CUDACOACH: GUIDING CUDA KERNEL OPTIMIZATION WITH DIVERSE STRATEGIES LEARNED FROM VERIFIED CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimizing CUDA kernels is critical for high-performance deep learning systems. Recently, multi-role iterative frameworks powered by large language models have advanced this optimization process by establishing an automated "planning-generation-execution" pipeline, which effectively enhances kernel correctness and performance. However, improving strategy planning within these frameworks requires addressing two challenges: (a) Off-the-shelf planners repeat generic strategies across kernels with different performance profiles, limiting the search for faster implementations. (b) Supervised training to improve these recommendations requires reliable strategy demonstrations, but successful kernel implementations alone do not provide explicit modification guidance. We introduce **CUDACoach**, a framework for learning repair and optimization strategies to guide a frozen coder. (a) To obtain reliable demonstrations for supervised initialization, we propose *verified strategy mining*, which reconstructs strategies from successful kernel edits and validates them through implementation by the code model. (b) To counter repetitive optimization strategies, we use a *two-level diversity reward* during reinforcement learning to encourage distinct technique combinations among candidates and down-weight techniques frequently used across training. We evaluate CUDACoach on all 250 KernelBench tasks with three downstream coders. With Qwen3-Coder-Plus as the coder, our method achieves 60.4% correctness and produces correct kernels faster than PyTorch on 19.2% of tasks, compared with 44.8% and 13.2% for single-stage iteration and 42.4% and 13.6% when the coder also serves as the planner. Ablations confirm the benefits of diversity-reward training and the multi-kernel context for kernel correctness and optimization.

1 INTRODUCTION

High-performance CUDA kernels are central to the training and inference efficiency of modern deep learning systems. Developing such high-performance CUDA kernels requires substantial hardware expertise and iterative profiling (Ouyang et al., 2025). Driven by this complexity, automating CUDA kernel generation and optimization via large language models (LLMs) has emerged as a critical research frontier (Chen et al., 2025b; Meta AI, 2025; Yu et al., 2026).

Recently, advanced automated CUDA kernel generation methods are mainly built upon the multi-role iterative framework (Zhang et al., 2025; Zhu et al., 2026; Chen et al., 2025a; Dong et al., 2025), which typically involve two key steps. The first step is the "strategy step," in which a planner LLM recommends acceleration or correction techniques based on profiling and error feedback. The second step is the "generation step," where a coder LLM synthesizes the optimized CUDA kernel guided by the recommendations from the first step.

However, existing methods fundamentally lack strategy diversity, consistently generating similar optimization strategies regardless of the underlying profiling messages, as they rely on off-the-shelf LLMs as generic planners without a learned mapping from profiling messages to specific acceleration techniques. To substantiate this drawback, we evaluate an off-the-shelf planner on 61 tasks. As illustrated in Figure1(a), the planner consistently suggests similar technique combinations for diverse kernels. Since these similar techniques are not tailored to the specific bottlenecks of each kernel,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

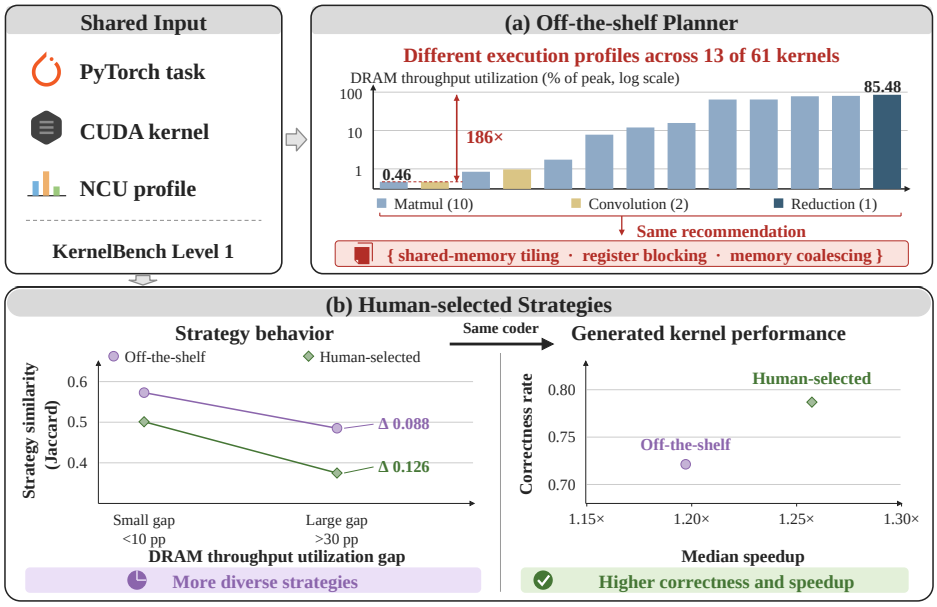

Figure 1: Motivation for diverse, kernel-specific strategies. (a) An off-the-shelf planner repeats the same strategy across kernels with different performance profiles. (b) Off-the-shelf and human-selected strategies. Left, Jaccard similarity of recommended technique sets within operator families. Right, generated-kernel performance with the same coder and downstream generation budget. Correctness rate is the fraction of three-candidate groups containing at least one correct implementation. Speedup is relative to the input kernel.

the optimization outcomes are highly inconsistent, with DRAM throughput utilization spanning a massive gap from 0.46% to 85.48% of peak bandwidth. Conversely, to examine whether more varied strategies can improve optimization, Figure1(b) compares these off-the-shelf recommendations with human-selected strategies. The results show that human-selected strategies yield more diverse technique sets, achieving higher correctness (78.7% vs. 72.1%) and better median speedups (1.26× vs. 1.20×) under the same generation budget. This contrast conclusively demonstrates that it is imperative for the planner to generate diverse, bottleneck-aware techniques.

To significantly boost the diversity of optimization strategies generated by the planner LLM, employing supervised fine-tuning (SFT) emerges as a highly compelling solution. While highly promising, applying SFT to the planner faces a critical hurdle: the scarcity of explicit strategy demonstrations. Supervised fine-tuning demands clear instructions on which strategies to apply given specific profiling or error feedback. Yet, successful kernel optimizations typically leave these strategic decisions implicitly entangled in the final code (Zhang et al., 2026). Therefore, an accelerated kernel cannot be directly used as an SFT target for the planner until the underlying strategic rationale is explicitly recovered.

We address the above two challenges with **CUDACoach**, a training framework for high-diversity LLM planners in the "strategy step". Specifically, to address the lack of explicit SFT labels, we propose *verified strategy mining* to extract candidate strategies by comparing an original kernel with an optimized revision. Each extracted strategy is then applied to the original kernel in isolation. We retain a strategy as SFT supervision only if the resulting kernel resolves functional errors or achieves higher performance while maintaining correctness. To achieve higher strategy diversity, we design a *two-level diversity reward* and fine-tune the planner via reinforcement learning. This reward explicitly downweights globally overused techniques and encourages diverse technique combinations for individual kernels.

To sum up, our contributions are as follows:

- We introduce **CUDACoach** to train an LLM planner that guides a frozen LLM coder in CUDA kernel optimization tasks, maximizing strategy diversity through verified strategy mining and a two-level diversity reward.

- We propose **verified strategy mining** to automatically construct annotated SFT datasets for LLM planners, by extracting explicit strategies from an original kernel and a better revision, and verifying their effectiveness through code generation and execution.
- We design a **two-level diversity reward** for reinforcement learning to encourage diverse strategy recommendations for kernel optimization. It assigns higher weights to techniques used less frequently across training and rewards diverse technique combinations among strategies sampled for the same kernel.
- Experiments on all 250 KernelBench tasks with three downstream coders demonstrate the effectiveness of CUDACoach. With Qwen3-Coder-Plus, it improves correctness from 44.8% to 60.4% and fast_1 from 13.2% to 19.2% over single-stage iteration. The diversity reward further increases strategy diversity from 0.440 to 0.510 and reduces the most common technique-set frequency from 16.9% to 6.7%.

2 Related Work

We organize related work on LLM-based GPU kernel generation into two categories. One trains kernel-specialized models, and the other builds training-free agentic frameworks that orchestrate off-the-shelf models.

2.1 Kernel-Specific Model Training

Training a model directly for the domain is one dominant approach. KernelLLM (Meta AI, 2025) fine-tunes Llama-3.1-8B on a PyTorch-to-Triton dataset and surpasses much larger general models. Because kernel generation exposes verifiable rewards such as correctness and speedup, most later work adopts reinforcement learning. These RL methods differ mainly in their reward and training design (Li et al., 2025a;b; Baronio et al., 2025; Kong et al., 2025; Woo et al., 2025). A parallel line specializes RL to one operator family such as sparse or dense matrix multiplication (Wang et al., 2026; Su et al., 2025). Other work scales this up through unified RL environments (Liu et al., 2026), end-to-end terminal agents (Dai et al., 2026), execution-verified skill libraries (Fu et al., 2026), and reasoning transfer into smaller models (Gong et al., 2025). Across this line, one model carries code understanding, bottleneck attribution, planning and implementation at once.

QiMeng-Kernel (Zhu et al., 2026) is closely related to our work. It trains a lightweight policy to select semantic optimization actions for a general-purpose LLM to implement. Each action combines an optimization type from an expert-designed set with a candidate code region. CUDACoach instead generates natural-language strategies with concrete modification guidance, using error feedback for kernel repair and performance profiles for optimization. Verified strategy mining provides supervision for planner initialization, while a two-level diversity reward encourages exploration of different optimization techniques during reinforcement learning.

2.2 Agentic Kernel Optimization Frameworks

Rather than training a model, training-free agentic systems orchestrate off-the-shelf LLMs through elaborate harnesses. Many coordinate specialized agents that plan, generate, and reflect in an iterative loop (Wang et al., 2025; Dong et al., 2025; Du et al., 2025; Wei et al., 2025), and some add analytics that map profiling diagnostics to code transformations (Qu et al., 2026; Zaeed et al., 2026). CudaForge (Zhang et al., 2025) pairs a Coder with a Judge from the same base model and shows that role decomposition alone improves performance. CUDAMaster (Han et al., 2026) extends this line to sparse and scientific computing workloads, filtering profiling metrics by a fixed bottleneck rule before handing them to an untrained planner agent.

A growing subset augments agents with experience and skill memories, ranging from context and trajectory stores to dual-level and self-evolving knowledge bases (KernelEvolve Team, Meta Platforms, 2025; Du et al., 2026; Sun et al., 2026; Zheng et al., 2026). KernelBlaster (Dong et al., 2026) uses memory-augmented in-context RL to select optimizations at inference time without weight updates, and AVO (Chen et al., 2026a) replaces evolutionary mutation and crossover with a knowledge-base-driven agent. A parallel methodological line treats strategy as an explicit intermediate representation (Chen et al., 2025a) and strengthens the surrounding pipeline with formalized code

evolution, reusable skills, quality-diversity search, bandit control, and harder benchmarks (Guo et al., 2025; Zhang et al., 2026; Wiedemann et al., 2026; Ran et al., 2025; Lange et al., 2025).

Across this category the harness is elaborate yet training-free, so diagnosing a kernel's bottleneck and choosing an optimization falls to an untrained model bounded by its priors. Our work learns this step rather than only orchestrating it.

3 METHOD

CUDACoach learns repair and optimization strategies that guide a frozen coder in iteratively improving CUDA kernels. Each iteration proposes a strategy from the current kernel and its execution feedback, implements that strategy with the frozen coder, and evaluates the result to produce the feedback for the next iteration. Figure 2 summarizes the framework, and Section 3.1 formalizes this loop together with the task-level objective it optimizes. The proposed framework contains the following three key components. First, verified strategy mining (Section 3.2) constructs supervision by recovering strategies from successful kernel edits and verifying their effectiveness through code generation and execution. Second, execution-aligned training (Section 3.3) initializes the planner by supervised fine-tuning and then applies reinforcement learning with execution-based task rewards and a two-level diversity reward. Third, compositional inference (Section 3.4) combines optimization techniques across implementations to iteratively improve kernels.

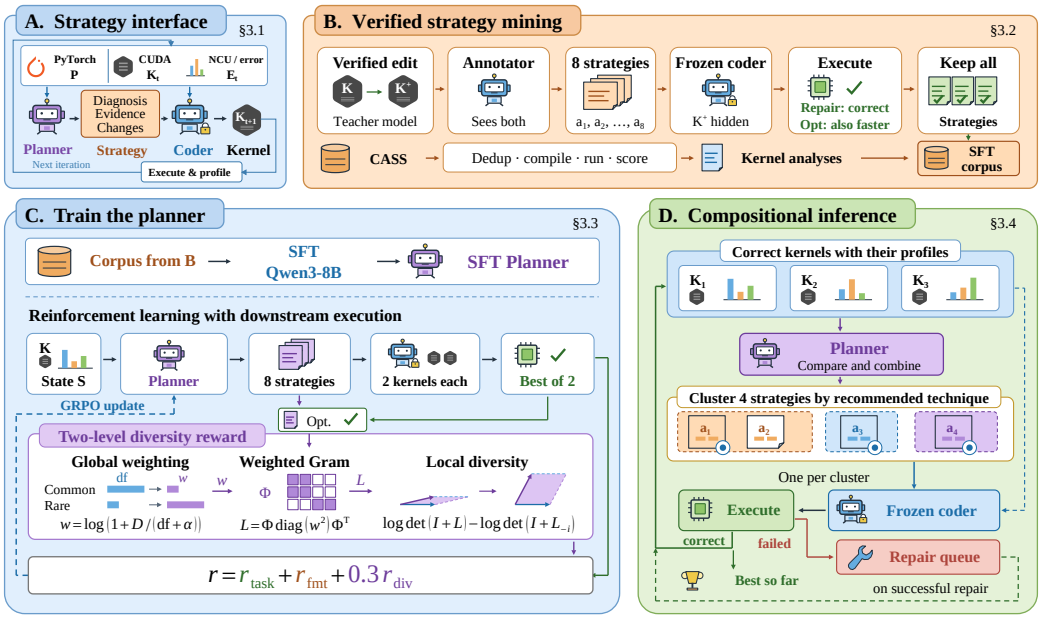

Figure 2: Overview of CUDACoach. (A) At each iteration the planner observes the current kernel together with its profile or error feedback and emits a strategy, which the coder realizes as a candidate kernel. (B) Verified strategy mining recovers strategies from successful kernel edits and validates them through code generation and execution. Verified strategies and CASS kernel analyses form the SFT corpus. (C) Supervised fine-tuning initializes the planner, after which reinforcement learning trains it with a combined reward of execution-based task and format terms plus a two-level diversity reward over the recommended techniques. (D) Compositional inference combines optimization techniques from multiple correct kernels and their profiles, while failed kernels undergo repair and the best correct kernel remains in the pool.

3.1 PROBLEM FORMULATION AND STRATEGY INTERFACE

Each task provides a PyTorch reference program P and target hardware H, and we seek a CUDA kernel that reproduces the outputs of P and minimizes latency on H. CUDACoach reaches this goal

by repairing and optimizing an existing kernel iteratively, as shown in Figure 2(A). At iteration t, the state $S_t = (P, K_t, E_t, H)$ contains the current kernel K_t and its execution feedback E_t, which is an error report for repair or a performance profile from Nsight Compute (NCU) (NVIDIA Corporation, 2026) for optimization. From this state the planner π_θ emits a natural-language strategy a_t which is a structured document. It states a diagnosis of the dominant bottleneck, the evidence supporting that diagnosis, and the changes to make. The frozen coder G_ϕ then realizes the strategy as a candidate kernel K_{t+1}, which is compiled and evaluated to produce the feedback for the next iteration. The strategy is the interface between the two models, and only the planner is trained.

Our goal is to learn strategies that restore correctness for failing kernels and maximize speedup over the current implementation for correct kernels. We formulate this goal as the following task objective:

$$\max_{\theta} \mathbb{E}_{\substack{S_t \sim \mathcal{D} \\ a_t \sim \pi_\theta(\cdot|S_t) \\ K_{t+1} \sim G_\phi(\cdot|S_t, a_t)}} \left[\begin{cases} \dfrac{\ell(K_t)}{\ell(K_{t+1})}, & V_P(K_t) = V_P(K_{t+1}) = 1, \\ V_P(K_{t+1}), & \text{otherwise.} \end{cases} \right]. \tag{1}$$

Here, $\mathcal{D}$ denotes the distribution of repair and optimization states. $V_P(K) \in \{0, 1\}$ indicates whether K compiles, runs, and passes numerical checks against P. $\ell(K)$ denotes its measured latency on H.

3.2 Verified Strategy Mining

Successful kernel edits provide examples of effective changes, but do not explicitly record the underlying strategies. Our verified strategy mining, shown in Figure 2(B), first constructs successful edits and then recovers strategies from them. We refer to the input kernel before each modification as the parent kernel. Starting from a kernel and its execution feedback in state S, we use Claude Opus 4.5 as the teacher model to repair an incorrect implementation or optimize a correct one. We execute the revised kernel and retain it as K^+ only if it restores correctness or reduces latency while preserving correctness. For each retained edit, an annotator uses S, K^+, and the teacher response to reconstruct eight candidate strategies.

To verify that these strategies can guide code generation, the frozen coder applies each one to the original kernel without access to K^+. We accept a repair strategy if the new implementation compiles, runs, and passes numerical checks. An optimization strategy must additionally produce an implementation faster than the parent. The new implementation need not reproduce the teacher's code. This test verifies whether the recovered strategy can guide a successful modification, rather than merely describe an existing one. We retain all verified strategies as supervision.

We also include auxiliary analyses of CUDA kernels extracted from CASS (Heakl et al., 2025). We deduplicate the kernels, perform compilation and runtime checks, and select them based on code quality and difficulty. Claude Opus 4.5 then generates a structured analysis of each kernel, and we keep the analyses that pass a quality filter. Appendix B details the construction and filtering procedures, and Table 5 gives the corpus composition.

3.3 Execution-Aligned Planner Training

We initialize the planner by supervised fine-tuning Qwen3-8B (Team, 2025) on the mixed corpus described in Section 3.2. During reinforcement learning, the planner samples $N = 8$ strategies for each state S, and the frozen coder generates $M = 2$ implementations per strategy. We compile and evaluate these implementations to compute task rewards based on correctness and speedup. Figure 2(C) summarizes the two stages, and Appendix C.1 gives the sampling and execution settings.

We define task rewards separately for repair and optimization. For a repair state, $r_{\text{code}}(S, K') = 2V_P(K') - 1$, giving $+1$ for correctness and -1 otherwise. For optimization, let $s = b(S)/\ell(K')$, where $\ell(K')$ is the candidate latency and $b(S)$ is the parent latency. The reward is

$$r_{\text{code}}(S, K') = \begin{cases} -1, & V_P(K') = 0, \\ 0, & V_P(K') = 1 \text{ and } s \le 1.1, \\ 0.5 + \min(\ln s, 2), & V_P(K') = 1 \text{ and } s > 1.1. \end{cases} \tag{2}$$

The three cases distinguish failure, correctness without sufficient improvement, and meaningful acceleration. The logarithmic cap limits extreme speedups. Strategy i receives the best reward among its downstream implementations, $r_{\text{task},i} = \max_j r_{\text{code}}(S, K'_{ij})$.

Our two-level diversity reward combines historical technique-frequency weighting with a within-group diversity bonus adapted from DQO (Chen et al., 2026b). A fixed annotator represents each strategy as a binary vector of recommended techniques.

To discourage repeated recommendations across training, we assign each technique u a weight based on its frequency in earlier batches:

$$w_u = \log\left(1 + \frac{D}{\mathrm{df}(u) + \alpha}\right), \tag{3}$$

where D counts previous optimization responses with nonempty technique labels, $\mathrm{df}(u)$ counts those containing u, and $\alpha = 0.5$. This gives lower weights to more frequently recommended techniques.

To encourage distinct technique combinations within each state, we consider optimization strategies with nonempty technique labels and at least one correct downstream implementation. Stacking their binary vectors gives $\Phi \in \{0,1\}^{k\times d}$, where k is the number of eligible strategies and d is the number of technique labels. We incorporate the historical weights into the Gram matrix

$$L = \Phi\,\mathrm{diag}(w_1^2, \ldots, w_d^2)\Phi^\top. \tag{4}$$

and assign each eligible strategy a bonus equal to its contribution to the group's log-determinant diversity:

$$r_{\mathrm{div},i} = \log\det(I_k + L) - \log\det(I_{k-1} + L_{-i}), \tag{5}$$

where I_k is the identity matrix and L_{-i} removes row and column i. Appendix C.3 specifies the technique annotation and the frequency updates.

We optimize the planner with GRPO (Shao et al., 2024) using the combined scalar reward

$$r_i = r_{\mathrm{task},i} + r_{\mathrm{fmt},i} + 0.3 r_{\mathrm{div},i}, \tag{6}$$

where $r_{\mathrm{fmt},i}$ rewards a valid reasoning block followed by a nonempty strategy, and $r_{\mathrm{div},i} = 0$ for ineligible strategies.

To handle difficult repair states, we use a difficulty-aware curriculum that gradually introduces harder states. For repair, adaptive replay revisits states with mixed correct and incorrect rollout outcomes to maintain task-reward contrast. Appendix C.2 provides the details.

3.4 Compositional Optimization at Inference

At inference, CUDACoach extends the strategy interface from a single kernel to a pool of correct kernels, as shown in Figure 2(D). At iteration t, the planner conditions on the pool $\mathcal{B}_t$ of available correct kernels and their performance profiles. This multi-kernel context supports combining complementary mechanisms across implementations.

To limit redundant code generation, we cluster sampled strategies based on their recommended techniques. We jointly select one representative per cluster to maximize the minimum pairwise distance between the selected strategies. The frozen coder then generates a candidate kernel for each selected strategy. Appendix D specifies the similarity measure and selection procedure.

Meanwhile, failed kernels in a separate queue $\mathcal{Q}_t$ undergo repair, with the planner generating repair strategies from error feedback for the frozen coder to implement. After evaluation, correct outputs and successful repairs are merged with the fastest correct kernel found so far to form $\mathcal{B}_{t+1}$, while failed implementations are carried forward in $\mathcal{Q}_{t+1}$ for subsequent repair. At the end of inference, we return the fastest correct kernel found across the initial seed and all iterations.

4 Experiments

We evaluate the final kernel performance of CUDACoach, examine the effects of diversity-reward training and the multi-kernel context, and analyze progress over successive iterations.

4.1 Experimental Setup

Benchmark and metrics. We evaluate CUDACoach on all 250 tasks across the three levels of KernelBench (Ouyang et al., 2025), covering individual operators, sequences of operators, and

end-to-end architectures. We report task-level correctness rate, fast_1, fast_2, and geometric mean speedup. The metric fast_p measures the fraction of all tasks with a correct implementation whose speedup exceeds p. Speedups are measured relative to PyTorch eager. Geometric mean speedup is computed over tasks with correct implementations, including those slower than the reference.

Baselines. We compare CUDACoach with best-of-N (BoN) sampling from a fixed initial kernel, single-stage iteration, and self-planned compositional iteration. BoN and single-stage iteration use Qwen3-Coder-Plus, Qwen3.8-Flash, and Kevin-32B (Baronio et al., 2025). The self-planned baseline uses Qwen3-Coder-Plus or Qwen3.8-Flash as both planner and coder, with the compositional inference procedure of Section 3.4 unchanged.

Training settings. We initialize the planner from Qwen3-8B and perform supervised fine-tuning on 32,904 examples, followed by GRPO training. Only the planner is updated, while the downstream coder remains frozen. Full training settings are provided in Appendix C.

Evaluation protocol. All methods use the same initial CUDA kernel generated by cudaLLM-8B for each task. BoN samples ten candidates from this fixed input. Iterative methods run for ten iterations and return the correct implementation with the highest measured speedup across the initial kernel and all iterations. Single-stage iteration generates four code candidates per iteration. Two-stage methods sample four optimization strategies per iteration and implement one per cluster. Kernels are evaluated in FP32 on a high-end datacenter GPU. Appendix D gives the detailed settings.

4.2 Main Results

Table 1 compares CUDACoach with fixed-parent BoN sampling, single-stage iteration, and self-planned compositional iteration across the three levels of KernelBench.

Table 1: KernelBench results by inference method. Corr. and F_1/F_2 are correctness rate and $\mathrm{fast}_1/\mathrm{fast}_2$ in percent. GM is geometric mean speedup over correct tasks with valid timings, including slowdowns. Bold marks the best values for each coder when methods differ.

		Level 1 (100 tasks)			Level 2 (100 tasks)			Level 3 (50 tasks)			Overall (250 tasks)		
Planner	Coder	Corr. ↑	F_1/F_2 ↑	GM ↑	Corr. ↑	F_1/F_2 ↑	GM ↑	Corr. ↑	F_1/F_2 ↑	GM ↑	Corr. ↑	F_1/F_2 ↑	GM ↑
Fixed-parent sampling (BoN-10)													
None	Qwen3-Coder-Plus	64.0	27.0/12.0	0.67	24.0	0.0/0.0	0.05	14.0	0.0/0.0	**0.18**	38.0	10.8/4.8	0.31
None	Qwen3.8-Flash	80.0	50.0/22.0	1.08	72.0	14.0/7.0	0.59	40.0	6.0/6.0	0.27	68.8	26.8/12.8	0.71
None	Kevin-32B	69.0	25.0/11.0	0.70	52.0	0.0/0.0	0.09	18.0	2.0/0.0	0.12	52.0	10.4/4.4	0.27
Single-stage iteration													
None	Qwen3-Coder-Plus	71.0	29.0/17.0	0.82	33.0	**3.0**/1.0	0.08	16.0	2.0/0.0	0.11	44.8	13.2/7.2	0.35
None	Qwen3.8-Flash	**99.0**	59.0/26.0	0.85	**93.0**	**37.0/26.0**	0.91	**50.0**	10.0/**8.0**	0.47	**86.8**	40.4/22.4	0.82
None	Kevin-32B	71.0	**43.0**/19.0	0.94	58.0	2.0/0.0	0.23	26.0	**4.0**/0.0	**0.16**	56.8	**18.8**/7.6	0.44
Two-stage compositional iteration													
Qwen3-Coder-Plus	Qwen3-Coder-Plus	67.0	33.0/14.0	0.93	30.0	1.0/0.0	0.14	18.0	0.0/0.0	0.17	42.4	13.6/5.6	0.47
Our planner	Qwen3-Coder-Plus	**74.0**	**43.0/20.0**	**1.09**	**64.0**	**3.0/3.0**	**0.25**	**26.0**	**4.0**/0.0	0.11	**60.4**	**19.2/9.2**	**0.48**
Our planner	Qwen3.8-Flash	94.0	**63.0/33.0**	**1.17**	84.0	33.0/25.0	**1.09**	46.0	**12.0/8.0**	**0.55**	80.4	**40.8/24.8**	**1.04**
Our planner	Kevin-32B	**74.0**	41.0/**21.0**	**1.10**	**64.0**	**5.0/2.0**	**0.24**	**30.0**	0.0/0.0	0.08	**61.2**	18.4/**9.2**	**0.45**

Overall performance. CUDACoach improves all four overall metrics over fixed-parent BoN with every downstream coder. The largest gain over single-stage iteration occurs with Qwen3-Coder-Plus, where correctness increases from 44.8% to 60.4% and fast_1 from 13.2% to 19.2%. The correctness improvement is most pronounced on Level 2, rising from 33.0% to 64.0%. CUDACoach achieves these results with 5,426 code generations, compared with 10,000 for single-stage iteration, because strategy clustering implements one representative per cluster.

Across all three coders, CUDACoach improves both fast_2 and geometric mean speedup over single-stage iteration. The geometric mean gain is largest with Qwen3.8-Flash, increasing from $0.82\times$ to $1.04\times$. With Kevin-32B, correctness also increases from 56.8% to 61.2%. Paired with our planner,

Qwen3-Coder-Plus further surpasses standalone Kevin-32B in correctness and fast_1, achieving 60.4% versus 56.8% and 19.2% versus 18.8%.

Effect of the planner choice. To assess the trained planner, we compare it with the self-planned baseline while keeping the downstream coder and compositional inference procedure fixed. With Qwen3-Coder-Plus as the coder, replacing it with our trained planner increases correctness by 18.0 percentage points and fast_1 by 5.6 points. In contrast, the self-planned baseline alone yields slightly lower correctness than single-stage iteration, despite a small increase in fast_1. These results indicate that introducing a separate strategy step is not sufficient on its own, and highlight the importance of the planner used within the same inference procedure.

4.3 Ablation Studies

We ablate diversity-reward training and the multi-kernel context using Qwen3-Coder-Plus under the same evaluation protocol. Strategy clustering, repair, and historical-best retention remain enabled in all variants. Table 2 reports final kernel performance.

Table 2: Ablations on all 250 KernelBench tasks with Qwen3-Coder-Plus as the coder. Corr. is correctness rate and F_1/F_2 denotes $\mathrm{fast}_1/\mathrm{fast}_2$, all in percent. GM is geometric mean speedup over correct tasks with valid timings, including slowdowns. Best values are shown in bold.

	Components		Kernel performance		
Configuration	Diversity reward	Multi-kernel	Corr. ↑	F_1/F_2 ↑	GM (×) ↑
Full method	Yes	Yes	**60.4**	**19.2/9.2**	0.48
w/o diversity reward	No	Yes	53.2	17.6/8.4	0.42
w/o multi-kernel context	Yes	No	52.8	15.2/8.4	**0.49**

Diversity-reward training. The diversity reward encourages more varied strategies and less repetition across tasks. On the same 89-task subset, Table 3 shows that first-round strategy diversity before clustering increases from 0.440 to 0.510, while the most common technique-set frequency falls from 16.9% to 6.7%. These changes accompany higher correctness on the full benchmark, from 53.2% to 60.4%, with smaller gains in fast_1 and fast_2, from 17.6% to 19.2% and 8.4% to 9.2%, respectively. Additional training dynamics are reported in Appendix F.

Table 3: Effect of diversity-reward training on first-round strategy diversity before clustering. Strategy diversity is the mean pairwise Jaccard distance among the technique sets of four strategies sampled for each input state, averaged across tasks. Top-1 frequency is the percentage of tasks where the most common technique set appears at least once. Both rows use the same 89-task subset. Best values are shown in bold.

Planner training	Strategy diversity ↑	Top-1 technique-set frequency (%) ↓
Without diversity reward	0.440	16.9
With diversity reward	**0.510**	**6.7**

Multi-kernel context. Restricting the planner and coder to the best correct kernel and its feedback reduces correctness from 60.4% to 52.8% and fast_1 from 19.2% to 15.2%. Access to multiple implementations therefore improves both correctness coverage and the fraction of tasks exceeding PyTorch eager performance. GM remains similar at $0.48\times$ with the context and $0.49\times$ without it, though the averages cover different correct-task sets. Section 4.4 traces this benefit across iterations, and Appendix E illustrates how a strategy combines information from two kernels.

4.4 Iterative Optimization Analysis

We analyze how iterative optimization benefits from retaining and combining diverse implementations over time. Figure 3 compares CUDACoach with single-stage iteration, self-planned compositional iteration, and its single-kernel variant using Qwen3-Coder-Plus as the coder. Each point reports performance from the best correct kernel found up to that iteration.

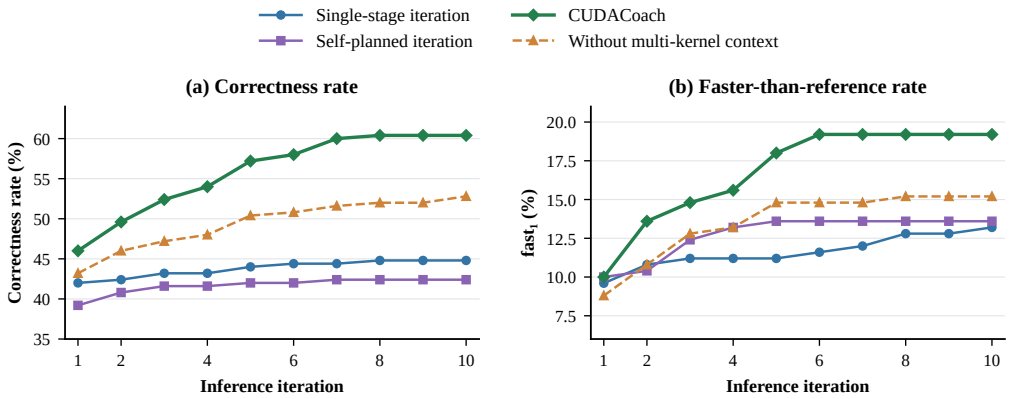

Figure 3: Correctness rate and fast_1 after each inference iteration with Qwen3-Coder-Plus as the coder. At iteration t, both rates are computed over all 250 tasks using the fastest correct kernel found up to that iteration, including the initial kernel.

Progress across iterations. CUDACoach achieves stronger acceleration results within fewer iterations. By iteration 6, it reaches 19.2% fast_1, exceeding the final rates of single-stage iteration and self-planned compositional iteration after 10 iterations, which are 13.2% and 13.6%, respectively. Its correctness also increases from 46.0% after iteration 1 to 60.4% by iteration 8. These results show that CUDACoach accelerates more tasks within fewer iterations while progressively expanding correctness coverage.

Effect of multi-kernel context. The multi-kernel context preserves diverse correct implementations, allowing later strategies to inherit optimization techniques and combine them using code and profiling evidence from different kernels. Its benefit becomes more pronounced over successive iterations. From iteration 1 to iteration 10, the correctness advantage over the single-kernel variant grows from 2.8 to 7.6 percentage points, while the fast_1 advantage grows from 1.2 to 4.0 points. By iteration 6, CUDACoach already reaches 19.2% fast_1, exceeding the single-kernel variant's final rate of 15.2% after 10 iterations. Appendix E further illustrates this process through a strategy that inherits and combines techniques from two parent kernels.

5 Conclusion

We presented CUDACoach, a framework that trains a planner to guide a frozen coder in CUDA kernel repair and optimization. Verified strategy mining turns successful kernel edits into execution-validated supervision, while reinforcement learning with a two-level diversity reward encourages varied optimization strategies. At inference time, the planner combines complementary techniques across implementations to support iterative improvement. Evaluation on all 250 KernelBench tasks with three downstream coders demonstrates the value of learned strategy guidance, with gains that vary across coders and metrics. With Qwen3-Coder-Plus, the trained planner improves both correctness and the fraction of tasks with kernels faster than PyTorch over single-stage iteration and self-planned iteration. Ablations further support the contributions of diversity-reward training and the multi-kernel context. Together, these findings establish strategy learning as a promising direction for improving automated kernel optimization without updating the downstream coder. Extending this approach to additional hardware architectures and production workloads remains an important direction for future work.

Limitations

Our main kernel optimization evaluation is limited to KernelBench on a high-end datacenter GPU with FP32 inputs. Transfer across GPU architectures and precision settings, as well as performance on other kernel benchmarks and production workloads, remains outside the scope of this evaluation.

Reproducibility Statement

To support reproducibility, Appendices B–D document data construction, training, reward computation, inference, and evaluation. Appendix G provides the complete planner prompts and technique catalogue.

AI Use Statement

We used large language models to assist with manuscript editing and training data construction. For manuscript editing, AI assistance focused on correcting grammar, refining wording, and improving clarity and readability. For training data construction, we used these models to generate candidate CUDA kernels, formulate repair and optimization strategies, and produce kernel analyses used as supervision. The models and their roles in data construction are described in Appendix B. The authors are responsible for the accuracy and integrity of the paper, including its technical claims, experimental results, and AI-assisted content.

References

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. *CoRR*, abs/2507.11948, 2025. doi: 10.48550/ARXIV.2507.11948. URL https://doi.org/10.48550/arXiv.2507.11948.

Jinwu Chen, Qidie Wu, Bin Li, Lin Ma, Xin Si, Yang Hu, Shouyi Yin, and Jun Yang. cuPilot: A strategy-coordinated multi-agent framework for CUDA kernel evolution. *CoRR*, abs/2512.16465, 2025a. doi: 10.48550/ARXIV.2512.16465. URL https://doi.org/10.48550/arXiv.2512.16465.

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger A. Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. AVO: Agentic variation operators for autonomous evolutionary search. *CoRR*, abs/2603.24517, 2026a. doi: 10.48550/ARXIV.2603.24517. URL https://doi.org/10.48550/arXiv.2603.24517.

Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. CUDA-LLM: LLMs can write efficient CUDA kernels. *CoRR*, abs/2506.09092, 2025b. doi: 10.48550/ARXIV.2506.09092. URL https://doi.org/10.48550/arXiv.2506.09092.

Yilei Chen, Souradip Chakraborty, Lorenz Wolf, Ioannis Paschalidis, and Aldo Pacchiano. Post-training large language models for diverse high-quality responses. In *International Conference on Learning Representations*, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/cfeb1fa78c5fb06be27c0980aa8ee581-Paper-Conference.pdf.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation. *CoRR*, abs/2602.24286, 2026. doi: 10.48550/ARXIV.2602.24286. URL https://doi.org/10.48550/arXiv.2602.24286.

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. *CoRR*, abs/2510.16996, 2025. doi: 10.48550/ARXIV.2510.16996. URL https://doi.org/10.48550/arXiv.2510.16996.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. KernelBlaster: Continual cross-task CUDA optimization via memory-augmented in-context reinforcement learning. *CoRR*, abs/2602.14293, 2026. doi: 10.48550/ARXIV.2602.14293. URL https://doi.org/10.48550/arXiv.2602.14293.

Jinye Du, Quan Yuan, Zuyao Zhang, Yanzhi Yi, Jiahui Hu, Wangyi Chen, Yiyang Zhu, Qishui Zheng, Wenxiang Zou, Xiangyu Chang, Zuohe Zheng, Zichun Ye, Chao Liu, Shanni Li, Renwei Zhang, Yiping Deng, Xinwei Hu, Xuefeng Jin, and Jie Zhao. AKG kernel agent: A multi-agent framework for cross-platform kernel synthesis. *CoRR*, abs/2512.23424, 2025. doi: 10.48550/ARXIV.2512.23424. URL https://doi.org/10.48550/arXiv.2512.23424.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, Yiming Yang, and Sean Welleck. AdaExplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation. *CoRR*, abs/2604.16625, 2026. doi: 10.48550/ARXIV.2604.16625. URL https://doi.org/10.48550/arXiv.2604.16625.

Dayuan Fu, Mohan Jiang, Tongyu Wang, Dian Yang, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. daVinci-kernel: Co-evolving skill selection, summarization, and utilization via RL for GPU kernel optimization. *CoRR*, abs/2606.16497, 2026. doi: 10.48550/ARXIV.2606.16497. URL https://doi.org/10.48550/arXiv.2606.16497.

Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. From large to small: Transferring CUDA optimization expertise via reasoning graph. *CoRR*, abs/2510.19873, 2025. doi: 10.48550/ARXIV.2510.19873. URL https://doi.org/10.48550/arXiv.2510.19873.

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. *CoRR*, abs/2510.03760, 2025. doi: 10.48550/ARXIV.2510.03760. URL https://doi.org/10.48550/arXiv.2510.03760.

Yuxuan Han, Menghao Guo, Zhengning Liu, Wenguang Chen, and Shi-Min Hu. Making LLMs optimize multi-scenario CUDA kernels like experts. *CoRR*, abs/2603.07169, 2026. doi: 10.48550/ARXIV.2603.07169. URL https://doi.org/10.48550/arXiv.2603.07169.

Ahmed Heakl, Sarim Hashmi, Gustavo Bertolo Stahl, Seung Hun Eddie Han, Salman H. Khan, and Abdulrahman Mahmoud. CASS: Nvidia to AMD transpilation with data, models, and benchmark. *CoRR*, abs/2505.16968, 2025. doi: 10.48550/ARXIV.2505.16968. URL https://doi.org/10.48550/arXiv.2505.16968.

KernelEvolve Team, Meta Platforms. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at meta. *CoRR*, abs/2512.23236, 2025. doi: 10.48550/ARXIV.2512.23236. URL https://doi.org/10.48550/arXiv.2512.23236.

Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. ConCuR: Conciseness makes state-of-the-art kernel generation. *CoRR*, abs/2510.07356, 2025. doi: 10.48550/ARXIV.2510.07356. URL https://doi.org/10.48550/arXiv.2510.07356.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *CoRR*, abs/2509.14279, 2025. doi: 10.48550/ARXIV.2509.14279. URL https://doi.org/10.48550/arXiv.2509.14279.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoTriton: Automatic Triton programming with reinforcement learning in LLMs. *CoRR*, abs/2507.05687, 2025a. doi: 10.48550/ARXIV.2507.05687. URL https://doi.org/10.48550/arXiv.2507.05687.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. *CoRR*, abs/2507.14111, 2025b. doi: 10.48550/ARXIV.2507.14111. URL https://doi.org/10.48550/arXiv.2507.14111.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for Triton kernel generations. *CoRR*, abs/2602.05885, 2026. doi: 10.48550/ARXIV.2602.05885. URL https://doi.org/10.48550/arXiv.2602.05885.

Meta AI. KernelLLM. https://huggingface.co/facebook/KernelLLM, 2025. Hugging Face model card; Llama-3.1-8B fine-tuned on the KernelBook dataset for PyTorch-to-Triton kernel generation.

NVIDIA Corporation. Nsight compute profiling guide, version 13.3. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html, 2026. Defines the Speed of Light throughput section, which reports achieved utilization as a percentage of the theoretical maximum for each compute and memory unit.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Qiuyi Qu, Yicheng Sui, Yufei Sun, Rui Chen, Xiaofei Zhang, Yuzhi Zhang, Haofeng Wang, Ge Lan, Ning Zhang, and Manfei Zhao. A two-stage GPU kernel tuner combining semantic refactoring and search-based optimization. *CoRR*, abs/2601.12698, 2026. doi: 10.48550/ARXIV.2601.12698. URL https://doi.org/10.48550/arXiv.2601.12698.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. KernelBand: Boosting LLM-based kernel optimization with a hierarchical and hardware-aware multi-armed bandit. *CoRR*, abs/2511.18868, 2025. doi: 10.48550/ARXIV.2511.18868. URL https://doi.org/10.48550/arXiv.2511.18868.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL https://doi.org/10.48550/arXiv.2402.03300.

Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L2: Surpassing cuBLAS performance for matrix multiplication through reinforcement learning. *CoRR*, abs/2512.02551, 2025. doi: 10.48550/ARXIV.2512.02551. URL https://doi.org/10.48550/arXiv.2512.02551.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. KernelSkill: A multi-agent framework for GPU kernel optimization. *CoRR*, abs/2603.10085, 2026. doi: 10.48550/ARXIV.2603.10085. URL https://doi.org/10.48550/arXiv.2603.10085.

Qwen Team. Qwen3 technical report. *CoRR*, abs/2505.09388, 2025. doi: 10.48550/ARXIV.2505.09388. URL https://doi.org/10.48550/arXiv.2505.09388.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. GEAK: Introducing Triton kernel AI agent and evaluation benchmarks. *CoRR*, abs/2507.23194, 2025. doi: 10.48550/ARXIV.2507.23194. URL https://doi.org/10.48550/arXiv.2507.23194.

Yaoyu Wang, Hankun Dai, Zhidong Yang, Junmin Xiao, and Guangming Tan. Mastering sparse CUDA generation through pretrained models and deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. OpenReview:VdLEaGPYWT.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for GPU kernel performance optimization. *CoRR*, abs/2509.07506, 2025. doi: 10.48550/ARXIV.2509.07506. URL https://doi.org/10.48550/arXiv.2509.07506.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. KernelFoundry: Hardware-aware evolutionary GPU kernel optimization. *CoRR*, abs/2603.12440, 2026. doi: 10.48550/ARXIV.2603.12440. URL https://doi.org/10.48550/arXiv.2603.12440.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. TritonRL: Training LLMs to think and code Triton without cheating. *CoRR*, abs/2510.17891, 2025. doi: 10.48550/ARXIV.2510.17891. URL https://doi.org/10.48550/arXiv.2510.17891.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of LLMs. *CoRR*, abs/2601.15727, 2026. doi: 10.48550/ARXIV.2601.15727. URL https://doi.org/10.48550/arXiv.2601.15727.

Mohammad Zaeed, Tanzima Z. Islam, and Vladimir Indic. Optimas: An intelligent analytics-informed generative AI framework for performance optimization. *CoRR*, abs/2604.23892, 2026. doi: 10.48550/ARXIV.2604.23892. URL https://doi.org/10.48550/arXiv.2604.23892.

Shuoming Zhang, Qiuchu Yu, Yangyu Zhang, Ruiyuan Xu, Xiyu Shi, Guangli Li, Xiaobing Feng, Huimin Cui, and Jiacheng Zhao. Learning when to optimize: Verified optimization skills from expert GPU-Kernel lineages. *CoRR*, abs/2605.28213, 2026. doi: 10.48550/ARXIV.2605.28213. URL https://doi.org/10.48550/arXiv.2605.28213.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. *CoRR*, abs/2511.01884, 2025. doi: 10.48550/ARXIV.2511.01884. URL https://doi.org/10.48550/arXiv.2511.01884.

Yujie Zheng, Zhuo Li, Shengtao Zhang, Hanjing Wang, Junjie Sheng, Jiaqian Wang, Junchi Yan, Weinan Zhang, Ying Wen, Bo Tang, and Muning Wen. Towards cold-start drafting and continual refining: A value-driven memory approach with application to NPU kernel synthesis. *CoRR*, abs/2603.10846, 2026. doi: 10.48550/ARXIV.2603.10846. URL https://doi.org/10.48550/arXiv.2603.10846.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. QiMeng-Kernel: Macro-thinking micro-coding paradigm for LLM-based high-performance GPU kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2026. arXiv:2511.20100.

A Diagnostic Analysis of Planner Recommendations

A.1 Per-Kernel Technique Assignments

We examine the fixed subset of 61 KernelBench Level 1 tasks with correct seed implementations used in Figure 1(a). For each task, Qwen3-Coder-Plus receives the PyTorch reference, current CUDA implementation, Nsight Compute profile, and a hardware description of the high-end datacenter GPU. We sample one response per task with temperature 0.6, top-p 0.95, and an output limit of 8,192 tokens. The prompt requests at most three recommended techniques. A separate Qwen3-Coder-Plus annotator maps the "Recommended techniques" section of each strategy to the fixed catalogue of 53 labels, normalizing equivalent expressions. The optimization and annotation prompts are provided in Appendix G. Figure 4 shows all 61 assignments.

DRAM throughput utilization is measured with `dram__throughput.avg.pct_of_peak_sustained_elapsed` (NVIDIA Corporation, 2026). The 61 responses cover 15 catalogue techniques and 24 distinct technique sets. The most frequent set combines shared-memory tiling, register blocking, and memory coalescing across ten matrix-multiplication, two convolution, and one reduction task. Within the matrix-multiplication family, ten of the 16 tasks receive this set, spanning 0.46% to 80.00% utilization.

These results highlight a tendency to reuse the same technique combinations across kernels with substantially different DRAM throughput utilization, reflecting limited diversity in strategy recommendations.

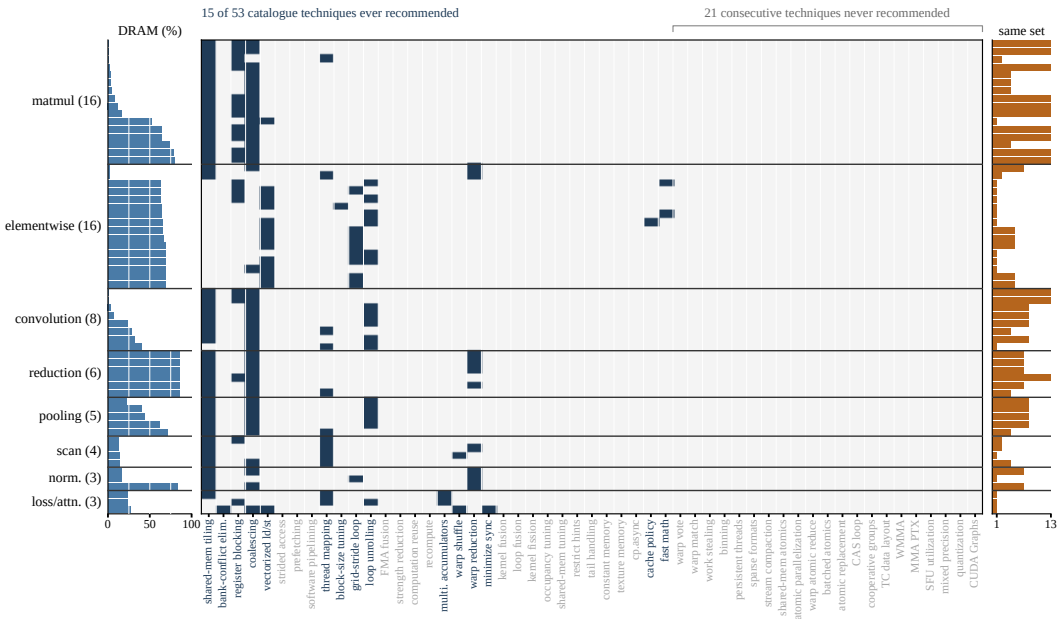

Figure 4: Per-kernel technique assignments for Figure 1(a). Rows are grouped by operator family and ordered within each family by DRAM throughput utilization. Table 4 identifies the rows by KernelBench task ID. Columns represent the 53 catalogue techniques, and filled cells indicate recommendations. Left bars show DRAM throughput utilization. Right bars count the tasks sharing each row's exact technique set.

Table 4: KernelBench Level 1 task IDs corresponding to the rows of Figure 4. Within each operator family, IDs are listed in the same top-to-bottom order as the figure.

Operator family	Task IDs in figure order
Matrix multiplication	10, 6, 3, 11, 14, 13, 7, 9, 18, 17, 12, 15, 1, 8, 16, 2
Elementwise activation	23, 24, 29, 21, 25, 88, 26, 30, 28, 22, 20, 27, 31, 32, 5, 19
Convolution	66, 60, 54, 59, 85, 82, 84, 83
Reduction	49, 52, 47, 51, 48, 53
Pooling	42, 41, 44, 43, 46
Scan	89, 91, 90, 93
Normalization	39, 38, 36
Loss and attention	99, 95, 98

A.2 Strategy Source Comparison

Figure 1(b) compares off-the-shelf Qwen3-Coder-Plus strategies with human-selected strategies on the same 61 task contexts. For the human-selected arm, we use three human-selected strategies per task. Each strategy is expressed as a natural-language implementation plan. The strategy-similarity analysis uses the selected strategies. Both arms use the same initial kernel and NCU profile.

Each arm contributes three fixed strategies per task. The off-the-shelf strategies use temperature 0.6 and top-p 0.95. A separate Qwen3-Coder-Plus annotator maps each strategy to the same fixed catalogue of 53 technique labels. For each pair of tasks within the same operator family, we compute Jaccard similarity for all $3 \times 3 = 9$ pairs of strategy technique sets and take the mean. We divide task pairs into small-gap and large-gap groups with absolute DRAM throughput utilization differences below 10 and above 30 percentage points, respectively. We include families with at least three task pairs in each group, leaving matrix multiplication, convolution, and elementwise activation. Within each family, we compute mean similarity for each group, then aggregate using the same weights for both groups, proportional to the number of large-gap task pairs. Separation is the aggregated small-gap similarity minus the aggregated large-gap similarity.

For the downstream comparison, the strategies are held fixed and passed to the same coder, Qwen3-Coder-Plus. It independently generates three implementations per strategy with temperature 0.6, top-p 0.95, and a 16,384-token output limit. The direct-code control receives the same task context without a planner strategy and generates nine implementations per task, grouped into three sets of three for aggregation. Thus, all arms have nine downstream code-generation attempts per task. This matches the code-generation budget, not the total token or API cost of planning and generation.

Correctness rate is the fraction of three-candidate groups containing at least one correct implementation. For speedup, we select the fastest correct implementation among each group's candidates and the parent kernel, average the three group speedups per task, and report the median across tasks. Speedups are measured relative to the parent kernel. All configurations follow the evaluation protocol in Appendix D.

Human-selected strategies have a mean within-task pairwise Jaccard distance of 0.490, compared with 0.371 for off-the-shelf strategies. Their small-gap and large-gap similarities are 0.501 and 0.375, giving a separation of 0.126. The corresponding off-the-shelf separation is 0.088. With the same frozen coder and downstream generation budget, human-selected strategies achieve 78.7% group correctness and $1.26\times$ median parent-relative speedup, compared with 72.1% and $1.20\times$ for the off-the-shelf strategies.

B Data Construction and Verification

The SFT and RL corpora use separately constructed task pools. Table 5 summarizes their composition. The following sections specify task selection, strategy verification, auxiliary analyses, and RL state construction.

We verify that no evaluation task appears in training. For every training task with an extractable PyTorch reference program, we compare it against the 250 KernelBench Level 1 to Level 3 tasks by exact text equality, by abstract syntax tree structure, and by the `Model` class alone. No comparison produces a match, so no training task was removed.

Table 5: Training corpus composition. SFT counts are accepted strategy or analysis examples. RL counts are kernel states before the training and validation split.

Stage	Data type	Count
SFT	Verified repair strategies	9,726
	Verified optimization strategies	18,478
	Auxiliary kernel analyses	4,700
	Total examples	**32,904**
RL	Repair states	6,063
	Optimization states	3,887
	Total states	**9,950**

B.1 Task Selection and Verified Edits

The supervised tasks are drawn from the public RL dataset released with ByteDance Seed's cudaLLM-8B.[1] The selected tasks cover 21 operator categories. Within each category, programs are grouped by their exact operator combinations. Each group contributes at most three programs at the shortest, median, and longest code lengths, using random seed 42. This produces an initial pool of 4,880 tasks.

To reduce timing variation for small workloads, a language model increases input dimensions while preserving the operator structure. We rerun the scaled programs and measure runtime variation over 100 PyTorch forward passes. The strict runtime coefficient of variation $\mathrm{CV} < 0.1$ criterion retains 4,443 tasks.

We generate initial implementations with cudaLLM-8B. The 4,443 seeds comprise 1,991 correct kernels and 2,452 failing kernels. Claude Opus 4.5 repairs failing seeds, restoring 815 kernels, of

[1] `https://huggingface.co/datasets/ByteDance-Seed/cudaLLM-data`

which 813 have NCU profiles and join the correct seeds for optimization. A further repair stage addresses failures introduced during optimization. This process produces 5,725 verified optimization edits and 2,137 successful repair edits. Optimization edits must preserve correctness and reduce latency relative to the parent.

For each verified edit, Qwen3-Coder-Plus reconstructs eight candidate strategies from the reference program, parent kernel, execution feedback, improved kernel, and teacher response. Each strategy follows the repair or optimization interface in Appendix G. The frozen Qwen3-Coder-Plus then implements each strategy from the input state S without access to the improved kernel K^+. Define $V_P(K)$ to indicate that a kernel compiles, runs, and passes numerical checks against the reference P, and let $\ell(K)$ denote its measured latency. For the new implementation $\widehat{K}$, the acceptance rule is

$$\mathrm{keep}(S,a)=\begin{cases}0, & V_P(\widehat{K})=0,\\ 1, & V_P(\widehat{K})=1 \text{ and } S \text{ is a repair state},\\ \mathbf{1}\Big[\ell(\widehat{K})<\ell(K)\Big], & V_P(\widehat{K})=1 \text{ and } S \text{ is an optimization state.}\end{cases} \tag{7}$$

The new kernel need not reproduce the teacher's code. All accepted strategies enter SFT. The final strategy corpus contains 18,478 optimization examples and 9,726 repair examples.

B.2 Auxiliary Kernel Analyses

The CASS branch starts with 60,554 extracted CUDA kernels. Code deduplication leaves 57,588, and NVCC compilation filtering leaves 52,225 for scoring. Runtime checks remove no further kernels. Qwen3-Coder-Plus evaluates implementation difficulty and code quality using the rubric in Table 6. These are source-code judgments, not measured performance or independent ratings of the generated analyses.

The preliminary selection retains difficulty-1 kernels with quality 5, difficulty-2 or difficulty-3 kernels with quality at least 4, and difficulty-4 or difficulty-5 kernels with quality at least 3. This selects 22,442 kernels for analysis by Claude Opus 4.5. We remove analyses with incomplete reasoning markup or unsupported CUDA intrinsic names. We then retain analyses whose source kernels have quality score 5, or quality score 4 and difficulty score greater than 3, yielding 4,700 auxiliary examples.

Table 6: CASS source-code scoring rubric. Quality is assessed relative to the functional complexity of the kernel, so an efficiently written simple kernel can receive the highest quality score.

Score	Implementation difficulty	Code quality
1	Independent, straight-line elementwise logic	Broken or logically invalid
2	Simple loops and standard aggregations	Poor practices, inefficiency, or unclear code
3	Fused or composite computation	Standard implementation with limited polish
4	Complex or indirect indexing and sparse structures	Well-written code following CUDA conventions
5	Global dependencies, complex synchronization, or irregular control flow	Concise, efficient, production-ready implementation for its difficulty

B.3 RL State Construction

The RL tasks come from CUDA Agent (Dai et al., 2026). We retain tasks with runtime CV at most 0.1, sample four CUDA candidates per task, and verify them. Failing candidates are grouped by error category. Correct candidates are grouped into performance bands whose largest and smallest speedups differ by at most a factor of 1.2. One candidate per group is retained to cover different failure modes and performance levels. These four candidates construct the state pool and are separate from strategy rollouts during RL.

The separate RL task pool contains 5,959 programs. After candidate verification and state selection, the RL state pool contains 9,950 states from 5,866 tasks. The states comprise 6,063 repair cases and 3,887 optimization cases. A split with random seed 42 assigns 9,453 states to training and 497 to validation. The version used by the reported checkpoint includes NCU context where available and appends the reasoning-format requirement to the user prompt.

C Training Details

C.1 Optimization and Sampling Settings

Table 7 summarizes planner optimization. SFT predicts assistant responses in the mixed corpus. RL uses scalar reward composition, token-mean loss aggregation, no KL penalty, policy ratio clipping in $[0.8, 1.2]$, and a dual-clip parameter of 3.

Table 7: Planner training settings. Both stages run on eight GPUs, and SFT uses BF16.

Setting	SFT	RL
Training duration	2 epochs	148 updates
Global batch size	64 examples	64 states
Learning rate	2×10^{-5}	5×10^{-6}
Warmup	1% of steps	3 updates
Token limit	32,768 per example	8,192 each for prompt and response
Mini-batch size	—	64

RL runs on one node with eight 80GB GPUs. During RL training, the planner samples eight strategies per state using temperature 1.0 and top-p 1.0. Validation samples four strategies per state. The frozen Qwen3-Coder-Plus uses temperature 0.6, top-p 0.95, and a 16,384-token output limit. It generates two implementations per training strategy and four per validation strategy. Correctness uses five input trials with tolerance 10^{-4}. Training timing uses five warmup and 20 measurement runs, compared with ten and 100 for validation.

C.2 Difficulty-Aware Curriculum and Adaptive Repair Replay

Before RL, we freeze the SFT planner and probe each training state with four independently sampled strategies. The frozen Qwen3-Coder-Plus generates two implementations per strategy. Sampling temperatures and execution checks follow the RL training settings in Appendix C.1. A repair strategy succeeds if either implementation passes compilation and numerical checks. An optimization strategy must additionally yield speedup greater than 1.1 under the execution protocol in Appendix C.3. Let $c(S) \in \{0, \ldots, 4\}$ count successful probe strategies. States with $c(S) = 4$ are easy, those with $1 \leq c(S) \leq 3$ are partially solvable, and those with $c(S) = 0$ are hard. Probing uses the existing NCU context and does not rerun profiling. The explicit reasoning-format requirement is added for RL after this probe.

The offline probe finds no successful strategy for 3,373 of the 5,765 repair states (58.5%) under its four-strategy, two-implementation budget. Table 8 gives the complete difficulty distribution. These counts describe the initial probe, not the outcomes of later RL updates, and motivate gradually introducing the hard repair states.

Table 8: Initial training-state difficulty under the frozen SFT planner and coder. Repair success requires correctness, whereas optimization success additionally requires speedup greater than 1.1.

State type	Easy	Partially solvable	Hard	Total
Repair	811	1,581	3,373	5,765
Optimization	784	1,934	970	3,688
Total	**1,595**	**3,515**	**4,343**	**9,453**

Each batch contains 43 repair states and 21 optimization states. Both use the proportions in Table 9, interpolated linearly between listed updates. States are sampled within difficulty groups with random seed 42. The 148-update schedule has 9,472 state slots rather than a single pass through the 9,453 training states, so some states repeat and others are not scheduled.

Adaptive replay uses repair task outcomes rather than format or diversity rewards. A repair rollout group is task-effective when between one and seven of its eight strategies yield a correct downstream implementation. Such groups provide variation in the binary task reward. Starting with updates

Table 9: Difficulty proportions in the base curriculum. The final mixture is retained from update 60 onward. Adaptive replay modifies only hard repair slots.

Update	Easy (%)	Partially solvable (%)	Hard (%)
1	80	20	0
9	40	60	0
10	40	55	5
20	25	65	10
30	15	65	20
40	5	65	30
50	5	55	40
60 onward	5	45	50

11 through 20, the controller measures their fraction over non-overlapping ten-update windows. If the fraction is below 0.40, it increases the replacement fraction by 0.25, up to 0.50. Otherwise, it decreases the fraction by 0.25, down to zero. The initial replacement fraction is zero, and the first adjustment occurs after update 20.

The buffer holds at most 2,048 previously task-effective repair states with $c(S) \leq 3$. Each state can be adaptively replayed at most twice. The controller replaces hard repair slots with buffered states using seed 42, excluding states already in the batch and prioritizing those replayed fewer times. If too few states are eligible, the remaining hard slots are left unchanged. Replay replaces scheduled hard repair states and generates fresh strategies and implementations rather than reusing stored responses or advantages. It preserves the batch size and update budget and does not affect optimization states.

C.3 Reward Computation and Policy Update

Section 3.3 defines the repair reward, the optimization reward in Eq. (2), and their aggregation into the strategy-level task reward. Output, compilation, runtime, and correctness failures all receive -1 in both phases. Sampling, numerical checks, and timing settings are specified in Appendix C.1.

The format reward is 1 when the response begins with exactly one `<think>` block whose trimmed content has at least five characters and is followed by a nonempty final strategy, and 0 otherwise.

The fixed Qwen3-Coder-Plus annotator maps only the *Recommended techniques* section to 53 named technique IDs and one shared `other` dimension, giving $d = 54$. Repeated mentions of a technique contribute one binary entry. The local eligible set contains non-truncated optimization responses with a correct downstream implementation and at least one assigned technique. Repair responses are excluded. If fewer than two responses are eligible, all diversity bonuses are zero. Ineligible responses receive zero diversity reward regardless of group size. The reward does not require a speedup once downstream correctness is satisfied.

The historical weights in Eq. (3) use counts from all earlier optimization responses with nonempty technique labels, including both correct and incorrect downstream outcomes. Each response contributes once to D and once to $\mathrm{df}(u)$ for each technique it contains. We use counts from before the current batch so that a response cannot change its own weight. After scoring, the current assignments are accumulated without decay. The first batch uses unit weights.

The weighted Gram matrix in Eq. (4) uses these frequency weights. We apply Eq. (5) without row normalization or bound-based rescaling, preserving the magnitude of the technique weights. The $\log(1 + k)$ upper bound in DQO assumes normalized response embeddings and does not apply to this weighted representation. Validation receives no diversity bonus. Validation diversity is measured with unit technique weights and does not update training frequencies.

We sum the task, format, and weighted diversity rewards as in Eq. (6), then quantize the result to increments of 0.01. Writing this quantized reward as $\bar{r}_i$, the group advantage is

$$A_i = \frac{\bar{r}_i - \mathrm{mean}_{j=1}^{N} \bar{r}_j}{\mathrm{std}_{j=1}^{N} \bar{r}_j + \epsilon}, \qquad \epsilon = 10^{-6}. \tag{8}$$

The standard deviation is the sample standard deviation over all N strategies from the same state. The advantage is shared across the generated response tokens in the clipped policy update. The reward components are not normalized separately.

We annotate four strategies per planner for each of the 91 initially correct KernelBench kernels using the RL annotation prompt with 53 named techniques and a shared `other` label. Two tasks have an empty technique set in one sampled strategy and are excluded from both groups, leaving 89 paired tasks.

D Compositional Inference Procedure

Baseline inputs and output budgets. BoN and single-stage iteration use identical initial CUDA kernels for all 250 tasks, verified by code hashes. BoN samples ten candidates from the fixed initial kernel and its feedback without incorporating other candidates. Its generated-only metrics select among these ten candidates and exclude the initial kernel from the output set. Single-stage iteration updates the parent using the best correct kernel found so far and includes the initial kernel in final selection. Reported iteration metrics use the archived selected candidate, which maximizes measured speedup. Because reference timings may vary across evaluations, this selection can differ from selecting the lowest absolute kernel latency. The BoN runs and the single-stage iteration batch use a 32,768-token total output budget. Actual request limits can be lower because of model context limits and token-budget margins.

Compositional inference. A shared seed kernel initializes either the correct-kernel pool or the repair queue according to its correctness. When the correct-kernel pool is nonempty, we obtain an NCU profile for each member and present the whole pool to the planner, asking it to compare code, execution feedback, and profiles, then combine compatible mechanisms into an actionable strategy. Earlier planner strategies are not included in the pool context. Four strategies are sampled independently per optimization iteration.

A fixed annotator builds a shared catalogue for the current strategy group from its *Recommended techniques* sections. This catalogue is local to the group and does not use the global frequency weights from training. For nonempty technique sets T_i and T_j, similarity is

$$\mathrm{sim}(i, j) = \frac{|T_i \cap T_j|}{\max(|T_i|, |T_j|)}. \tag{9}$$

Complete-link clustering merges groups only when every cross-group pair has similarity at least $\tau = 0.5$. We select one strategy per cluster, jointly maximizing the minimum pairwise distance between representatives. The sum of pairwise distances breaks ties. Unclassified strategies remain singleton clusters so annotation failure cannot suppress their implementation.

The frozen coder receives the full correct-kernel pool and produces one implementation per selected strategy. Each queued failure receives one repair attempt per iteration, and newly generated failures are deferred to the next iteration. Pool updates remove exact code duplicates and retain the historical best without evicting other distinct outputs of the iteration. All generation and execution costs, including repair attempts, are counted when evaluating inference.

Decoding and budgets. We evaluate the planner checkpoint taken after the final training update. The planner decodes with temperature 1.0, top-p 1.0, and a 16,384-token output limit, matching its reinforcement learning sampling settings. The coder decodes with temperature 0.6, top-p 0.95, and a 32,768-token output limit, and the technique annotator decodes greedily with a 4,096-token output limit. Each request is issued once and retried at most twice. A single task receives at most 40 coder generations in total. Optimization and repair draw from this shared budget, optimization first, and the loop stops once the budget is exhausted. The shared seed kernels are sampled once per task from cudaLLM-8B with temperature 0.6 and a 32,768-token output limit.

Profiling and evaluation. Each correct kernel in the pool is profiled with Nsight Compute over five kernel launches under a 300-second timeout. The profile passed to the planner reports 24 metrics covering occupancy limits, instruction throughput for the FP32 and tensor pipelines, DRAM traffic and throughput, L1 and L2 hit rates and throughput, warp stall reasons, and branch divergence. Repair

prompts carry no profile, because a failing kernel has no valid measurement. Missing NCU evidence is rendered as `(no NCU data)`, and a profiling failure does not override the candidate's measured correctness or latency. Evaluation uses FP32 inputs, five correctness trials, ten warmup runs, and 100 timing runs on a high-end datacenter GPU, with a 600-second per-candidate timeout and random seed 42. The implementation rejects detected high-level PyTorch compute shortcuts in generated kernels.

For CUDACoach the cap of 40 covers optimization and repair together. Planner calls and profiling are additional and differ across methods, so the counts compare code generations rather than total computation.

E Compositional Inference Case Study

We illustrate how compositional inference inherits and refines techniques from parent strategies using KernelBench task 40 (`LayerNorm`), with our planner and Qwen3-Coder-Plus as the coder. Among multi-parent iterations that produce a new best kernel faster than PyTorch eager, this case has the largest improvement over its best parent. The initial kernel fails correctness checks and is repaired at $0.93\times$ speedup. Subsequent optimization reaches $6.34\times$, with the decisive iteration improving over its best parent by $1.50\times$. Figure 5 shows this iteration and the overall trajectory. Table 10 records the pool and clustering dynamics. The ten iterations use 21 optimization generations and 10 repair attempts within the 40-generation budget.

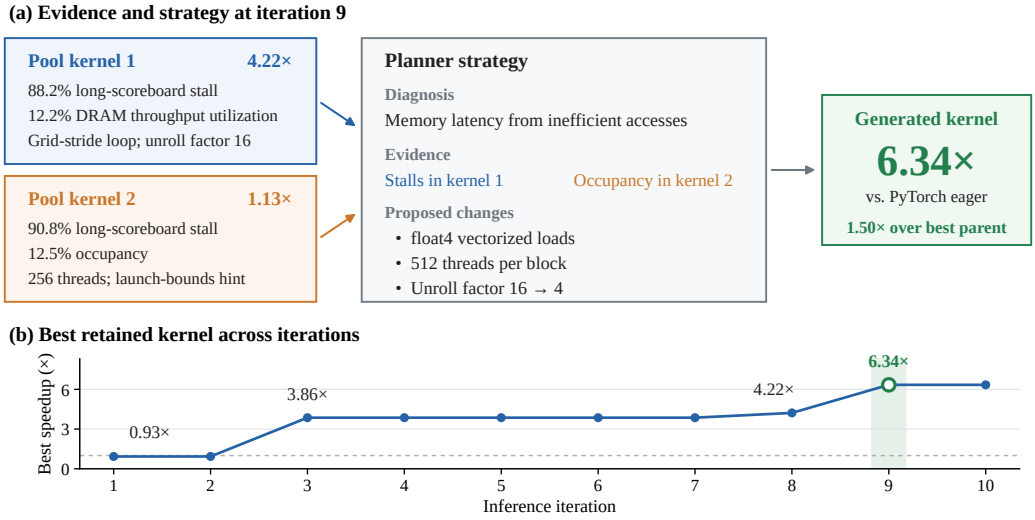

Figure 5: Compositional inference on KernelBench task 40 (`LayerNorm`). (a) At iteration 9, the planner builds on techniques inherited from two parent strategies and uses their implementations' profiles to propose further changes. The resulting kernel achieves $6.34\times$ speedup over PyTorch eager and $1.50\times$ over the best parent. (b) Best retained speedup across ten iterations, with iteration 9 highlighted. The dashed line denotes PyTorch eager.

At iteration 9, the pool contains two implementations with speedups of $4.22\times$ and $1.13\times$. The first parent strategy emphasizes grid-stride loops, loop unrolling, and simplified indexing, whereas the second emphasizes block-size tuning and `__launch_bounds__` for register allocation. Their implementations use 1,024 and 256 threads per block, respectively, both with an unrolling factor of 16. The new strategy inherits elements of both parent strategies and revises them using evidence from the two profiles. Its diagnostic explicitly identifies:

Code-specific evidence.

- **High long scoreboard stalls**: 88.24% (candidate 1) and 90.8% (candidate 2) indicate severe memory latency issues
- **Poor occupancy**: 12.49% in candidate 2 due to reduced block size of 256 threads

Table 10: Inference trajectory for KernelBench task 40 (`LayerNorm`). Four strategies are sampled in every iteration after the first, and one kernel is generated per cluster. Iteration 1 has an empty pool because the seed is incorrect and only repair runs.

Iteration	Correct kernels in pool	Clusters	Best speedup (×)
1	0	—	0.93
2	1	2	0.93
3	2	3	3.86
4	2	1	3.86
5	3	2	3.86
6	3	4	3.86
7	5	2	3.86
8	3	2	4.22
9	2	3	6.34
10	3	2	6.34

The resulting implementation retains grid-stride traversal and warp-based reduction, carries forward the launch-bounds constraint as `__launch_bounds__(512, 4)`, and adds `float4` vectorization while reducing the unrolling factor to 4. It reaches 6.34× speedup after passing correctness checks. This case illustrates inheritance and refinement of techniques from parent strategies rather than a direct copy of either implementation. The comparison with the slower parent also contributes diagnostic information for the next optimization step.

F RL Validation Dynamics

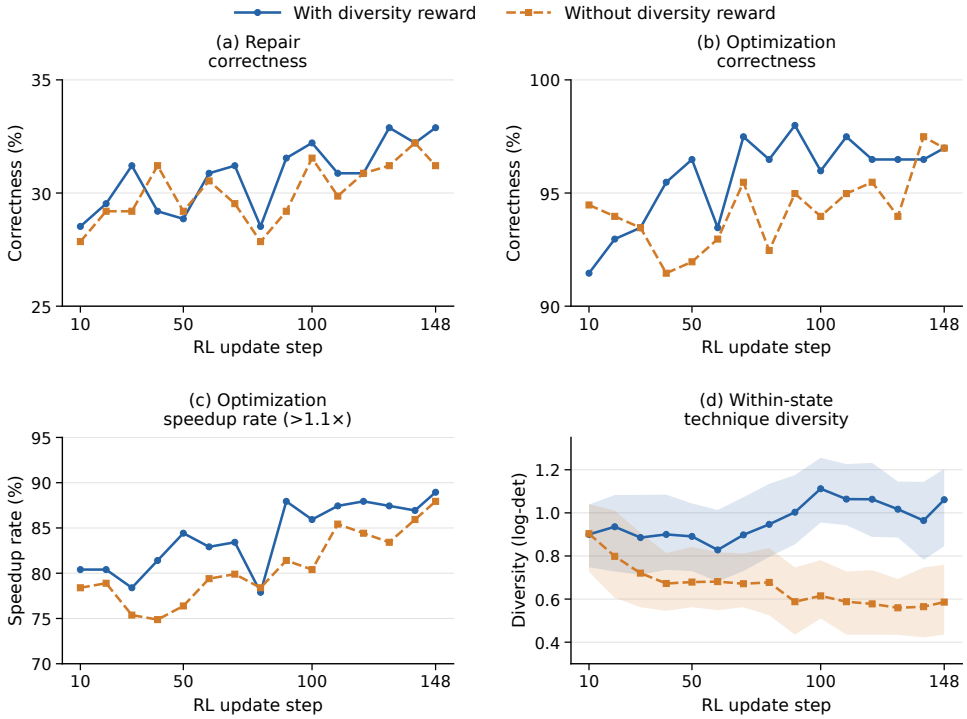

Figure 6: Validation metrics during RL training with and without the diversity reward. Panel (d) measures within-state technique diversity as the leave-one-out log-determinant over the technique sets recommended for the same state, aggregated per state, with the line at the median across states and the band spanning the interquartile range.

We compare training with and without the diversity reward on a fixed validation set of 298 repair states and 199 optimization states. At each checkpoint, the planner generates four strategies per state, and the frozen coder produces four implementations per strategy. Correctness measures the fraction of states with at least one correct implementation among the 16 candidates. The optimization speedup rate measures the fraction of all optimization states with at least one correct implementation exceeding $1.1\times$ speedup over the input CUDA kernel, matching the training reward threshold. Candidates marked as infrastructure failures count as unsuccessful. We evaluate every ten RL updates and at the final update 148.

Panels (a) to (c) of Figure 6 show higher optimization correctness with the diversity reward at every evaluated checkpoint from update 40 through update 130, and a higher speedup rate at 14 of the 15 checkpoints, with a smaller difference in repair correctness. The optimization gap narrows by update 148, when both runs reach 96.98% correctness and speedup rates are 88.94% with the reward and 87.94% without it.

Panel (d) separates the two runs more sharply. The two distributions coincide at update 10, after which within-state diversity declines without the reward and increases with it. From update 90 onward the interquartile ranges no longer overlap, and at update 148 the medians are 1.06 with the reward and 0.59 without it.

G Planner Prompts and Technique Catalogue

The single-kernel interface uses separate repair and optimization templates. The repair prompt supplies the reference program, failing kernel, error log, and GPU description. The optimization prompt supplies the reference program, current kernel, Nsight Compute report, and hardware description. Both request a strategy rather than a complete CUDA implementation. The multi-kernel context used for compositional inference is described in Appendix D.

The fixed reward-annotation catalogue contains 53 named techniques, organized into 27 core and 26 conditional entries, plus one collapsed `other` dimension for uncatalogued mechanisms. This scoring catalogue is distinct from the candidate shortlist in the optimization prompt. In particular, it includes pointer-aliasing hints and tail handling beyond the prompt shortlist. At inference, clustering instead uses a group-local catalogue. Annotation identifiers are used for scoring and selection, not passed to the coder. At inference, the coder receives the raw planner response, whereas the RL reward path extracts the final strategy after the reasoning block. In the prompts below, ILP means *instruction-level parallelism*.

The following panels reproduce the complete prompt wording and variable placeholders. Headings and code blocks are typeset for readability. Placeholders are filled with task-specific code, feedback, and hardware information. Panel titles and continuation labels are presentation aids, not part of the prompts.

Repair Strategy Prompt

System message

You are an expert in CUDA kernel debugging and repair. Your job is to produce a precise repair instruction that can guide a downstream code-generation model to fix the same bug. Your instruction must localise the error source and lay out an actionable, ordered repair plan. Do not write the fixed code.

User message

Task

You will be given a reference PyTorch implementation, a buggy CUDA-extension implementation, an error log, and the target GPU's hardware context.

Produce a repair instruction for a downstream model that will attempt to fix the same buggy CUDA code given the same inputs. The instruction will be injected directly into that model's prompt alongside the torch code, the buggy CUDA code, and the error log.

Repair Strategy Prompt (continued)

Your output must contain exactly the following three sections:

Root Cause:
Identify the dominant bug that best explains the observed failure. Explain precisely what the buggy implementation does wrong relative to the reference PyTorch code, including which operation, kernel, wrapper function, or data flow is incorrect and why it produces wrong results or errors. Only describe bugs that are evidenced by the error log and the buggy code. Do not speculate about unrelated issues.

Error Location:
Write bullet points that localise the error source in the buggy CUDA code. Each bullet must name the exact code region to inspect, such as the kernel name, wrapper function, launch site, forward() call path, indexing formula, shared-memory region, or a short distinctive code pattern. If exact line numbers are unavailable, use symbol names plus nearby statement patterns so the downstream model can find the location reliably.

Repair Plan:
Write a numbered repair plan that the downstream model can execute directly. Each step must (a) name the exact code region to change (kernel / function / launch site / indexing rule / wrapper / forward() call path), and (b) describe the change applied there. Keep it concrete and minimal. Cover kernel logic fixes, indexing corrections, function signature changes, wrapper updates, and forward() call-site changes — include a step only when it is actually required by the repair. When the fix involves shared memory, specify the layout explicitly — including how the buffer is partitioned across threads and the total allocation size. When function signatures change, state the old and new signatures explicitly and note all call sites that must be updated. You may include short code snippets or pseudo-code only when the fix is non-obvious or involves precise indexing.

Do not suggest replacing the CUDA extension with torch.* or nn.* calls for the main computation, but you may reference torch semantics as the target behavior to align with.

Reference PyTorch Code
`{torch_code}`

Buggy CUDA Code
`{cuda_code}`

Error Log
`{error_log}`

Target GPU
`{gpu_block}`

Figure 7: System and user messages for generating a kernel repair strategy. Orange placeholders are filled with task-specific inputs.

Optimization Strategy Prompt

System message

You are a senior CUDA performance engineer. Analyse the NCU profiling metrics and the current kernel, then reconstruct a reusable optimization instruction for a downstream CUDA-generation model.

User message

Task
A CUDA kernel is functionally correct but slow. You will be given a reference PyTorch implementation, an current CUDA kernel, and NCU profiling metrics.

Write a reusable optimization instruction for a downstream model.

Optimization Strategy Prompt (continued)

Be specific about what to change and why.
Do NOT write the optimized code — only give instructions.
Diagnose the bottleneck primarily from the current CUDA code and its NCU metrics.
Do not invent bottlenecks or techniques that are not supported by the code and profiling metrics.
Your diagnosis should be grounded in concrete code structure such as loops, indexing formulas, kernel decomposition, launch configuration, memory layout, or intermediate tensor materialization.
Select at most 1-3 optimisation techniques from the candidate categories above, and only if they are directly applicable here.

Output Format

Use exactly the following section headers:

Primary bottleneck
Write the dominant performance problem in the current CUDA implementation.

Code-specific evidence
Write bullet points that cite concrete loop structures, indexing formulas, launch patterns, memory-access behavior, or intermediate tensor materialization from the current CUDA code, and include NCU metrics that support the diagnosis.

Task-specific insight
Write bullet points describing any semantic, shape, workload, or data-layout properties that should guide the optimisation.

Recommended techniques
Write bullet points where each item names a concrete optimization technique and briefly explains why it is appropriate here.

Modification plan
Write a numbered plan describing the structural code changes that should be made. The steps should be actionable and concrete. You should include code snippets or pseudo-code fragments in backticks when they help clarify what needs to change, but do not output full optimized code.

Reference PyTorch Code
`{torch_code}`

Current CUDA Code
`{cuda_code}`

NCU Profiling Metrics
`{ncu_metrics}`

Target GPU
`{gpu_block}`

Candidate Optimisation Categories
`{technique_guide}`

Output ONLY the formatted instruction with the exact headers above, nothing else.

Figure 8: System and user messages for generating a kernel optimization strategy from code and profiling feedback.

The `{technique_guide}` placeholder is expanded with the following candidate shortlist.

Optimization Technique Shortlist

User message component

Use the following optimisation categories as a layered shortlist of candidate directions. Prefer techniques from Core techniques. Use Conditional techniques only when the current code structure, tensor shapes, or NCU metrics clearly justify them.

Core techniques:

- Shared-memory reuse / tiling
- Bank-conflict elimination
- Register blocking / register reuse
- Memory coalescing / layout-aware access
- Vectorized load/store
- Strided access optimization
- Prefetching
- Software pipelining
- Thread mapping / work decomposition
- Thread block size tuning
- Grid-stride loop
- Loop unrolling
- FMA fusion
- Strength reduction
- Computation reuse
- Recompute to reduce memory traffic
- Multiple accumulators for ILP
- Warp shuffle
- Warp reduction
- Minimize __syncthreads
- Kernel fusion
- Loop fusion
- Kernel fission
- Register limit / occupancy tuning
- Shared memory tuning

Conditional techniques (use only if clearly applicable here):

- Constant memory for small read-only broadcasted data
- Texture memory for special cached read patterns
- Async copy / cp.async when using staged shared-memory pipelines
- Cache policy control when cache behavior is a clear bottleneck
- Fast math functions only when transcendental ops exist and precision tradeoff is acceptable
- Warp vote / ballot
- Warp match
- Work queue / stealing
- Binning / bucketing
- Persistent threads
- Sparse formats
- Stream compaction
- Shared-memory atomics
- Atomic-based parallelization
- Warp-level atomic reduction
- Batched atomics
- Atomic replacement
- CAS loop optimization
- Cooperative groups
- Tensor Core data-layout optimization
- WMMA
- MMA PTX
- SFU utilization

Optimization Technique Shortlist (continued)

- Mixed precision
- Quantization
- CUDA Graphs

Do NOT recommend a technique unless you can point to the exact loop, indexing pattern, memory-access pattern, synchronization pattern, or tensor-shape property in the current CUDA code that should change.
Do NOT produce a checklist of generic CUDA advice.
Do NOT recommend host-device transfer, multi-stream, multi-GPU, NCCL, or system-level runtime techniques for this prompt.

Figure 9: Candidate directions inserted into the optimization prompt through the technique-guide placeholder.

The RL data-preparation pipeline appends the following requirement to the user prompt. It is separate from the base templates used in the diagnostic study.

RL Reasoning-Format Requirement

User message suffix

Thinking Format Requirement
Your response must start with `<think>`. Write your reasoning process inside `<think>...</think>`, then after `</think>` provide the final answer using exactly the required section headers. Do not leave the think block empty.

Figure 10: Reasoning-format requirement appended to the user message for RL training.

The following system message defines the 53-entry scoring catalogue. Its user message is a JSON object containing a `responses` list. Each item supplies `response_id` and the extracted `recommended_techniques` text. The annotator uses temperature 0. Its `other:name` outputs are collapsed to the shared `other` dimension for the RL reward.

Technique Annotation Prompt

System message

You label the Recommended techniques of alternative responses for one same task using a FIXED catalogue of CUDA optimization mechanisms.

The catalogue below is closed and shared across all requests. Select the id whose mechanism matches what the response actually proposed. Use the same id for the same mechanism wherever it appears, even if wording differs or it targets a different tensor, variable, dimension, or code location. Never assign an id to a response that did not explicitly propose it.

Assign one id per distinct core optimization choice the response recommends. Default to keeping a tightly coupled method as one id. Do not assign a separate id merely for a target, benefit, condition, or an implementation qualifier. Treat a jointly stated compiler-option bundle as one item. Do not infer from the Modification plan or the current code; read only the Recommended techniques.

If a response clearly proposes a concrete mechanism that no catalogue id covers, emit "other:<short lower-case name>" for it.

CATALOGUE:

CORE techniques:
- shared_memory_reuse_tiling: Shared-memory reuse / tiling
- bank_conflict_elimination: Bank-conflict elimination
- register_blocking_register_reuse: Register blocking / register reuse

Technique Annotation Prompt (continued)

- memory_coalescing_layout_aware_access: Memory coalescing / layout-aware access
- vectorized_load_store: Vectorized load/store
- strided_access_optimization: Strided access optimization
- prefetching: Prefetching
- software_pipelining: Software pipelining
- thread_mapping_work_decomposition: Thread mapping / work decomposition
- thread_block_size_tuning: Thread block size tuning
- grid_stride_loop: Grid-stride loop
- loop_unrolling: Loop unrolling
- fma_fusion: FMA fusion
- strength_reduction: Strength reduction
- computation_reuse: Computation reuse
- recompute_to_reduce_memory_traffic: Recompute to reduce memory traffic
- multiple_accumulators: Multiple accumulators for ILP
- warp_shuffle: Warp shuffle
- warp_reduction: Warp reduction
- minimize_syncthreads: Minimize __syncthreads
- kernel_fusion: Kernel fusion
- loop_fusion: Loop fusion
- kernel_fission: Kernel fission
- register_limit_occupancy_tuning: Register limit / occupancy tuning
- shared_memory_tuning: Shared memory tuning
- pointer_aliasing_hints_restrict: Pointer aliasing hints (__restrict__)
- tail_remainder_handling: Tail / remainder handling

CONDITIONAL techniques:

- constant_memory: Constant memory for small read-only broadcasted data
- texture_memory: Texture memory for special cached read patterns
- async_copy_cp_async: Async copy / cp.async when using staged shared-memory pipelines
- cache_policy_control: Cache policy control when cache behavior is a clear bottleneck
- fast_math_functions: Fast math functions only when transcendental ops exist and precision tradeoff is acceptable
- warp_vote_ballot: Warp vote / ballot
- warp_match: Warp match
- work_queue_stealing: Work queue / stealing
- binning_bucketing: Binning / bucketing
- persistent_threads: Persistent threads
- sparse_formats: Sparse formats
- stream_compaction: Stream compaction
- shared_memory_atomics: Shared-memory atomics
- atomic_based_parallelization: Atomic-based parallelization
- warp_level_atomic_reduction: Warp-level atomic reduction
- batched_atomics: Batched atomics
- atomic_replacement: Atomic replacement
- cas_loop_optimization: CAS loop optimization
- cooperative_groups: Cooperative groups
- tensor_core_data_layout_optimization: Tensor Core data-layout optimization
- wmma: WMMA
- mma_ptx: MMA PTX
- sfu_utilization: SFU utilization
- mixed_precision: Mixed precision
- quantization: Quantization
- cuda_graphs: CUDA Graphs

Return JSON only in this exact schema:

Technique Annotation Prompt (continued)

```
{"techniques":[{"id":"<catalogue id or
↪  other:name>","name":"<mechanism as
↪  stated>"}],"responses":[{"response_id":"<input
↪  id>","technique_ids":["<id>"]}]}
```

The "techniques" array is a deduplicated catalogue: list each id at most once, even when several responses word that mechanism differently or apply it with different parameters, widths, or tile shapes. Give it one general "name" for the shared mechanism rather than quoting any single response. Which responses proposed it is expressed only through the "responses" array.

Use catalogue ids verbatim as the "id" field. Return every input response_id exactly once.

Figure 11: Fixed-catalogue annotation prompt used to map recommended techniques to shared labels for scoring.