CUDA Agent: Large-Scale Agentic RL for High-Performance CUDA Kernel Generation

Weinan Dai[1,2,3*], **Hanlin Wu**[1,2,3*], **Qiying Yu**[1,2,3*], **Huan-ang Gao**[1,2,3*], **Jiahao Li**[1], **Chengquan Jiang**[1], **Weiqiang Lou**[1], **Yufan Song**[1], **Hongli Yu**[1,2,3*], **Jiaze Chen**[1,3], **Wei-Ying Ma**[2,3], **Ya-Qin Zhang**[2,3], **Jingjing Liu**[2,3], **Mingxuan Wang**[1,3], **Xin Liu**[1], **Hao Zhou**[2,3†]

[1]ByteDance Seed [2]Institute for AI Industry Research (AIR), Tsinghua University
[3]SIA-Lab of Tsinghua AIR and ByteDance Seed

[*]Equal contributions, [†]Corresponding Authors

Abstract

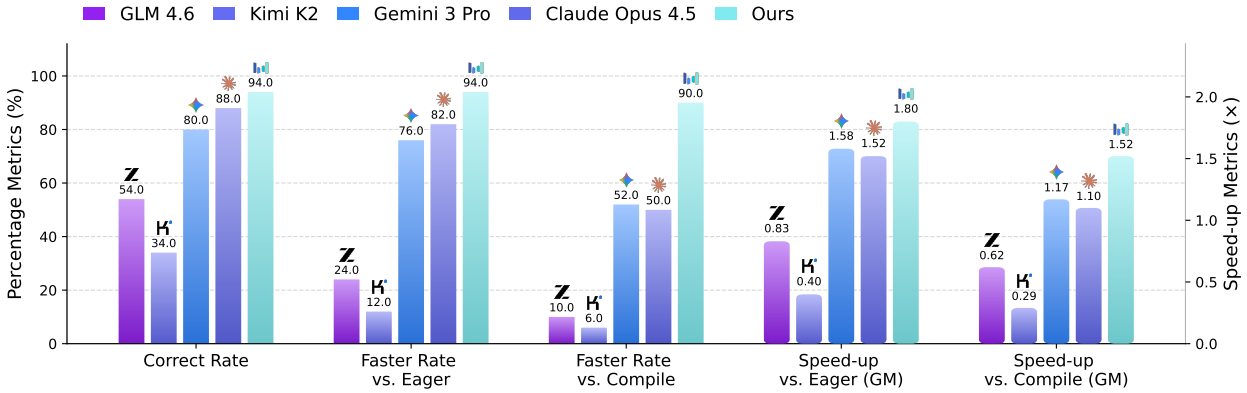
GPU kernel optimization is fundamental to modern deep learning but remains a highly specialized task requiring deep hardware expertise. Despite strong performance in general programming, large language models (LLMs) remain uncompetitive with compiler-based systems such as `torch.compile` for CUDA kernel generation. Existing CUDA code generation approaches either rely on training-free refinement or fine-tune models within fixed multi-turn execution-feedback loops, while both paradigms fail to fundamentally improve the model's intrinsic CUDA optimization ability, resulting in limited performance gains. We present CUDA Agent, a large-scale agentic reinforcement learning system that develops CUDA kernel expertise through three components: a scalable data synthesis pipeline, a skill-augmented CUDA development environment with automated verification and profiling to provide reliable reward signals, and RL algorithmic techniques enabling stable training. CUDA Agent achieves state-of-the-art results on KernelBench, delivering 100%, 100%, and 92% faster rate over `torch.compile` on KernelBench Level-1, Level-2, and Level-3 splits, outperforming the strongest proprietary models *e.g.* Claude Opus 4.5 and Gemini 3 Pro by about 40% on the hardest Level-3 setting.

Date: February 26, 2026
Correspondence: zhouhao@air.tsinghua.edu.cn
Project Page: https://nwiad.github.io/CUDA-Agent.github.io/

1 Introduction

GPU kernels are a foundational component of modern deep learning infrastructure [12, 17], with NVIDIA's CUDA architecture currently dominating the AI hardware ecosystem. Despite its widespread adoption, the development and optimization of high-performance CUDA kernels remain highly challenging, which requires a deep understanding of GPU microarchitectural features [16], and sophisticated profiling toolkits [4].

Although Large Language Models (LLMs) have demonstrated human-comparable proficiency in general software development tasks [23], existing CUDA kernel generation approaches [4, 12] still remain uncompetitive even with automatic optimization tools such as `torch.compile` in CUDA kernel code generation [17], let



alone human experts.

One line of prior work on CUDA code generation focuses on designing training-free workflows [5, 6, 8], which rely on hand-designed refinement heuristics guided by execution feedback. However, these methods do not remedy the fundamental lack of CUDA-coding abilities in the base models, causing performance gains to be significantly capped by the model's intrinsic capabilities. Another line of research [1, 4, 11, 14] attempts to fine tune base models within a fixed multi-turn refinement loop driven by code execution feedback. However, such methods waste context length by including all previous solutions and constrain the agent's autonomy to learn debugging, search, and profiling strategies.

To overcome these limitations, we introduce CUDA Agent, a large-scale CUDA agent training system that systematically enhances the base model's CUDA kernel coding capabilities by making contributions in three complementary dimensions: data, agent environment, and reinforcement learning (RL) algorithmic techniques.

To support large-scale reinforcement learning, we design a scalable data synthesis pipeline that generates training problems spanning a wide range of difficulty levels, enabling effective curriculum-based RL training. For the agent environment, we adopt the agent skills paradigm [2], equipping the model with a structured specification that formalizes the standard workflow for writing, validating and optimizing CUDA kernels, together with automated test and profiling scripts for execution-based feedback. Notably, we implement rigorous correctness and performance tests, along with system-level permission isolation, to prevent reward hacking and ensure accurate reward signals. As for RL algorithmic improvement, we analyze the sources of instability in RL optimization and propose a multi-stage warm-up strategy for both the actor and critic models, enabling stable training of large language models for 150 steps.

Integrating these components, CUDA Agent successfully scales to a context length of 128k tokens and supports up to 200 interaction turns, achieving state-of-the-art performance. Specifically, CUDA Agent achieves speedups of **100**%, **100**%, and **92**% over `torch.compile` on the Level-1, Level-2, and Level-3 splits of KernelBench [17], outperforming advanced proprietary models such as Claude Opus 4.5 and Gemini 3 Pro[1] by approximately **40**% in the Level-3 split.

Our contributions can be summarized as follows:

- We introduce CUDA Agent, a large-scale agentic reinforcement learning system for automatic CUDA kernel generation, which systematically improves the base model's CUDA coding and optimization abilities through scalable data synthesis, a skill-augmented CUDA kernel development environment, and RL techniques designed for stable long-context, multi-turn agentic training.
- CUDA Agent achieves state-of-the-art performance on KernelBench, outperforming `torch.compile` in the majority of test cases and delivering the largest speedups across multiple difficulty levels. These results establish LLM-based kernel generation as a competitive—and often superior—alternative to traditional compiler-driven kernel optimization.

[1]We found that models in the ChatGPT-5 series (5, 5.1, and 5.2) were not amenable to evaluation on this task, as they consistently declined to respond to CUDA-related prompts.

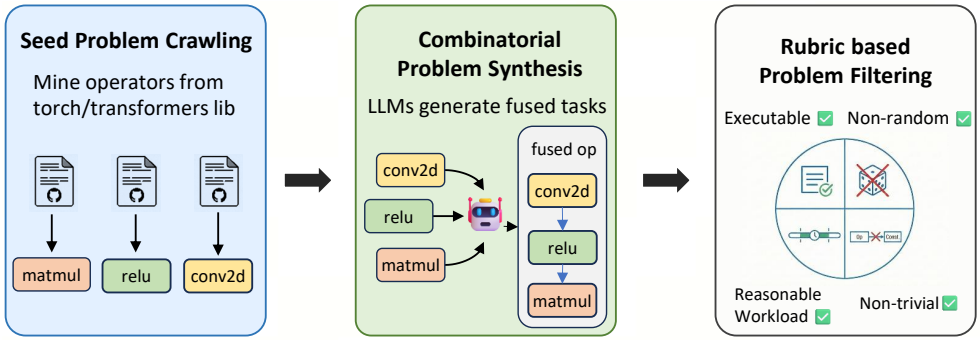


Figure 1 Overview of the three-stage data collection pipeline. We first crawl seed operators from PyTorch and Transformer libraries to build a repository of fundamental computational primitives. Next, an LLM performs combinatorial synthesis to generate fused, multi-operator tasks. Finally, a rubric-based filtering stage retains only executable, deterministic, non-trivial problems with reasonable workloads to ensure data quality and reliable evaluation.

2 Related Works

2.1 Training Free Systems for Kernel Generation

Several recent works explore designing with explicit search to address the irregular landscape of kernel optimization. STARK [5] employs a strategic team of agents with planning, coding, and debugging roles that explore a tree-structured search space on KernelBench, iteratively refining kernels using compilation, correctness checks, and timing feedback. ReGraphT [6] presents a training-free, retrieval-augmented framework that distills CUDA optimization trajectories from large language models into a reasoning graph, which is then searched via Monte Carlo Graph Search to guide smaller models toward competitive performance. EvoEngineer [8] formulates CUDA kernel optimization as a constrained code evolution problem and applies an LLM-driven evolutionary loop that iteratively edits and validates kernels to improve performance while preserving correctness. CudaForge [26] introduces a training-free two-agent system where a Judge uses Nsight Compute profiling and hardware specifications to diagnose bottlenecks and provide targeted optimization feedback to a Coder, achieving consistent speedups across GPUs.

Although these works demonstrate considerable performance gains, they heavily rely on the base model's CUDA coding capability. Furthermore, those test-time scaling approaches are also orthogonal and could be applied to our model.

2.2 Fine-tuning LLM for Kernel Generation

In parallel, another line of research seeks to improve CUDA kernel generation by training base models via supervised fine-tuning or reinforcement learning. Kevin [4] introduces a multi-turn reinforcement learning framework for CUDA kernels that models the iterative developer workflow, achieving notable gains in both correctness and performance over its QwQ-32B base model on KernelBench. CUDA-L1 [14] proposes a contrastive reinforcement learning framework that evaluates multiple CUDA kernel variants using execution-based rewards. However, their training and evaluation are conducted on the same KernelBench dataset without a proper train–test split. This data leakage renders the reported results not directly comparable to ours. ConCuR [11] synthesize and curate CUDA kernels with reasoning traces, and uses the resulting dataset to fine-tune QwQ-32B into KernelCoder, achieving performance improvment among open-source models.

Overall, these approaches remain fundamentally constrained by the scarcity of high-quality training data, limited training scale, and hand-designed optimization loops, which collectively cap their performance improvements. A more detailed analysis and comparison of these related works can be found in Appendix C.

3 Method

Our large-scale agentic reinforcement learning system CUDA Agent is built upon three key components: 1) a scalable data collection pipeline; 2) a skill-integrated and non-hackable training environment with robust

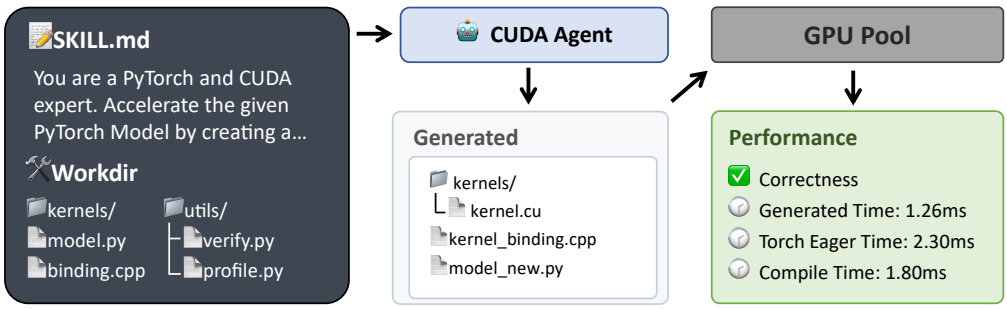


Figure 2 Overview of the agent loop.

reward scheduling designing; and 3) proposed reinforcement learning algorithmic techniques for stable training. Each component plays a crucial role in achieving the final strong empirical results.

3.1 Scalable Training Data Synthesis Pipeline

The scarcity of high-performance CUDA kernels creates a significant bottleneck for supervised fine-tuning, as manual implementation of expert-level reference code is prohibitively expensive. To overcome this, we employ reinforcement learning (RL) for training CUDA Agent. RL necessitates a vast and diverse corpus of reference operators implemented in PyTorch to serve as training tasks. Since existing public datasets lack the requisite diversity and scale, we develop a **scalable data collection pipeline** that systematically expands the task space through seed problem crawling, LLM-based combinatorial synthesis, and rigorous execution-based filtering (Figure 1).

The key observation for our combination based data synthesis is composing multiple operators into fused tasks yields valuable new CUDA-agent training problem. This is because the combined problem is often not equivalent to trivially optimizing each operator in isolation and then chaining them. Fusion reshapes the optimization landscape by avoiding intermediate global-memory materialization, coupling stages through shared register/SMEM/occupancy constraints, and requiring a unified parallel mapping and data layout that may favor downstream consumption.

Seed Problem Crawling First, we mine reference operator implemented in PyTorch from the `torch` and `transformers` libraries, establishing a comprehensive seed problem set. Each operator is represented as a Python class with initialization and forward methods. These operator classes are widely used and well-maintained. We therefore exclude individually maintained repositories that lack sufficient code quality.

Combinatorial Problem Construction Next, to expand the dataset and introduce higher complexity, we utilize LLMs to synthesize aggregated operators. Specifically, the LLM is prompted to sample no more than 5 operator classes from the `torch` library. The sampled operator classes are composed sequentially by stacking them into a single computational layer. We do not sample operator classes from the `transformers` library, as these operators are typically higher-level modules that already encapsulate multiple primitive operations.

Problems Filtering Finally, we implement a rigorous data selection process to filter out problems that are either too easy or too difficult, based on execution feedback. We validate each operator against four criteria: (1) The operator must execute successfully in both Eager and Compile modes. (2) To ensure reproducibility, we exclude operators with inherent stochasticity. (3) For anti-hacking, we verify that outputs for different inputs are neither constant values nor numerically indistinguishable. (4) To filter out trivial or excessively heavy tasks, we restrict the execution time in eager mode to the range of 1 ms to 100 ms. Furthermore, we exclude operators that exhibit high similarity to KernelBench test cases; the similarity distribution is provided in section A.

In total, the filtered synthesized training dataset contains 6,000 samples, forming CUDA-Agent-Ops-6K[2], a curated operator-level dataset for training CUDA-capable agents. Additional details on data format, contamination analysis, and dataset composition are provided in section A.

[2] https://huggingface.co/datasets/BytedTsinghua-SIA/CUDA-Agent-Ops-6K

3.2 Skill-Integrated Agent Loop

Agent Loop As CUDA kernel coding naturally arises as a subtask of coding agents, we design our agent loop to align with the widely adopted OpenHands framework [22] to ensure the generalizability. The LLM is provided with a standard suite of shell utilities—`BashTool`, `GlobTool`, `MultiEditTool`, and `TodoWriteTool`—to fully support CUDA coding development (see section B for the full list). On top of this, our agent loop (visualized in figure 2) follows the ReAct-style paradigm [24], interleaving reasoning, action execution, and observation to enable iterative coding, debugging, and performance optimization.

CUDA Coding Skill Inspired by the idea of Agent Skills [2], we deliberately put the CUDA coding specific instructions and tools (e.g. the profiling tool to compare the performance of generated kernel against `torch.compile`) as Agent Skills format. We also design a CUDA kernel coding specific instructions `SKILL.md` formulating the standard process to optimize the CUDA kernel (original text can be found in section B):

1. Analyze the performance of the native PyTorch implementation using the provided `profile.py` script. This step identifies performance bottlenecks and optimization opportunities, such as excessive kernel launches and suboptimal memory access patterns.
2. Implement custom CUDA operators by rewriting the model in `model_new.py` and developing the corresponding CUDA kernel source files and binding code, targeting the performance-critical operators identified in the analysis stage.
3. Compile and evaluate the optimized model in the provided GPU sandbox environment, and iteratively refine the CUDA kernel implementations until both numerical correctness and performance requirements are met.
4. Repeat the optimization process starting from Step 2 until the final implementation achieves at least a 5% speedup over the `torch.compile` baseline while passing all numerical correctness checks.

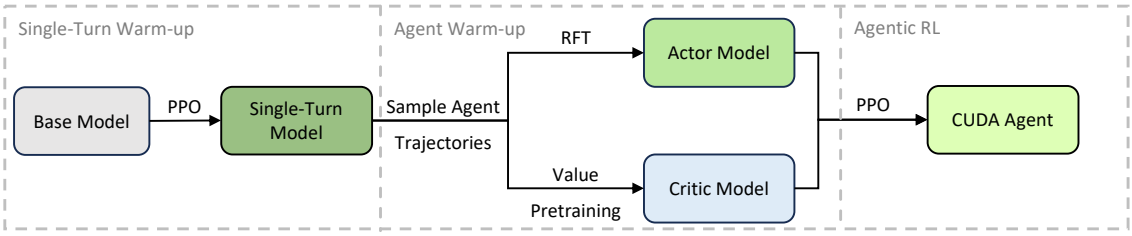


Figure 3 Overview of training pipeline. Following a single-turn RL warm-up stage, the sampled trajectories are used to initialize actor model and critic model before agentic RL stage.

Robust Reward Scheduling Existing RL approaches for CUDA generation [1, 4, 14] use speedup over baselines as the reward signal. However, we observe that this approach suffers from outliers and bias toward easy kernels. This is because operators vary substantially in optimization difficulty, making raw speedup an unreliable proxy for code quality. To address this, we propose a normalized, robust reward scheme to remedy this and jointly optimize correctness and execution latency.

We assign a reward score $r \in \{-1, 1, 2, 3\}$ based on correctness and performance:

$$r = \begin{cases} -1 & \text{if correctness check fails} \\ 3 & \text{if } b(t, t_{\text{eager}}) \wedge b(t, t_{\text{compile}}) \\ 2 & \text{if } b(t, t_{\text{eager}}) \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

where t is the generated kernel's runtime, t_{native} and t_{compile} are the runtimes of PyTorch's eager implementation and `torch.compile` version respectively, and $b(t, t_0) = \mathbb{I}\left[(t_0 - t)/t_0 > 5\%\right]$ indicates a significant speedup over baseline t_0. We validate this design in section 4.3.2 that it significantly outperforms the commonly used speedup reward.

Efforts to Avoid Reward Hacking We note that previous work Lange et al. [12] suffer from the hacking issue [9]. Specifically, our system enforces the following constraints to avoid reward hacking: 1) the provided Python scripts for correctness verification and performance profiling are protected via file permission controls, preventing the agent from modifying or interfering with the evaluation logic; 2) to avoid trivial fallbacks, we enforce execution-time constraints using context managers that explicitly forbid invoking fallback implementations from `torch.nn.functional`, ensuring that performance gains originate solely from the generated CUDA kernels; 3) for each problem, we validate kernel outputs against five randomly sampled inputs, strictly following the KernelBench evaluation protocol to ensure functional correctness; 4) the profiling pipeline is carefully engineered with proper device synchronization, warm-up iterations, and repeated measurements with averaging, substantially reducing measurement noise and metric fluctuation; and 5) the agent is not provided with any web search or external information retrieval tools, ensuring that all solutions are derived purely from local execution environment.

3.3 Algorithmic Improvements for Stable RL Training

We observe that our initial RL trial could only train stably for 17 steps before the model's performance collapsed. We identified the root cause of this instability and propose warming up both the actor and critic models to adapt to the model's distribution. With this modification, our RL trial can stably train for 200 steps with consistent reward growth.

The Root Cause of Training Instability stems from a severe domain distribution mismatch, where the base model's learned prior deviates significantly from the data distribution required for CUDA kernel coding. This is because the CUDA coding data accounts for less than 0.01% of pretraining data [10, 13]. This distribution gap results in numerous sampled low-probability and CUDA kernel code tokens. Furthermore, when training and inference engines use different numerical precisions (e.g., BF16 vs FP16), those low-probability tokens result in large importance sampling ratio variance because small numerical errors in computing token probablities $\pi_\theta(y_t)$ near the precision floor (e.g., $\pi_\theta(a_t \mid s_t) \approx 10^{-9}$) cause the importance ratio $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ to fluctuate wildly or explode (similar to the discussion in Liu et al. [15]).

To achieve stable reinforcement learning, we propose a simple yet effective warm-up strategy: initializing both the actor and critic models using agent trajectories generated by the base model after single-turn RL, as illustrated in figure 3.

Single-Turn Warm-up We first perform single-turn RL on base model to enhance its capability in CUDA kernel generation. We use PPO for optimization, where the base model serves as the policy and value network.

Actor Initialization We then adopt a Rejection Fine-Tuning (RFT) stage on agent trajectories to initialize the actor model π_θ. The resulting model from single-turn RL is used to collect CUDA agent trajectories by running the agent in the agent loop from section 3.2. Next, we apply RFT on the collected CUDA agent trajectories. Rejection sampling is performed to retain only high-quality rollouts according to the following rubrics: (1) **Outcome filtering:** we only keep trajectories that achieve a positive reward ($R > 0$). (2) **Pattern filtering:** we discard trajectories that exhibit inefficient or invalid behaviors, such as redundant multi-turn loops or hallucinations that violate the predefined tool-call schema.

The filtered trajectories are then used to optimize the actor via standard supervised fine-tuning with the following objective:

$$\mathcal{L}_{\text{RFT}}(\theta) = -\mathbb{E}_{\tau \sim \mathcal{D}'} \left[\sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t, a_{<t}) \right], \tag{2}$$

where $\tau = (s_0, s_1, \ldots, s_{T-1})$ denotes a filtered CUDA agent trajectory, π_θ is the policy parameterized by θ, and $\mathcal{D}'$ represents the dataset after rejection sampling.

Critic Initialization We perform Value Pretraining to initialize the critic, specifically, we utilize the sampled trajectory data comprising state sequences and their corresponding outcome rewards to pretrain the critic network. Let $\tau = (s_0, s_1, \ldots, s_{T-1})$ denote a trajectory where s_t represents the state at token position t, and let r denote the outcome reward assigned at the final token (i.e., $r_t = 0$ for $t < T-1$ and $r_{T-1} = r$). We

Table 1 Main Results on KernelBench. We report Pass Rate, Faster Rate (percentage of kernels faster than baseline), and Geometric Mean Speed-up. Metrics are reported relative to both PyTorch Eager and PyTorch Compile baselines. Overall metrics are weighted by the number of problems in each level (Level 1: 100, Level 2: 100, Level 3: 50). **Bold** indicates the best performance.

			Faster Rate (↑)		Speed-up (Geomean, ×)	
Subset	Model	Pass Rate	vs. Eager	vs. Compile	vs. Eager	vs. Compile
Overall†	Seed1.6 (base model)	74.0%	43.6%	27.2%	0.95×	0.69×
	GLM 4.6	75.6%	44.8%	19.2%	0.78×	0.57×
	Kimi K2	66.8%	40.8%	22.8%	0.93×	0.66×
	Gemini 3 Pro	91.2%	87.6%	69.6%	1.92×	1.42×
	Claude Opus 4.5	95.2%	90.4%	66.4%	1.99×	1.46×
	CUDA Agent (ours)	**98.8%**	**98.4%**	**96.8%**	**2.60**×	**2.11**×
Level 1	Seed1.6 (base model)	90.0%	63.0%	51.0%	1.65×	1.25×
	GLM 4.6	86.0%	57.0%	32.0%	0.99×	0.73×
	Kimi K2	85.0%	56.0%	39.0%	1.43×	1.00×
	Gemini 3 Pro	95.0%	90.0%	72.0%	1.99×	1.51×
	Claude Opus 4.5	96.0%	88.0%	72.0%	2.03×	1.54×
	CUDA Agent (ours)	**100.0%**	**99.0%**	**97.0%**	**2.48**×	**1.87**×
Level 2	Seed1.6 (base model)	74.0%	40.0%	16.0%	0.68×	0.50×
	GLM 4.6	76.0%	43.0%	11.0%	0.60×	0.42×
	Kimi K2	65.0%	40.0%	15.0%	0.93×	0.65×
	Gemini 3 Pro	93.0%	91.0%	76.0%	2.03×	1.46×
	Claude Opus 4.5	98.0%	97.0%	69.0%	2.24×	1.60×
	CUDA Agent (ours)	**100.0%**	**100.0%**	**100.0%**	**3.27**×	**2.80**×
Level 3	Seed1.6 (base model)	42.0%	12.0%	2.0%	0.60×	0.40×
	GLM 4.6	54.0%	24.0%	10.0%	0.83×	0.62×
	Kimi K2	34.0%	12.0%	6.0%	0.40×	0.29×
	Gemini 3 Pro	80.0%	76.0%	52.0%	1.58×	1.17×
	Claude Opus 4.5	88.0%	82.0%	50.0%	1.52×	1.10×
	CUDA Agent (ours)	**94.0%**	**94.0%**	**90.0%**	**1.80**×	**1.52**×

compute target values using Generalized Advantage Estimation [19]:

$$V_t^{\text{targ}} = V_\phi(s_t) + \hat{A}_t, \quad \text{where} \quad \hat{A}_t = \sum_{l=0}^{T-1-t} (\gamma\lambda)^l \delta_{t+l}, \tag{3}$$

and $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ is the temporal difference error with $V_\phi(s_T) = 0$. We set $\gamma = 1$ and $\lambda = 0.95$ in our experiments. We optimize the critic parameters ϕ by minimizing the mean squared error:

$$\mathcal{L}_{\text{VP}}(\phi) = \frac{1}{2}\mathbb{E}_{\tau\sim\mathcal{D}}\left[\frac{1}{T}\sum_{t=0}^{T-1}\left(V_\phi(s_t) - V_t^{\text{targ}}\right)^2\right], \tag{4}$$

where $\mathcal{D}$ denotes the collection of agent trajectories. We ablate the proposed initialization stage in section 4.3.3.

RL Algorithm We employ PPO [18] to optimize the actor model π_θ. Let π_θ denote the policy to be optimized, and $\pi_{\theta_{\text{old}}}$ denote the policy used for trajectory sampling. We maximize the expected return using the clipped surrogate objective:

$$\begin{aligned}\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_{\tau\sim\mathcal{D}}\Bigg[\frac{1}{T}\sum_{t=0}^{T-1}\min\big(&\rho_t(\theta)\hat{A}_t,\\ &\text{clip}(\rho_t(\theta), 1-\epsilon_{\text{lower}}, 1+\epsilon_{\text{higher}})\hat{A}_t\big)\Bigg]\end{aligned} \tag{5}$$

where $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the importance sampling ratio between the current and old policies, a_t denotes the action (i.e., token) taken at position t, $\epsilon_{\text{lower}} = 0.2, \epsilon_{\text{higher}} = 0.28$ following Yu et al. [25].

Table 2 Ablation Study. Comparison between the full model and leave-one-out variants **under agent loop evaluation**. We analyze the contributions of (1) the agent loop, (2) robust reward design, (3) RFT, and (4) Value Pretraining. For variants without RFT or Value Pretraining, we report results from the final validation step before training collapse.

			Faster Rate (↑)		Speed-up (Geomean, ×)	
Subset	Model	Pass Rate	vs. Eager	vs. Compile	vs. Eager	vs. Compile
Overall†	w/o Agent Loop	77.1%	43.5%	14.1%	0.89×	0.69×
	w/o Robust Reward	96.8%	90.4%	60.4%	1.70×	1.25×
	w/o RFT	95.6%	82.0%	49.8%	1.56×	1.05×
	w/o Value Pretraining	98.6%	85.0%	50.9%	1.49×	1.00×
	CUDA Agent (ours)	**98.8%**	**98.4%**	**96.8%**	**2.60**×	**2.11**×

4 Experiments

4.1 Experiment Settings

Training Settings for RL. We leverage Seed1.6 [20] as the base model, a Mixture-of-Experts (MoE) model with 23B active and 230B total parameters. We set the global batch size to 1024, matching the mini-batch size for online PPO updates. Learning rate for actor and critic is 3×10^{-6} and 6×10^{-6}. These hyper-parameters are shared between single-turn RL and agentic RL. Context window length is 32768 for single-turn RL and 131072 for agentic RL. We impose a maximum of 150 agent turns during training rollouts, which is relaxed to 200 agent turns during evaluation. The model is trained over 150 training steps.

Sandbox Environment To ensure accurate speedup measurement for reliable reward signals and efficient GPU utilization, we design a CPU–GPU resource–decoupled sandbox architecture. A Docker-based terminal sandbox handles CPU-centric tasks (e.g., kernel compilation), while the agent leverages pre-defined CUDA Agent skill scripts to dispatch verification and profiling jobs to a dedicated GPU sandbox pool (128 NVIDIA H20 GPUs). This process-level isolation and exclusive resource allocation eliminate inter-process interference, ensuring stable latency measurements and guaranteed HBM capacity.

Benchmark. We conduct our evaluations on KernelBench [17], utilizing the Level 1 to Level 3 subsets which comprise a total of 250 distinct operator tasks. To ensure fair comparison, we adapt these tasks from their original single-file format into our multi-file development environment (Section 3.2).

Baseline Models. We compare our approach against frontier proprietary coding models: **Claude Opus 4.5** [3] and **Gemini 3 Pro** [7], as well as two powerful open-source coding models: **GLM 4.6** [27] and **Kimi K2** [21]. These models currently hold leading positions on major coding agent leaderboards and serve as strong baselines for general-purpose reasoning and coding capabilities. For fair comparison, baseline models are evaluated under the same agent loop mentioned in Section 3.2.

Evaluation Metrics. We assess performance using three key metrics: (1) **Pass Rate**, the percentage of tasks where the agent generates a kernel that successfully compiles and passes functional correctness checks; (2) **Faster Rate**, the percentage of tasks where the generated kernel is correct and achieves a faster execution time than the baseline (`eager` and `compile` modes); and (3) **Speed-up**, the geometric mean of the execution speed-up ratio relative to the baselines, computed exclusively for correct solutions. To determine the final metrics for each task, we extract the best-performing solution from the agent's interaction trajectory, specifically the one that achieves the maximum speed-up relative to `torch.compile`.

4.2 Main Results

Table 1 summarizes the performance of CUDA Agent against strong proprietary model baselines on KernelBench. Our analysis yields three primary insights regarding the efficacy of agentic RL for CUDA kernel optimization.

First, when compared to strong proprietary models, CUDA Agent delivers substantially stronger CUDA kernel coding performance. Although Claude Opus 4.5 and Gemini 3 Pro achieve respectable Pass Rates (91.2%–95.2%), their faster rates remain low at 66%–69%, indicating that general-purpose LLMs often produce

naïve kernels that fail to outperform `torch.compile`. In contrast, CUDA Agent attains a 98.8% Pass Rate and a 96.8% faster rate, demonstrating that specialized RL training enables consistently correct and highly optimized CUDA implementations.

Second, compared to static `torch.compile`, CUDA Agent prove that learned optimization policies can consistently outperform static compiler heuristics, particularly in complex scenarios like operator fusion. This is most evident in Level 2 tasks (Operator Sequences), where CUDA Agent achieves a perfect 100% faster rate and a massive 2.80× speed-up over `torch.compile`. Traditional compilers rely on predefined, rule-based patterns for kernel fusion which often struggle with non-trivial operator combinations. By contrast, CUDA Agent explores a much larger design space through its iterative agent loop, discovering hardware-specific memory access patterns and tiling strategies that remain inaccessible to static backends.

4.3 Ablation Studies

4.3.1 Impact of Skill-Integrated Agent Loop

To assess the critical role of the interactive environment in policy learning, we train two separate models under distinct protocols: (1) **single-turn model**, trained using a standard code generation objective where the model predicts the final kernel and bindings in a single turn without execution feedback, serving as a warm-up stage in our training pipeline, and (2) **CUDA Agent**, trained following the complete training pipeline, allowing the policy to see compilation errors and profiler feedback in multi-turn interactions.

Table 2 highlights the limitations of single-turn code generation and underscores the necessity of our skill-integrated agent loop. Removing the agent loop causes a substantial drop in both correctness and optimization quality. More importantly, the resulting kernels are not only less optimized but often regress in performance. Being exposed to compilation errors, runtime failures, and profiler feedback, the agent can iteratively diagnose mistakes and refine transformations across turns.

4.3.2 Impact of Reward Design

To determine the optimal shaping of reward for kernel optimization, we evaluate two different reward formulations: (1) **speed-up reward**, a continuous reward signal defined as $r_s = t_{\text{compile}}/t_{\text{gen}}$ for correct solutions and -1 for incorrect ones; and (2) **our robust reward schedule** (section 3.2), which assigns discrete values for achieving specific performance milestones .

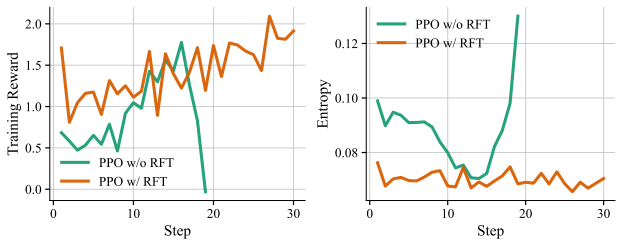
As shown in Table 2, reward design has a pronounced effect on optimization outcomes. Replacing our robust reward schedule with a raw Speed-up Reward (w/o Robust Reward) yields comparable functional correctness, but substantially weaker optimization performance.

These results indicate that a normalized, milestone-based reward is better aligned with the goal of producing consistently faster kernels. By assigning credit to clear performance targets rather than directly regressing on noisy runtime ratios, the policy more reliably discovers transformations that translate into real speed-ups relative to both eager execution and compiler baselines.

4.3.3 Impact of Multi-Stage Training

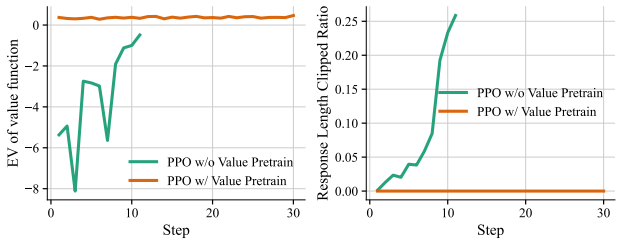
As shown in Table 2, removing either RFT or Value Pretraining leads to substantial degradation in optimization performance, despite largely preserved pass rates. More importantly, both ablations exhibit training instability and eventual collapse, motivating a closer analysis of the two stages below.

Rejection Sampling Fine-Tuning (RFT) provides a critical prior that prevents policy collapse. As shown in Figure 4a, removing the RFT stage results in a rapid and catastrophic collapse of training rewards. To diagnose the underlying cause, we examine the policy entropy in Figure 4b, which reveals a sharp increase coinciding with the reward collapse. This entropy surge indicates that the policy distribution becomes increasingly diffuse, producing incoherent and poorly structured outputs. By initializing the policy with a strong behavioral prior, RFT constrains the entropy growth during reinforcement learning and keeps the optimization trajectory within a well-structured output distribution.



(a) Training Reward. **(b) Actor Entropy.**

Figure 4 Ablation: RFT. Removing RFT causes training reward to collapse. The concurrent increase in actor entropy suggests that the policy becomes increasingly diffuse and poorly structured.



(a) Explained Variation of Value Function. **(b)** Response Length Clipped Ratio.

Figure 5 Ablation: Value Pretraining. Without Value Pretraining, the critic fails to learn a meaningful value function, as reflected by low explained variance. This leads to inefficient exploration, manifested as excessively long interaction trajectories.

Value Pretraining is indispensable for providing reliable advantage estimates and preventing pathological search behaviors. Without an initialized critic, the model fails to capture the value landscape of the multi-turn interaction states (Figure 5a). This poor estimation leads to an explosion in trajectory length (Figure 5b), as the uninitialized critic fails to penalize fruitless or redundant search paths. Value Pretraining ensures that the critic can immediately provide accurate feedback, guiding the agent toward efficient optimization paths and avoiding the computational instability of near-infinite interaction loops.

5 Conclusion

We introduced CUDA Agent, a large-scale agentic reinforcement learning system that endows large language models with the ability to generate and optimize CUDA kernels under realistic, execution-driven development workflows. By jointly scaling data synthesis, agent environments, and stability-oriented RL training, CUDA Agent moves LLMs beyond syntactic code generation toward hardware-aware performance optimization, achieving consistent gains over `torch.compile` and strong proprietary models on KernelBench. These results suggest a broader consequence: equipping foundation models with structured environments and reliable execution-based rewards can transform them from passive code generators into active systems optimizers, opening a path toward automating performance-critical software development in GPU computing.

References

[1] Anonymous. Mastering sparse CUDA generation through pretrained models and deep reinforcement learning. In The Fourteenth International Conference on Learning Representations, 2026. URL https://openreview.net/forum?id=VdLEaGPYWT.

[2] Anthropic. Equipping agents for the real world with agent skills, 2025. URL https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills. Accessed: 2026-01-26.

[3] Anthropic. Claude Opus 4.5: System Card. https://www.anthropic.com/claude-opus-4-5-system-card, November 2025. Accessed: 2026-01-28.

[4] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. arXiv preprint arXiv:2507.11948, 2025. URL https://arxiv.org/abs/2507.11948.

[5] Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. Stark: Strategic team of agents for refining kernels. arXiv preprint arXiv:2510.16996, 2025. URL https://arxiv.org/abs/2510.16996.

[6] Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. From large to small: Transferring cuda optimization expertise via reasoning graph, 2025. URL https://arxiv.org/abs/2510.19873.

[7] Google DeepMind. Gemini 3 Pro: Model Card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf, November 2025. Accessed: 2026-01-28.

[8] Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. Evoengineer: Mastering automated cuda kernel code evolution with large language models, 2025. URL https://arxiv.org/abs/2510.03760.

[9] Horse. This example from their paper, which is claimed to have 150x speedup, is actually 3x slower if you bench it. X (formerly Twitter), February 2025. URL https://x.com/main_horse/status/1892473238036631908. Accessed: 2025-01-27.

[10] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. arXiv preprint arXiv:2211.15533, 2022.

[11] Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. Concur: Conciseness makes state-of-the-art kernel generation, 2025. URL https://arxiv.org/abs/2510.07356.

[12] Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. Technical report, Technical report, Sakana AI, 02 2025, 2025.

[13] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161, 2023.

[14] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. arXiv preprint arXiv:2507.14111, 2025. URL https://arxiv.org/abs/2507.14111.

[15] Jiacai Liu, Yingru Li, Yuqian Fu, Jiawei Wang, Qian Liu, and Zhuo Jiang. When speed kills stability: Demystifying RL collapse from the training-inference mismatch, September 2025. URL https://richardli.xyz/rl-collapse.

[16] NVIDIA. Nvidia h100 tensor core gpu architecture, 2022. URL https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c.

[17] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? arXiv preprint arXiv:2502.10517, 2025. URL https://arxiv.org/abs/2502.10517.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[19] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018. URL https://arxiv.org/abs/1506.02438.

[20] Bytedance Seed. Seed 1.6 tech introduction. URL https://seed.bytedance.com/en/seed1_6.

[21] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

[22] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. arXiv preprint arXiv:2407.16741, 2024.

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

[24] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In The eleventh international conference on learning representations, 2022.

[25] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. arXiv preprint arXiv:2503.14476, 2025.

[26] Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. arXiv preprint arXiv:2511.01884, 2025. URL https://arxiv.org/abs/2511.01884.

[27] Zhipu AI. Glm-4.6: Advanced agentic, reasoning and coding capabilities. https://z.ai/blog/glm-4.6, 2025. Official blog post introducing the GLM-4.6 model.

Appendix

A Details of Collected Training Data

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import *
from transformers.activations import ACT2FN

class FNetPredictionHeadTransform(nn.Module):
    ... # omitted for brevity

class FNetLMPredictionHead(nn.Module):
    ... # omitted for brevity

class Model(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.predictions = FNetLMPredictionHead(config)
        self.seq_relationship = nn.Linear(config.hidden_size, 2)

    def forward(self, sequence_output, pooled_output):
        prediction_scores = self.predictions(sequence_output)
        seq_relationship_score = self.seq_relationship(pooled_output)
        return prediction_scores, seq_relationship_score

def get_inputs():
    sequence_output = torch.randn(32, 128, 512, dtype=torch.float32)
    pooled_output = torch.randn(32, 512, dtype=torch.float32)
    return (sequence_output, pooled_output,)

def get_init_inputs():
    import transformers
    config = transformers.FNetConfig(...) # omitted for brevity
    return (config,)
```

(a) `transformers` operator class

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

class Model(nn.Module):
    def __init__(
        self, in_channels, out_channels, kernel_size,
        stride, padding, output_padding, bias_shape
    ):
        super().__init__()
        self.softmax = nn.Softmax(dim=1)
        self.conv_transpose = nn.ConvTranspose2d(
            in_channels, out_channels, kernel_size, stride=stride,
            padding=padding, output_padding=output_padding
        )
        self.bias = nn.Parameter(torch.randn(bias_shape))

    def forward(self, x):
        x = self.softmax(x)
        x = self.conv_transpose(x)
        return x + self.bias

batch_size = 512
in_channels = 3
out_channels = 16
height, width = 32, 32
kernel_size = 3
stride = 2
padding = 1
output_padding = 1
bias_shape = (out_channels, 1, 1)

def get_inputs():
    return [torch.randn(batch_size, in_channels, height, width)]

def get_init_inputs():
    return [in_channels, out_channels, kernel_size, stride, padding, output_padding, bias_shape]
```

(b) Combinatorial `torch` operator class

Figure 6 Examples of operator classes in our training data.

Operator Format. Each training sample is represented as a Python class implemented in PyTorch. Specifically, each operator is defined as a subclass of `torch.nn.Module`, with an `__init__` method for parameter initialization and a `forward` method specifying the computation. In addition to the operator class, each sample includes two auxiliary functions: `get_init_inputs()`, which constructs the inputs required to instantiate the operator class, and `get_inputs()`, which generates runtime inputs for the `forward` method. Together, the operator class and these auxiliary input-generation functions form a self-contained and executable training task. Figure 6 shows representative examples of operator classes used in our training data.

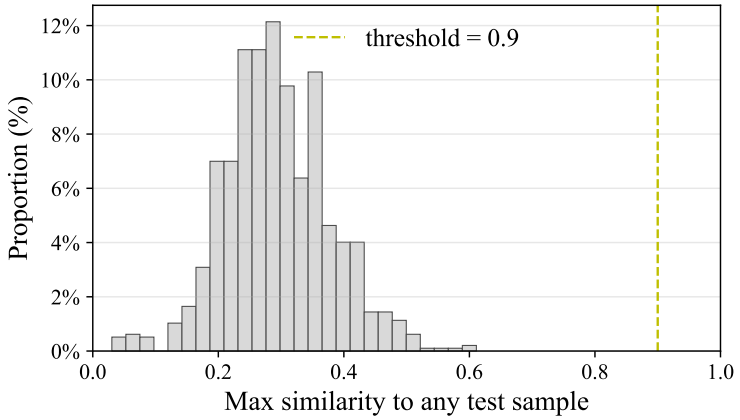


Figure 7 Distribution of the maximum AST similarity between each training sample and all evaluation samples.

Data Contamination Check. To prevent overlap between training data and evaluation benchmarks, we perform an explicit decontamination step using an off-the-shelf AST-based code similarity tool. Specifically, we extract the `Model` class from each program and compute pairwise structural similarity between training and evaluation samples using `PythonASTSimilarity`. A training sample is removed if its maximum similarity to any evaluation program exceeds 0.9. After this filtering, we confirm that no training sample exhibits high AST-level similarity with the evaluation set. Figure 7 shows the distribution of the maximum AST similarity between each training sample and all evaluation samples. The majority of training samples exhibit low similarity scores, and no sample exceeds the decontamination threshold after filtering.

Table 3 Composition of the final training dataset

Operator category	Proportion
`torch` operators ×1	3.40%
`torch` operators ×2	83.77%
`torch` operators ×3	7.62%
`torch` operators ×4	2.80%
`torch` operators ×5	1.23%
`transformers` operator	1.18%

Final Dataset Composition. Table 3 summarizes the composition of the final training dataset after data synthesis and filtering. The majority of training samples are composite operator classes constructed by sequentially stacking between one and five operator classes from the `torch` library. We report the distribution of composite operators by composition depth, where a k-op composition denotes an operator class formed by composing k primitive operator classes. In addition, the dataset includes a set of operator classes directly taken from the `transformers` library, which are included as standalone operators and are not involved in compositional construction. This distribution reflects a deliberate balance between simple operators, moderately complex compositions, and higher-level `transformers` modules.

B Details of Agent Loop

B.1 Full List of Provided Tools

We equip the agent with a structured toolset that abstracts common developer operations into callable interfaces. These tools define the action space of the agent and serve as the only mechanisms through which it can interact with the system and the external world.

The agent is provided with a suite of tools for controlled interaction with the local execution environment:

Bash. Executes shell commands in a persistent session under strict safety constraints (e.g., command quoting rules and directory validation). It enables compilation, dependency management, and program execution.

Read / Write. Provide read-only and write access to local files. Write operations are guarded by a read-before-write policy to prevent blind overwriting.

Edit / MultiEdit. Support deterministic string-level code modifications. *Edit* performs single replacements, while *MultiEdit* enables multiple atomic edits within one file, ensuring consistency across dependent code changes.

Glob. Performs fast file discovery using glob patterns (e.g., `**/*.py`), allowing the agent to navigate large codebases efficiently.

Grep. A structured code search interface based on ripgrep, supporting regex search, file-type filtering, and contextual line retrieval.

NotebookEdit. Enables structured modification of Jupyter notebook cells, allowing the agent to operate in mixed code–analysis environments.

Together, these tools allow the agent to perform typical software engineering operations, including code inspection, refactoring, compilation, and debugging.

BashOutput. Streams incremental outputs from background shell processes, enabling the agent to monitor long-running jobs (e.g., training or compilation).

KillBash. Terminates background shell sessions when jobs hang or resources must be reclaimed.

B.2 Original SKILL.md Content

```
You are a PyTorch and CUDA expert. Accelerate the given PyTorch Model by creating a
high-performance CUDA C++ extension, targeting the best possible performance with a minimum
requirement of 5\% faster than torch.compile baseline.

## 1. CRITICAL RESTRICTIONS

### STRICTLY FORBIDDEN

* **NO torch operators in C++**: NEVER use `torch::*` or `torch::nn::functional::*` in bindi
ng.cpp or .cu files
* **NO torch operations in model_new.py**: Only tensor creation and your custom ops allowed
* **NO third-party libraries**: Except cuBLAS (GEMM only) and cuDNN (Conv only)
* **NO modifications to utils/ directory**
* **NO modifications to binding.cpp or binding_registry.h**: These are fixed infrastructure

### ALLOWED ONLY

* **C++**: Raw CUDA kernels (for custom ops), cuBLAS (for GEMM), cuDNN (MANDATORY
for Conv/ConvTranspose)
* **Python**: torch.tensor creation, custom extension ops, tensor properties (.shape, .device)
* **Memory**: torch::empty_like for allocation only
* **Focus**: Implement kernels in `kernels/` directory only

## 2. WORKSPACE STRUCTURE

```
.
binding_registry.h    # Do NOT modify - registration system
binding.cpp           # Do NOT modify - main module binding
kernels/              # YOUR WORK: Implement all kernels here (initially empty)
utils/                # DO NOT modify - Testing and profiling tools
model.py              # Do NOT modify -Original PyTorch model (provided by user)
model_new.py          # YOUR WORK: Your optimized model using custom ops.
```

File Types and Usage

* **`.cu` files**: CUDA kernels with `__global__` functions (custom implementations)
* **`.cpp` files**: cuDNN/cuBLAS API calls (NO custom kernels)
* **`_binding.cpp` files**: PyTorch tensor handling and Python bindings

3. UNIFIED WORKFLOW

Step 1: Analysis (ALWAYS FIRST)

```bash
# Use the sandbox util to analyze optimization targets and metrics

# Record these critical metrics:
# - Minimum target time (torch.compile baseline × 0.95)
# - Torch.compile baseline time (for reference)
# - Kernel count (fusion opportunities)
# - Compute capability (for TORCH_CUDA_ARCH_LIST)
```
Step 2: Implementation
Step 3: Compile and Test

```bash
```
```

```
# Compile with architecture-specific optimizations
TORCH_CUDA_ARCH_LIST=MAJOR.MINOR bash utils/compile.sh


# If correctness fails: debug and fix
# If performance insufficient: continue to Step 4
```

Step 4: Performance Optimization (IF NEEDED)

4.1 Profiling and Analysis

4.2 Optimization Strategy (Priority Order)

Priority 1: Algorithmic (>50% impact)

* Kernel fusion - reduce memory traffic
* Shared memory tiling - improve data reuse
* Memory coalescing - consecutive access patterns

Priority 2: Hardware Utilization (20-50% impact)

* Vectorized loads (float2/float4)
* Warp-level primitives (__shfl_sync, __ballot_sync)
* Occupancy tuning (block size, register usage)

Priority 3: Fine-tuning (<20% impact)

* Instruction-level parallelism
* Mixed precision (FP16/TF32)
* Prefetching and double buffering

Only when within 1.2x of target and algorithmic options exhausted:

```python
# tune_kernel.py - NO recompilation needed
import time, torch, cuda_extension

configs = [
    (0, "256_threads_16_tile"),
    (1, "128_threads_32_tile"),
    (2, "512_threads_8_tile")
]

# Test input
x = torch.randn(batch_size, features).cuda()

# Benchmark each config
best_config, best_time = 0, float('inf')
for config_id, name in configs:
    # Warmup
    for _ in range(10):
        cuda_extension.my_kernel_forward(x, config=config_id)
    torch.cuda.synchronize()

    # Measure
    start = time.perf_counter()
    for _ in range(100):
        cuda_extension.my_kernel_forward(x, config=config_id)
    torch.cuda.synchronize()
    elapsed = time.perf_counter() - start

    print(f"Config {name}: {elapsed:.4f}s")
    if elapsed < best_time:
        best_time, best_config = elapsed, config_id

print(f"Best: config {best_config} ({best_time:.4f}s)")
# Update model_new.py with best_config
```

```

### Step 5: Iteration Requirements

#### Correctness Failures

**MUST iterate until correctness passes - NO EXCEPTIONS**

1. Debug the specific failing kernel
2. Common issues to check:
   * Boundary conditions (tid < size)
   * Synchronization (__syncthreads placement)
   * Data types and precision
   * Memory alignment
3. Fix in kernels/*.cu and *_binding.cpp ONLY
4. Recompile and test

#### Performance Optimization

**GOAL: Achieve the best possible performance (the faster, the better!)**
**MINIMUM: Must be at least 5% faster than torch.compile baseline**

For each iteration:

1. **Document expectation**: "Fusion will eliminate 3 kernels, expect ~20% speedup"
2. **Apply optimization aggressively**: Don't revert to slow versions
3. **Debug if correctness fails**: Fix the optimized version
4. **Measure and analyze**: Understand why performance changed
5. **Continue optimizing**: Even if you meet the minimum, keep pushing for better performance

**Iteration strategy**:

* First 1-2 iterations: Achieve the minimum 5% improvement
* Next 3-5 iterations: Push for maximum possible speedup
* Continue until no further improvements possible or diminishing returns

**Remember**: The goal is the BEST possible performance, not just meeting the minimum!

## C Discussion of Concurrent Works

In this appendix, we provide a more detailed discussion of concurrent and closely related works, clarifying differences in problem settings, evaluation protocols, and training assumptions that make direct comparison with our method either inappropriate or non-informative.

*STARK.* STARK adopts Claude Sonnet 4 as its base model and builds a structured multi-agent system consisting of specialized roles for planning, coding, and debugging. These agents follow a fixed execution program to explore a tree-structured search space. In contrast, our method employs a single agent that autonomously invokes tools, gathers feedback, and performs iterative kernel correction and optimization. Although both methods are evaluated on KernelBench, the substantial difference between a fixed multi-agent pipeline and a single, autonomous agent makes direct comparison less meaningful. In addition, STARK does not explicitly specify the GPU hardware used for runtime evaluation, which further complicates precise performance comparison.

*ReGraphT.* ReGraphT focuses on a distinct research question: transferring the optimization capabilities of large language models to smaller models via retrieval-augmented reasoning graphs and Monte Carlo Graph Search. As its primary goal is model compression and capability distillation rather than maximizing absolute kernel generation performance, we do not include direct performance comparisons with ReGraphT.

*EvoEngineer.* EvoEngineer formulates kernel optimization as an evolutionary code editing process driven by LLM feedback. However, its evaluation is conducted on only 91 problems selected from the 250 tasks in KernelBench, rather than the full benchmark. This partial coverage introduces selection bias and limits
```

the representativeness of the reported results, making direct comparison with our full-benchmark evaluation inappropriate.

CudaForge. CudaForge adopts a two-agent workflow built on OpenAI-o3, where a Judge agent leverages Nsight Compute profiling signals and hardware specifications to diagnose performance bottlenecks and provide targeted optimization feedback to a Coder agent that applies code edits. Similar to STARK, this approach relies on predefined agent roles and a fixed interaction protocol. In contrast, our method employs a single agent that autonomously decides when to invoke tools, interprets feedback, and iteratively revises kernels in a self-directed manner.

Kevin. Kevin introduces a multi-turn reinforcement learning framework that explicitly models the iterative CUDA development workflow. However, this approach partitions KernelBench into subsets for training and evaluation, effectively training on benchmark-derived tasks. As a result, the reported gains may partially reflect benchmark-specific adaptation rather than generalizable kernel generation capability, which limits the fairness of direct comparison with methods trained without access to KernelBench data.

CUDA-L1. CUDA-L1 directly constructs supervised fine-tuning data from KernelBench reference implementations and further applies reinforcement learning using execution-based rewards on the same benchmark tasks. This practice results in significant data leakage between training and evaluation, rendering the reported performance not directly comparable to approaches, such as ours, that strictly avoid using KernelBench for training.

ConCuR. ConCuR synthesizes CUDA kernels with reasoning traces generated by a Kevin-32B model and uses the resulting dataset to fine-tune KernelCoder. As previous mentioned, Kevin-32B is trained on a KernelBench subset, so the performance of KernelCoder do not reflect training from independently curated or real-world kernel optimization data.

D Case Study

we conduct an in-depth case study on CUDA Agent's optimization trajectories over Level 1 to Level 3 of KernelBench. For each level, we analyze a representative example and distill the key optimization applied by CUDA Agent.

D.1 Common Optimization Patterns

Across all three difficulty levels in KernelBench, we observe several recurring optimization patterns in the trajectories generated by CUDA Agent.

Algebraic Simplification and Operator Reduction. A dominant pattern is the use of algebraic reasoning to simplify high-level tensor expressions and reduce computational complexity. In Case D.2, an explicit diagonal matrix multiplication is reduced to row-wise scaling, eliminating an entire matrix construction and GEMM. In Case D.3, a matrix multiplication followed by reduction is transformed into a reduction over weights followed by a dot product. Such transformations replace generic, high-cost operators with simpler computations that are better aligned with the underlying mathematical structure.

Kernel Fusion. Another recurring pattern is kernel fusion for eliminating intermediate tensors. Sequences of logically related operations are merged into a single kernel to avoid intermediate tensor materialization and reduce kernel launch overhead. Examples include collapsing multiple arithmetic stages (summation, dot product, division, and scaling) into a single kernel in Case D.3, and fusing element-wise residual addition with activation functions in Case D.4. Fusion improves data locality and reduces global memory traffic.

Memory Access Optimization. Efficient memory usage emerges as a key theme across cases. The generated kernels emphasize coalesced global memory accesses, reduced memory footprint by avoiding large intermediate tensors, and judicious use of shared memory for intra-block reductions. Vectorized loads (e.g., `float4`) are employed when beneficial to maximize memory bandwidth utilization.

Hardware-Aware Optimization. For complex models, CUDA Agent demonstrates explicit awareness of GPU hardware capabilities and tailors the generated implementation accordingly. In Case D.4, the system enables TF32 computation for both matrix multiplications and convolution operations, allowing the execution to leverage Tensor Cores on modern GPUs. By selectively adopting lower-precision arithmetic where it is performance-critical yet numerically acceptable, CUDA Agent achieves substantial speedups without requiring manual intervention.

Library-Aware Optimization. In addition to custom kernel generation, CUDA Agent effectively leverages highly optimized vendor libraries when appropriate. In Case D.4, the system identifies opportunities to invoke fused cuDNN APIs, such as convolution with bias and activation, to replace multiple separate operators. By mapping high-level model semantics onto optimized library primitives, CUDA Agent reduces kernel launch overhead and benefits from mature, hardware-tuned implementations provided by cuDNN.

Overall, these common patterns illustrate how CUDA Agent systematically bridges algorithmic insight and low-level performance engineering to optimize diverse workloads ranging from simple linear algebra to full neural network building blocks.

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

class Model(nn.Module):
    """
    Simple model that performs a matrix multiplication of a diagonal matrix with another matrix.
    C = diag(A) * B
    """
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, A, B):
        """
        Performs the matrix multiplication.

        Args:
            A (torch.Tensor): A 1D tensor representing the diagonal of the diagonal matrix. Shape: (N,).
            B (torch.Tensor): A 2D tensor representing the second matrix. Shape: (N, M).

        Returns:
            torch.Tensor: The result of the matrix multiplication. Shape: (N, M).
        """
        return torch.diag(A) @ B

M = 4096
N = 4096

def get_inputs():
    A = torch.rand(N)
    B = torch.rand(N, M)
    return [A, B]

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

```

Figure 8 Reference operator for diagonal `matmul` (Case D.2).

```cuda
#include <cuda_runtime.h>

__global__ void diagonal_multiply_kernel(float* output, const float* A, const float* B) {
    int idx = blockIdx.x * 128 + threadIdx.x;
    int total_elements = 4096 * 4096;

    while (idx < total_elements) {
        int row = idx / 4096;
        output[idx] = A[row] * B[idx];
        idx += gridDim.x * 128;
    }
}

extern "C" void diagonal_multiply_launcher(float* output, const float* A, const float* B, cudaStream_t stream) {
    int blocks = 1024;
    diagonal_multiply_kernel<<<blocks, 128, 0, stream>>>(output, A, B);
}
```

Figure 9 Diagonal `matmul` kernel implementation (Case D.2).

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

class ModelNew(nn.Module):
    """
    Optimized model that performs diagonal multiplication without creating the diagonal matrix.
    C[i, j] = A[i] * B[i, j]
    """
    def __init__(self):
        super(ModelNew, self).__init__()

    def forward(self, A, B):
        """
        Performs the optimized diagonal multiplication.

        Args:
            A (torch.Tensor): A 1D tensor representing the diagonal of the diagonal matrix. Shape: (N,).
            B (torch.Tensor): A 2D tensor representing the second matrix. Shape: (N, M).

        Returns:
            torch.Tensor: The result of the multiplication. Shape: (N, M).
        """
        return cuda_extension.diagonal_multiply_forward(A, B)
```

Figure 10 Custom operator for diagonal `matmul` (Case D.2).

D.2 Example of Level 1: Model that performs matrix multiplication of a diagonal matrix with another matrix

In this case, the reference model computes the product of a diagonal matrix constructed from a vector and a dense matrix. Although mathematically simple, this formulation incurs unnecessary computational and memory overhead by explicitly materializing the diagonal matrix and invoking a general matrix multiplication (GEMM). Figures figures 8 to 10 illustrate the reference operator, the optimized CUDA kernel, and the resulting custom operator for this example. The Python/C++ binding is omitted for brevity. This custom operator achieves 73.31× speed-up versus Torch Compile.

CUDA Agent exploits the algebraic structure of diagonal matrix multiplication. Specifically, left-multiplying a matrix B by a diagonal matrix defined by vector A is equivalent to scaling each row i of B by the scalar $A[i]$. This observation reduces the computation from a matrix–matrix multiplication to an element-wise broadcast multiplication, lowering the time complexity from $O(N^2M)$ to $O(NM)$.

Based on this simplification, CUDA Agent implements a custom CUDA kernel that directly performs the row-wise scaling without constructing the intermediate diagonal matrix. The kernel uses a grid-stride loop to efficiently cover all elements of the output matrix, and each thread independently multiplies one element of B by the corresponding diagonal entry from A. This approach fuses the diagonal construction and matrix multiplication into a single kernel, significantly reducing kernel launch overhead and global memory traffic.

Overall, this case illustrates a common optimization pattern identified by CUDA Agent: recognizing implicit structure in high-level tensor expressions and replacing generic operators with specialized kernels that directly implement the underlying mathematical operation.

D.3 Example of Level 2: Model that performs matrix multiplication, division, summation, and scaling

The second case involves a model composed of a sequence of operations, including matrix multiplication, division, summation, and scaling. Executed naively, this pipeline materializes large intermediate tensors and performs redundant computations. Figures figures 11 to 13 show the reference operator, the fused CUDA kernel, and the final custom operator produced by CUDA Agent. The Python/C++ binding is omitted for brevity. This custom operator achieves 24.04× speed-up versus Torch Compile.

CUDA Agent begins with an algebraic rearrangement of the computation. By exploiting the linearity of summation and matrix multiplication, the operation can be rewritten as

$$\sum_j \frac{x_i \cdot w_j^T}{2} = x_i \cdot \left(\sum_j w_j^T \right) /2,$$

transforming a matrix–matrix multiplication followed by a reduction into a column-wise reduction of the weight matrix followed by a dot product. This significantly reduces both the number of floating-point operations and memory accesses.

CUDA Agent implement this transformation using two custom CUDA kernels. The first kernel computes the column-wise sum of the weight matrix with fully coalesced memory accesses. The second kernel performs the dot product between the input vector and the reduced weight vector, while simultaneously applying division and scaling. These operations are fused into a single kernel to avoid intermediate writes to global memory.

To further improve performance, the dot product kernel leverages vectorized memory access using `float4` loads, maximizing memory bandwidth utilization. A shared-memory tree reduction is used within each thread block to accumulate partial sums efficiently, reducing synchronization overhead and avoiding expensive global atomics.

This case demonstrates how CUDA Agent systematically combines algorithmic simplification with kernel fusion and low-level CUDA optimizations to collapse a multi-operator computation graph into a small number of highly efficient kernels.

```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 11 Reference operator for matrix multiplication, division, summation, and scaling (Case D.3).

```cpp
#include <cuda_runtime.h>

__global__ void sum_weight_kernel(const float* weight, float* sum_weight) {
    int tid = blockIdx.x * 128 + threadIdx.x;

    if (tid < 8192) {
        float sum = 0.0f;

        // Each thread processes one input_size element (tid)
        // and sums over hidden_size
        for (int j = 0; j < 8192; j++) {
            // Calculate base index for this j and tid
            int base = j * 8192 + tid;
            sum += weight[base];
        }

        sum_weight[tid] = sum;
    }
}

__global__ void dot_product_kernel(const float* x, const float* sum_weight, float scaling_factor, float* output) {
    int tid = threadIdx.x;
    int block = blockIdx.x;

    __shared__ float smem[128];

    if (tid < 128 && block < 1024) {
        float dot = 0.0f;

        // Use float4 to load more data at once
        for (int k = tid * 4; k < 8192; k += 128 * 4) {
            float4 xv = *reinterpret_cast<float4*>(const_cast<float*>(x) + block * 8192 + k);
            float4 swv = *reinterpret_cast<float4*>(const_cast<float*>(sum_weight) + k);

            dot += xv.x * swv.x;
            dot += xv.y * swv.y;
            dot += xv.z * swv.z;
            dot += xv.w * swv.w;
        }

        smem[tid] = dot;
        __syncthreads();

        // Reduce within block
        for (int s = 64; s > 0; s >>= 1) {
            if (tid < s) {
                smem[tid] += smem[tid + s];
            }
            __syncthreads();
        }

        if (tid == 0) {
            output[block] = smem[0] * (scaling_factor / 2.0f);
        }
    }
}

extern "C" void fused_sum_dot_launcher(
    const float* x,
    const float* weight,
    float scaling_factor,
    float* output) {

    // First sum the weight
    float* sum_weight;
    cudaMalloc(&sum_weight, 8192 * sizeof(float));

    sum_weight_kernel<<<(8192 + 128 - 1)/128, 128>>>(weight, sum_weight);

    // Then compute dot product
    dot_product_kernel<<<1024, 128>>>(x, sum_weight, scaling_factor, output);

    cudaFree(sum_weight);
}
```

Figure 12 Fused `sum-then-dot-product` kernel implementation (Case D.3).

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

class ModelNew(nn.Module):
    """
    Optimized model that fuses sum, dot product, and scaling operations.
    """
    def __init__(self, input_size, hidden_size, scaling_factor):
        super(ModelNew, 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 = cuda_extension.fused_sum_dot_forward(x, self.weight, self.scaling_factor)
        return x
```

Figure 13 Custom operator for matrix multiplication, division, summation, and scaling (Case D.3).

D.4 Example of Level 3: ResNet BasicBlock

The third case studies a ResNet BasicBlock, representative of realistic deep learning workloads composed of convolutions, normalization, nonlinearities, and residual connections. The reference implementation follows the standard PyTorch execution path, resulting in multiple kernel launches for convolution, batch normalization, bias addition, activation, and residual addition. Figures figures 14 to 18 present the reference ResNet BasicBlock, the optimized cuDNN-based convolution, the fused add–ReLU kernel, and the resulting custom operator. The Python/C++ binding is omitted for brevity. This custom operator achieves 3.59× speed-up versus Torch Compile.

CUDA Agent applies several complementary techniques. First, it manually folds the BatchNorm parameters into the preceding convolution weights and bias in the Python model, eliminating the BatchNorm operator entirely at inference time. This reduces both kernel launches and memory accesses while preserving numerical equivalence.

Second, by using `cudnnConvolutionBiasActivationForward`, CUDA Agent modifies the custom cuDNN convolution wrapper to enable convolution, bias addition, and ReLU activation to be executed within a single cuDNN kernel. This further reduces kernel launch overhead and improves data locality. We explicitly enable TF32 computation for cuDNN and matrix multiplication operations, allowing the implementation to leverage Tensor Cores on Hopper GPUs.

CUDA Agent also explores switching the data layout from NCHW to NHWC to better align with Tensor Core requirements. However, the required layout conversions introduced significant overhead, offsetting the potential gains. As a result, the final implementation retains the NCHW layout while using NCHW-compatible cuDNN APIs.

Finally, CUDA Agent fuses the residual addition and the final ReLU activation into a single custom CUDA kernel, replacing two separate element-wise operators. This kernel computes the element-wise sum of the main and residual branches and immediately applies the ReLU function.

Together, these optimizations reduce the number of kernel launches, improve arithmetic intensity, and better

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

class Model(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        """
        :param in_channels: Number of input channels
        :param out_channels: Number of output channels
        :param stride: Stride for the first convolutional layer
        :param downsample: Downsample layer for the shortcut connection
        """
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels * self.expansion, kernel_size=1, stride=stride, bias=False),
            nn.BatchNorm2d(out_channels * self.expansion),
        )
        self.stride = stride

    def forward(self, x):
        """
        :param x: Input tensor, shape (batch_size, in_channels, height, width)
        :return: Output tensor, shape (batch_size, out_channels, height, width)
        """
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out

# Test code
in_channels = 3
out_channels = 64
stride = 1
batch_size = 10
num_classes = 1000

def get_inputs():
    return [torch.rand(batch_size, in_channels, 224, 224)]

def get_init_inputs():
    return [in_channels, out_channels, stride]

```

Figure 14 Reference operator for Resnet BasicBlock (Case D.4).

```cpp
// Pure cuDNN convolution implementation
#include <cudnn.h>
#include <cuda_runtime.h>
#include <vector>
#include <memory>
#include <stdexcept>
#include <string>

#define CHECK_CUDNN(status) \
    do { \
        cudnnStatus_t err = (status); \
        if (err != CUDNN_STATUS_SUCCESS) { \
            throw std::runtime_error(std::string("cuDNN error: ") + cudnnGetErrorString(err)); \
        } \
    } while(0)

struct CudnnDeleter {
    void operator()(cudnnTensorDescriptor_t d) { cudnnDestroyTensorDescriptor(d); }
    void operator()(cudnnFilterDescriptor_t d) { cudnnDestroyFilterDescriptor(d); }
    void operator()(cudnnConvolutionDescriptor_t d) { cudnnDestroyConvolutionDescriptor(d); }
};

extern "C" void conv_cuda_forward(
    cudnnHandle_t handle,
    const float* input, const float* weight, const float* bias, float* output,
    const int* input_dims, const int* weight_dims, const int* output_dims,
    const int* stride, const int* padding, const int* dilation,
    int ndim, void* workspace, size_t ws_size,
    bool use_relu) {

    // Create descriptors
    cudnnTensorDescriptor_t input_desc, output_desc, bias_desc = nullptr;
    cudnnFilterDescriptor_t weight_desc;
    cudnnConvolutionDescriptor_t conv_desc;

    CHECK_CUDNN(cudnnCreateTensorDescriptor(&input_desc));
    CHECK_CUDNN(cudnnCreateTensorDescriptor(&output_desc));
    CHECK_CUDNN(cudnnCreateFilterDescriptor(&weight_desc));
    CHECK_CUDNN(cudnnCreateConvolutionDescriptor(&conv_desc));

    // Set descriptors using 4D helper for common case (NCHW)
    int n = input_dims[0];
    int c = input_dims[1];
    int h = input_dims[2];
    int w = input_dims[3];

    CHECK_CUDNN(cudnnSetTensor4dDescriptor(input_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, n, c, h, w));
    CHECK_CUDNN(cudnnSetFilter4dDescriptor(weight_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, weight_dims[0], weight_dims[1], weight_dims[2], weight_dims[3]));
    CHECK_CUDNN(cudnnSetConvolution2dDescriptor(conv_desc, padding[0], padding[1], stride[0], stride[1], dilation[0], dilation[1], CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));

    // Enable Tensor Cores / TF32
    CHECK_CUDNN(cudnnSetConvolutionMathType(conv_desc, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION));

    int out_n, out_c, out_h, out_w;
    CHECK_CUDNN(cudnnGetConvolution2dForwardOutputDim(conv_desc, input_desc, weight_desc, &out_n, &out_c, &out_h, &out_w));
    CHECK_CUDNN(cudnnSetTensor4dDescriptor(output_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, out_n, out_c, out_h, out_w));

    float alpha = 1.0f;
    float beta = 0.0f;

    // Get algorithm
    cudnnConvolutionFwdAlgo_t algo;
    int requestedAlgoCount = 1;
    int returnedAlgoCount;
    cudnnConvolutionFwdAlgoPerf_t perfResults;
    CHECK_CUDNN(cudnnGetConvolutionForwardAlgorithm_v7(handle, input_desc, weight_desc, conv_desc, output_desc, 1, &requestedAlgoCount, &perfResults));
    algo = perfResults.algo;

    if (bias) {
        CHECK_CUDNN(cudnnCreateTensorDescriptor(&bias_desc));
        std::vector<int> b_dims(ndim, 1), b_str(ndim, 1);
        b_dims[1] = weight_dims[0];
        b_str[0] = weight_dims[0]; // Stride logic matching NCHW
        CHECK_CUDNN(cudnnSetTensorNdDescriptor(bias_desc, CUDNN_DATA_FLOAT, ndim, b_dims.data(), b_str.data()));

        cudnnActivationDescriptor_t act_desc;
        CHECK_CUDNN(cudnnCreateActivationDescriptor(&act_desc));
        if (use_relu) {
            CHECK_CUDNN(cudnnSetActivationDescriptor(act_desc, CUDNN_ACTIVATION_RELU, CUDNN_NOT_PROPAGATE_NAN, 0.0));
        } else {
            CHECK_CUDNN(cudnnSetActivationDescriptor(act_desc, CUDNN_ACTIVATION_IDENTITY, CUDNN_NOT_PROPAGATE_NAN, 0.0));
        }

        CHECK_CUDNN(cudnnConvolutionBiasActivationForward(
            handle, &alpha, input_desc, input, weight_desc, weight, conv_desc, algo, workspace, ws_size,
            &beta, output_desc, output, bias_desc, bias, act_desc, output_desc, output));

        CHECK_CUDNN(cudnnDestroyActivationDescriptor(act_desc));
        CHECK_CUDNN(cudnnDestroyTensorDescriptor(bias_desc));
    } else {
        CHECK_CUDNN(cudnnConvolutionForward(
            handle, &alpha, input_desc, input, weight_desc, weight, conv_desc, algo,
            workspace, ws_size, &beta, output_desc, output));
    }

    // Cleanup
    CHECK_CUDNN(cudnnDestroyTensorDescriptor(input_desc));
    CHECK_CUDNN(cudnnDestroyTensorDescriptor(output_desc));
    CHECK_CUDNN(cudnnDestroyFilterDescriptor(weight_desc));
    CHECK_CUDNN(cudnnDestroyConvolutionDescriptor(conv_desc));
}
```

Figure 15 cuDNN convolution implementation, part 1 (Case D.4).

```
extern "C" size_t conv_cuda_workspace_size(
    cudnnHandle_t handle, const int* input_dims, const int* weight_dims,
    const int* output_dims, const int* stride, const int* padding,
    const int* dilation, int ndim) {

    cudnnTensorDescriptor_t in_d, out_d;
    cudnnFilterDescriptor_t w_d;
    cudnnConvolutionDescriptor_t c_d;

    CHECK_CUDNN(cudnnCreateTensorDescriptor(&in_d));
    CHECK_CUDNN(cudnnCreateTensorDescriptor(&out_d));
    CHECK_CUDNN(cudnnCreateFilterDescriptor(&w_d));
    CHECK_CUDNN(cudnnCreateConvolutionDescriptor(&c_d));

    int n = input_dims[0], c = input_dims[1], h = input_dims[2], w = input_dims[3];
    int k = weight_dims[0]; // out_channels

    CHECK_CUDNN(cudnnSetTensor4dDescriptor(in_d, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, n, c, h, w));
    CHECK_CUDNN(cudnnSetFilter4dDescriptor(w_d, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, k, c, weight_dims[2], weight_dims[3]));
    CHECK_CUDNN(cudnnSetConvolution2dDescriptor(c_d, padding[0], padding[1], stride[0], stride[1], dilation[0], dilation[1], CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
    CHECK_CUDNN(cudnnSetConvolutionMathType(c_d, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION));

    int out_n, out_c, out_h, out_w;
    CHECK_CUDNN(cudnnGetConvolution2dForwardOutputDim(c_d, in_d, w_d, &out_n, &out_c, &out_h, &out_w));
    CHECK_CUDNN(cudnnSetTensor4dDescriptor(out_d, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, out_n, out_c, out_h, out_w));

    cudnnConvolutionFwdAlgo_t algo;
    int requestedAlgoCount = 1;
    cudnnConvolutionFwdAlgoPerf_t perfResults;
    CHECK_CUDNN(cudnnGetConvolutionForwardAlgorithm_v7(handle, in_d, w_d, c_d, out_d, 1, &requestedAlgoCount, &perfResults));
    algo = perfResults.algo;

    size_t size = 0;
    CHECK_CUDNN(cudnnGetConvolutionForwardWorkspaceSize(handle, in_d, w_d, c_d, out_d, algo, &size));

    CHECK_CUDNN(cudnnDestroyTensorDescriptor(in_d));
    CHECK_CUDNN(cudnnDestroyTensorDescriptor(out_d));
    CHECK_CUDNN(cudnnDestroyFilterDescriptor(w_d));
    CHECK_CUDNN(cudnnDestroyConvolutionDescriptor(c_d));

    return size;
}
```

Figure 16 `cuDNN` convolution implementation, part 2 (Case D.4).

```
#include <cuda_runtime.h>
#include <algorithm>

__global__ void fused_add_relu_kernel(float* output, const float* input1, const float* input2, int total_elements) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < total_elements / 4; i += stride) {
        float4 v1 = reinterpret_cast<const float4*>(input1)[i];
        float4 v2 = reinterpret_cast<const float4*>(input2)[i];
        float4 res;

        res.x = fmaxf(v1.x + v2.x, 0.0f);
        res.y = fmaxf(v1.y + v2.y, 0.0f);
        res.z = fmaxf(v1.z + v2.z, 0.0f);
        res.w = fmaxf(v1.w + v2.w, 0.0f);

        reinterpret_cast<float4*>(output)[i] = res;
    }

    int remaining_start = (total_elements / 4) * 4;
    for (int i = remaining_start + idx; i < total_elements; i += stride) {
        output[i] = fmaxf(input1[i] + input2[i], 0.0f);
    }
}

extern "C" void fused_add_relu_launcher(float* output, const float* input1, const float* input2, int total_elements, cudaStream_t stream) {
    int block_size = 256;
    int grid_size = (total_elements + block_size - 1) / block_size;
    if (grid_size > 4096) grid_size = 4096;

    fused_add_relu_kernel<<<grid_size, block_size, 0, stream>>>(output, input1, input2, total_elements);
}
```

Figure 17 Fused `add-relu` kernel implementation (Case D.4).

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

class ModelNew(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super(ModelNew, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels * self.expansion, kernel_size=1, stride=stride, bias=False),
            nn.BatchNorm2d(out_channels * self.expansion),
        )
        self.stride = stride

    def forward(self, x):
        # Enable TF32 precision
        torch.backends.cuda.matmul.allow_tf32 = True
        torch.backends.cudnn.allow_tf32 = True

        identity = x

        # Fold batch norm 1 into conv1
        gamma = self.bn1.weight
        beta = self.bn1.bias
        mean = self.bn1.running_mean
        var = self.bn1.running_var
        eps = self.bn1.eps

        inv_std = torch.rsqrt(var + eps)
        folded_weight = self.conv1.weight * inv_std[:, None, None, None]
        folded_bias = beta - mean * inv_std

        # Conv1 with folded batch norm and relu
        out = cuda_extension.conv_forward(x, folded_weight, folded_bias,
                                          list(self.conv1.stride),
                                          list(self.conv1.padding),
                                          list(self.conv1.dilation),
                                          use_relu=True)

        # Fold batch norm 2 into conv2
        gamma = self.bn2.weight
        beta = self.bn2.bias
        mean = self.bn2.running_mean
        var = self.bn2.running_var
        eps = self.bn2.eps

        inv_std = torch.rsqrt(var + eps)
        folded_weight = self.conv2.weight * inv_std[:, None, None, None]
        folded_bias = beta - mean * inv_std

        # Conv2 with folded batch norm
        out = cuda_extension.conv_forward(out, folded_weight, folded_bias,
                                          list(self.conv2.stride),
                                          list(self.conv2.padding),
                                          list(self.conv2.dilation),
                                          use_relu=False)

        # Downsample with folded batch norm
        if self.downsample is not None:
            # Fold batch norm into downsample conv
            gamma = self.downsample[1].weight
            beta = self.downsample[1].bias
            mean = self.downsample[1].running_mean
            var = self.downsample[1].running_var
            eps = self.downsample[1].eps

            inv_std = torch.rsqrt(var + eps)
            folded_weight = self.downsample[0].weight * inv_std[:, None, None, None]
            folded_bias = beta - mean * inv_std

            identity = cuda_extension.conv_forward(x, folded_weight, folded_bias,
                                                   [1, self.stride],
                                                   [0, 0],
                                                   [1, 1],
                                                   use_relu=False)

        # Fused Add + ReLU
        out = cuda_extension.fused_add_relu_forward(out, identity)

        # Disable TF32 precision
        torch.backends.cuda.matmul.allow_tf32 = False
        torch.backends.cudnn.allow_tf32 = False

        return out
```

Figure 18 Custom operator for Resnet BasicBlock (Case D.4).

exploit hardware acceleration features. This case highlights CUDA Agent 's ability to integrate high-level graph transformations with library-level fusion and custom kernel design in complex, real-world neural network blocks.

E Limitations

Our study has two main limitations. First, we do not compare CUDA Agent against more sophisticated compiler frameworks such as TVM. While these systems can potentially provide stronger baselines, they are difficult to integrate into a large-scale RL training loop with thousands of rollouts due to their substantial tuning overhead and complex deployment requirements; we therefore focus on `torch.compile` as a widely adopted, training-friendly baseline. Second, our training pipeline relies on a large GPU pool with process-level isolation, which incurs considerable computational and engineering cost. This reliance on massive GPU resources may limit accessibility for broader research community, and we leave exploring more resource-efficient training strategies as important directions for future work.