STARK: STRATEGIC TEAM OF AGENTS FOR REFINING KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

The efficiency of GPU kernels is central to the progress of modern AI, yet optimizing them remains a difficult and labor-intensive task due to complex interactions between memory hierarchies, thread scheduling, and hardware-specific characteristics. While recent advances in large language models (LLMs) provide new opportunities for automated code generation, existing approaches largely treat LLMs as single-shot generators or naive refinement tools, limiting their effectiveness in navigating the irregular kernel optimization landscape. We introduce an LLM agentic framework for GPU kernel optimization that systematically explores the design space through multi-agent collaboration, grounded instruction, dynamic context management, and strategic search. This framework mimics the workflow of expert engineers, enabling LLMs to reason about hardware trade-offs, incorporate profiling feedback, and refine kernels iteratively. We evaluate our approach on KernelBench, a benchmark for LLM-based kernel optimization, and demonstrate substantial improvements over baseline agents: our system produces correct solutions where baselines often fail, and achieves kernels with up to 16× faster runtime performance. These results highlight the potential of agentic LLM frameworks to advance fully automated, scalable GPU kernel optimization.

1 INTRODUCTION

Artificial intelligence (AI) has advanced at an unprecedented pace, transforming both research and real-world applications. While innovations in model architectures and training algorithms have been central to this progress, the efficiency of the computational infrastructure that executes them is equally critical. At the core of modern AI systems are *GPU kernels*, which implement fundamental operations such as matrix multiplication and convolution. Even modest improvements in GPU kernel efficiency can translate into significant reductions in training time, inference latency, and deployment cost, making kernel optimization a cornerstone for sustaining AI's rapid growth.

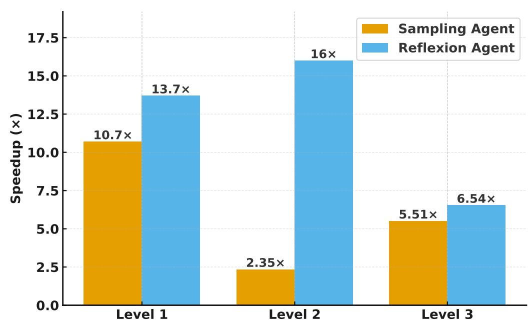

Figure 1: Speedup of `STARK` over baseline agents on KernelBench (L1–L3) with same number of attempts. Bars report GPU wall-clock speedups (×) relative to the *Sampling* and *Reflexion* agents; higher is better. `STARK` reaches up to 16× over Reflexion (L2) and 10.7× over Sampling (L1).

Despite their importance, designing and optimizing GPU kernels remains a major challenge. The performance of a kernel depends on subtle interactions between thread scheduling, memory hierarchy utilization, synchronization, and hardware-specific characteristics. Small changes in tiling strategies, loop unrolling, or memory alignment can yield disproportionate effects on runtime. As a result, the kernel optimization landscape is highly irregular, architecture-dependent, and difficult to navigate. Existing approaches largely fall into two categories: *manual optimization* by expert engineers, which is effective but labor-intensive and difficult to scale; and *automated compilers and domain-specific languages* (DSLs) such as TVM and Triton (Chen et al., 2018; Tillet et al., 2019), which apply heuristics or search but often struggle with irregular operators and hardware variability (Zheng et al., 2020a;b).

The rapid progress of large language models (LLMs) opens a new opportunity for kernel optimization. Beyond their ability to generate correct code, LLMs can be guided to reason about hardware trade-offs, adapt to profiling feedback, and iteratively refine implementations. However, prior work has mostly treated LLMs as single-shot code generators or simple refinement tools (Ouyang et al., 2025), which underutilizes their potential for structured exploration of the kernel design space. To build a more powerful agent, we identify and address three critical limitations in existing methods:

1. **Naive exploration strategy.** Current agents typically refine code linearly, learning only from the immediately preceding attempt. This simplistic process neglects the rich history of prior attempts and fails to effectively balance the exploration-exploitation trade-off, often getting trapped in local optima.
2. **Monolithic agent design.** Kernel optimization is a multifaceted task requiring distinct capabilities for planning, implementation, and reflection. By assigning all these responsibilities to a single, generalist LLM, current agents operate inefficiently.
3. **Planning-implementation gap.** We observe a failure mode particularly acute in this domain: LLMs frequently devise a correct high-level optimization plan (e.g., "apply memory tiling") but fail to translate it into valid low-level CUDA code. This gap stems from the relative scarcity of expert-level kernel code in the models' training data.

To address these limitations, we introduce STARK (*Strategic Team of Agents for Refining Kernels*), a novel framework for automated GPU-kernel optimization. Our contributions are threefold:

- **Collaborative multi-agent workflow.** We design a workflow with specialized agents for planning, coding, and reflection, mirroring an expert development cycle and overcoming the inefficiencies of monolithic designs.
- **Bridging the planning–implementation gap.** We propose two mechanisms—*grounded instruction* and *dynamic context windows*—that translate high-level strategies into precise, actionable code edits, ensuring robust coordination across agents.
- **Strategic search for refinement.** We incorporate a search policy that balances exploration and exploitation over prior attempts, enabling systematic discovery of strong kernels.

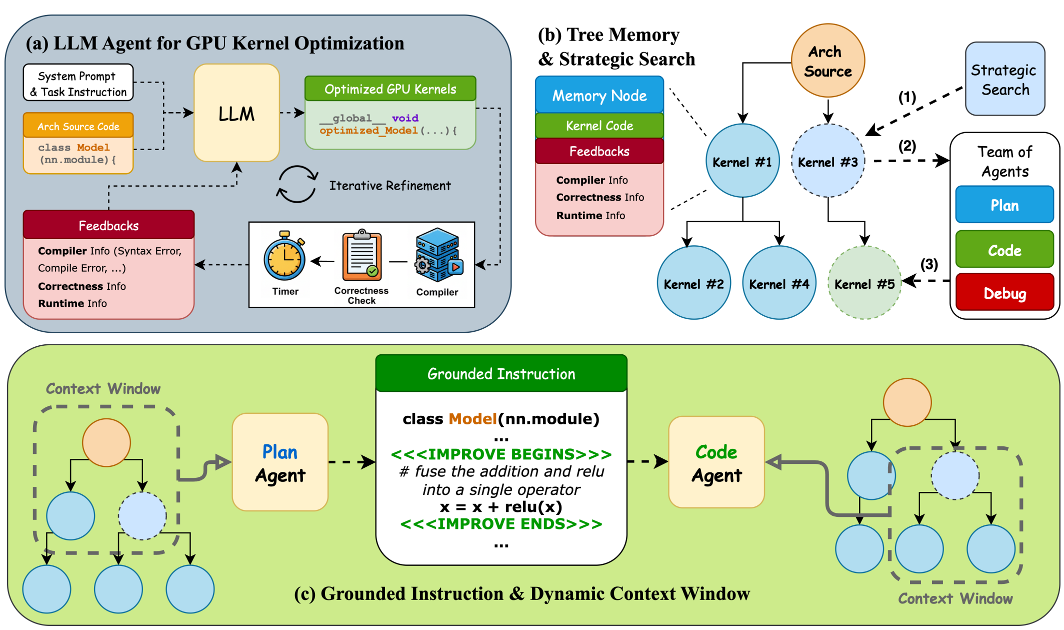

Figure 2: Overview of STARK. **(a)** Prior LLM-based kernel optimizers rely on a monolithic agent with local iterative refinement. **(b)** STARK replaces this with a collaborative multi-agent workflow (plan/code/debug) coupled with strategic search over a tree memory. **(c)** The plan agent issues *grounded instructions* that anchor edits to code spans; *dynamic context windows* surface role-specific history; and the debug agent repairs failures. See Section 4 for details.

We evaluate our framework on **KernelBench** Ouyang et al. (2025), a benchmark designed to assess LLM-based GPU kernel optimization. Experiments show that combining these improvements leads to an agent system significantly more competitive than the baseline agents in both runtime performance and success rate across diverse kernel problems, authoring competitive kernels for the challenging problems in KernelBench where the baseline agents struggle to even find a working solution. Notably, `STARK` achieves more than $10\times$ speedup over kernels produced by the baseline agents (i.e., the optimized kernels run in under one-tenth the time of the baseline.). Overall, our work suggests that LLM-driven agents represent a promising step toward fully automated GPU kernel optimization.

2 Related Work

Due to space constrain, we only review the most relevant prior work here and defer the complete discussion to Appendix E.

The optimization of GPU kernels has progressed from empirical auto-tuning frameworks that perform black-box parameter searches (van Werkhoven, 2019; Nugteren & Codreanu, 2015) and compiler-based approaches with static heuristics (Yang et al., 2010), to the use of machine learning (ML). ML-based techniques have been used to replace hand-tuned heuristics in production compilers (Trofin et al., 2021), learn cost models to guide optimization (Chen et al., 2018), and even learn directly from raw source code without manual feature engineering (Cummins et al., 2017). A significant leap was the use of deep reinforcement learning to discover fundamentally new algorithms, as demonstrated by AlphaTensor's success in finding faster matrix multiplication methods (Fawzi et al., 2022). While powerful, these prior works either optimize within a fixed search space or operate in purely formal domains. Our work addresses these limitations by operating directly on source code to implement novel, structural changes.

The emergence of powerful Large Language Models (LLMs) has revolutionized programmatic interaction with source code, demonstrating a remarkable proficiency in generating code for diverse applications from competitive programming to compiler testing (Gu, 2023; Zhong & Wang, 2024; Jain et al., 2025). This capability has catalyzed a paradigm shift away from single-shot code generation and toward the development of autonomous LLM agents. An agent enhances a base LLM with planning, memory, and tool-use capabilities to direct its own workflow (Weng, 2023). The success of frameworks like SWE-agent in independently resolving complex GitHub issues has validated the power of this approach for software engineering (SWE) (Yang et al., 2024). While the application of LLM agents to SWE is a burgeoning field of research (Yang et al., 2024; Antoniades et al., 2024; Yang et al., 2025), their potential in the specialized domain of GPU kernel optimization remains largely unexplored. To fill this gap, we designed `STARK`, an agent framework with capabilities tailored to the unique challenges of this domain.

3 Preliminary

3.1 LLMs and Autoregressive Generation

Given an input sequence $x = (x_1, x_2, \ldots, x_n)$ (e.g., the task instruction) as the context, an LLM p_θ with parameters θ generates an output sequence $y = (y_1, y_2, \ldots, y_m)$ where $y_t \in \mathcal{Y}, t \in \{1, \ldots, m\}$ are tokens. Pretrained on a massive corpus of text, LLMs autoregressively generate the next token y_t conditioning on x and all the previously generated token $y_{<t} = (y_1, \ldots, y_{t-1})$. Specifically, at each time t, the LLM first computes the logits $z_\theta(y|y_{<t}, x)$ for each token y in the vocabulary $\mathcal{Y}$ and generate y_t following the conditional distribution

$$p_\theta(y_t|y_{<t}, x) = \frac{\exp(z_\theta(y_t|y_{<t}, x)/\tau)}{\sum_{y' \in \mathcal{Y}} \exp(z_\theta(y'|y_{<t}, x)/\tau)}. \tag{1}$$

The temperature parameter $\tau > 0$ modulates the randomness of an LLM's output. Higher values of τ flatten the next token distribution in Equation 1, encouraging creative and diverse responses. Conversely, lower values sharpen the distribution, promoting deterministic and high-fidelity outputs.

This trade-off is critical in complex tasks, as different sub-problems demand different behaviors. For instance, **planning** and **exploration** benefit from a high temperature to generate novel strategies, whereas tasks requiring precision and factual correctness, such as **code implementation**, necessitate a low temperature to ensure reliability. A single agent with a fixed temperature is ill-equipped to handle this dichotomy. This observation is a core motivation for `STARK`'s multi-agent design, which allows specialized agents to operate at distinct temperatures tailored to their roles, i.e., a high τ for the creative plan agent and a low τ for the precise code agent.

3.2 KernelBench

KernelBench Ouyang et al. (2025) is a recently proposed benchmark specifically designed for assessing LLM-based GPU kernel optimization. Unlike prior evaluations that focus only on code correctness or small-scale operator tests, KernelBench provides a principled and reproducible testbed that measures both correctness and runtime efficiency across a broad spectrum of GPU workloads. KernelBench comprises a suite of optimization tasks, categorized into three difficulty levels. For each task, the objective is to create a custom GPU kernel that is functionally equivalent to a provided PyTorch reference implementation while minimizing its wall-clock execution time. See an example of the KernelBench task in Appendix D.

Specifically, **Level 1** tasks focus on single, common operators such as matrix multiplication and convolution, serving as a baseline for fundamental optimization capabilities; **Level 2** tasks comprise tasks with multiple operators fused into a single kernel, testing the ability to manage more complex dataflows and scheduling; **Level 3** tasks represent the highest difficulty, featuring popular full ML architectures such as the ResNet (He et al., 2016) and LSTM (Hochreiter & Schmidhuber, 1997), which involve highly irregular computations and intricate memory access patterns that are challenging for both human experts and automated systems to optimize effectively.

4 STARK: Strategic Team of Agents for Refining Kernels

Framework Overview. We now present `STARK`, an agentic framework for GPU-kernel optimization. `STARK` organizes kernel refinement into three layers: (i) a *multi-agent workflow* that separates planning, coding, and debugging, (ii) *coordination mechanisms* with *grounded instruction* to anchor planned edits to concrete code spans and *dynamic context windows* that surface role-specific history (e.g., prior attempts, failures, profiler feedback) to each agent, and (iii) a *strategic search* policy that balances exploration and exploitation across iterative attempts. Notably, multi-agent workflow and grounded instruction improve reliability even under a single-attempt budget, whereas dynamic context windows and strategic search deliver most of their gains when multiple attempts are allowed. Figure 2 provides an overview; the following subsections detail each component in turn.

4.1 Multi-Agent Collaboration

Optimizing GPU kernels is inherently multifaceted and mirrors expert team workflows. A single agent typically fails to balance correctness, performance, and exploration across a vast, irregular design space. In particular, *strategy discovery* (e.g., fusion, vectorization, shared-memory tiling) benefits from higher-temperature generation that encourages diversity whereas *strategy realization*, i.e., committing those ideas to code, requires low-temperature precision to avoid errors. We therefore adopt a multi-agent framework that enables role specialization through LLMs.

Multi-Agent Design (MAD). Specifically, `STARK` decomposes kernel optimization into three roles – *plan*, *code*, and *debug*. Using a role-specific context window (Section 4.4) with selected prior attempts and execution outcomes, the **plan** agent proposes targeted transformations to either the source kernel or a candidate chosen by the strategic search policy (Section 4.2), emitting *grounded instructions* (Section 4.3) that anchor edits to explicit code spans. The **code** agent consumes grounded instructions and translates them into executable GPU-kernel code, conditioning on its own context window to improve adherence and code quality. The **debug** agent repairs promising but failing candidates by consulting the plan agent's instructions and compiler/runtime diagnostics, producing a working kernel that realizes the intended transformation.

Benefits of MAD. Role specialization lets each agent use prompts and base LLMs matched to its objective. In our instantiation, we choose `Claude Sonnet 4` with temperature τ=0.8 for the plan agent to encourage strategy diversity, and the same model with τ=0.1 for the code and debug agents to enforce precision. Despite this simple setup, MAD already performs strongly (see Section 5). We underscore that because the design is modular, we can swap in planners with richer kernel-optimization priors or code-specialist reasoning models to further improve results. In addition, modularity also exposes bottlenecks. We observe that the dominant bottleneck is code-synthesis fidelity: LLMs often need multiple attempts to faithfully implement a given instruction. Finally, MAD makes targeted post-training straightforward: we can fine-tune the base LLM for a specific agent (e.g., the code agent) *without* affecting the others, improving stability and predictability. However,, a systematic study of agent-specific post-training is orthogonal to our core contributions and is left to future work.

4.2 Strategic Search with Tree Memory

Prior LLM-driven kernel optimizers typically use either *best-of-K* sampling that generates multiple candidates independently and select the fastest correct one or *iterative refinement* which repeatedly edits the latest kernel (Ouyang et al., 2025). However, best-of-K is unguided and wasteful: all the new attempts ignore feedback from earlier attempts and repeatedly probe redundant regions of the design space. On the other hand, iterative refinement is feedback-aware but *myopic*: by building only on the most recent candidate, it is prone to getting trapped in narrow, suboptimal basins.

To address these limitations, `STARK` reframes kernel optimization as **strategic search** over a persistent **tree memory**. We maintain a search tree T whose nodes store candidates and their observations (runtime, correctness, and compiler diagnostics). The root represents the source architecture; each edge corresponds to applying a grounded instruction from the plan agent and realizing it via the code agent (or repairing via the debug agent). Each node n is assigned a score $s(n)$ reflecting competitiveness; in our implementation we use the straightforward kernel runtime as $s(n)$ and treat *lower is better*. For kernels that are incorrect or failing to compile, we give them scores of $+\infty$. At each step, we (1) *select* a node to expand using a strategic policy, (2) *expand* by invoking the plan/code (or debug) agents to produce a child candidate, (3) *evaluate* for correctness and runtime, and (4) *record* results in T to inform subsequent selections. This converts ad-hoc trial-and-error into a directed, feedback-driven process.

Policy choice and an adapted ϵ-greedy rule. We compared representative search policies including Monte-Carlo Tree-Search (MCTS), evolutionary, greedy, and ϵ-greedy policies and found that ϵ-greedy consistently performs best under the same budget constraint. Importantly, we observe that kernel optimization poses domain-specific challenges that are root dominance (it is very challenging to even outperform the source architecture in the root node) and frequent compilation/runtime failures. To address these challenges, we adapt the canonical rule as follows: (1) **Root throttling:** cap the number of direct children of the root at n_{root} to avoid redundant first-hop edits; once the cap is reached, the root is ineligible for selection; (2) **Dead-branch pruning:** if a node has more than n_{child} children and all current children fail, mark the node ineligible to prevent wasting trials; (3) **High exploration rate:** use a relatively large ϵ (empirically 0.3–0.4) to counteract local traps; (4) **Leaf-biased exploration:** with probability ϵ, sample uniformly from expandable leaves (not only failing nodes), encouraging discovery beyond the immediate failure set.

4.3 Grounded Instruction

We introduce grounded instruction for kernel enhancement. The plan agent must not only propose an optimization, but also insert **explicit span anchors** in the kernel source that mark exactly where the change should occur. Each anchor is a short, machine-checkable tag (i.e, `<<<IMPROVE BEGINS>>>` ... `<<<IMPROVE ENDS>>>`) wrapped around the target site, such as a load/store, loop body, or the launch configuration. The code agent consumes this annotated scaffold and resolves each anchor by emitting concrete CUDA that realizes the instruction. Grounded instruction tightens plan–code alignment, curbs hallucinated guidance, and narrows the coder's search space. It also improves traceability: every proposal leaves a visible, verifiable footprint in the final code. In practice, we observe fewer misinterpretations and markedly fewer faulty kernels. Despite its simplicity, the mechanism is especially effective on Level 3 KernelBench tasks with deeper architectures (e.g., VGG).

4.4 Dynamic Context Window

Past attempts provide rich, actionable signals for subsequent decisions, but different agents benefit from different *views* of this history. We therefore maintain a *dynamic, agent-specific context window* that is rebuilt at each selection step for different agents. See Figure 3 for a visual demonstration. Throughout this section, let node i be the node selected by the search policy defined in Section 4.2. We use $\mathcal{W}(i)$ to denote the context window containing a subset of historical attempts and their evaluation outcomes (e.g., compiler information and runtime). As we always include the source architecture as part of the prompt for agents, $\mathcal{W}(i)$ always includes the root node n_{root}. For a naive search algorithm without dynamic context window, $\mathcal{W}(i) = \{i, n_{\text{root}}\}$ only includes node i in addition to the root.

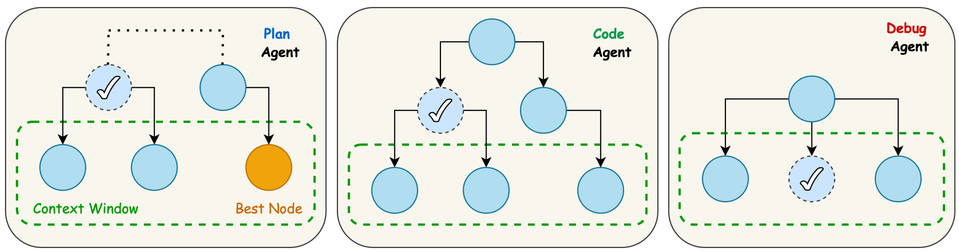

Figure 3: Dynamic Context Window. Nodes with ✓'s represent selected nodes.

Tree relations. We use tree relations to build agent-specific context windows. Let $p(i)$ be the parent of node i. Define the *siblings* of i as $\mathcal{S}(i) = \{\, j : p(j) = p(i) \,\}$. Moreover, define the set of child nodes of a node i as $\mathcal{D}(i)$. We also maintain a small global leaderboard $\mathcal{C}$ of top-performing nodes.

Plan agent (local & contrastive global context). For a selected node i, the plan agent conditions on a context window $\mathcal{W}_{\text{plan}}(i)$ that aggregates node i's children and a small set of global leaders from the leaderboard $\mathcal{C}$. Formally,

$$\mathcal{W}_{\text{plan}}(i) \;=\; \{i, n_{\text{root}}\} \,\cup\, \mathcal{D}(i) \,\cup\, \text{Top}\!-\!r(\mathcal{C}),$$

where $\mathcal{D}(i)$ contains all evaluated children of i with their observations, and $\text{Top}\!-\!r(\mathcal{C})$ returns the r highest-scoring distinct kernels from the global leaderboard (excluding i's subtree) to discourage duplication.

This design serves three purposes. (i) **reflection**: the plan agent can revise or stack its prior instructions rather than rediscovering them; (ii) **ambition calibration**: top competitors prevent redundant exploration and provide transferable motifs such as warp-shuffle reductions, vectorized LD/ST, and shared-memory tiling; (ii) **capability estimation**: by inspecting how past instructions were realized or failed by the code agent, the next instruction is adapted to what the code agent can reliably execute, *improving first-pass success and avoiding instructions beyond current ability*. To achieve this, we explicitly require the plan agent to adapt its instruction to the code agent's demonstrated capabilities observed in $\mathcal{D}(i)$.

Code agent (extended context). For kernel code emission at node i, the code agent conditions on

$$\mathcal{W}_{\text{code}}(i) \;=\; \{i, n_{\text{root}}\} \,\cup\, \mathcal{D}(i) \,\cup\, \{j : p(j) \in S(i)\}.$$

The nodes in $\{j : p(j) \in S(i)\}$ are essentially the children of node i's siblings. Our insight is that these nodes typically share near-identical scaffolds with node i from a common planning lineage, so successful patches and micro-optimizations transfer with high probability; conversely, seeing failures in closely related contexts helps the coder avoid repeating the same mistakes. Hence, this extended window serves two aims: (i) **reduce implementation errors** by letting the coder imitate successful patches from closely related scaffolds and avoid previously observed failure modes; (ii) **surface stronger implementations** by transferring micro-optimizations (e.g., warp-shuffle motifs, vectorized LD/ST, shared-memory tiling) that have already worked on cousin nodes.

Debug agent (local context). For fault repair, we construct the context window for the debug code as

$$\mathcal{W}_{\text{debug}}(i) \;=\; \{i, n_{\text{root}}\} \,\cup\, \mathcal{S}(i),$$

We choose this design mainly for two reasons. Most fixes are structural and local, e.g., off-by-one guards, stride/indexing alignment, launch-parameter tweaks, or shared-memory sizing often transfer directly among siblings that share the same scaffold. Moreover, restricting the window to $\mathcal{S}$ avoids distracting the debug agent with globally unrelated kernels, improving precision and reducing hallucinated edits.

4.5 Framework Overview

Here we provide an overview of our framework STARK and describe its execution process. Algorithm 1 presents its pseudocode.

At a high level, STARK repeatedly (i) selects a promising node (a prior attempt) from a search tree, (ii) builds agent-specific context windows from local history and global leaders, (iii) asks the *planning agent* to propose a concrete optimization along with *grounded instruction* anchors inserted into a scaffold, (iv) asks the *code agent* to realize those anchors into an executable kernel, (v) if the selected node has a problematic kernel, we build the debugger's dynamic context window and request a minimal fix. The new attempt is evaluated, appended as a child node, and the leaderboard $\mathcal{C}$ is updated. We repeat this process until we reach a pre-specified max attempts B.

Algorithm 1 STARK: Strategic Team of Agents for Refining Kernels

Require: Budget B (max attempts), selection policy π_{select} (adapted ε-greedy), leaderboard size r

Initialize search tree T with root n_{root} (PyTorch reference)
Initialize leaderboard $\mathcal{C} \leftarrow \{n_{\text{root}}\}$
for $t = 1, 2, \ldots, B$ **do**
  $i \leftarrow \pi_{\text{select}}(T, \mathcal{C})$ ▷ pick a node to refine
  if HasBug(i) **then** ▷ compile fail or unit-test fail recorded at i
    $\mathcal{W}_{\text{debug}}(i) \leftarrow$ BuildContextDebug(i, T)
    $\mathsf{kernel}' \leftarrow$ DebugAgent($\mathcal{W}_{\text{dbg}}$, i.kernel, i.logs)
    (ok, correct, runtime, logs) $\leftarrow$ Evaluate(kernel') ▷ compile, correctness check, timing
    (plan, anchors) $\leftarrow$ (i.plans, i.anchors)
  else
    $\mathcal{W}_{\text{plan}}(i) \leftarrow$ BuildContextPlan$(i, T, \mathcal{C})$
    (plan, anchors) $\leftarrow$ PlanAgent($\mathcal{W}_{\text{plan}}$)
    $\mathcal{W}_{\text{code}}(i) \leftarrow$ BuildContextCode(i, T)
    $\mathsf{kernel}' \leftarrow$ CodeAgent($\mathcal{W}_{\text{code}}$, plan, anchors)
    (ok, correct, runtime, logs) $\leftarrow$ Evaluate(kernel')
  end if
  $j \leftarrow$ AddChild(T, i, kernel', plan, anchors, ok, correct, runtime, logs)
  $\mathcal{C} \leftarrow$ UpdateLeaders($\mathcal{C}$, j, r)
end for
return Best($\mathcal{C}$) ▷ fastest correct, grounded kernel

5 Experiments

We use KernelBench (Ouyang et al., 2025), a recently proposed benchmark consisting of comprehensive and challenging GPU kernel tasks, to validate the effectiveness of our proposed approaches.

Baselines and Metrics. We compare our framework STARK with the following list of approaches:

- **Torch Eager**: the out-of-box PyTorch modules without any compilation or optimization.
- **Torch Compile** : We use `torch.compile` to produce optimized versions of the given PyTorch modules. While `torch.compile` offers different compilation modes, we compare to two of the most representative and competitive ones – **default** and **max-autotune**.
- **Sampling Agent**: the single agent framework originally proposed and used by KernelBench to evaluate the difficulty of the tasks in KernelBench and the ability of LLMs to

write efficient kernels. This agent repeatedly samples responses when given the source model to optimize and chooses the best generated custom kernel as the solution.

- **Reflexion Agent**: this agent follows the Reflexion paradigm (Shinn et al., 2023), where at each optimization step, it tries to update its last attempt using its corresponding observations such as the compiler and runtime information.

We report the following metrics to comprehensively understand the agents' performances: (i) $\mathbf{Fast}_1$ rate is the percentage of the problems for which the agent can generate kernels that are *at least* as fast as the torch baselines; (ii) **Success** rate represents the percentage of the problems for which the agent can generate compiled and correct kernels; (iii) **Speed**: To better understand how good the generated kernels are, we also report the average speed across all tasks.

Comparison with Torch Baselines. In Table 1, we present the results about success rate, $\mathbf{Fast}_1$ rate and speed over all 3 levels of KernelBench challenges. For each task, we let all agents to have a maximum of $B = 30$ attempts. Due to limited computation resource, we evaluate on the representative subset of KernelBench (Ouyang et al., 2025). We use `Claude Sonnet 4` as the base LLMs for all the LLM-based baselines and our agents. Due to space constraint, we defer implementation and evaluation details to Appendix B.

The results in Table 1 demonstrate that our proposed framework, `STARK`, consistently outperforms both the Sampling and Reflexion baselines across all KernelBench difficulty levels. At Level 1, `STARK` not only achieves a perfect 100% success rate but also delivers up to a $3.0\times$ speedup over Torch Eager baselines, while Sampling and Reflexion agents frequently generate kernels that are slower than the baselines. This advantage becomes even more pronounced at Level 2, where the complexity of the kernels increases. Here, `STARK` maintains a perfect success rate and achieves speedups of $2.7\times$, whereas the Reflexion agent, despite attaining 100% correctness, produces kernels that run slower than the baseline. At Level 3, which involves the most irregular and challenging tasks, both Sampling and Reflexion degrade significantly, with success rates falling and runtimes dropping below baseline. In contrast, `STARK` continues to maintain full success while producing kernels that outperform the Torch implementations by up to $1.6\times$. These results highlight that `STARK` not only generates correct kernels but also delivers substantial performance improvements, even as task difficulty increases.

		Torch Eager		Default		Max-autotune	
Level 1	**Success** ↑	$\mathbf{Fast}_1$ ↑	**Speed** ↑	$\mathbf{Fast}_1$ ↑	**Speed**↑	$\mathbf{Fast}_1$ ↑	**Speed**↑
Sampling Agent	57.1%	14.3%	0.81×	7.1%	0.46×	7.1%	0.81×
Reflexion Agent	92.6%	28.6%	1.24×	14.3%	0.57×	35.7%	0.92×
`STARK`	100%	71.4%	3.03×	78.6%	2.37×	78.6%	2.76×
Level 2	**Success**	$\mathbf{Fast}_1$	**Speed**	$\mathbf{Fast}_1$	**Speed**	$\mathbf{Fast}_1$	**Speed**
Sampling Agent	87.5%	50%	1.06×	37.5%	0.91×	37.5%	0.91×
Reflexion Agent	100%	75%	0.88×	62.5%	0.78×	62.5%	0.78×
`STARK`	100%	100%	2.69×	87.5%	2.51×	87.5%	2.52×
Level 3	**Success**	$\mathbf{Fast}_1$	**Speed**	$\mathbf{Fast}_1$	**Speed**	$\mathbf{Fast}_1$	**Speed**
Sampling Agent	100%	50%	0.87×	12.5%	0.67×	12.5%	0.66×
Reflexion Agent	67.5%	25%	0.79×	12.5%	0.62×	12.5%	0.61×
`STARK`	100%	87.5%	1.58×	87.5%	1.27×	87.5%	1.26×

Table 1: Performance of LLM Agents on the KernelBench Tasks. $\mathbf{Fast}_1$ represents the percentage of problems for which the agent can generate custom kernels that are correct and as fast as the Torch baselines (higher is better). Speed is computed as the ratio of the kernel runtime of the baseline to that of the generated kernel.

Comparison between Agents. We investigate deeper into the behavior of our agent `STARK` with the two baseline agents to better understand their optimization behaviors. A deeper analysis of compile and correctness rates, shown in Table 2, provides further insight into why `STARK` succeeds where baselines struggle. While all agents achieve relatively high compile rates (mostly above 80%), the fraction of kernels that are both compilable and correct varies widely. The Sampling agent, for example, compiles over 90% of its outputs on Level 1 but only 43% of these are functionally correct. Reflexion improves correctness slightly through iterative refinement, but its correctness rate remains below 55% at all levels. In contrast, STARK achieves the highest correctness rates across the board, reaching 61.2% on Level 2 tasks. This suggests that STARK 's structured planning and feedback-driven refinement not only increase the chance of generating efficient kernels but also reduce wasted attempts on invalid or incorrect code. Finally, Figure 1 highlights the dramatic runtime improvements of STARK relative to baseline agents. On Level 1 tasks, STARK achieves over a $10\times$ speedup compared to Sampling and a $13.7\times$ speedup over Reflexion. On Level 2, these gains rise as high as $16\times$, and even at the most challenging Level 3 tasks STARK maintains 5–$6\times$ improvements. These relative gains indicate that while baselines occasionally achieve correctness, they rarely deliver true runtime efficiency. By contrast, STARK 's ability to jointly optimize for correctness and speed allows it to close both gaps simultaneously. Taken together, these findings confirm that multi-agent collaboration and strategic search are key enablers for scaling LLMs to the demands of GPU kernel optimization.

	Compile Rate↑			**Correct Rate↑**		
KernelBench Level	1	2	3	1	2	3
Sampling Agent	90.8%	97.0%	84.9%	43%	44.0%	15.1%
Reflexion Agent	86.0%	86.2%	78.9%	48.3%	53.4%	28.4%
STARK	84.5%	90.7%	83.4%	50.6%	61.2%	35.5%

Table 2: Percentages of Successfully Compiled and Correct Kernels.

Ablations. We ablate the agentic components of our system. We compare (i) **Search Agent**, which is a single-agent model equipped with our strategic search, and (ii) **MA-only**, which employs the multi-agent workflow (plan/code/debug with grounded instruction) using best-of-K sampling instead of search. As shown in Table 3, both variants outperform the *Sampling* baseline, confirming that each component helps. When combined in STARK, the effects compound: strategic search exploits the structured proposals produced by the multi-agent workflow, yielding the largest gains.

	Torch Eager		**Default**		**Max-autotune**	
	Fast$_1$ ↑	**Speed↑**	**Fast$_1$ ↑**	**Speed↑**	**Fast$_1$ ↑**	**Speed↑**
Sampling Agent	50%	0.87×	12.5%	0.67×	12.5%	0.66×
Search Agent	67.5%	0.89×	25%	0.71×	25%	0.70×
MA-Only	67.5%	1.11×	25%	0.92×	25%	0.91×
STARK	87.5%	1.58×	87.5%	1.27×	87.5%	1.26×

Table 3: Ablation on the Proposed Agentic Features.

6 Conclusion

In this work, we introduced an agentic framework for GPU kernel optimization that combines multi-agent role play, dynamic context management, and strategic search. Our evaluation on KernelBench demonstrated that the proposed framework consistently outperforms baseline methods in both success rate and runtime efficiency, across tasks of varying complexity. These results highlight the value of moving beyond single-agent or unguided sampling approaches, and point to the promise of collaborative, feedback-driven optimization. Looking forward, we envision that agentic LLM frameworks will play an increasingly important role in automated system optimization. Extending our approach to broader classes of operators, diverse hardware architectures, and cross-kernel scheduling decisions are natural directions for future research. More broadly, our work suggests that multi-agent LLMs can meaningfully accelerate the co-design of AI algorithms and infrastructure, pushing the boundaries of what is possible in efficient large-scale computation.

Ethics Statement

This study complies with the ICLR Code of Ethics. All datasets employed are publicly available and open-source under licenses that permit research use. No private or personally identifiable information was accessed, and no new data were collected from human subjects. The research does not pose privacy, security, or fairness concerns. The authors declare no conflicts of interest and no external sponsorship.

Reproducibility Statement

We document all algorithmic and implementation details in the paper and appendix, including the exact prompts for every agent, full hyperparameters, and ablation settings. Upon acceptance, we will release an open-source repository with configuration files, an environment file and Dockerfile, and step-by-step commands to recreate every table and figure.

References

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.

Ian Buck. Gpu program optimization. In Matt Pharr (ed.), *GPU Gems 2*. NVIDIA, 2008.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Hangu Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232. IEEE, 2017.

DeepMind. Discovering faster matrix multiplication algorithms with alphatensor. `https://www.deepmind.com/blog/discovering-faster-matrix-multiplication-algorithms-with-alphatensor`, 2022. Accessed: 2024.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco JR Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Qiuhan Gu. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 2201–2203, 2023.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Stijn Heldens and Ben van Werkhoven. Kernel launcher: C++ library for optimal-performance portable cuda applications. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 556–565. IEEE, 2023.

Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. Baco: A fast and portable bayesian compiler optimization framework, 2023. URL `https://arxiv.org/abs/2212.11142`.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=chfJJYC3iL`.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024.

Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning gemm kernels for the fermi gpu. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2054, 2012.

LangChain Inc. Langgraph: Build resilient language agents as graphs, 2025. URL `https://github.com/langchain-ai/langgraph`. Version 0.6.7; accessed 2025-09-24.

Teodor V. Marinov, Alekh Agarwal, and Mircea Trofin. Offline imitation learning from multiple baselines with applications to compiler optimization, 2024. URL `https://arxiv.org/abs/2403.19462`.

Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 846–851. IEEE, 2015.

NVIDIA Developer. Improving gemm kernel auto-tuning efficiency on nvidia gpus with heuristics and cutlass 4.2. `https://developer.nvidia.com/blog/improving-gemm-kernel-auto-tuning-efficiency-on-nvidia-gpus-with-heuristics-and-cu` 2024. Accessed: 2024.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=yeoN1iQT1x`.

Ari Rasch, Markus Haidl, and Sergei Gorlatch. Atf: A generic auto-tuning framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications (HPCC)*, pp. 64–71. IEEE, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Philippe Tillet, Hsiang-Tsung 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.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. In *Future Generation Computer Systems*, volume 90, pp. 347–358. Elsevier, 2019.

Lilian Weng. Llm-powered autonomous agents. `https://lilianweng.github.io/posts/2023-06-23-agent/`, 2023. Accessed: 2024.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Thirty-eighth Conference on Neural Information Processing Systems*, 2024.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 86–97. ACM, 2010.

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 *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 863–879, 2020a.

Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 859–873, 2020b.

Li Zhong and Zilong Wang. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 21841–21849, 2024.

A Use of Large Language Models (LLMs)

Large Language Models (LLMs) were used solely as general-purpose assistive tools to improve the clarity and readability of the manuscript. Specifically, we used an LLM to help rephrase and polish text that we had already drafted.

B Implementation and Evaluation

Agent Implementation. We use `Claude Sonnet 4` as the base LLMs for all agent baselines and agents of `STARK`. For both sampling and reflexion agents, we follow KernelBench to set temperature $\tau = 0.7$ during generating, with other generation parameters such as top-p set to the default value. For `STARK`, we use `Claude Sonnet 4` with temperature $\tau = 0.8$ for the plan agent, and $\tau = 0.1$ for the code and debug agents. For all tasks, all agent baselines and `STARK` have a maximum of $B = 30$ attempts to optimize each task. Regarding the hyperparameters of `STARK`, we choose the root throtting number to be 5, dead-branch pruning number to be 3, $\epsilon = 0.3$ for the search, $r = 2$ for the leaderboard $\mathcal{C}$. To prevent exploding context to the LLMs, we set an upper limit for the number of nodes in the dynamic context window: whenever the dynamic context window has more than 5 nodes, we randomly sample 5 from all the nodes in the window. We implement `STARK` with LangGraph (LangChain Inc., 2025).

Runtime Evaluation. We evaluate all the Pytorch baselines and LLM-generated kernels on the same NVIDIA A100 40GB GPU. We use the source code provided by KernelBench at its official repo[1] to benchmark the kernels' runtime. In particular, to ensure stable measurement, runtime is measured with CUDA events after warm-up runs using fixed input shapes; we choose a large number of 100 warm-up runs to ensure accurate measurement.

C Prompts

Our prompts follow the templates of KernelBench (Ouyang et al., 2025), which has four components: *system message*, *in-context example*, *architecture source code*, and *instruction*.

As we have multiple agents in `STARK` with different tasks, they require different prompts to fulfill their tasks. Specifically, we reuse the system message and in-context example from KernelBench for all agents and include the architecture source code regardless of which node is selected for optimization. To motivate the agents to use the already optimized modules such as cuBLAS, we include an additional instruction in the system message to consider using existing highly optimized kernels. We show the system prompt in Figure 7, the in-context example in Figures 11 and 12. In addition, we include the information within the *dynamic context window* and *role-specific instructions* for different agents. See Figure 4 for the prompt template of `STARK`. We show the role-specific instruction in Figures 8, 9, and 10.

D Example KernelBench Tasks

Here we show three examples of KernelBench tasks, one for each level. See Figures 13, 14, and 15 for example tasks in Level 1, 2, and 3. We refer interested readers to Ouyang et al. (2025) for the complete list.

E Related Work

Optimizing GPU kernels to extract maximum performance from underlying hardware is a long-standing and formidable challenge. The vast, non-convex, and hardware-specific search space of possible kernel implementations necessitates sophisticated optimization strategies. The evolution of these strategies can be broadly categorized into three paradigms: empirical auto-tuning, compiler- and model-driven optimization, and most recently, generative approaches using Large Language

[1] `https://github.com/ScalingIntelligence/KernelBench`

```
{System Message}

Here's an example to show you the syntax of inline embedding
custom CUDA operators in torch: The example given architecture is:
{Example Architecture Source Code}
The example new arch with custom CUDA kernels looks like this:
{Example New Architecture Source Code}

You are given the following architecture:
{Architecture Source Code}

Here is your latest attempt:
{Source Code of the Selected Node}

[Dynamic Context Window] You should use the following observations
 regarding your historical attempts to provide better
implementations:
- Learn from the failed examples to avoid bugs and write
successful kernels.
- Learn from the successful examples to design improved kernels.

**Kernel Source Code #1**
{Source Code of Historical Attempt}

**Compiler Observation**
{Compiler Log}

**Kernel Execuation Result**
{Runtime or Correctness Error}

**Kernel Source Code #2**
[...skipped]

{Role-specific Instruction}
```

Figure 4: Prompt Template for Agents.

Models (LLMs). Our work builds upon this trajectory by introducing a fully autonomous agent that manages the entire optimization lifecycle.

E.1 Empirical and Compiler-Based Optimization

The foundational approach to GPU performance tuning is empirical auto-tuning, which treats the problem as a black-box search over a set of tunable parameters, such as thread block dimensions, memory tiling factors, and loop unrolling factors (van Werkhoven, 2019). Traditional methods often rely on an exhaustive or brute-force search, where thousands of potential kernel configurations are generated, compiled, and benchmarked to identify the top performer (Kurzak et al., 2012).While effective, this process is prohibitively time-consuming; for instance, an exhaustive search for a single GEMM kernel can take over 700 minutes to complete (NVIDIA Developer, 2024).

To mitigate this cost, heuristic-driven methods prune the search space. NVIDIA's `nvMatmulHeuristics`, for example, uses a predictive model to recommend a small subset of high-potential configurations, achieving near-optimal performance in a fraction of the time (NVIDIA Developer, 2024). Frameworks like Kernel Tuner (van Werkhoven, 2019), ATF (Rasch et al., 2017), and CLTune (Nugteren & Codreanu, 2015) provide robust environments for orchestrating these searches and support more advanced strategies like Bayesian Optimization, which builds a probabilistic performance model to guide the search more intelligently (Hellsten et al., 2023; Heldens & van Werkhoven, 2023).

Concurrently, compiler-based approaches aim to automate optimization through a series of program transformations applied to an intermediate representation (Yang et al., 2010). GPU compilers em-

```
## System Message

You are an expert in writing efficient code.
You write custom CUDA kernels to replace the pytorch operators in the given architecture to get speedups.

You have complete freedom to choose the set of operators you want to replace. You may make the decision to replace some operators with custom CUDA kernels and leave others unchanged. You may replace multiple operators with custom implementations, consider operator fusion opportunities (combining multiple operators into a single kernel, for example, combining matmul+relu), or algorithmic changes (such as online softmax). You are only limited by your imagination.

You should consider using CUDA's existing highly optimized kernels and operations whenever appropriate. Try building on these optimized blocks and further improve it with your custom kernels.
```

Figure 5: System Message for All Agents.

```
## System Message

You act as an LLM agent specializing in GPU optimization. Your goal is to speed up a given PyTorch architecture by generating custom CUDA kernels to replace its existing operators.
You may freely choose which operators to target, whether to rewrite one operator, many operators, or none. You can pursue kernel fusion opportunities (e.g., folding linear layers and elementwise ops together) or restructure the computation algorithmically to achieve higher throughput or lower memory traffic.

There are no constraints on what transformations you may proposeinnovation is encouraged.

Use CUDAs existing highly optimized kernel libraries whenever advantageous; treat them as strong baselines that your custom kernels can refine, fuse, or extend to achieve even greater efficiency.
```

Figure 6: Alternate System Message $V1$ for STARK.

ploy passes for memory coalescing, data prefetching, vectorization, and loop optimizations to adapt naive code to the hardware architecture (Buck, 2008). While these approaches excel at finding optimal configurations within a predefined search space, they cannot fundamentally alter the kernel's algorithm. Our work introduces an agent that reasons about performance bottlenecks to implement novel, structural code changes, moving beyond simple parameter tuning.

E.2 Machine Learning for Code Optimization

Machine learning (ML) has emerged as a powerful tool to transcend the limitations of hand-crafted heuristics. Early work focused on using ML to make better decisions within existing compiler and tuning frameworks. Systems like TVM employ a learned cost model to predict the performance of kernel variants, guiding the search process and avoiding exhaustive empirical evaluation (Chen et al., 2018). More recent efforts have integrated ML directly into production compilers. Google's MLGO framework uses reinforcement learning (RL) to train policies for classic compiler optimizations like function inlining and register allocation, demonstrating significant improvements in code size and performance over decades-old, manually-tuned heuristics in LLVM (Trofin et al., 2021; Marinov

```
## System Message

You are a GPU kernel engineer tasked with accelerating a PyTorch model by selectively replacing its operators with custom CUDA kernels.
You have full discretion in deciding which operators should be rewritten, fused, or left as-is. You may implement multiple custom kernels, explore operator fusion (e.g., matmul + activation), or introduce more efficient algorithmic variants (such as online softmax or tiled reductions).

Your design space is unrestrictedcreativity and performance intuition should guide your choices.

Whenever beneficial, leverage CUDAs high-performance primitives (tensor cores, cutlass, cuBLAS, cuDNN, etc.). Build upon these optimized components and push performance further with your own implementations.
```

Figure 7: Alternate System Message $V2$ for STARK.

et al., 2024). These models can learn from massive code corpora and discover complex feature interactions that are opaque to human experts (Cummins et al., 2017).

A more profound application of ML has been in algorithmic discovery. DeepMind's AlphaTensor framed the search for faster matrix multiplication algorithms as a single-player game, using a deep RL agent based on AlphaZero to navigate the enormous search space of tensor decompositions (Fawzi et al., 2022). This approach successfully discovered novel, provably correct algorithms that outperform human-derived state-of-the-art methods, including improving upon Strassen's algorithm for 4×4 matrices for the first time in over 50 years (Fawzi et al., 2022; DeepMind, 2022). This work marked a critical shift from using ML to *configure* existing optimization strategies to using it to *invent* new ones from first principles. However, AlphaTensor operated in a clean, formal mathematical domain. Translating this power to the messy, syntactic, and hardware-constrained domain of GPU kernel programming presents a distinct challenge. Our work addresses this by employing an agent that operates directly on source code, navigating the complexities of syntax, compilation, and hardware-specific performance characteristics.

E.3 LLM-Powered Autonomous Agents

The capabilities of LLMs have given rise to a new paradigm of autonomous agents. An LLM agent uses a core LLM as its "brain" or controller, augmented with capabilities for planning, memory, and tool use to perform complex tasks autonomously (Weng, 2023). The key distinction from simple LLM prompting is the agent's ability to decompose a high-level goal into a sequence of manageable subtasks, execute them iteratively, and use reflection to gauge progress and self-correct. This agentic workflow involves the LLM interacting with an external environment through a set of tools, such as a code interpreter or a web search API, to gather information and perform actions. While this paradigm is powerful for general problem-solving, its application to specialized domains like software engineering requires tailored tools and reasoning processes. Our work specializes this agentic concept for the domain of performance optimization, which presents unique challenges not found in general-purpose agent tasks.

E.4 LLMs for Code Optimization and Generation

The advent of powerful LLMs has opened a new frontier in performance engineering. To grant LLMs greater autonomy, the agentic paradigm has been adapted specifically for software engineering. The success of systems like SWE-agent, which autonomously resolves complex bugs in large GitHub repositories, has demonstrated the viability of this approach (Yang et al., 2024). SWE-agent equips an LLM with a specialized Agent-Computer Interface (ACI) containing tools for file navigation, editing, and test execution, enabling it to perform long-horizon tasks far beyond the scope of

```
## Instruction

- Optimize the architecture named Model with custom CUDA operators!
- Give explicit and actonable advice to improve the efficiency, in terms of the GPU wall-clock time, of the architecture named Model.

- Give ONE advice of the top priority! Don’t over-request.
- Include necessary details such as how to change pointers and indices or how to achieve shared memory tiling so that your advice can be correctly implemented.

- After your advice, modify and return the given source code in the following way:
    - Identify the code block whose efficiency can be improved (that is where your advice should be implemented)
      and mark it with comments ’<<<IMPROVE BEGIN>>>’ at the beginning and ’<<<IMPROVE END>>>’ at the end
    - The markers ’<<<IMPROVE BEGIN>>>’ and ’<<<IMPROVE END>>>’ should be valid comments for the marked coding language. For example, when marking source code of custom kernels, you need to use comments for the C++ language as ’// <<<IMPROVE BEGIN>>>’ and ’// <<<IMPROVE END>>>’; when marking source code of Python, you should use ’## <<<IMPROVE BEGIN>>>’ and ’## <<<IMPROVE END>>>’
    - Add your advice as comments at the identified code block to help the following agent’s implementation
    - There will be another agent focusing on improving the efficiency of the identified code block.
    - Return the complete code block with the identified code block as its subpart.

- You should consider using CUDA’s existing highly optimized kernels and operations whenever appropriate. Try building on these optimized blocks and further improve it with your custom kernels.
- When presented with multiple prior attempts, you should consider exploration of more diverse optimization strategies.
- Pay careful attention to the implementation agent’s capability demonstrated from the historical implementations.
- Adjust your advice accordingly to ensure that it can successfully implement.
```

Figure 8: Instruction for the Plan Agent.

simple code generation (Yang et al., 2024; Jimenez et al., 2024). While these agents are a significant step towards autonomous software engineering, their focus has primarily been on functional correctness, such as bug fixing. Our work extends this agentic software engineering paradigm to the non-functional, performance-oriented domain of GPU kernel optimization. We introduce an agent that not only interacts with a codebase but also with hardware profiling tools, allowing it to autonomously diagnose performance bottlenecks, form hypotheses, and conduct experiments to iteratively improve kernel efficiency, thus acting as a true autonomous performance engineer.

```
## Instruction

Optimize the architecture named Model with custom CUDA operators!

- Think about the given advice from human experts and implement the ones that you believe are correct and you are confident implementing.
- Only focus on the code block marked with '<<<IMPROVE BEGIN>>>' and '<<<IMPROVE END>>>'
- Write custom cuda kernel to replace the pytorch operators within the marked code block to improve its efficiency
- Name your optimized output architecture ModelNew.
- Output the new code in codeblocks.
- Explain your implementation and how you follow the advice.
- Using the given tool to return your final structured answer.

Please generate real code, NOT pseudocode, make sure the code compiles and is fully functional.

NO testing code!
```

Figure 9: Instruction for the Code Agent.

```
## Instruction

Fix the issues of your implementation named ModelNew, which should improve efficiency of the source model named Model.

- ModelNew and Model should have the same functionality, that is, the same input-output mapping.
- The given architecture ModelNew ieither does not compile, or has run-time error, or has different functionality to the source Model.
- Use the given observations to infer bugs and then fix them.
- Explain how the bugs happen and how you fix it.
- Return the fixed bug-free code and name your optimized output architecture ModelNew.
```

Figure 10: Instruction for the Debug Agent.

```python
## Example Architecture Source Code
import torch
import torch.nn as nn
import torch.nn.functional as F


class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self, a, b):
        return a + b


def get_inputs():
    # randomly generate input tensors based on the model architecture
    a = torch.randn(1, 128).cuda()
    b = torch.randn(1, 128).cuda()
    return [a, b]


def get_init_inputs():
    # randomly generate tensors required for initialization based on the model architecture
    return []
```

Figure 11: In-context Example Architecture for All Agents.

```
## Example New Architecture Source Code

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.cpp_extension import load_inline

# Define the custom CUDA kernel for element-wise addition
elementwise_add_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void elementwise_add_kernel(const float* a, const float
* b, float* out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        out[idx] = a[idx] + b[idx];
    }
}

torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor
b) {
    auto size = a.numel();
    auto out = torch::zeros_like(a);

    const int block_size = 256;
    const int num_blocks = (size + block_size - 1) / block_size;

    elementwise_add_kernel<<<num_blocks, block_size>>>(a.data_ptr<
float>(), b.data_ptr<float>(), out.data_ptr<float>(), size);

    return out;
}
"""

elementwise_add_cpp_source = (
    "torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::
Tensor b);"
)

# Compile the inline CUDA code for element-wise addition
elementwise_add = load_inline(
    name="elementwise_add",
    cpp_sources=elementwise_add_cpp_source,
    cuda_sources=elementwise_add_source,
    functions=["elementwise_add_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)


class ModelNew(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.elementwise_add = elementwise_add

    def forward(self, a, b):
        return self.elementwise_add.elementwise_add_cuda(a, b)
```

Figure 12: In-context Optimized Example Architecture for All Agents.

```python
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a LogSoftmax activation.
    """
    def __init__(self, dim: int = 1):
        super(Model, self).__init__()
        self.dim = dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Applies LogSoftmax activation to the input tensor.

        Args:
            x (torch.Tensor): Input tensor of shape (batch_size,
                dim).

        Returns:
            torch.Tensor: Output tensor with LogSoftmax applied,
                same shape as input.
        """
        return torch.log_softmax(x, dim=self.dim)

batch_size = 4096
dim = 393216

def get_inputs():
    x = torch.rand(batch_size, dim)
    return [x]

def get_init_inputs():
    return []  # No special initialization inputs needed
```

Figure 13: Example KernelBench Level 1 Task.

```python
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Model that performs a matrix multiplication, division,
        summation, and scaling.
    """
    def __init__(self, input_size, hidden_size, scaling_factor):
        super(Model, self).__init__()
        self.weight = nn.Parameter(torch.randn(hidden_size,
            input_size))
        self.scaling_factor = scaling_factor

    def forward(self, x):
        """
        Args:
            x (torch.Tensor): Input tensor of shape (batch_size,
                input_size).
        Returns:
            torch.Tensor: Output tensor of shape (batch_size,
                hidden_size).
        """
        x = torch.matmul(x, self.weight.T)  # Gemm
        x = x / 2  # Divide
        x = torch.sum(x, dim=1, keepdim=True) # Sum
        x = x * self.scaling_factor  # Scaling
        return x


batch_size   = 1024
input_size   = 8192
hidden_size  = 8192
scaling_factor = 1.5

def get_inputs():
    return [torch.rand(batch_size, input_size)]

def get_init_inputs():
    return [input_size, hidden_size, scaling_factor]
```

Figure 14: Example KernelBench Level 2 Task.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class NewGELU(nn.Module):
    """
    Implementation of the GELU activation function currently in Google BERT repo (identical to OpenAI GPT).
    Reference: Gaussian Error Linear Units (GELU) paper: https://arxiv.org/abs/1606.08415
    """
    def __init__(self):
        super(NewGELU, self).__init__()

    def forward(self, x):
        return 0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x + 0.044715 * torch.pow(x, 3.0))))

class CausalSelfAttention(nn.Module):
    """
    A vanilla multi-head masked self-attention layer with a projection at the end.
    It is possible to use torch.nn.MultiheadAttention here but I am including an
    explicit implementation here to show that there is nothing too scary here.
    """

    def __init__(self, n_embd, n_head, attn_pdrop, resid_pdrop, max_seqlen):
        super().__init__()
        assert n_embd % n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(n_embd, 3 * n_embd)
        # output projection
        self.c_proj = nn.Linear(n_embd, n_embd)
        # regularization
        self.attn_dropout = nn.Dropout(attn_pdrop)
        self.resid_dropout = nn.Dropout(resid_pdrop)
        # causal mask to ensure that attention is only applied to the left in the input sequence
        self.register_buffer("bias", torch.tril(torch.ones(max_seqlen, max_seqlen))
                                     .view(1, 1, max_seqlen, max_seqlen))
        self.n_head = n_head
        self.n_embd = n_embd
        [...skipped]

class Model(nn.Module):
    """ an unassuming Transformer block """

    [...skipped]

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlpf(self.ln_2(x))
        return x

[...skipped]
```

Figure 15: Example KernelBench Level 3 Task.