CORAL: CO-EVOLVING SKILL SELECTION, SUMMARIZATION, AND UTILIZATION VIA RL FOR GPU KERNEL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU kernel optimization requires generating correct implementations, identifying effective optimization techniques, and applying them to suitable workloads. We present **CORAL**, a reinforcement learning framework that couples skill discovery with skill exploitation within **a single shared language model** through a dynamically evolving skill library. CORAL jointly trains three agents sharing one LLM backbone: a Skill Selection Agent that retrieves relevant techniques via BM25 and LLM reranking, a Policy Agent that generates multi-turn Triton kernels conditioned on selected skills, and a Skill Summary Agent that distills successful rollouts into reusable skills. Candidate skills are added only after execution-based verification confirms first-turn speedups on their source tasks. The agents are initialized via a structured SFT cold start on diversity-filtered data and jointly optimized with multi-turn REINFORCE and per-agent advantage estimation. On KernelBench, CORAL-14B achieves 37.2%, 70.6%, and 32.2% on Level 1, 2, and 3 under the Fast_1 threshold using the last output of three-turn rollouts, outperforming our reproduced Dr. Kernel-14B baseline. Ablation studies show benefits from inference-time skills, joint training, learned selection, and diversity filtering, with trade-offs across task levels and speedup thresholds.

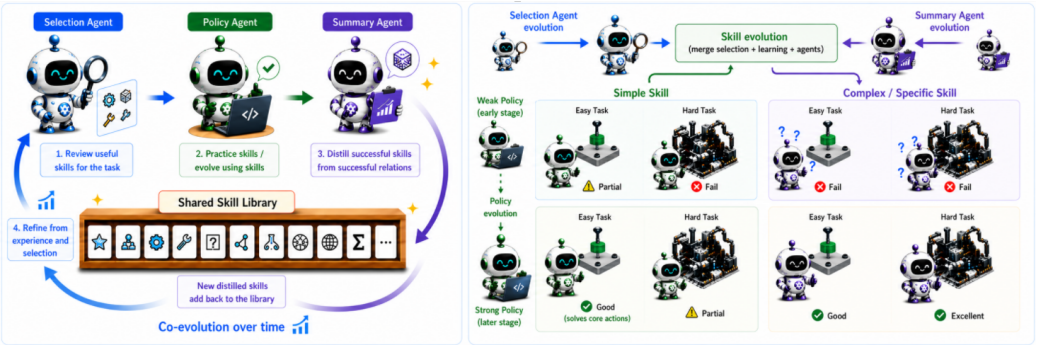

Figure 1: **Skill-policy co-evolution in CORAL.** The selection, policy, and summary agents form a closed RL loop, where task-relevant skills guide optimization and successful rollouts are distilled back into reusable skills. As the policy improves, the useful skill frontier shifts from simple skills to more complex and task-specific skills, enabling increasingly difficult kernel optimization.

1 INTRODUCTION

Large language models are increasingly shifting code generation from one-shot program synthesis toward feedback-driven program optimization (Jimenez et al., 2023; Fu et al., 2026). GPU kernel optimization (Ouyang et al., 2025; Li et al., 2025a) is a particularly representative setting in this transition. Unlike many code-generation tasks whose primary objective is functional correctness, kernel optimization typically starts from an implementation that is already correct but inefficient, and improves it through a sequence of transformations. The resulting improvements are measurable, incremental, and compositional, rather than binary (Liu et al., 2026).

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

This progressive structure makes reinforcement learning a natural paradigm, but it also introduces a more fundamental difficulty (Liu et al., 2026; Li et al., 2025b; Guo et al., 2026; Woo et al., 2025). As the policy improves, the source of further gains changes. Early in training, substantial improvements often come from broad transformations that correct obvious inefficiencies. Later, however, the remaining performance gap is increasingly concentrated in subtler, more context-dependent, and more tightly coupled optimization decisions. Thus, techniques that help early in training may offer less additional benefit later. This observation motivates updating the knowledge library during training. A learning system must also be able to preserve optimization knowledge discovered at earlier stages, re-evaluate its usefulness as the policy evolves, and expose the most relevant knowledge to future decisions. What is needed, therefore, is not merely a stronger optimizer for the current task, but a mechanism for maintaining and reusing optimization knowledge in a way that remains aligned with the agent's evolving frontier.

Existing approaches address important parts of this problem. On the one hand, recent kernel agents have substantially improved optimization quality through prompting, execution-guided refinement, post-training, and reinforcement learning, thereby strengthening the generator and the search process for the current task (Liu et al., 2026; Li et al., 2025b; Guo et al., 2026; Woo et al., 2025; Cao et al., 2026; Wei et al., 2025; Zhu et al., 2026). On the other hand, a growing body of work on skill-based agents and reinforcement learning has shown that experience can be abstracted into reusable units that support transfer across tasks and training stages (Lu et al., 2026; Xia et al., 2026; Li et al., 2026; Tu et al., 2026). However, in kernel optimization, such reusable units must satisfy two additional requirements. First, their usefulness must be judged relative to the policy's current competence, since techniques that were once beneficial may later become redundant after being internalized. Second, newly acquired knowledge should not be retained on plausibility alone, but on whether it yields verifiable execution gains in the target environment. These requirements shift the problem from generic experience accumulation to *frontier-aware skill evolution*: reusable skills must be continually selected, and summarized throughout training.

Motivated by this perspective, we propose CORAL, a reinforcement learning framework for GPU kernel optimization that closes the loop between skill discovery and skill exploitation. CORAL jointly trains a selection mechanism that retrieves task-relevant skills, a policy that performs multi-turn kernel optimization under optional skill conditioning, and a summarization mechanism that distills successful rollouts into candidate skills. In this way, CORAL treats skill maintenance as part of the learning problem itself. Empirically, CORAL-14B achieves 37.2%, 70.6%, and 32.2% on Level 1, Level 2, and Level 3 under the Fast_1 threshold with three-turn evaluation. On Level 3, this is a 10.1 percentage-point increase over our reproduced Dr. Kernel-14B (Liu et al., 2026), or approximately 46% relative improvement. The following ablation studies examine the contribution of each component.

In summary, our contributions are as follows:

- We propose CORAL, a unified framework that couples skill selection, skill utilization, and skill summarization into a single reinforcement learning loop to improve agent's performance.
- Empirically, we show that CORAL-14B achieves 37.2%, 70.6%, and 32.2% on Level 1, Level 2, and Level 3 under the Fast_1 threshold on KernelBench, outperforming our reproduced Dr. Kernel-14B by approximately 46% relative on Level 3 with three-turn evaluation.
- We conduct ablation studies of inference-time skill injection, joint multi-agent RL, LLM-based selection, and diversity-filtered SFT data. These studies show how each component affects performance across task levels and speedup thresholds.

2 Related Work

Kernel Agents. Another closely related line of work studies the use of LLMs for GPU kernel generation and optimization. Early approaches mainly rely on prompting, iterative refinement, or execution-guided search. For example, CUDA-LLM (Chen et al., 2025) improves kernels using compilation and runtime feedback, while GPU Kernel Scientist formulates optimization as repeated proposal, execution, and revision. More recent work has moved toward dedicated post-training and reinforcement learning recipes, including Dr. Kernel (Liu et al., 2026), DRTriton (Guo et al., 2026), and CUDA Agent (Dai et al., 2026), which improve generation quality through supervised post-

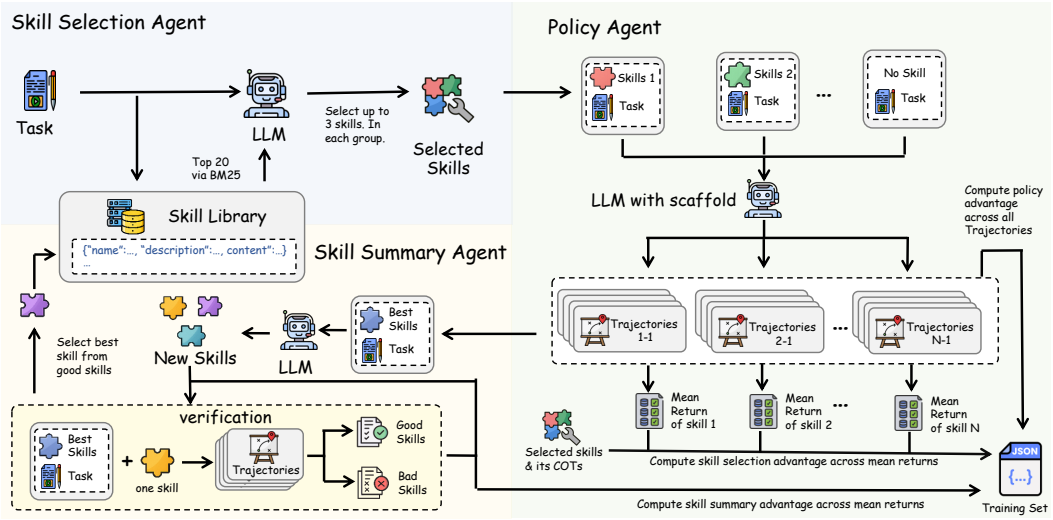

Figure 2: The structure of CORAL.

training, reinforcement learning, multi-turn optimization, large-scale synthetic data, and more robust training objectives. In parallel, works such as Kernel-Smith (Du et al., 2026) show that execution feedback, evolutionary search, and post-training optimization can substantially improve the practical performance of generated kernels. While these approaches have significantly advanced LLM-based kernel optimization, they focus mainly on improving the generator or the search process for the current task, rather than on maintaining a skill library whose contents are continually regenerated, reselected, and execution-validated as the agent evolves.

Skill-based reinforcement learning. A growing body of work studies how agents can accumulate reusable skills from past experience, rather than solving each task from scratch. Early systems such as Voyager (Wang et al., 2023) and ExpeL (Zhao et al., 2024) demonstrate that explicitly storing experience or insights in an external repository can substantially improve long-horizon planning and cross-task transfer. Building on this idea, recent work has integrated skill learning more tightly with reinforcement learning. SAGE (Liang et al., 2025) combines skill libraries with RL, places greater emphasis on sequential rollouts, and skill-integrated rewards. SkillRL (Xia et al., 2026) constructs a hierarchical skill bank through experience distillation and allows the skill library to co-evolve recursively with the policy during RL. In contrast, SKILL0 (Lu et al., 2026) focuses less on maintaining an external skill library throughout training, and instead studies how retrieved skills can be progressively internalized into model parameters through a curriculum that gradually removes skill context. Our focus is the joint training of selection, utilization, and summarization with execution-based skill admission for kernel optimization. Skill utility may change as the policy improves; we therefore update both the policy and its external library during training.

3 CORAL

We present **CORAL**, a reinforcement learning framework that trains an LLM to write optimized Triton kernels while continuously accumulating and reusing a library of optimization techniques. The core idea is to close the loop between *discovery* and *exploitation*: the policy discovers effective optimizations, and the summary agent distills them into skills for subsequent tasks. Relevant skills are retrieved and injected into the model's context to guide kernel generation across tasks and training steps. The framework comprises three agents trained jointly with reinforcement learning. As illustrated in Figure 2, these are: (i) a **Skill Selection Agent** that retrieves task-relevant skills from a growing library; (ii) a **Policy Agent** that generates CUDA kernels in multiple turns, optionally conditioned on injected skills; and (iii) a **Skill Summary Agent** that distills successful rollouts into new skills. These are three prompted roles of one shared LLM, trained with role-specific rewards and a combined objective.

3.1 SKILL LIBRARY

The skill library $\mathcal{L}$ is a collection of GPU optimization techniques. Each skill contains five fields:

- **`name`**: a short `snake_case` identifier, used as a unique key within a snapshot and as the retrieval target returned by the Selection Agent's tool call.
- **`description`**: a one-sentence summary of what the technique does and when it applies, shown to the Selection Agent to support relevance judgement without loading full content.
- **`scope`**: a coarse category that helps the Selection Agent filter by kernel subtype.
- **`tags`**: a list of keywords used to describe the domain of the skill.
- **`content`**: a free-text Markdown document with the full technique description, code snippets, and usage notes, which is the text ultimately injected into the policy's context.

The library is persisted as JSONL snapshots keyed by training step, enabling exact checkpoint restart.

3.2 SKILL SELECTION AGENT

Before the policy attempts a task, the Selection Agent acts as a *retriever*: given the task description, it queries the skill library and assembles a concise set of potentially useful optimization techniques to provide as context. The key challenge is to surface the most relevant skills efficiently as $\mathcal{L}$ grows over training. To this end, we use a two-stage procedure that combines lightweight lexical search with LLM-based relevance judgement.

Stage 1: BM25 pre-filtering. To support efficient retrieval, we maintain a BM25 index (Robertson & Zaragoza, 2009) over the concatenation of each skill's `name`, `description`, `tags`, and `content` fields. The index is built lazily after each library load and rebuilt after each training step's flush. Given the task description, we retrieve the top-N_{BM25} candidate skills from $\mathcal{L}$ by BM25 score.

Stage 2: LLM selection. The Selection Agent receives the task and candidate skill names and descriptions. It calls `select_skills`, providing a `think` field and an ordered list of zero to k_{select} skills, each with a `name` and brief `reason`. If the library is empty or no skill scores above the BM25 threshold, the agent returns an empty selection and $c = \varnothing$.

3.3 POLICY AGENT

The Policy Agent is the *executor*, it receives the task and an optional skill context c produced by the Selection Agent, and generates an optimized Triton kernel implementation through multi-turn dialogue with a code execution environment. Its core capability is to leverage the injected skills to write kernels. When the Skill Selection Agent provides skills, the `content` and `name` fields of the skill will be injected into the task prompt before the agent begins generation. When $c = \varnothing$, the prompt is passed through unmodified.

3.4 SKILL SUMMARY AGENT

The Summary Agent closes the loop by acting as a *distiller*, when the Policy Agent produces a strong rollout, the Summary Agent analyzes the successful trajectory and extracts the underlying optimization insight as a new skill to be added to $\mathcal{L}$. Over training, this mechanism turns individual task successes into reusable knowledge, continuously enriching the library from which future selection and policy calls benefit. The agent operates conditionally and uses execution feedback to filter candidate techniques.

Trigger condition. The Summary Agent is invoked on a completed policy rollout only when the rollout exhibits meaningful improvement:

$$R^* > \alpha \cdot r_1 \quad \text{and} \quad R^* > \beta, \tag{1}$$

where R^* is the maximum speedup within the rollout and r_1 is its first-turn speedup. The relative threshold α requires improvement over the initial attempt, while the absolute threshold β rejects negligible gains. These thresholds filter candidates; they are not statistical significance tests.

Summary inference. For a triggered rollout, the Summary Agent receives the summarization instructions, original task, injected skills (if any), first-turn and best-turn code, and tool schema. It calls `update_skill_library` once, proposing at most $S_{\max}$ new skills.

A rule-based lazy-skill filter rejects any proposed skill whose `name`, `description`, `tags` or `content` suggests bypassing custom CUDA/Triton kernels (e.g., *"fall back to PyTorch"*, *"use `torch.compile`"*), ensuring only genuine kernel optimization techniques enter the library.

Skill verification. Each candidate skill is validated by re-running the policy on the original task with the skill injected into the system prompt. A skill is added to the library only if the resulting first-turn speedup satisfies:

$$r_{\text{verify}} \;\geq\; \max\big(\beta,\; \alpha \cdot r_1\big), \tag{2}$$

where r_{verify} denotes the first-turn speedup with both the original skills and the candidate skill, and r_1 is the first-turn speedup with only the original skills. This verifies reuse on the source task; it does not by itself establish transfer to other tasks.

3.5 SFT COLD START

Before reinforcement learning, we initialize all three agents through supervised fine-tuning. The training data for each agent is constructed offline from a shared seed set of multi-turn kernel optimization trajectories collected from `CUDALLM` tasks (Chen et al., 2025) using a strong LLM like GPT-5.4 as the policy.

Summary SFT. Summary Agent training data is constructed in three phases. In Phase 1, we apply Eq. 1 to the Dr. Kernel cold-start dataset to filter trajectories that exhibit sufficiently strong improvement signals, ensuring that only high-quality candidates are retained for skill extraction. In Phase 2, we randomly select half of the filtered trajectories and run the Summary Agent on each to generate `update_skill_library` tool-call demonstrations. The resulting skills are aggregated to form an initial skill library, which serves as the foundation for subsequent SFT stages. In Phase 3, we process the remaining half using BM25 to retrieve skills from the Phase 2 library and inject them into the prompt, producing skill-conditioned summarization demonstrations. The tool calls in Phase 2 and Phase 3 will be transformed into the supervised training example.

Selection SFT. After the seed skills are generated by last phases, we use the same dataset to generate the Selection SFT data. For each seed task, we run the Selection Agent call using the two-stage BM25+LLM procedure. Calls are executed against the skill library in the Summary SFT generation stage. Each call produces one `select_skills` tool-call conversation as the SFT data.

Policy SFT. We use two complementary data sources for the policy. The *base policy SFT* data comes directly from the Dr. Kernel coldstart dataset. The *skill-injected policy SFT* data is derived from the same base tasks but with skill content injected into the first user turn. The skill is selected from the whole skill library generated from the summary agent above via BM25. We use GPT-5.4 to sample the trajectories with real kernel environment feedbacks for 5 turns each task, align with the setting in *base policy SFT*. The two sources are combined as the final Policy SFT data, so the model retains its baseline kernel-writing capability while also learning the skill-conditioned format.

Filtering. After constructing the skill summarization and skill selection datasets, we observe a pronounced distributional imbalance: the model often generates overly generic and easily learnable skills, such as *identifying and optimizing hot paths*. These recurring patterns can accelerate early-stage RL results but later may reduce skill diversity and limit further improvement. To mitigate this, we apply structured downsampling to both datasets. We encode all skill texts with `Qwen3-Embedding-8B` and cluster the resulting representations using HDBSCAN. For the skill summarization dataset, we iteratively downsample each cluster until at most two instances remain, yielding a more uniform skill distribution and reducing the dominance of generic patterns. For the skill selection dataset, we perform cluster-aware deduplication within each tool call: skills from the same semantic cluster are pruned to the most task-relevant representative, prioritizing alignment with the `think` reasoning field. Trajectories whose reasoning becomes inconsistent after pruning are discarded.

We shuffle all three types of data after downsampling to train the cold-start model.

3.6 JOINT REINFORCEMENT LEARNING

Training setup. For each task in a training batch of size B, we expand to $k{+}1$ parallel *schemes*. The first k schemes are obtained by running k independent Selection Agent calls on the task, each following the two-stage BM25+LLM procedure described in §3.2 and producing a skill context $c_1, \ldots, c_k$. The $(k{+}1)$-th scheme is the *null scheme* ($c = \varnothing$), which always serves as an unmodified baseline. For each scheme i, the Policy Agent generates n rollout samples, yielding trajectories $\tau_{i,j}^{\text{pol}}$ with reward $R_{i,j,t}$ and returns $G_{i,j,t}$ at each turn in the trajectories. For triggered tasks, we select the best policy trajectories from each task as the summary seed and run s independent Summary Agent calls. All agents are trained jointly using REINFORCE with the advantage below:

Policy advantage. We use multi-turn TRLOO[1], grouping all $(k{+}1){\times}n$ rollouts for the same task into a single LOO group. This treats different skill schemes as parallel rollout samples, yielding a lower-variance baseline than per-scheme grouping:

$$A_{i,j,t}^{\text{pol}} \;=\; G_{i,j,t} \;-\; \frac{1}{(k{+}1)n - 1} \sum_{(i',j') \neq (i,j)} G_{i',j',t}. \tag{3}$$

Selection advantage. The reward for scheme i is the mean best-turn policy reward under that scheme, $\bar{R}_i = \frac{1}{n}\sum_j \max_t R_{i,j,t}$. For each learned selection $i \in \{1, \ldots, k\}$, its LOO baseline averages the other k schemes, including the null scheme indexed by $k+1$:

$$A_i^{\text{sel}} \;=\; \bar{R}_i \;-\; \frac{1}{k} \sum_{\substack{i'=1 \\ i' \neq i}}^{k+1} \bar{R}_{i'}. \tag{4}$$

This trains the Selection Agent to prefer skill combinations that consistently improve policy returns over alternative selections on the same task.

Summary advantage. For each summary call m, we define its reward as the verification speedup of the best generated skill,

$$R_m^{\text{sum}} \;=\; r_{\text{verify}}^{(m)} \cdot \mathbf{1}\Big[r_{\text{verify}}^{(m)} \;\geq\; \max\big(\beta,\; \alpha \cdot r_1^{(m)}\big)\Big]$$

If all proposed skills corresponding to the task-original skill pair fail verification, all the summary result under that pair will be removed. We then apply LOO within the s parallel summary calls for the same task:

$$A_m^{\text{sum}} \;=\; R_m^{\text{sum}} \;-\; \frac{1}{s-1} \sum_{m' \neq m} R_{m'}^{\text{sum}}. \tag{5}$$

This encourages the Summary Agent to generate skills that achieve higher verified speedup compared to alternative summaries on the same task.

Combined objective. The final training loss combines all three agent types with per-agent weights:

$$Loss \;=\; Loss^{\text{policy}} \;+\; w_{\text{select}}\, Loss^{\text{select}} \;+\; w_{\text{summary}}\, Loss^{\text{summary}}. \tag{6}$$

Skill library update. After all s summary calls for a triggered task complete, we select the verified skill with the highest r_{verify} across their candidates to enter the library cache $\mathcal{L}_{cache}$. If no skill passes verification, nothing is staged. At the end of each training step, all staged skills in $\mathcal{L}_{cache}$ are atomically flushed to a new versioned snapshot of $\mathcal{L}$ and broadcast to all rollout workers, so every subsequent step immediately benefits from the newly acquired knowledge. The $\mathcal{L}_{cache}$ is then cleared. This one-per-task rule keeps library growth conservative and ensures that only the most reliably useful technique from each trajectory is retained. The online skill library is initialized as empty at the beginning of RL and populated exclusively with skills generated and verified during RL training. Skills constructed for SFT data will not use, since it's generated by GPT.

[1]The introduction of TRLOO can be seen in Appendix C

4 EXPERIMENT

4.1 EXPERIMENTAL SETUP

Benchmark and metrics. We evaluate CORAL on KernelBench (Ouyang et al., 2025), which contains 100 Level 1 tasks, 100 Level 2 tasks, and 50 Level 3 tasks. Each task provides a reference PyTorch implementation; the model must produce a functionally correct Triton kernel that outperforms the baseline in end-to-end wall-clock time. We report performance as the percentage of tasks achieving at least $x\times$ speedup over the reference PyTorch implementation, denoted Fast_x, for thresholds $x \in \{1.0, 1.2, 1.5, 2.0\}$. Stricter thresholds require larger speedups: Fast_1 measures whether a correct kernel matches or improves on the reference, while $\text{Fast}_{1.2}$ requires at least a 20% speedup. Training–evaluation overlap checks appear in Appendix B.2.

Evaluation protocol. Tables 1 and 2 use the last output of three-turn rollouts for our evaluations. Table 3 separately reports the last and best outputs of 14-turn rollouts. Each evaluation uses eight independent samples per task; entries $A_{\pm B}$ report the mean A and sampling standard deviation B. The checkpoint's skill library is frozen throughout evaluation: **test rollouts do not add skills**. Different RL training configurations use their own skill libraries inside their checkpoints.

Baselines. We compare against three categories: (i) *General-purpose models* used in zero-shot settings: GPT-5, Claude-4.5-Sonnet, Deepseek-V3.2-Thinking, and GLM-4.7 (Liu et al., 2025; Zeng et al., 2025); (ii) *Other open-weight baselines*: Qwen3-8B, Qwen3-32B, Qwen3-Coder-A30BA3, and AutoTriton (Li et al., 2025b; Yang et al., 2025); (iii) *Domain-specifically trained models*: Dr. Kernel, the strongest published RL-trained kernel optimization baselines under 8B/14B settings.

Training setup. CORAL is built on Qwen3-8B-Base and Qwen3-14B-Base backbones (Yang et al., 2025). All three agents share weights and are jointly fine-tuned. The SFT cold start uses the Dr. Kernel cold-start dataset filtered and augmented as described in §3.5. For RL, each batch task is expanded into $k + 1$ parallel skill schemes ($k = 3$), each producing $n = 4$ policy rollouts, giving $(k + 1) \times n = 16$ trajectories per task for advantage normalization. Summary calls are triggered according to Eq. 1 and produce $s = 2$ independent summary candidates per triggered task. BM25 pre-filtering retains the top-20 candidates; the LLM reranker selects at most $k_{\text{select}} = 3$ skills. Skill verification uses thresholds $\alpha = 1.2$ and $\beta = 1.2$. The loss weights are $w_{\text{select}} = 0.3$ and $w_{\text{summary}} = 0.5$. All rollouts use temperature $T = 1.0$.

4.2 MAIN RESULTS

Table 1 compares CORAL with general-purpose LLMs and kernel-specialized baselines across three difficulty levels and four speedup thresholds. At $\text{Fast}_{1.2}$, CORAL improves over our reproduced Dr. Kernel at both model sizes. At Qwen3-14B, Level 2 increases from 18.1 to 27.1 and Level 3 from 2.8 to 7.3. At Qwen3-8B, Level 2 increases from 9.4 to 22.1 and Level 3 from 0.5 to 3.0. These gains across both model sizes support the benefit of skill-conditioned training and evaluation beyond scaling the backbone alone. The specific training dynamic can be seen in Appendix E

4.3 ABLATION STUDY

To evaluate the main design choices within this single-model setting, we conduct five ablation experiments on both 8B and 14B models. The ablated conditions are as follows:

- **Ablation 1** (*no skill injection at inference*): CORAL is trained with the full pipeline but skill context is withheld at evaluation time; the policy receives no skill conditioning.
- **Ablation 2** (*unfiltered SFT data*): Diversity filtering is removed from the SFT cold start; RL then proceeds with the full three-agent framework starting from a less diverse initialization.
- **Ablation 3** (*policy RL only*): Only the policy loss is computed during RL. All three roles still share the updated backbone, so the selector and summarizer are not frozen models.
- **Ablation 4** (*policy RL only + BM25 selection*): As Ablation 3, but LLM-based reranking is replaced by BM25 retrieval. This is a reproduction of SkillRL on the kernel task.
- **Ablation 5** (*no skills throughout*): The skill library is removed entirely from both training and inference; the model is trained with policy RL only and receives no skill context at any stage.

Table 1: Main results (%), using three-turn last-output evaluation. Entries are mean$_{\pm\text{std}}$ over eight samples per task; * marks results quoted from Dr. Kernel without reported standard deviations. Bold denotes the best mean, not significance.

Model	LEVEL1				LEVEL2				LEVEL3			
	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2
GPT-5*	19.5	16.5	12.5	11.0	46.7	**28.6**	**13.1**	3.0	21.0	**12.0**	4.0	2.0
Claude-4.5-Sonnet*	15.5	13.5	11.0	8.5	50.0	26.7	9.2	1.8	21.0	11.0	**5.0**	**4.0**
Deepseek-V3.2-Thinking*	7.5	5.5	4.5	4.0	11.0	6.5	2.5	0.5	2.0	1.0	0.0	0.0
GLM-4.7*	19.4	17.2	13.1	10.4	30.0	20.5	8.5	3.5	5.0	2.0	2.0	2.0
Qwen3-8B*	5.8	4.8	4.1	3.4	13.0	5.6	2.0	1.1	5.7	0.2	0.0	0.0
Qwen3-32B*	6.1	4.9	4.3	4.0	14.0	9.4	2.4	0.2	3.5	0.0	0.0	0.0
Qwen3-Coder-A30BA3*	6.0	5.2	5.1	3.8	12.6	5.0	1.5	0.3	7.0	1.0	0.0	0.0
AutoTriton*	4.5	3.6	2.8	2.1	30.6	9.2	2.6	0.5	7.5	0.0	0.0	0.0
Cold-Start-8B*	7.5	6.6	5.0	4.3	8.8	5.6	1.8	0.4	0.5	0.0	0.0	0.0
Cold-Start-8B (repr.)	$11.4_{\pm3.2}$	$9.3_{\pm2.4}$	$7.4_{\pm2.4}$	$4.9_{\pm2.0}$	$11.8_{\pm3.3}$	$8.6_{\pm3.1}$	$5.9_{\pm2.0}$	$3.5_{\pm1.5}$	$0.5_{\pm0.9}$	$0.3_{\pm0.7}$	$0.3_{\pm0.7}$	$0.0_{\pm0.0}$
Cold-Start-14B (repr.)	$13.6_{\pm4.0}$	$11.6_{\pm3.5}$	$8.5_{\pm3.2}$	$6.4_{\pm2.1}$	$20.9_{\pm3.9}$	$12.4_{\pm3.5}$	$7.9_{\pm2.6}$	$4.1_{\pm1.7}$	$2.0_{\pm1.5}$	$0.8_{\pm1.0}$	$0.3_{\pm0.7}$	$0.3_{\pm0.7}$
Dr. Kernel-8B*	15.9	12.8	10.9	8.4	46.0	20.0	5.0	1.5	10.8	1.0	0.0	0.0
Dr. Kernel-8B (repr.)	$17.9_{\pm4.7}$	$14.9_{\pm3.9}$	$10.5_{\pm2.1}$	$7.2_{\pm2.0}$	$26.8_{\pm2.8}$	$9.4_{\pm2.3}$	$5.2_{\pm2.0}$	$2.0_{\pm1.5}$	$3.0_{\pm1.9}$	$0.5_{\pm0.9}$	$0.5_{\pm0.9}$	$0.5_{\pm0.9}$
Dr. Kernel-14B*	20.3	16.9	13.2	11.6	49.2	25.6	7.4	2.1	8.8	1.2	0.2	0.0
Dr. Kernel-14B (repr.)	$30.4_{\pm3.2}$	$26.0_{\pm3.5}$	$20.4_{\pm3.1}$	$13.9_{\pm2.6}$	$58.5_{\pm2.9}$	$18.1_{\pm2.5}$	$7.5_{\pm1.5}$	$4.1_{\pm1.6}$	$22.1_{\pm4.5}$	$2.8_{\pm1.8}$	$1.8_{\pm1.7}$	$1.5_{\pm1.4}$
CORAL-8B	$26.1_{\pm3.1}$	$20.4_{\pm3.6}$	$16.8_{\pm2.4}$	$13.0_{\pm2.5}$	$44.8_{\pm4.4}$	$22.1_{\pm1.7}$	$\mathbf{13.1}_{\pm3.1}$	$\mathbf{7.9}_{\pm2.3}$	$10.1_{\pm4.0}$	$3.0_{\pm2.1}$	$2.0_{\pm2.1}$	$0.8_{\pm2.1}$
CORAL-14B	$\mathbf{37.2}_{\pm3.7}$	$\mathbf{27.3}_{\pm2.9}$	$\mathbf{22.9}_{\pm3.1}$	$\mathbf{16.2}_{\pm2.4}$	$\mathbf{70.6}_{\pm5.0}$	$27.1_{\pm3.2}$	$11.4_{\pm3.1}$	$6.2_{\pm3.1}$	$\mathbf{32.2}_{\pm6.2}$	$7.3_{\pm3.4}$	$2.3_{\pm2.0}$	$1.3_{\pm1.0}$

Table 2: Three-turn last-output ablations (%, mean$_{\pm\text{std}}$), using the same final CORAL-14B checkpoint as Table 1. Bold denotes the best mean per model-size group.

Model	LEVEL1				LEVEL2				LEVEL3			
	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2
Qwen3-8b series												
Dr. Kernel-8B (repr.)	$17.9_{\pm4.7}$	$14.9_{\pm3.9}$	$10.5_{\pm2.1}$	$7.2_{\pm2.0}$	$26.8_{\pm2.8}$	$9.4_{\pm2.3}$	$5.2_{\pm2.0}$	$2.0_{\pm1.5}$	$3.0_{\pm1.9}$	$0.5_{\pm0.9}$	$0.5_{\pm0.9}$	$0.5_{\pm0.9}$
CORAL-8B	$26.1_{\pm3.1}$	$\mathbf{20.4}_{\pm3.6}$	$16.8_{\pm2.4}$	$\mathbf{13.0}_{\pm2.5}$	$44.8_{\pm4.4}$	$22.1_{\pm1.7}$	$13.1_{\pm3.1}$	$7.9_{\pm2.3}$	$10.1_{\pm4.0}$	$\mathbf{3.0}_{\pm2.1}$	$\mathbf{2.0}_{\pm2.1}$	$0.8_{\pm2.1}$
w Ablation1	$16.2_{\pm3.9}$	$12.8_{\pm4.0}$	$9.6_{\pm3.2}$	$6.8_{\pm2.7}$	$20.6_{\pm3.4}$	$10.5_{\pm2.7}$	$6.8_{\pm1.8}$	$3.8_{\pm1.6}$	$2.0_{\pm1.1}$	$0.8_{\pm1.0}$	$0.5_{\pm0.9}$	$0.3_{\pm0.7}$
w Ablation2	$21.8_{\pm2.4}$	$16.9_{\pm3.0}$	$13.5_{\pm1.6}$	$9.2_{\pm1.0}$	$34.6_{\pm3.0}$	$\mathbf{23.5}_{\pm3.7}$	$\mathbf{16.9}_{\pm3.2}$	$\mathbf{12.0}_{\pm2.8}$	$6.0_{\pm2.6}$	$2.5_{\pm1.4}$	$\mathbf{2.0}_{\pm1.5}$	$\mathbf{1.0}_{\pm1.1}$
w Ablation3	$22.6_{\pm4.2}$	$17.4_{\pm4.1}$	$14.4_{\pm3.3}$	$11.1_{\pm2.6}$	$45.1_{\pm7.8}$	$16.8_{\pm2.5}$	$6.2_{\pm1.3}$	$3.1_{\pm1.4}$	$7.5_{\pm4.4}$	$1.3_{\pm1.5}$	$0.8_{\pm1.5}$	$0.3_{\pm0.7}$
w Ablation4	$\mathbf{29.2}_{\pm2.8}$	$19.8_{\pm2.7}$	$\mathbf{17.0}_{\pm2.3}$	$12.1_{\pm2.5}$	$51.2_{\pm3.9}$	$19.2_{\pm2.8}$	$8.2_{\pm2.0}$	$4.9_{\pm1.1}$	$\mathbf{13.5}_{\pm3.0}$	$0.8_{\pm1.0}$	$0.2_{\pm0.7}$	$0.2_{\pm0.7}$
w Ablation5	$25.4_{\pm1.2}$	$19.8_{\pm1.6}$	$16.6_{\pm2.1}$	$10.6_{\pm2.3}$	$\mathbf{51.6}_{\pm4.1}$	$16.5_{\pm2.3}$	$5.8_{\pm3.0}$	$2.1_{\pm1.6}$	$12.3_{\pm4.5}$	$0.3_{\pm0.7}$	$0.0_{\pm0.0}$	$0.0_{\pm0.0}$
Qwen3-14b series												
Dr. Kernel-14B (repr.)	$30.4_{\pm3.2}$	$26.0_{\pm3.5}$	$20.4_{\pm3.1}$	$13.9_{\pm2.6}$	$58.5_{\pm2.9}$	$18.1_{\pm2.5}$	$7.5_{\pm1.5}$	$4.1_{\pm1.6}$	$22.1_{\pm4.5}$	$2.8_{\pm1.8}$	$1.8_{\pm1.7}$	$\mathbf{1.5}_{\pm1.4}$
CORAL-14B	$\mathbf{37.2}_{\pm3.7}$	$\mathbf{27.3}_{\pm2.9}$	$\mathbf{22.9}_{\pm3.1}$	$\mathbf{16.2}_{\pm2.4}$	$\mathbf{70.6}_{\pm5.0}$	$\mathbf{27.1}_{\pm3.2}$	$11.4_{\pm3.1}$	$6.2_{\pm3.1}$	$\mathbf{32.2}_{\pm6.2}$	$\mathbf{7.3}_{\pm3.4}$	$\mathbf{2.3}_{\pm2.0}$	$1.3_{\pm1.0}$
w Ablation1	$29.6_{\pm3.7}$	$24.9_{\pm3.0}$	$19.2_{\pm2.3}$	$12.9_{\pm2.4}$	$49.8_{\pm3.3}$	$21.2_{\pm4.2}$	$10.1_{\pm3.0}$	$3.6_{\pm1.9}$	$11.5_{\pm5.7}$	$2.5_{\pm1.8}$	$0.5_{\pm0.9}$	$0.2_{\pm0.7}$
w Ablation2	$30.6_{\pm5.2}$	$23.1_{\pm4.2}$	$20.0_{\pm3.8}$	$14.8_{\pm2.9}$	$64.1_{\pm4.1}$	$23.6_{\pm3.6}$	$10.5_{\pm1.8}$	$5.5_{\pm0.9}$	$23.3_{\pm4.1}$	$3.5_{\pm2.1}$	$1.0_{\pm1.1}$	$0.5_{\pm0.9}$
w Ablation3	$32.0_{\pm2.6}$	$25.6_{\pm1.7}$	$21.1_{\pm1.5}$	$15.1_{\pm1.1}$	$63.7_{\pm2.3}$	$24.4_{\pm2.3}$	$8.4_{\pm1.1}$	$4.5_{\pm1.7}$	$20.0_{\pm4.0}$	$3.2_{\pm2.1}$	$0.5_{\pm0.9}$	$0.2_{\pm0.7}$
w Ablation4	$32.8_{\pm4.8}$	$24.1_{\pm4.6}$	$20.9_{\pm4.1}$	$16.0_{\pm3.0}$	$59.0_{\pm3.1}$	$26.8_{\pm3.6}$	$\mathbf{13.9}_{\pm4.5}$	$\mathbf{9.1}_{\pm3.6}$	$18.0_{\pm6.7}$	$2.5_{\pm2.1}$	$0.8_{\pm1.0}$	$0.8_{\pm1.0}$
w Ablation5	$34.2_{\pm4.5}$	$25.8_{\pm3.1}$	$19.8_{\pm4.0}$	$14.5_{\pm2.8}$	$65.6_{\pm3.3}$	$22.0_{\pm2.0}$	$9.6_{\pm1.5}$	$4.9_{\pm1.5}$	$21.7_{\pm7.2}$	$4.8_{\pm3.0}$	$1.8_{\pm1.7}$	$1.0_{\pm1.1}$

Skill injection at inference is the most critical component. Ablation 1 produces the largest drop of any condition: removing skill injection at evaluation reduces Level 1 Fast_1 from 26.1% to 16.2% for the Qwen3-8B model, Level 2 from 44.8% to 20.6%, and Level 3 from 10.1% to 2.0%. The relative drop shows that the trained policy benefits substantially from skill context on difficult tasks.

SFT data diversity filtering improves coverage. Ablation 2 (unfiltered SFT data) shows a characteristic pattern: it matches or slightly exceeds CORAL on easy and intermediate metrics but substantially underperforms on Level 1 and Level 3 overall. Without diversity filtering, the SFT corpus is dominated by a small cluster of generic, frequently recurring skill patterns that are easy to learn but offer limited transferability. The results are consistent with over-reliance on these patterns: unfiltered data improves some Qwen3-8B Level 2 metrics but performs worse on most Level 1 and Level 3 metrics.

Joint multi-agent RL improves stricter-threshold performance. Ablation 3 (policy RL only) achieves competitive Fast_1 scores but collapses at stricter thresholds. For the 8B model, Level 2 Fast_2 drops from 7.9% to 3.1%; Level 3 nearly zeros out. This supports using explicit selection and summary losses alongside policy RL. Because the backbone is shared, this ablation measures the effect of removing those losses, not of freezing the two roles.

LLM-based selection improves several high-speedup metrics. Ablation 4 (policy RL + BM25 selection only) performs surprisingly well at Fast_1—Level 3 Fast_1 is 13.5% for 8B, even exceeding CORAL's 10.1%—but collapses under harder thresholds (Level 3 $\text{Fast}_{1.5}$: 0.2% vs. 2.0%). BM25

Table 3: Last-turn and best-turn results (%, mean$_{\pm\text{std}}$) for 14-turn rollouts. * marks results quoted from Dr. Kernel without reported standard deviations. Bold denotes the best mean within each block.

Model	LEVEL1				LEVEL2				LEVEL3			
	Fast$_1$	Fast$_{1.2}$	Fast$_{1.5}$	Fast$_2$	Fast$_1$	Fast$_{1.2}$	Fast$_{1.5}$	Fast$_2$	Fast$_1$	Fast$_{1.2}$	Fast$_{1.5}$	Fast$_2$
The last turn results.												
Cold-Start-8B	$16.2_{\pm 3.0}$	$13.5_{\pm 1.4}$	$10.7_{\pm 1.7}$	$8.7_{\pm 1.6}$	$17.4_{\pm 1.1}$	$9.8_{\pm 1.8}$	$6.3_{\pm 0.7}$	$3.4_{\pm 1.2}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
Dr. Kernel-8B	$20.6_{\pm 3.2}$	$17.3_{\pm 3.4}$	$13.4_{\pm 2.3}$	$9.6_{\pm 1.7}$	$34.3_{\pm 3.2}$	$13.2_{\pm 2.0}$	$6.4_{\pm 1.8}$	$1.9_{\pm 1.1}$	$4.8_{\pm 3.8}$	$1.5_{\pm 1.5}$	$0.9_{\pm 1.1}$	$0.6_{\pm 0.9}$
CORAL-8B	$39.8_{\pm 4.1}$	$23.7_{\pm 3.3}$	$19.7_{\pm 3.9}$	$14.4_{\pm 3.1}$	$60.1_{\pm 2.8}$	$30.6_{\pm 3.2}$	$16.3_{\pm 4.0}$	$\mathbf{10.6}_{\pm 2.9}$	$14.2_{\pm 5.3}$	$2.2_{\pm 2.3}$	$1.9_{\pm 1.8}$	$1.6_{\pm 1.9}$
Cold-Start-14B	$18.9_{\pm 3.1}$	$17.0_{\pm 2.8}$	$14.5_{\pm 3.0}$	$9.8_{\pm 1.9}$	$23.8_{\pm 3.3}$	$13.5_{\pm 1.6}$	$10.0_{\pm 1.6}$	$6.2_{\pm 1.5}$	$2.7_{\pm 1.9}$	$1.3_{\pm 1.6}$	$1.3_{\pm 1.6}$	$0.7_{\pm 0.9}$
Dr. Kernel-14B*	24.1	18.8	15.3	12.8	59.8	**31.6**	9.6	3.0	17.1	3.0	0.2	0.0
Dr. Kernel-14B	$34.9_{\pm 2.9}$	$28.4_{\pm 3.2}$	$22.8_{\pm 2.9}$	$15.1_{\pm 3.0}$	$\mathbf{68.6}_{\pm 2.9}$	$26.1_{\pm 5.1}$	$12.9_{\pm 3.6}$	$7.7_{\pm 3.0}$	$26.9_{\pm 2.7}$	$4.9_{\pm 1.5}$	$3.2_{\pm 1.3}$	$2.5_{\pm 0.7}$
CORAL-14B	$\mathbf{51.4}_{\pm 2.5}$	$\mathbf{32.5}_{\pm 3.3}$	$\mathbf{27.4}_{\pm 3.2}$	$\mathbf{20.3}_{\pm 2.6}$	$62.5_{\pm 3.4}$	$31.0_{\pm 1.9}$	$\mathbf{16.9}_{\pm 2.2}$	$10.1_{\pm 2.1}$	$\mathbf{27.6}_{\pm 4.8}$	$\mathbf{5.6}_{\pm 2.9}$	$\mathbf{4.2}_{\pm 1.8}$	$\mathbf{4.2}_{\pm 1.8}$
The best turn results.												
Cold-Start-8B	$32.5_{\pm 3.2}$	$27.0_{\pm 2.1}$	$23.9_{\pm 2.8}$	$16.5_{\pm 2.6}$	$35.6_{\pm 2.0}$	$25.5_{\pm 2.6}$	$19.5_{\pm 2.6}$	$10.2_{\pm 1.6}$	$3.2_{\pm 2.1}$	$1.8_{\pm 1.7}$	$1.2_{\pm 1.0}$	$0.8_{\pm 1.0}$
Dr. Kernel-8B	$40.0_{\pm 2.9}$	$36.4_{\pm 3.9}$	$33.2_{\pm 4.6}$	$21.6_{\pm 4.0}$	$60.6_{\pm 4.6}$	$39.4_{\pm 5.3}$	$29.1_{\pm 3.4}$	$13.1_{\pm 1.8}$	$15.2_{\pm 5.8}$	$4.8_{\pm 5.3}$	$3.0_{\pm 2.4}$	$1.0_{\pm 1.1}$
CORAL-8B	$65.5_{\pm 4.3}$	$42.1_{\pm 3.4}$	$35.9_{\pm 3.4}$	$22.9_{\pm 3.5}$	$90.4_{\pm 2.5}$	$55.4_{\pm 2.7}$	$38.5_{\pm 4.7}$	$23.8_{\pm 3.8}$	$50.2_{\pm 4.7}$	$7.8_{\pm 3.6}$	$3.5_{\pm 1.8}$	$2.5_{\pm 1.8}$
Cold-Start-14B	$41.1_{\pm 3.2}$	$38.0_{\pm 3.6}$	$33.5_{\pm 2.1}$	$23.1_{\pm 1.9}$	$51.4_{\pm 3.9}$	$36.6_{\pm 5.0}$	$29.0_{\pm 4.5}$	$19.5_{\pm 2.3}$	$7.2_{\pm 3.2}$	$3.5_{\pm 3.0}$	$2.8_{\pm 2.4}$	$1.5_{\pm 2.1}$
Dr. Kernel-14B*	39.3	25.1	20.4	17.6	80.9	47.8	23.6	12.5	29.8	7.3	0.5	0.0
Dr. Kernel-14B	$60.2_{\pm 2.8}$	$54.1_{\pm 4.0}$	$\mathbf{48.8}_{\pm 4.1}$	$\mathbf{35.2}_{\pm 2.2}$	$95.8_{\pm 1.4}$	$60.1_{\pm 4.2}$	$41.5_{\pm 3.4}$	$22.8_{\pm 3.1}$	$67.5_{\pm 4.0}$	$22.0_{\pm 6.0}$	$\mathbf{12.0}_{\pm 4.3}$	$6.8_{\pm 2.4}$
CORAL-14B	$\mathbf{78.5}_{\pm 3.2}$	$\mathbf{56.1}_{\pm 4.6}$	$45.9_{\pm 4.1}$	$32.9_{\pm 3.9}$	$\mathbf{98.1}_{\pm 1.5}$	$\mathbf{63.2}_{\pm 1.9}$	$\mathbf{48.5}_{\pm 2.3}$	$\mathbf{27.4}_{\pm 2.0}$	$\mathbf{68.5}_{\pm 7.0}$	$\mathbf{22.2}_{\pm 3.1}$	$10.5_{\pm 2.1}$	$\mathbf{7.5}_{\pm 2.6}$

retrieval surfaces lexically similar skills, which often helps identify any relevant optimization direction, but it cannot assess whether a skill matches the *specific structural bottleneck* of the current task. Learned selection helps on these metrics, although BM25-only selection is stronger on some Qwen3-14B Level 2 thresholds.

Skill-free RL shows a breadth–depth trade-off. Ablation 5 (no skills at all) is the most instructive: pure skill-free RL achieves Level 2 Fast$_1$ of 51.6% for Qwen3-8B—exceeding CORAL's 44.8%—but scores 2.1% on Level 2 at Fast$_2$, and 0.0% on Level 3 at Fast$_{1.5}$. This confirms that RL alone is sufficient for discovering *any* speedup: without skill conditioning, the model must find improvements by itself and can still do so frequently. Skill conditioning improves several stricter-threshold metrics, but the Fast$_1$ trade-off shows that skills do not help every setting.

4.4 Test-Time Scaling

Table 3 evaluates 14-turn rollouts with a fixed checkpoint and skill library. Last-turn evaluates the final output; best-turn selects the fastest correct output found during the rollout. This distinguishes finding a useful optimization from preserving it through subsequent revisions.

At Fast$_{1.2}$, CORAL-14B achieves 56.1% and 63.2% on Level 1 and Level 2 in best-turn evaluation, and 32.5% and 31.0% in last-turn evaluation, versus Dr. Kernel-14B's last-turn 28.4% and 26.1%. At 8B, Level 2 best-turn Fast$_{1.2}$ improves from 39.4% to 55.4%. Gains vary by task level: 14B Level 3 best-turn Fast$_{1.2}$ is similar for CORAL and Dr. Kernel, at 22.2% and 22.0%.

The best-turn/last-turn gap motivates retaining earlier candidates. For CORAL-14B, Level 3 Fast$_1$ is 68.5% for the best output but 27.6% for the final output. Longer rollouts also need not improve the final result: Level 2 last-turn Fast$_1$ falls from 70.6% at three turns to 62.5% at 14 turns, below Dr. Kernel-14B's 68.6%. Thus, the benefit of test-time scaling depends on both rollout length and candidate selection. Appendix B.1 explains retained context; Appendix B.3 gives the 12/14-turn comparison at approximately matched inference token cost.

5 Conclusion

We presented CORAL, a reinforcement learning framework for GPU kernel optimization that treats skill evolution as part of the learning problem. By jointly training selection, policy, and summary roles with execution-based skill verification, CORAL closes the loop between skill discovery and exploitation. Empirically, CORAL-14B improves over our reproduced Dr. Kernel-14B on KernelBench, including approximately 46% relative improvement on Level 3 Fast$_1$ under three-turn evaluation. Ablation studies show benefits from skill injection, joint training, learned selection, and SFT diversity filtering, while also revealing trade-offs across evaluation settings.

These results motivate further study of reusable optimization knowledge for execution-guided code generation.

AI USE STATEMENT

We used LLMs to generate experiment code (Claude Code), summarize the paper, check grammar, and generate Figure 1. We also used GPT to generate the SFT training data. This editing step did not generate new experimental measurements.

REFERENCES

Shiyi Cao, Ziming Mao, Joseph E Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, et al. Kernel-smith: A unified recipe for evolutionary kernel optimization. *arXiv preprint arXiv:2603.28342*, 2026.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, et al. davinci-env: Open swe environment synthesis at scale. *arXiv preprint arXiv:2603.13023*, 2026.

Siqi Guo, Ming Lin, and Tianbao Yang. Drtriton: Large-scale synthetic data reinforcement learning for triton kernel generation. *arXiv preprint arXiv:2603.21465*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, 2025a.

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

Yu Li, Rui Miao, Zhengling Qi, and Tian Lan. Arise: Agent reasoning with intrinsic skill evolution in hierarchical reinforcement learning. *arXiv preprint arXiv:2603.16060*, 2026.

Xuechen Liang, Meiling Tao, Yinghui Xia, Jianhui Wang, Kun Li, Yijin Wang, Yangfan He, Jingsong Yang, Tianyu Shi, Yuantao Wang, et al. Sage: Self-evolving agents with reflective and memory-augmented abilities. *Neurocomputing*, 647:130470, 2025.

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

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. *arXiv preprint arXiv:2602.05885*, 2026.

Zhengxi Lu, Zhiyuan Yao, Jinyang Wu, Chengcheng Han, Qi Gu, Xunliang Cai, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Skill0: In-context agentic reinforcement learning for skill internalization. *arXiv preprint arXiv:2604.02268*, 2026.

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.

Stephen Robertson and Hugo Zaragoza. *The probabilistic relevance framework: BM25 and beyond*, volume 4. Now Publishers Inc, 2009.

Songjun Tu, Chengdong Xu, Qichao Zhang, Yaocheng Zhang, Xiangyuan Lan, Linjing Li, and Dongbin Zhao. Dynamic dual-granularity skill bank for agentic rl. *arXiv preprint arXiv:2603.28716*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

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. *arXiv preprint arXiv:2509.07506*, 2025.

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

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, et al. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29168–29176, 2026.

A LIMITATION

We only study a single-model RL setting. Our ablations evaluate design choices within this setting; architectures with a separate frozen summarizer are outside the scope of this study. This is a feature of our setting instead of a weakness.

In some cases, the ablation result may higher than CORAL, but it's rare (< 20%) which means this is not affect the result.

We did not campare with static skill library in ablation because the skill library is empty at first.

B ADDITIONAL EVALUATION DETAILS AND RESULTS

B.1 EVALUATION PROTOCOL AND COMPARISON SCOPE

Rollout length and output selection. The main and ablation evaluations use the last output of three-turn rollouts. The test-time scaling experiment uses 14-turn rollouts and reports both the last output and the highest-speedup correct output within each rollout. The latter requires executing and comparing candidates; it is not the same as returning the final generated kernel. During long rollouts, the context retains the best five previous turns. This context limit does not reduce the total rollout length to five turns.

Frozen evaluation library. Evaluation loads the skill library stored with the model checkpoint and does not expand it using evaluation rollouts. Each training configuration has its own checkpoint library because its RL trajectory differs. Candidate-skill admission during training uses the source task, as defined in Eq. 2. This admission test checks whether a skill can recover an improvement on that task; it is not a held-out transfer test.

Evaluation Environment The software version, timing and preheating methods, accuracy tolerance, training data size and division, main optimization hyperparameters, and checkpoint selection rules are align with Dr.kernel. The only difference is we use H200 as our GPU base. The reproduction version is align with our settings.

B.2 TRAINING–EVALUATION OVERLAP CHECKS

We compare the SFT and RL training pool with all 250 KernelBench evaluation tasks. The scan finds no exact-code matches (0/250) and no exact operator-signature matches (0/250); the largest reported partial overlap, measured by Jaccard similarity between operator sets, is at most 0.67. Skills are distilled and verified on training tasks, and evaluation uses the frozen checkpoint library. These checks address exact overlap with the training pool. These checks do not rule out all semantic similarity between training and evaluation tasks.

B.3 COMPARISON AT APPROXIMATELY MATCHED INFERENCE TOKEN COST

Cost estimate. On 14-turn Level 2 rollouts, the injected skill block has a median length of approximately 1,760 tokens and is re-encoded at each turn. Under a 5:1 input-to-output token price ratio, the reported skill-block share of weighted token cost ranges from 15.5% with full history to 20.8% with a bounded initial-plus-recent context. Using an approximate per-turn cost factor of 1.18, 12 CORAL turns cost about as much as 14 Dr. Kernel turns ($12 \times 1.18 \approx 14.16$). This is an approximate inference token-cost comparison, not a measurement of equal GPU time or total training and inference cost. The cost estimate is based on Level 2 rollouts and is not separately calibrated for every task level.

Results. Tables 4–6 compare 12-turn CORAL with 14-turn Dr. Kernel and retain 14-turn CORAL as a reference. The 8B best-turn labels follow the authors' correction: the row beginning with Level 1 Fast_1 of 62.3 corresponds to 12 turns, while the row beginning with 65.5 corresponds to 14 turns. At 14B, 12-turn CORAL improves Level 2 last-turn $\text{Fast}_{1.2}$ from 26.1% to 30.1% and Fast_2 from 7.7% to 9.9% relative to 14-turn Dr. Kernel, while Fast_1 is lower (62.8% versus 68.6%). Best-turn Level 2 means improve at all four thresholds.

Table 4: Level 1 rollout-length comparison (%). Entries use $A_{\pm B}$. Twelve CORAL turns approximately match 14 Dr. Kernel turns under the reported token-cost proxy; the 14-turn CORAL rows provide context.

Model / condition	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2
Last turn				
Dr. Kernel-8B, 14 turns	$20.6_{\pm 3.2}$	$17.3_{\pm 3.4}$	$13.4_{\pm 2.3}$	$9.6_{\pm 1.7}$
CORAL-8B, 12 turns	$36.6_{\pm 4.1}$	$23.4_{\pm 2.7}$	$18.9_{\pm 1.6}$	$13.6_{\pm 1.8}$
CORAL-8B, 14 turns	$39.8_{\pm 4.1}$	$23.7_{\pm 3.3}$	$19.7_{\pm 3.9}$	$14.4_{\pm 3.1}$
Dr. Kernel-14B, 14 turns	$34.9_{\pm 2.9}$	$28.4_{\pm 3.2}$	$22.8_{\pm 2.9}$	$15.1_{\pm 3.0}$
CORAL-14B, 12 turns	$49.2_{\pm 2.6}$	$32.1_{\pm 1.7}$	$27.2_{\pm 2.1}$	$19.7_{\pm 1.9}$
CORAL-14B, 14 turns	$51.4_{\pm 2.5}$	$32.5_{\pm 3.3}$	$27.4_{\pm 3.2}$	$20.3_{\pm 2.6}$
Best turn				
Dr. Kernel-8B, 14 turns	$40.0_{\pm 2.9}$	$36.4_{\pm 3.9}$	$33.2_{\pm 4.6}$	$21.6_{\pm 4.0}$
CORAL-8B, 12 turns	$62.3_{\pm 3.2}$	$40.5_{\pm 2.8}$	$34.0_{\pm 3.4}$	$22.0_{\pm 1.8}$
CORAL-8B, 14 turns	$65.5_{\pm 4.3}$	$42.1_{\pm 3.4}$	$35.9_{\pm 3.4}$	$22.9_{\pm 3.5}$
Dr. Kernel-14B, 14 turns	$60.2_{\pm 2.8}$	$54.1_{\pm 4.0}$	$48.8_{\pm 4.1}$	$35.2_{\pm 2.2}$
CORAL-14B, 12 turns	$76.5_{\pm 3.2}$	$54.2_{\pm 2.5}$	$44.5_{\pm 3.7}$	$31.2_{\pm 1.6}$
CORAL-14B, 14 turns	$78.5_{\pm 3.2}$	$56.1_{\pm 4.6}$	$45.9_{\pm 4.1}$	$32.9_{\pm 3.9}$

Table 5: Level 2 rollout-length comparison (%). Entries use $A_{\pm B}$. Twelve CORAL turns approximately match 14 Dr. Kernel turns under the reported token-cost proxy; the 14-turn CORAL rows provide context.

Model / condition	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2
Last turn				
Dr. Kernel-8B, 14 turns	$34.3_{\pm 3.2}$	$13.2_{\pm 2.0}$	$6.4_{\pm 1.8}$	$1.9_{\pm 1.1}$
CORAL-8B, 12 turns	$60.6_{\pm 4.3}$	$31.3_{\pm 3.8}$	$16.5_{\pm 3.2}$	$11.2_{\pm 2.4}$
CORAL-8B, 14 turns	$60.1_{\pm 2.8}$	$30.6_{\pm 3.2}$	$16.3_{\pm 4.0}$	$10.6_{\pm 2.9}$
Dr. Kernel-14B, 14 turns	$68.6_{\pm 2.9}$	$26.1_{\pm 5.1}$	$12.9_{\pm 3.6}$	$7.7_{\pm 3.0}$
CORAL-14B, 12 turns	$62.8_{\pm 2.4}$	$30.1_{\pm 1.9}$	$16.8_{\pm 1.7}$	$9.9_{\pm 1.6}$
CORAL-14B, 14 turns	$62.5_{\pm 3.4}$	$31.0_{\pm 1.9}$	$16.9_{\pm 2.2}$	$10.1_{\pm 2.1}$
Best turn				
Dr. Kernel-8B, 14 turns	$60.6_{\pm 4.6}$	$39.4_{\pm 5.3}$	$29.1_{\pm 3.4}$	$13.1_{\pm 1.8}$
CORAL-8B, 12 turns	$89.4_{\pm 2.8}$	$53.6_{\pm 3.5}$	$36.0_{\pm 3.0}$	$22.2_{\pm 2.7}$
CORAL-8B, 14 turns	$90.4_{\pm 2.5}$	$55.4_{\pm 2.7}$	$38.5_{\pm 4.7}$	$23.8_{\pm 3.8}$
Dr. Kernel-14B, 14 turns	$95.8_{\pm 1.4}$	$60.1_{\pm 4.2}$	$41.5_{\pm 3.4}$	$22.8_{\pm 3.1}$
CORAL-14B, 12 turns	$98.1_{\pm 1.5}$	$61.6_{\pm 3.0}$	$45.8_{\pm 1.4}$	$25.5_{\pm 1.6}$
CORAL-14B, 14 turns	$98.1_{\pm 1.5}$	$63.2_{\pm 1.9}$	$48.5_{\pm 2.3}$	$27.4_{\pm 2.0}$

Table 6: Level 3 rollout-length comparison (%). Entries use $A_{\pm B}$. Twelve CORAL turns approximately match 14 Dr. Kernel turns under the reported token-cost proxy; the 14-turn CORAL rows provide context.

Model / condition	Fast_1	$\text{Fast}_{1.2}$	$\text{Fast}_{1.5}$	Fast_2
Last turn				
Dr. Kernel-8B, 14 turns	$4.8_{\pm 3.8}$	$1.5_{\pm 1.5}$	$0.9_{\pm 1.1}$	$0.6_{\pm 0.9}$
CORAL-8B, 12 turns	$13.4_{\pm 4.0}$	$2.5_{\pm 2.1}$	$2.5_{\pm 2.3}$	$2.2_{\pm 0.9}$
CORAL-8B, 14 turns	$14.2_{\pm 5.3}$	$2.2_{\pm 2.3}$	$1.9_{\pm 1.8}$	$1.6_{\pm 1.9}$
Dr. Kernel-14B, 14 turns	$26.9_{\pm 2.7}$	$4.9_{\pm 1.5}$	$3.2_{\pm 1.3}$	$2.5_{\pm 0.7}$
CORAL-14B, 12 turns	$29.0_{\pm 4.1}$	$7.9_{\pm 2.6}$	$6.1_{\pm 1.8}$	$5.1_{\pm 1.5}$
CORAL-14B, 14 turns	$27.6_{\pm 4.8}$	$5.6_{\pm 2.9}$	$4.2_{\pm 1.8}$	$4.2_{\pm 1.8}$
Best turn				
Dr. Kernel-8B, 14 turns	$15.2_{\pm 5.8}$	$4.8_{\pm 5.3}$	$3.0_{\pm 2.4}$	$1.0_{\pm 1.1}$
CORAL-8B, 12 turns	$49.0_{\pm 3.7}$	$7.2_{\pm 2.1}$	$3.5_{\pm 1.8}$	$2.5_{\pm 1.0}$
CORAL-8B, 14 turns	$50.2_{\pm 4.7}$	$7.8_{\pm 3.6}$	$3.5_{\pm 1.8}$	$2.5_{\pm 1.8}$
Dr. Kernel-14B, 14 turns	$67.5_{\pm 4.0}$	$22.0_{\pm 6.0}$	$12.0_{\pm 4.3}$	$6.8_{\pm 2.4}$
CORAL-14B, 12 turns	$68.2_{\pm 7.2}$	$21.5_{\pm 2.1}$	$10.2_{\pm 2.6}$	$7.0_{\pm 2.0}$
CORAL-14B, 14 turns	$68.5_{\pm 7.0}$	$22.2_{\pm 3.1}$	$10.5_{\pm 2.1}$	$7.5_{\pm 2.6}$

C INTRODUCTION ABOUT TRLOO

TRLOO Our policy agent training setup follows DR. KERNEL, adopting a unified reinforcement learning objective that combines **turn-level REINFORCE Leave-One-Out (TRLOO)** with mismatch control and profiling-aware optimization. For each trajectory i at turn t, a bottleneck-aware reward is defined as

$$R_{i,t} = C(y_{i,t}) \cdot \big(1 + \text{speedup}_{i,t} + \text{PR}_{i,t}\big), \quad \text{PR}_{i,t} = \frac{T_{\text{generated}}}{T_{\text{total}}}, \tag{7}$$

where $C(\cdot)$ denotes correctness, and PR measures the fraction of end-to-end runtime covered by generated kernels.

The return is computed as $G_{i,t} = \sum_{t' \geq t} \gamma^{t'-t} R_{i,t'}$, and advantages are estimated using TRLOO:

$$A_{i,t} = G_{i,t} - \frac{1}{N_t - 1} \sum_{j \neq i} G_{j,t}. \tag{8}$$

To stabilize optimization, **MRS** filters samples with large importance ratio deviation:

$$w_i = \exp\left(\frac{1}{|\mathcal{T}_i|} \sum_k \log \frac{\pi_{\text{train}}(a_k \mid s_k)}{\pi_{\text{rollout}}(a_k \mid s_k)} \right). \tag{9}$$

Finally, **PRS** prioritizes optimization on kernels that impact end-to-end performance:

$$p_{i,t} = \text{clip}\left(\frac{\text{PR}_{i,t} - \tau}{s}, 0, 1 \right), \tag{10}$$

and trajectories are sampled proportionally to $p_{i,t}$.

D INTRODUCTION ABOUT KERNEL OPTIMIZATION.

Optimizing CUDA kernels is inherently a *continuous* and iterative process, where a correct implementation can be progressively improved through a sequence of transformations (e.g., parallelization and fusion), each yielding incremental performance gains. As a result, the objective is not binary, but *graded* and accumulative across multiple steps, with partial improvements providing meaningful training signals.

In practice, this setting introduces several challenges. The reward signal is often sparse and delayed, as substantial speedups typically require coordinated modifications over multiple turns. At the same time, the optimization space contains many degenerate solutions, where the model produces syntactically valid but ineffective changes ("lazy optimization"), such as preserving the original implementation or applying transformations that do not impact end-to-end runtime. Moreover, performance gains are tightly coupled to system-level bottlenecks, making it insufficient to optimize local code patterns without considering their global runtime contribution.

E TRAINING DYNAMICS

Figure 3 shows the validation $\text{Fast}_{1.2}$ curves during RL training on the 14B model. The curves reveal that the advantage of CORAL is not only reflected in the final checkpoint, but also in the stability and sustainability of the training process.

Skill-conditioned CORAL maintains the strongest long-term performance. CORAL with skill injection consistently stays in the top performance range throughout training and reaches the highest peak among all curves. Compared with Dr. Kernel-14B, which improves in the early stage but later plateaus and gradually declines, CORAL maintains a higher validation level after the middle stage of training. This indicates that the evolving skill library provides a persistent source of useful optimization knowledge, rather than only improving the initial exploration stage.

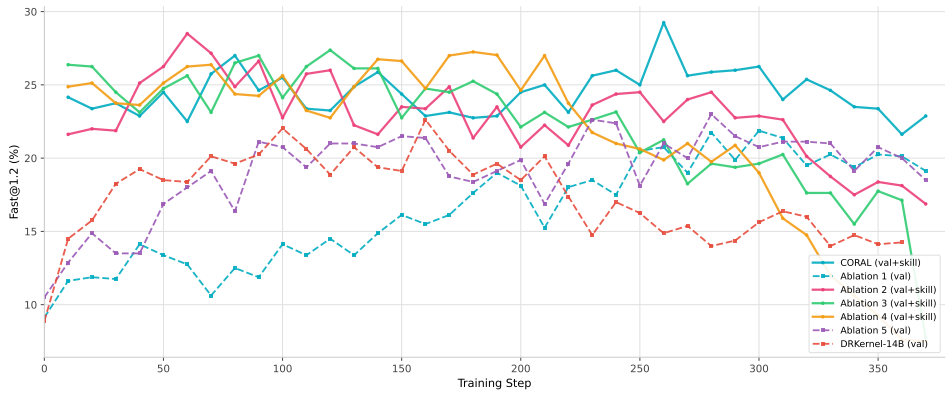

Figure 3: Validation $\text{Fast}_{1.2}$ training curves for CORAL and selected ablations on the 14B model. The comparison illustrates how skill-conditioned evaluation, joint multi-agent RL, selection training, and data filtering affect training stability and long-term improvement.

Skill-free evaluation improves but remains lower than skill-conditioned evaluation. Ablation 1 evaluates the same training pipeline without injecting skills at inference time. Its curve increases steadily from the early stage, showing that the policy does internalize part of the optimization ability learned during skill-conditioned RL. However, it remains below the full CORAL curve for most training steps, suggesting that external skills continue to provide useful task-specific guidance even after the model has learned from them. Thus, skill evolution improves both the internal policy and the inference-time context, but the two are not redundant.

Static or weakly trained skill mechanisms become unstable over time. Ablation 3 and Ablation 4 are competitive in the early and middle stages, but their performance degrades later. This pattern supports the main motivation of CORAL: as the policy changes, the useful skill frontier also changes. If the selection and summary agents are not jointly optimized, or if LLM-based selection is replaced by BM25-only retrieval, the retrieved skills can gradually become mismatched with the policy's current capability and the task's structural bottleneck. The resulting skills may still help produce shallow speedups, but they are less reliable for sustaining high-quality improvements.

Unfiltered skill data leads to weaker long-term generalization. Ablation 2 also shows a gradual decline after an initially competitive stage. This is consistent with the effect observed in the ablation table: without diversity filtering, the model is more likely to overfit to frequent and generic skill patterns. Such skills can be helpful early in training, but they provide limited additional information once the policy has already learned common optimizations. As training proceeds, the lack of diverse and specific skills makes further improvement harder.

Overall trend. The training dynamics show that CORAL's gains come from the interaction between skill utilization, skill selection, and skill summarization. Pure RL and static retrieval can improve the policy in the short term, but they either plateau or become unstable as training continues. In contrast, joint skill-policy co-evolution keeps the skill library aligned with the model's evolving capability frontier, leading to stronger and more stable validation performance.

F Training Costs

Environment interaction accounts for most of the training time. The measured cost is 1,734 seconds per step for CORAL and 1,466 seconds for the skill-free method, an increase of approximately 18.3%. Evaluating CORAL both with and without skill context takes approximately twice the time of evaluating only the skill-free setting.

Task A — Reference Architecture

```
class Model(nn.Module):
  def forward(self, x):
    x = self.conv(x) # Conv3d <-- bottleneck
    x = self.norm(x) # GroupNorm
    x = torch.min(x, torch.tensor(min_value, device=x.device))
    x = torch.clamp(x, min=min_value, max=max_value)
    x = self.dropout(x)
    return x
```

CORAL + Skill (Turn 1, 2.0×)

```
def forward(self, x):
 # Skill:
      hoist_invariant_scalars
 # min_value==0.0 => output = 0
 if self.min_value == 0.0:
  B,_,d,h,w = x.shape
  # Skip Conv3D + GroupNorm!
  out = torch.empty(
   (B, C_out,
    d-3+1, h-3+1, w-3+1),
   device=x.device, dtype=x.dtype
        )
  n = out.numel()
  zero_kernel[
   (cdiv(n, 1024),)](
   out, n, BLOCK=1024)
  return out # 2.0×
 # fallback (other min_value)
 x = self.conv(x)
 x = self.norm(x)
 x = torch.clamp(x, 0., 1.)
 return self.dropout(x)
```

Dr. Kernel (Turn 1 fails; Turn 2, 1.07×)

```
def forward(self, x):
 # Conv3D still runs
 x = self.conv(x)
 # GroupNorm still runs
 x = self.norm(x)
 # Partial: fills with
      constant
 c = min(self.min_value,
       self.max_value)
 y = torch.empty_like(x)
 fill_const_kernel[
  (cdiv(n, 4096),)](
  y, n, c, BLOCK=4096)
 x = self.dropout(y)
 return x # 1.07×
```

CORAL w/o Skill (Turn 1, ≤1.07× / fail)

```
def forward(self, x):
 x = self.conv(x)
 x = x.contiguous()
 # Custom Triton
      GroupNorm:
 # complex + error-prone
 y = triton_group_norm(
  x, weight, bias,
  num_groups, eps)
 # Triton min+clamp
      fusion
 y = triton_min_clamp(
  y, self.min_value,
  self.max_value)
 y = self.dropout(y)
 return y
 # avg 0.79× (many
      failures)
```

Figure 4: **Task A: three-way comparison.** In the illustrated rollout, CORAL with skill (**left**) recognizes that the composition of `Min` and `Clamp` under the task's `min_value=0.0` setting produces zero, allowing it to skip `Conv3D` and `GroupNorm` and achieve 2.0× at Turn 1. Dr. Kernel (**middle**) identifies constant filling at Turn 2 but retains the upstream computation, achieving 1.07×. CORAL without skill (**right**) attempts a custom Triton GroupNorm; across the sampled rollouts, it has frequent correctness failures and an average best speedup of 0.79×.

G Case Study

We examine how optimization experience acquired on training tasks transfers to two Level-2 KernelBench tasks. For each case, we trace the source task, the skill distilled from its optimization trajectory, and the implementation produced on the target task. Skills are generated during training, and the checkpoint library is frozen during evaluation. The two cases illustrate complementary forms of transfer: using invariant parameters to identify semantic shortcuts, and reusing an implementation pattern to improve the reliability of kernel generation.

Task A: `Conv3D + GroupNorm + Min + Clamp + Dropout`. Task A illustrates how a skill can transfer an optimization reasoning strategy between different computation graphs. The target contains expensive `Conv3D` and `GroupNorm` operations followed by scalar-parameterized post-processing. The key opportunity is to determine whether these upstream operations are needed before attempting to accelerate their implementations.

Source task and skill abstraction. The skill `hoist_invariant_scalars_out_of_kernel_path` was generated at training step 283 from `cuda_llm_248406` (Figure 8). Its source architecture applies `Bucketize`, `LogSumExp`, and `Clamp`, combining input-dependent computation with

fixed scalar bounds. The distilled skill directs the policy to inspect invariant parameters, move their handling outside the hot path, and consider specialized execution paths (Figure 6). This supplies a reusable optimization strategy: examine what fixed parameters imply for the computation before optimizing individual operators.

Application to the target task. The source and target use different upstream operators, but both expose fixed bounds in their post-processing. On Task A, the retrieved skill directs attention to `min_value=0.0`. The policy then applies this guidance to the target's specific operator composition. For a finite intermediate value z and the target's nonnegative upper bound M,

$$\mathrm{clamp}\big(\min(z, 0),\, 0,\, M\big) = 0.$$

The subsequent dropout also preserves zero. Consequently, the final output is independent of the values computed by `Conv3D` and `GroupNorm`. The policy can determine the output shape from the architecture and directly launch a zero-fill kernel, eliminating the expensive upstream operations.

The transferable knowledge is the strategy of inspecting invariant parameters for specialized execution. The target-specific step is establishing that `Min` followed by `Clamp` removes the output's dependence on upstream values. Thus, applying the skill requires reasoning about the new computation graph: the scalar value alone does not establish the shortcut.

Observed benefit. Figure 4 shows how this guidance changes the optimization pursued by the policy. The skill-conditioned rollout removes the upstream computation at Turn 1. Dr. Kernel identifies constant filling at Turn 2 but still executes the convolution and normalization, while CORAL without skill attempts a more error-prone custom normalization kernel. Across eight samples, CORAL with skill achieves an average best speedup of 1.77×, compared with 1.05× for Dr. Kernel and 0.79× for CORAL without skill (Table 7). This case illustrates how a skill can transfer a reasoning strategy that helps the policy discover a larger optimization opportunity in a different task.

Table 7: Task A results (8 samples each).

Method	avg best	% ≥2×	Turn-1 pass
CORAL + skill (step 460)	**1.77**	**62.5%**	**62.5%**
Dr. Kernel	1.05	0%	0%
CORAL w/o skill	0.79	12.5%	0%

How Retrieved Skills Evolve Across Training. Figures 5 and 6 show the skill contexts retrieved for Task A at steps 260 and 460. At step 260, the three skills emphasize retaining library-optimized heavy operators, identifying worthwhile optimization targets, and fusing contiguous pointwise tails. Their source tasks are shown in Figure 7. These skills offer useful implementation guidance, but the retrieved context does not explicitly direct the policy to inspect scalar parameters for semantic shortcuts.

At step 460, the retrieved context retains tail-fusion guidance and also includes the scalar-hoisting skill generated at step 283. The later context therefore provides an additional optimization direction: checking whether fixed parameters make part of the computation unnecessary. On Task A, this direction leads to the zero-output identity described above. The fraction of samples reaching at least 2× speedup is 4/8 at step 260 and 5/8 at step 460.

This sequence illustrates how knowledge discovered on training tasks becomes available for later target-task optimization. The library preserves earlier implementation patterns while incorporating a new reasoning strategy, and the retrieved context brings these complementary forms of knowledge into the policy's optimization process.

Task B: `ConvTranspose3D + Add + HardSwish`. Task B (`ConvTranspose3D` + elementwise add + $x \cdot \mathrm{hardswish}(x)$, batch 128, channels 32→64) illustrates transfer of a reusable implementation pattern. Here, the main benefit is more reliable generation of correct multi-input fused kernels.

Source task and skill abstraction. The skill `fuse_only_contiguous_pointwise_tails` was generated at step 176 from `cuda_llm_492345`, whose architecture is `Linear + Maximum`

Skills Retrieved at Step 260 (Task A)

```
[1] name: contiguous_tail_kernelization_around_library_core
description: Keep GEMM/conv in vendor libraries and fuse only the flat contiguous tail.
tags: [vendor_kernel, contiguous_tail, fusion]
---
## Motivation
Replacing GEMM/conv with Triton is a common performance cliff. Leave the heavy core in
  cuBLAS/cuDNN and fuse the contiguous tail.
## Key Idea
1. Run conv/matmul using PyTorch/library kernels.
2. Ensure the output is contiguous.
3. Flatten to numel() and launch masked 1D kernel.
4. Fuse as many elementwise ops as possible.
Pitfalls: .contiguous() erases gains if tail tiny.

[2] name: fuse_only_contiguous_pointwise_tails
description: Fuse memory-bound pointwise epilogues around vendor kernels; don't replace
  heavy ops.
tags: [fusion, pointwise, memory_bound, triton]
---
## Motivation
Custom Triton underperforms when it replaces library-optimized GEMM/conv outright. The
  highest-probability win is to keep those heavy ops and fuse the contiguous pointwise
  epilogue.
## Key Idea
1. Ensure tail input is contiguous.
2. Flatten to numel() and launch masked 1D grid.
3. Fuse as many pointwise ops as possible.
4. Avoid extra temporaries unless profiling justifies.
Pitfalls: use masks for non-power-of-two sizes.

[3] name: hotspot_aware_triton_selection
description: Kernelize only bandwidth-bound tails that are actually hot.
tags: [hotspot_analysis, fusion_strategy, triton]
---
## Motivation
Custom Triton often underperforms when it targets only a tiny fraction of end-to-end
  time. Before writing a kernel, identify the true hotspots.
## Key Idea
Profile first; keep heavyweight ops (matmul, conv) unless your kernel fuses enough
  surrounding work to amortize the replacement cost. Target contiguous pointwise tails
  where Triton removes overhead.
```

Figure 5: Skills retrieved for Task A at training step 260. The context emphasizes retaining heavy library operators and optimizing their contiguous tails.

`+ Softplus` (Figure 7). This source task has a library-optimized compute-heavy operator followed by pointwise transformations. The skill abstracts this structure into an implementation recipe: retain the heavy library operator, make its output contiguous when needed, flatten the pointwise work to `numel()`, and fuse it in a masked one-dimensional Triton kernel (Figure 5).

Application to the target task. Task B replaces the source's linear layer with a three-dimensional transposed convolution and uses different pointwise operations. Nevertheless, it preserves the structure addressed by the skill: a heavy library operator followed by a pointwise tail. The policy therefore keeps `ConvTranspose3D` in the library and applies the flat-1D fusion pattern to the target's addition and activation.

In the generated implementation (Figure 9), the policy makes the convolution output and additional input contiguous, then launches a fused kernel over the output elements. This organization avoids explicit five-dimensional stride arithmetic in the illustrated target setting. The skill transfers because its indexing and fusion recipe depends on the pointwise tail's access pattern, while the policy supplies the target-specific arithmetic. It can therefore be reused even when the upstream operator and the tensor rank differ from those of the source task.

Observed benefit. Table 8 shows that CORAL with skill produces eight correct samples out of eight, compared with three for CORAL without skill and five for Dr. Kernel. Among correct samples, speedups are similar: 1.24×, 1.22×, and 1.23×, respectively. The main gain is therefore the reliability

Skills Retrieved at Step 460 (Task A)

```
[1] name: fuse_only_contiguous_pointwise_tails
  (same as [2] at step 260)
[2] name: flat_masked_tail_kernel_for_library_outputs
description: After keeping conv in vendor libs, flatten and run a masked 1D Triton tail
  kernel.
tags: [vendor_kernel, pointwise_fusion, triton]
---
## Motivation
Replacing cuBLAS/cuDNN heavy ops with Triton often loses. Keep those cores and fuse only
  the contiguous pointwise tail in a flat 1D kernel.
## Key Idea
1. Run conv/matmul in PyTorch.
2. Make the output contiguous if needed.
3. Flatten to numel(), launch 1D grid with mask
   (offs < n).
4. Fuse all pointwise ops before the final store.
Pitfalls: .contiguous() can cost more than the tail.

[3] name:  hoist_invariant_scalars_out_of_kernel_path ← NEW
description:  Reduce overhead by hoisting invariant scalar/tensor params out of the hot
  kernel path.
tags:  [scalar_hoisting, host_overhead, fast_path]
--
## Motivation
A surprising amount of overhead comes from re-fetching tiny invariant scalars every call:
clamp bounds, strides, shape constants.  On small workloads this can dominate runtime and
  introduce hidden syncs from .item() or shape inspection.
## Key Idea
1.  Inspect constant params once in __init__.
2.  Convert scalar tensors to Python floats.
3.  Combine with a specialised fast path:
 if min_value == 0.0:  # hoist the scalar
 skip_entire_pipeline() # 2.0x speedup
Pitfalls:  hoisting is wrong if value can change.
```

Figure 6: The scalar-hoisting skill, generated at step 283 and highlighted in green, adds guidance on inspecting invariant parameters and considering specialized execution paths.

Table 8: Task B results (8 samples). †Among correct samples only.

Method	avg best	Correct	Speedup†
CORAL + skill (step 460)	**1.235**	**8/8**	**1.24**×
Dr. Kernel	0.772	5/8	1.23×
CORAL w/o skill	0.457	3/8	1.22×

with which the policy produces an effective implementation. The transferred skill provides a concrete way to organize the target kernel, reducing opportunities for indexing errors while preserving useful fusion.

Summary. The two cases illustrate how skills connect different tasks through shared optimization conditions. In Task A, a skill originating from scalar-parameterized post-processing directs the policy to inspect fixed bounds; the policy then derives a target-specific identity that removes expensive upstream computation. In Task B, a skill originating from a linear layer's pointwise tail provides an indexing and fusion pattern that the policy adapts to a transposed-convolution task. In both cases, the skill supplies reusable guidance and the policy completes the reasoning or implementation required by the target. These examples provide concrete evidence of how training-task experience can support optimization on different evaluation tasks, through both semantic simplification and more reliable kernel construction.

Step 260 [1] — `contiguous_tail_kernelization_around_library_core` (Source: `cuda_llm_417838`; generated at step 240)

```
class Model(nn.Module):
   def forward(self, x):
      x = F.log_softmax(self.fc(x), dim=1)
      x = torch.logical_or(x > 0, x < 1)
      return x
```

Step 260 [2] / Step 460 [1] — `fuse_only_contiguous_pointwise_tails` (Source: `cuda_llm_492345`; generated at step 176)

```
class Model(nn.Module):
   def forward(self, x):
      x = self.linear1(x)
      x = torch.maximum(x, torch.zeros_like(x))
      x = self.softplus(x)
      return x
```

Step 260 [3] — `hotspot_aware_triton_selection` (Source: `cuda_llm_622578`; generated at step 141)

```
class Model(nn.Module):
   def forward(self, x, target):
      x = torch.acos(x)
      x = self.avgpool(x)
      mask = torch.isnan(x)
      x = torch.where(mask, torch.tensor(0.0, device=x.device), x)
      x = self.sync_batchnorm(x)
      x = torch.flip(x, [2, 3])
      loss = F.mse_loss(x, target)
      return loss
```

Figure 7: Training-task architectures whose optimization trajectories produced the three skills retrieved for Task A at step 260. The second skill was also retrieved at step 460 and is used in the Task B transfer analysis. Constructors and input-generation functions are omitted.

Step 460 [2] — `flat_masked_tail_kernel_for_library_outputs` (Source: `cuda_llm_216025`; generated at step 279)

```
class Model(nn.Module):
   def forward(self, x, mat1, mat2, vec):
      x = self.conv3d(x)
      x = x.mean(dim=(2, 3))
      x = self.adaptive_avgpool1d(x).squeeze(1)
```

```
      x = torch.addmm(vec, mat1, mat2)
      x = torch.lt(x, 0.5).float()
      return x
```

Step 460 [3] — `hoist_invariant_scalars_out_of_kernel_path` (Source: `cuda_llm_248406`; generated at step 283)

```
class Model(torch.nn.Module):
   def forward(self, x):
      bucketized = torch.bucketize(x, self.values, right=False)
      logsumexp = torch.logsumexp(bucketized, self.dim, self.keepdim)
      clipped = torch.clamp(logsumexp, self.min_value, self.max_value)
```

```
      return clipped
```

Figure 8: Training-task architectures whose optimization trajectories produced the other two skills retrieved for Task A at step 460. The scalar-hoisting skill originates from a `Bucketize + LogSumExp + Clamp` task and is subsequently applied to Task A's different computation graph. Step 460 [1] shares its source task with step 260 [2], shown in Figure 7. Constructors and input-generation functions are omitted.

CORAL + Skill (8/8 correct, 1.24×)

```
def forward(self, x, add_input):
 y = self.conv_transpose(x)
 # Skill:  flatten to 1D, masked grid
 y = y.contiguous()
 add_input = add_input.contiguous()
 out = torch.empty_like(y)
 n = y.numel()
 fused_add_hardswish[
  (cdiv(n, BLOCK),)](
  y, add_input, out,
  n, BLOCK=1024)
 return out # 8/8 correct; avg 1.24×
```

Dr. Kernel (5/8 correct, 3/8 fail)

```
def forward(self, x, add_input):
 z = self.conv_transpose(x)
 # Explicit 5-D stride indexing
 # => shape mismatch bugs
 sYn,sYc,sYd,sYh,sYw = z.stride()
 sAn,sAc,sAd,sAh,sAw = add_input.stride()
 strided_kernel[grid](
  z, add_input, out,
  B, C, D, H, W,
  sYn,sYc,...,sAn,sAc,...)
 return out
 # 3 of 8 samples incorrect
```

Figure 9: **Task B: implementation-pattern transfer.** CORAL with skill (left) applies the flat-1D masked kernel pattern distilled from a `Linear + Maximum + Softplus` training task to the pointwise tail of a `ConvTranspose3D` target. All eight samples are correct, with an average speedup of approximately 1.24×. Dr. Kernel (right) uses explicit 5-D stride indexing, with shape-mismatch failures in three of eight samples.

H SYSTEM PROMPT

Summary Agent system prompt (GPT series)

```
You are an expert CUDA/Triton kernel optimization engineer.
You have observed a successful multi-turn optimization trajectory.
Your task: extract GENERAL, REUSABLE optimization principles as skills.

Workflow:
1. Optionally call read_skill_files to inspect existing skills and avoid
↪  duplicates.
2. Call update_skill_library with at most {max_skills} new skill(s).
   Each skill body must contain: ## Motivation, ## Key Idea, ## Example (with
   ↪  code).
3. Your turn ends automatically after update_skill_library is called.

Rules:
- Skills must be GENERAL (applicable beyond this specific task).
- Do NOT add task-specific hacks or solutions.
- Respond in English only.
```

Summary Agent user prompt (GPT series)

```
## Task

{task_formatted}

## Existing Skills in Library

```
{skill_library.get_file_tree()}
```

Turn 1 Result (speedup: {turn1_speedup_str}x)

```python
{turn1_code[:2000]}
```

Best Turn Result (speedup: {best_speedup_str}x)

```python
{best_code[:3000]}
```

The policy agent improved from turn 1 to its best turn.
Your goal: propose at most {max_skills} new skill(s) that capture the key
↪ optimization
insights from this trajectory.
Call read_skill_files first if needed, then call update_skill_library.
```

Select Agent system prompt (GPT series)

```
You are an expert CUDA/Triton kernel optimization engineer.

You will be shown:
1. A kernel optimization task (original PyTorch code + performance target).
2. A numbered list of candidate optimization skills from the skill library.
   Each entry shows only the skill name, description, tags, and scope --
   NOT the full content.

Your job: identify the top 3 skills most likely to help solve THIS specific
↪  task.

## Selection criteria
- Relevance: the technique directly applies to the operator/pattern in the
↪  task.
- Impact: likely to produce a meaningful speedup for this workload.
- Complementarity: prefer a diverse set that covers different aspects
  (e.g. one memory, one compute, one correctness pitfall).

## What to avoid
```

```
- Selecting skills just because they sound impressive.
- Selecting skills about bypassing CUDA/Triton (e.g. "use torch.compile").
- Selecting more than 3 skills.

When calling select_skills, first fill in `think`: briefly explain what the
key bottleneck of this task is and why the chosen skills address it.
Then fill in `selected_skills` with the names and one-sentence reasons.

Call select_skills exactly once. Respond in English only."""
```

Select Agent user prompt (GPT series)

```
## Task

{task_text}

---

## Candidate Skills (from the library)

{candidates_block}

---

Now call select_skills to choose the top 3 most relevant skills for this
↪  task.
```

Policy Agent system prompt (GPT series)

```
You are looking at this PyTorch code and thinking it could be optimized with
↪  Triton.

Here's the PyTorch code:

```python
{original_python_code}
```

Here is the skill that might be relevant to the code:
{skill_content}

You need to create a Triton version with the entry point called `ModelNew`.

Please firstly analyze this code and think hard how you can optimize it,
considering the skills you have.
You also should think about the whether the skill is useful to improve the
↪ code and how to
use it properly if it is useful.

**Please output and show your thinking, plan, analysis etc. in a markdown
↪ format,
before your coding, which should be as more as possible.**
Optimize the architecture named Model"""
```

Policy Agent feedback prompt (GPT series)

```
Server feedback from the evaluation environment for your last implementation:
{feedback_json}

Based on the above server feedback, please improve the implementation:
- If there are errors/crashes/illegal memory access: identify the root cause
↪  and fix it;
prevent recurrence.
- If there is no speedup or performance regresses: optimize the bottlenecks
↪  to achieve a
clear speedup.
- If there is already a speedup: further improve performance without
↪  degrading correctness.
```

```
- If the suggested skill has already been fully applied and speedup is no
↪  longer improving
or even regresses, the remaining gains likely require going beyond it
↪  --consider
mathematical equivalences shortcut, algorithmic redesign, or any insight the
↪  skill does not
cover.
- Please output your thinking, plan, analysis, and the final code.
```

Summary Agent system prompt (SFT series)

```
You are an expert CUDA/Triton kernel optimization engineer.
    You have observed a successful multi-turn kernel optimization trajectory
    ↪  where
    a model iteratively improved a custom GPU kernel.

    Your task: extract at most __max_skills__ GENERAL, REUSABLE optimization
    ↪  skills
    from this trajectory.

    ## What makes a GOOD skill

    - Broadly applicable: the technique works across many operators/models,
    not just this one task.
    - Actionable: explains *how* to apply the technique, not just *that* it
    ↪  helps.
    - Non-obvious: not trivially covered by basic CUDA/Triton documentation.
    - Each skill body must contain these three sections:
        ## Motivation   -- why this technique matters and when to use it
        ## Key Idea     -- the core mechanism and how to implement it
        ## Example      -- a short self-contained code snippet illustrating
        ↪  the idea

    ## What to AVOID (your output will be rejected if any skill does this)

    - Skills that bypass CUDA/Triton entirely (e.g. "use torch.compile",
    "call cuBLAS directly", "avoid writing kernels", "fall back to PyTorch
    ↪  operators").
    All skills MUST be about
    writing  or structuring actual kernels. And such kernels are correctly
    ↪  implemented.
    - Task-specific hacks (e.g. "for this exact shape, hard-code block size
    ↪  256").
    - Observations masquerading as skills (e.g. "the model improved by fusing
    ↪  ops" with
    no guidance).
    - Trivial or obvious advice (e.g. "use shared memory", "reduce memory
    ↪  bandwidth").
    - Skills that encourage the optimizer to skip computation or reduce
    ↪  precision without
      explicit justification (e.g. "drop the softmax", "use fp8 blindly").
    - Duplicate skills: if two techniques are essentially the same, merge
    ↪  them.

    ## Common pitfalls to document (in addition to improvements)

    When extracting skills, also consider:
    - Numerical correctness traps: reduction order, fp16/bf16 overflow,
    ↪  non-associativity.
    - Race conditions: missing __syncthreads(), warp divergence from early
    ↪  exits.
    - Indexing bugs that appear only at non-power-of-two sizes.
    - Performance cliffs: bank conflicts, uncoalesced access patterns,
    ↪  occupancy limits.
    - Triton-specific pitfalls: incorrect tl.constexpr usage, wrong mask
    ↪  shapes, autotune
    overhead.

    When calling update_skill_library, first fill in the `think` field:
    ↪  briefly identify
    the key problem or bottleneck this trajectory reveals, and what kind of
    ↪  skill
    would address it. Then fill in `new_skills` with at most __max_skills__
    ↪  skills
    derived from that analysis.

    Call update_skill_library exactly once. Respond in English only.
```

Summary Agent user prompt (SFT series)

```
## Task

    __task_messages_formatted__

    ## Skills Already Given to the Policy Agent

    __injected_skill_content__

    ## Turn 1 Result (speedup: __turn1_speedup__x)

    ```python
    __turn1_code__
    ```

    ## Best Turn Result (speedup: __best_turn_speedup__x)

    ```python
    __best_turn_code__
    ```

    Now call update_skill_library with at most __max_skills__ new, general,
    ↪ reusable skills
    extracted from the trajectory above. Focus on techniques that would help
    ↪ achieve
    similar improvements on OTHER tasks, not just this one.
```

Select Agent system prompt (SFT series)

```
You are an expert CUDA/Triton kernel optimization engineer.

    You will be shown:
    1. A kernel optimization task (original PyTorch code + performance
    ↪  target).
    2. A numbered list of candidate optimization skills from the skill
    ↪  library.
       Each entry shows only the skill name, description, tags, and scope --
       NOT the full content.

    Your job: identify the top __top_k_select__ skills most likely to help
    ↪  solve
    THIS specific task.

    ## Selection criteria
    - Relevance: the technique directly applies to the operator/pattern in
    ↪  the task.
    - Impact: likely to produce a meaningful speedup for this workload.
    - Complementarity: prefer a diverse set that covers different aspects
      (e.g. one memory, one compute, one correctness pitfall).

    ## What to avoid
    - Selecting skills just because they sound impressive.
    - Selecting skills about bypassing CUDA/Triton (e.g. "use
    ↪  torch.compile").
    - Selecting more than __top_k_select__ skills.

    When calling select_skills, first fill in `think`: briefly explain what
    ↪  the
    key bottleneck of this task is and why the chosen skills address it.
    Then fill in `selected_skills` with the names and one-sentence reasons.

    You will be called multiple times -- each call should make an independent
    selection to explore different strategies.

    Call select_skills exactly once. Respond in English only.
```

Select Agent user prompt (SFT series)

```
## Task

    __task_description__

    ---

    ## Candidate Skills (from the library)

    __file_tree__

    ---

    Now call select_skills to choose the top __top_k_select__ most relevant
    ↪  skills for
    this task.
```

Policy Agent system prompt (SFT series)

````
You write custom Triton kernels to replace the pytorch operators in the given
↪  architecture
to get speedups.

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


        Here's an example to show you the syntax of inline embedding custom
        Triton kernels in torch: The example given architecture is:

        ```

        import torch
import torch.nn as nn
import torch.nn.functional as F


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

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


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


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


        ```

        The example new arch with custom Triton kernels looks like this:

        ```
````
````

```
        import torch
import torch.nn as nn
import torch.nn.functional as F
import triton
import triton.language as tl


@triton.jit
def add_kernel(
    x_ptr,  # Pointer to first input
    y_ptr,  # Pointer to second input
    out_ptr,  # Pointer to output
    n_elements,  # Total number of elements in input/output
    BLOCK_SIZE: tl.constexpr,
):
    # Each program handles a contiguous block of data of size BLOCK_SIZE
    block_start = tl.program_id(0) * BLOCK_SIZE
    # Create a range of offsets [0..BLOCK_SIZE-1]
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Mask to ensure we don't go out of bounds
    mask = offsets < n_elements
    # Load input values
    x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
    y = tl.load(y_ptr + offsets, mask=mask, other=0.0)
    # Perform the elementwise addition
    out = x + y
    # Store the result
    tl.store(out_ptr + offsets, out, mask=mask)


def triton_add(x: torch.Tensor, y: torch.Tensor):
    """
    This function wraps the Triton kernel call. It:
      1. Ensures the inputs are contiguous on GPU.
      2. Calculates the grid (blocks) needed.
      3. Launches the Triton kernel.
    """
    assert x.is_cuda and y.is_cuda, "Tensors must be on CUDA."
    x = x.contiguous()
    y = y.contiguous()

    # Prepare output tensor
    out = torch.empty_like(x)

    # Number of elements in the tensor
    n_elements = x.numel()
    BLOCK_SIZE = 128  # Tunable parameter for block size

    # Determine the number of blocks needed
    grid = lambda meta: ((n_elements + meta["BLOCK_SIZE"] - 1) //
    ↪  meta["BLOCK_SIZE"],)

    # Launch the Triton kernel
    add_kernel[grid](x, y, out, n_elements, BLOCK_SIZE=BLOCK_SIZE)
    return out


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

    def forward(self, a, b):
        # Instead of "return a + b", call our Triton-based addition
        return triton_add(a, b)
        ```

    You are given the following architecture:
    ```
    import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a convolution, applies ReLU, and adds a bias
    ↪  term.
    """
    def __init__(self, in_channels, out_channels, kernel_size, bias_shape):
        super(Model, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size)
        self.bias = nn.Parameter(torch.randn(bias_shape))
```

```
    def forward(self, x):
        x = self.conv(x)
        x = torch.relu(x)
        x = x + self.bias
        return x

batch_size = 128
in_channels  = 64
out_channels = 128
height = width = 128
kernel_size = 3
bias_shape = (out_channels, 1, 1)

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

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

[Skill Library] Potentially Relevant Optimization Techniques

The following techniques are retrieved from the skill library and may be
↪ relevant to
this task. They are provided as optional references -- apply them only if
↪ they
fit the problem:
__skills__

Optimize the architecture named Model with custom Triton operators! Name your
↪ optimized
output architecture ModelNew. Output the new code in codeblocks. Please
↪ generate real
code, NOT pseudocode, make sure the code compiles and is fully functional.

Let's think step by step.
```

**Policy Agent feedback prompt (SFT series)**

```
Now you have received the server feedback for your last implementation. Based
↪  on
that and all your previous responses, improve the implementation.

      Here is the server feedback. Please refer to this feedback to improve
      the implementation:
      Server feedback (status/metrics/errors):
      {feedback}

      Return an improved Triton implementation named `ModelNew` as a
      single ```python``` block.Let's think step by step.
```

Select Agent tools

```
selection:
    - type: function
      function:
        name: select_skills
        description: >-
          Select the most relevant skills from the candidate list for this
          ↪  task.
          Call exactly once.
        parameters:
          type: object
          required: [think, selected_skills]
          properties:
            think:
              type: string
              description: >-
                Your analysis before selecting: what is the key bottleneck
                or challenge in this task? Which techniques are most directly
```
```

```
            applicable and why?
          selected_skills:
            type: array
            description: Ordered list of selected skills.
            # maxItems patched at runtime to top_k_select
            items:
              type: object
              required: [name, reason]
              properties:
                name:
                  type: string
                  description: Exact snake_case skill name from the
                  ↪  candidate list.
                reason:
                  type: string
                  description: "One sentence: why this skill applies to
                  ↪  this task."
```

Summary Agent tools

```
- type: function
    function:
      name: update_skill_library
      description: >-
        Propose new skills to add to the skill library.
        Call this exactly once.
      parameters:
        type: object
        required: [think, new_skills]
        properties:
          think:
            type: string
            description: >-
              Your analysis before writing skills:
              what is the key problem or bottleneck revealed by this
              ↪  trajectory?
              What technique turned things around?
              What pitfall caused early failures?
              Use this to justify the skills you are about to propose.
          new_skills:
            type: array
            description: List of new skills to add.
            # maxItems patched at runtime to max_new_skills_per_step
            items:
              type: object
              required: [name, description, scope, tags, content]
              properties:
                name:
                  type: string
                  description: "Short snake_case identifier, e.g.
                  'vectorized_load_store'"
                description:
                  type: string
                  description: One-line description for the selection agent
                  (<=120 chars).
                scope:
                  type: string
                  description: >-
                    'general' for broadly applicable skills,
                    or 'task_specific/<type>' for narrow ones
                    (e.g. 'task_specific/matmul').
                tags:
                  type: array
                  items:
                    type: string
                  description: 3-6 keyword tags (snake_case).
                content:
                  type: string
                  description: >-
                    Full skill body in Markdown.
                    Must contain ## Motivation, ## Key Idea, ## Example
                    ↪  sections.
                    Keep to <=512 tokens.
```