realization, hardware-local admission, and scoped rejection.

# 3 METHOD

## 3.1 OVERVIEW

EVOK is an evidence-centric multi-agent framework for automatic GPU kernel generation. As illustrated in Figure 1, it contains two core components. An *evidence-guided staged optimization* strategy decomposes generation into route planning, candidate realization, and local admission, separating semantic opportunity from implementation choice and measured performance. An *evidence-aware multi-agent collaboration* mechanism assigns these decisions to specialized agents and coordinates them through structured evidence.

Across the staged workflow, the current task state is progressively completed from a semantic route to a realized candidate and finally to hardware-local validation. We store historical outcomes in a runtime ledger $\mathcal{L}$ as structured evidence:

$$e_h = \langle \mathcal{S}, \mathcal{B}, \mathcal{V}_h \rangle. \tag{1}$$

Here, $\mathcal{S}$ captures semantic validity, $\mathcal{B}$ captures the realization blueprint and candidate implementation, and $\mathcal{V}_h$ captures signed validation on GPU $h$. We use the following taxonomy: seeds are unverified hypotheses; observations are measured records for retrieval or accounting; signed evidence is locally admitted and provenance-scoped; eager-positive/eager-negative are accounting labels relative to PyTorch eager; and promoted evidence is signed $\mathcal{V}_h^+$ admitted by gates, while failures or sub-gate outcomes become $\mathcal{V}_h^-$. The ledger $\mathcal{L}$ may store all records, but only signed evidence can promote a route–blueprint pair. In figures, unprimed states denote ledger records and primed states denote current-task states.

## 3.2 EVIDENCE-GUIDED STAGED OPTIMIZATION

GPU kernel generation couples three qualitatively different decisions: which semantic transformation exposes an optimization opportunity, which implementation family realizes that transformation, and whether the resulting candidate is advantageous on the target GPU. Optimizing them jointly obscures the source of progress and allows noisy code or timing feedback to redirect semantic search. EVOK instead imposes a hierarchy of route planning, candidate realization, and local admission. Each stage fixes one decision class and supplies explicit constraints to the next, progressively reducing the admissible design space.

### 3.2.1 ROUTE PLANNING

Route planning establishes the semantic foundation that bounds subsequent realization. The central difficulty is not code syntax but identifying a transformation that removes work while preserving the operator contract. Textually different kernels can retain the same redundant computation, whereas a concise semantic rewrite can eliminate an intermediate tensor, a reduction, or an entire launch. The stage combines evidence seeding with guarded route selection and emits the semantic state $\mathcal{S}$.

Expert routes, prior implementations, NCU observations, and historical kernels enter as seeds; target-local successes and failures enter as signed evidence. Retrieval from the ledger matches semantic structure and available shape facts while preserving the originating task, implementation, backend, and hardware scope:

$$\mathcal{E}(T, h) = \{e \in \mathcal{L} \mid \mathrm{match}(\mathcal{S}_e, T) \wedge \mathrm{scope}(e, T, h)\}. \tag{2}$$

In implementation, $\mathrm{match}(\mathcal{S}_e, T)$ checks operator-graph structure, shape/dtype/layout compatibility, and route-level semantic cues, while $\mathrm{scope}(e, T, h)$ verifies backend, hardware, and provenance compatibility. Retrieved evidence is ranked by structural match first, scope compatibility second, and evidence quality as a final tie-breaker. The result is a set of plausible starting points rather than a stored solution for the target operator. When no validated evidence is available, EVOK falls back to guarded route templates and bounded LLM realization.

LLM-derived routes enter only as unverified offline seeds; online LLM calls begin after a route passes its deterministic guard. Each route is accepted only if a deterministic predicate over operator pattern, shape facts, dtype/layout constraints, and excluded semantic cases holds. Admissible opportunities are then classified into routes such as constant-result collapse, singleton simplification, aggregate-before-materialize, reduction-tail CSE, linear-algebra composition, normalization, softmax, and convolution tails. Let $\mathcal{T}$ denote the guarded template pool, including a conservative direct-generation template. Route selection is therefore a constrained decision over both retrieved evidence and template fallback:

$$\begin{aligned}\mathcal{R}(T,h) =& \{\mathcal{S}_e \mid e \in \mathcal{E}(T,h),\ P_e(T)=1\} \\ & \cup \{\mathcal{S}_\tau \mid \tau \in \mathcal{T},\ P_\tau(T)=1\}.\end{aligned} \tag{3}$$

$$\mathcal{S}^\star = \arg\max_{\mathcal{S}\in\mathcal{R}(T,h)} \big(\mathrm{pos}(\mathcal{S}) - \mathrm{neg}(\mathcal{S}) - \lambda\,\mathrm{cost}(\mathcal{S})\big). \tag{4}$$

The direct-generation template keeps $\mathcal{R}(T,h)$ nonempty for targets with valid PyTorch references. In Eq. 4, $\mathrm{pos}$ is weighted promoted support; $\mathrm{neg}$ is weighted scoped failure or sub-gate support; $\mathrm{cost}$ estimates implementation, tuning, and compile/test burden; and $\lambda$ trades support against cost. Exact-scope records dominate broader matches, and negative evidence only down-ranks candidates within its recorded scope. The selected route yields preconditions, exclusions, and realization alternatives; passing a guard authorizes exploration rather than correctness.

Figure 2(a) illustrates this stage on a Linear–Sigmoid–Sum operator. A candidate-centric agent may repeatedly regenerate kernels for the original three-step program. EVOK instead retrieves evidence for the same semantic pattern and rewrites the target into a row-wise reduction,

$$Y[i,0] = \sum_j \sigma(X[i,:]W[j,:]^\top + b[j]). \tag{5}$$

The semantic claim removes hidden-activation materialization from the optimization target and supplies the next stage with a constrained objective.

### 3.2.2 CANDIDATE REALIZATION

Candidate realization converts the selected route into a bounded implementation family $\mathcal{B}$. LLM calls are used only as a bounded realization operator within this stage, where they fill implementation choices under an already authorized route. The stage preserves the semantic decision made during route planning while resolving how that decision should be expressed as executable code. As shown in Figure 1, realization consists of an evidence-backed implementation choice followed by target-local tuning.

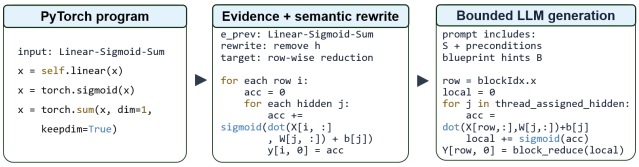
Three realization paths define the admissible implementation choices:

$$\begin{aligned}\mathcal{B}^\star \in\ & \mathrm{Rewrite}(\mathcal{S}^\star) \cup \mathrm{Reuse}(\mathcal{S}^\star, \mathcal{E}) \\ & \cup\, \mathrm{LLM}(\mathcal{S}^\star, B).\end{aligned} \tag{6}$$

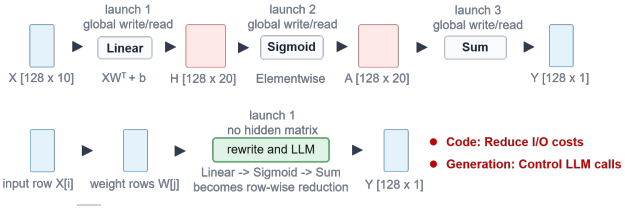
*Semantic rewrite* materializes deterministic transformations whose output structure follows directly from the selected route. *Evidence reuse* adapts a validated blueprint from a related operator or hardware context rather than copying a finished kernel. *LLM generation* is invoked when the route is established but no compatible blueprint is available. In all cases, the candidate set is bounded by the selected route, the target-local tuning space $\Theta_h$, and the search budget $B$:

$$\begin{aligned}\mathcal{C}(T,h) \subseteq\ & \{\mathrm{mat}(T, \mathcal{S}^\star, \mathcal{B}^\star, \theta) \mid \theta \in \Theta_h\}, \\ & |\mathcal{C}(T,h)| \le B.\end{aligned} \tag{7}$$

Equations 6 and 7 preserve strong implementation priors while retaining generation as a controlled fallback.




(a) Semantic rewrite with bounded LLM generation.




(b) Fused row-wise realization for Linear–Sigmoid–Sum.

Figure 2: Evidence-guided realization of Linear–Sigmoid–Sum.

Target-local tuning instantiates the lower half of the realization stage in Figure 1. The backend path fixes the execution substrate and fusion boundary, including custom CUDA, library-informed compositions, compiler paths, or hybrid realizations. The execution layout then selects block mapping, tile size, vector width, reduction structure, and memory layout under the semantic preconditions established by route planning. Runtime reuse covers opportunities that emerge across repeated launches, such as cached workspaces, in-place updates, and CUDA Graph replay (NVIDIA, 2026c). Figure 2(b) instantiates the running example as a fused row-wise CUDA kernel: one block owns a batch row, threads accumulate assigned hidden indices, and a block reduction writes the output once. The resulting blueprint removes two intermediate global-memory tensors and reduces three launches to one.

#### 3.2.3 Local Admission

Local admission separates a generated candidate from reusable performance evidence. The target GPU is part of the decision context: a semantically valid implementation may compile successfully yet remain unstable or slower than an optimized framework path. Admission therefore combines implementation screening, differential correctness testing, repeated benchmarking, and a baseline-relative promotion gate.

Each candidate is checked against its declared implementation form and provenance before compilation and differential testing. Numerical agreement is evaluated over the task inputs and tolerances; compilation failures, incorrect outputs, and unstable executions are attributable candidate outcomes, while infrastructure failures are excluded.

Correct candidates are benchmarked repeatedly against PyTorch eager on the target GPU and are promoted only when correctness, implementation-form, provenance, stability, baseline-validity, and repeated-speedup gates hold. Promotion turns an unverified route–blueprint pair into signed local validation rather than a global performance claim: passing candidates become $\mathcal{V}_h^+$ with their validated operating context, while incorrect, unstable, inadmissible, or sub-gate candidates become $\mathcal{V}_h^-$ with a failure category and task-, shape-, dtype-, layout-, backend-, and hardware-specific scope. Exact incompatibilities block a matched realization; empirical performance failures act as scoped penalties and remain soft outside their observed context. A later positive counterexample supersedes such a penalty only within its validated scope. Routes and blueprints may initialize realization on another GPU, but differences in tensor-core, register pressure, or shared-memory limits trigger target-local retuning and are written back as new scoped validation.

### 3.3 Evidence-Aware Multi-Agent Collaboration

The staged decomposition separates three decision authorities: semantic authorization, code realization, and empirical admission. EVOK assigns them to specialized agents and restricts interaction to typed evidence transitions, preventing semantic preconditions, local timing results, and debugging traces from being conflated across stages.

#### 3.3.1 MULTI-AGENT DESIGN

The three agents correspond to the stages in the upper region of Figure 1, but their roles are defined by both their decision authority and the evidence fields they produce:

- **Evidence Agent:** performs evidence retrieval, route guard checking, and semantic route selection. It decides which optimization routes are admissible for the target operator and emits a semantic decision $\mathcal{S}'$ with explicit preconditions and excluded cases.
- **Realization Agent:** turns the selected route into bounded candidate implementations. It selects among deterministic rewrites, blueprint reuse, template-based realization, and LLM generation, then emits a realization record $\mathcal{B}'$ together with candidate kernels.
- **Verification Agent:** compiles, tests, benchmarks, and admits candidates on the target GPU. It preserves the submitted $\mathcal{S}'$ and $\mathcal{B}'$, and writes hardware-local positive or negative validation $\mathcal{V}_h^+$ or $\mathcal{V}_h^-$.

#### 3.3.2 STRUCTURED EVIDENCE EXCHANGE

The evidence record in Eq. 1 defines the communication interface. The semantic component $\mathcal{S} = \langle R, P \rangle$ records route $R$ and preconditions $P$; the realization component $\mathcal{B} = \langle I, A \rangle$ records implementation form $I$ and bounded actions $A$; signed validation stores satisfied gates in $\mathcal{V}_h^+$ or a failure category and scope in $\mathcal{V}_h^-$.

Figure 1 distinguishes retrieved history from the state of the current task. Unprimed projections supply the Evidence Agent with semantic and validation context and the Realization Agent with blueprint and validation context. Forward transitions carry the current semantic decision $\langle \mathcal{S}', -, - \rangle$, then the completed blueprint $\langle \mathcal{S}', \mathcal{B}', - \rangle$; local admission writes the complete result $\langle \mathcal{S}', \mathcal{B}', \mathcal{V}_h' \rangle$. Only finalized typed decisions cross agent boundaries. Raw prompt histories, rejected intermediate edits, and infrastructure failures do not become reusable optimization state.

This interface enforces two invariants. The Realization Agent cannot silently relax semantic preconditions to accommodate an implementation, and the Verification Agent cannot promote a timing result without its originating route, blueprint, and provenance. Conversely, measured failures return as scoped search constraints rather than unstructured debugging transcripts.

## 4 EXPERIMENTS

The evaluation follows the EVOK workflow: end-to-end comparison measures target-GPU acceleration, ablation separates evidence quality and search cost, mechanism coverage attributes gains to route and realization choices, and ledger-size sensitivity tests reusable evidence.

### 4.1 EVALUATION SETUP

**Software and Hardware.** All experiments use CUDA 13.0 (NVIDIA, 2025), PyTorch 2.9.1 (PyTorch Foundation, 2025), and GPT-5.5 (OpenAI, 2026). We report results on Tesla T4, RTX 4090, and RTX PRO 6000, which differ in memory bandwidth, launch overhead, tensor-core capability, and register pressure.

**Baselines and Benchmark.** Baselines include PyTorch eager and `torch.compile` default/max-autotune (PyTorch Foundation, 2026), direct LLM generation (DeepSeek V3.2 (Liu et al., 2025), ChatGPT 5.1 (OpenAI, 2025), ChatGPT 5.5 (OpenAI, 2026)), AutoTriton (Li et al., 2025), CudaForge (Zhang et al., 2025), and EGG (Han et al., 2026). Agentic baselines use the same GPT-5.5 backend as EVOK. All methods are evaluated on KernelBench Level 1–3 (Ouyang et al., 2025), which contains 250 PyTorch-defined operators. Slash-separated triplets in Table 1 follow the RTX 4090/RTX PRO 6000/Tesla T4 order.

**Evidence Ledger.** Before optimizing a target, EVOK forms an eligible ledger from guarded route-template seeds and historical traces, normalized as semantic route, realization blueprint, hardware-local validation, and scoped failure category. Same-operator/same-GPU records are removed before retrieval, so evidence can guide route or blueprint selection but cannot import target results. Construction is online and amortized: generation, compile, test, and benchmark costs are counted as

Table 1: Comparison against PyTorch eager on three GPUs (RTX 4090/RTX PRO 6000/Tesla T4).

| Method | Level 1 | | | Level 2 | | | Level 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Success(%) | $Fast_1$(%) | Speedup | Success(%) | $Fast_1$(%) | Speedup | Success(%) | $Fast_1$(%) | Speedup |
| Torch Compile | 100 | 72/81/79 | 1.1/1.3/1.2 | 100 | 84/75/75 | 1.4/1.1/1.3 | 100 | 92/86/72 | 1.4/1.2/1.4 |
| Torch Compile(max-autotune) | 100 | 66/59/63 | 1.8/1.5/1.6 | 100 | 87/78/82 | 2.7/2.3/2.6 | 100 | 84/82/86 | 1.5/1.7/1.3 |
| DeepSeek V3.2 | 34/-/- | 11/-/- | 1.0/-/- | 40/-/- | 17/-/- | 0.9/-/- | 36/-/- | 14/-/- | 0.8/-/- |
| ChatGPT 5.1 | 60/-/- | 18/-/- | 0.9/-/- | 60/-/- | 32/-/- | 1.1/-/- | 66/-/- | 24/-/- | 0.9/-/- |
| ChatGPT 5.5 | 97/88/90 | 45/40/40 | 1.2/1.4/1.5 | 92/98/92 | 54/72/15 | 3.7/3.0/2.4 | 82/88/74 | 20/62/28 | 1.0/1.3/1.0 |
| AutoTriton | 36/46/24 | 9/16/11 | 1.2/1.3/1.2 | 55/54/43 | 22/35/31 | 1.0/1.1/1.1 | 56/44/48 | 26/12/26 | 0.8/0.9/1.1 |
| CudaForge | 100 | 60/60/56 | 1.5/1.6/1.4 | 100 | 94/92/92 | 2.3/2.1/2.1 | 100 | 78/76/76 | 1.5/1.5/1.4 |
| EGG | 100 | 78/70/71 | 2.0/2.1/1.9 | 100 | 100/98/100 | 3.0/3.2/2.7 | 100 | 94/94/96 | 1.6/1.7/1.7 |
| **EVOK** | **100** | **100/100/98** | **3.5/3.6/4.1** | **100** | **100** | **7.2/7.4/8.6** | **100** | **100/100/98** | **4.3/4.5/3.6** |

search cost; insertion only normalizes and indexes records. Table 3 varies this filtered ledger: Empty keeps template fallback; partial rows use fixed-seed stratified samples over route, hardware scope, and evidence polarity; Full uses all eligible records.

**Metrics.** Following KernelBench (Ouyang et al., 2025) and EGG (Han et al., 2026), we report success, $Fast_1$, and mean speedup over PyTorch eager. Success counts correct tasks; $Fast_1$ counts correct tasks faster than eager. Speedup is the arithmetic mean over correct tasks within each level; cross-level ablations and sensitivity results macro-average the three level means.

### 4.2 Overall Performance

Table 1 reports the end-to-end comparison across three GPUs under the filtered-ledger protocol above. Every reported kernel is realized or adapted anew and must pass fresh correctness and timing admission on the target GPU. Under the reported KernelBench tests, EVOK passes all correctness checks, reaches 98–100% $Fast_1$ coverage, and achieves 3.5–8.6× mean speedup over PyTorch eager. Compared with `torch.compile`, EVOK raises $Fast_1$ coverage from 59–92% to 98–100% and mean speedup from 1.1–2.7× to 3.5–8.6×.

The separation is largest on fused and compound operators: EVOK raises Level-2 speedup from 1.1–2.7× under compiler baselines to 7.2–8.6×. Direct LLM baselines can generate correct kernels, but ChatGPT 5.5's $Fast_1$ coverage drops to 15–72% on Level 2–3 because route preconditions and failure boundaries remain implicit. Against EGG, EVOK raises mean speedup from 1.6–3.2× to 3.5–8.6× and reaches 98–100% $Fast_1$ coverage compared with EGG's 70–100%. The key factor is the optimization carrier: typed evidence records preserve reusable routes, blueprints, measurements, and negative cases beyond the current candidate trajectory.

### 4.3 Component Ablation

Figure 3 isolates how evidence quality affects coverage and search cost. Direct LLM generation reaches 27.6–57.2% $Fast_1$ and 1.6–2.0× average speedup. Search without evidence recall improves $Fast_1$ to 47.2–58.8% but needs 6.7–7.6 LLM calls per operator, while random recall reaches only 51.6–69.6%. Adding scoped negative evidence raises $Fast_1$ to 68.0–76.4% by suppressing repeated compile, correctness, instability, and slow-kernel failures. Positive evidence then reaches 86.8–92.0% $Fast_1$ with 3.4–3.8 calls per operator. Full EVOK reaches 98.8–100.0% $Fast_1$ and 5.0–5.4× speedup while still saving 2.2–3.1 calls per operator relative to no-recall search.

### 4.4 Mechanism Coverage

Table 2 links speedups to evidence-guided decisions. Work-removing routes provide the largest gains: aggregate-before-materialize covers 25 tasks with 24 $Fast_1$ wins and 6.7× speedup, while constant collapse covers 15 tasks with 14 wins and 6.3× speedup. Many gains therefore come from changing the optimization target before code generation.

Implementation reuse appears in 131 tasks, yields 128 $Fast_1$ wins, and averages 5.8× speedup, indicating that validated blueprints transfer beyond their originating task. Reduction-tail CSE and CUDA Graph replay capture complementary effects: semantic work reduction and launch-overhead

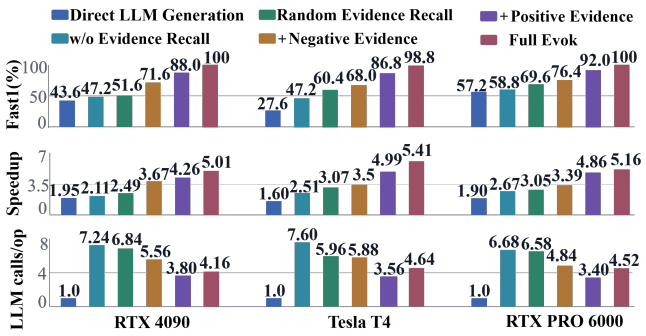



Figure 3: Ablation across three GPUs.

Table 2: Mechanism coverage on RTX 4090.

| Route / Mechanism | Tasks | $Fast_1$ | Avg. Speedup |
|---|---|---|---|
| Constant collapse | 15 | 14 | 6.3 |
| Singleton simplification | 16 | 15 | 5.5 |
| Reduction-tail CSE | 62 | 59 | 3.7 |
| Aggregate-before-materialize | 25 | 24 | 6.7 |
| Library-backed path | 52 | 50 | 1.9 |
| Implementation reuse | 131 | 128 | 5.8 |
| CUDA Graph replay | 52 | 50 | 3.8 |

Table 3: Ledger-size sensitivity on RTX 4090.

| Evidence subset | $Fast_1$ (%) | Speedup (×) | Calls/task |
|---|---|---|---|
| Empty | 48.0 | 2.1 | 7.2 |
| 25% stratified | 60.8 | 3.1 | 6.4 |
| 50% stratified | 85.6 | 3.9 | 5.1 |
| 75% stratified | 94.0 | 4.8 | 4.5 |
| Full | 100.0 | 5.0 | 4.2 |

amortization. LLM calls operate inside this bounded realization stage; local admission decides whether the resulting candidate becomes evidence.

### 4.5 Ledger Sensitivity and Evidence Transfer

Table 3 tests whether evidence acts as reusable optimization state rather than passive logs by varying the filtered ledger size on RTX 4090 under the same route guards, bounded realization, and target-local admission.

Increasing the ledger fraction improves all metrics: $Fast_1$ rises from 48.0% to 100.0%, speedup from 2.1× to 5.0×, and calls/task falls from 7.2 to 4.2. Appendix transfer analysis further removes same-operator or target-GPU records: EVOK still reaches 96.0% $Fast_1$ when all same-operator records are excluded and 91.2% under cross-platform-only transfer, showing that cross-platform evidence remains useful while target-local evidence reduces tuning cost.

This result also clarifies the boundary between reuse and verification. Cross-platform evidence is valuable because semantic routes and implementation blueprints often encode operator-level structure that survives changes in memory bandwidth, launch overhead, and register pressure. However, the final timing decision is deliberately not shared as a portable fact: every imported route must still be locally realized, compiled, tested, and benchmarked before promotion. The ledger therefore serves as a reusable prior over the search space, while the target-local gate remains the source of correctness and performance authority.

## 5 Conclusion

This paper introduced EVOK, an evidence-centric multi-agent framework for LLM GPU kernel generation. Evidence-guided staged optimization separates route planning, candidate realization, and local admission, while evidence-aware agent collaboration preserves their decisions as typed semantic, realization, and hardware-local validation records. KernelBench experiments show that this organization improves acceleration coverage across three GPUs, establishing structured evidence and staged agent collaboration as effective foundations for reusable LLM-based kernel optimization.

The broader implication is that kernel generation systems should treat measured outcomes as reusable program-optimization state rather than transient debugging feedback. Positive records provide route and blueprint priors, negative records delimit unproductive regions, and target-local gates keep portability claims tied to actual compilation, correctness, and timing evidence. This separation is what allows EVOK to reuse prior work without replacing local validation.

## REFERENCES


Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 929–947, 2024.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. arXiv:2507.11948.

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

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pp. 1469–1481, 2023.

Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. TensorIR: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 804–817, 2023.

Yaochen Han, Ke Fan, Hongxu Jiang, Wanqi Xu, Weiyu Xie, Runhua Zhang, Chenhui Zhu, and Yixiang Zhang. EGG: An expert-guided agent framework for kernel generation, 2026. arXiv preprint arXiv:2606.26758.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 21314–21328, 2022.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. AutoTriton: Automatic triton programming with reinforcement learning in llms, 2025. arXiv:2507.05687.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

NVIDIA. *CUDA Toolkit 13.0 Release Notes*, 2025. `https://docs.nvidia.com/cuda/archive/13.0.0/cuda-toolkit-release-notes/index.html`.

NVIDIA. *cuBLAS Library*, 2026a. `https://docs.nvidia.com/cuda/cublas/`.

NVIDIA. *CUDA Programming Guide*, 2026b. `https://docs.nvidia.com/cuda/cuda-programming-guide/`.

NVIDIA. *CUDA Programming Guide: CUDA Graphs*, 2026c. `https://docs.nvidia.com/cuda/cuda-programming-guide/index.html#cuda-graphs`.

NVIDIA. *NVIDIA cuDNN Documentation*, 2026d. `https://docs.nvidia.com/deeplearning/cudnn/latest/`.

OpenAI. *GPT-5.1 Model*, 2025. `https://developers.openai.com/api/docs/models/gpt-5.1`.

OpenAI. *GPT-5.5 Model*, 2026. `https://developers.openai.com/api/docs/models/gpt-5.5`.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

PyTorch Foundation. *PyTorch 2.9.1 Release*, 2025. https://github.com/pytorch/pytorch/releases/tag/v2.9.1.

PyTorch Foundation. *torch.compile Documentation*, 2026. https://docs.pytorch.org/docs/2.9/generated/torch.compile.html.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 31094–31116, 2023.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation*, pp. 37–54, 2021.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. TileLang: A composable tiled programming model for AI systems, 2025. arXiv preprint arXiv:2504.17577.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. TritonRL: Training llms to think and code triton without cheating, 2025. arXiv:2510.17891.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538, 2022.

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

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiao-Ping Zhang, Yuhan Dong, and Yu Wang. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024.