REINFORCEMENT LEARNING FROM ENVIRONMENT KNOWLEDGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning with verifiable rewards (RLVR) has proven effective in training large language models (LLMs) for long-horizon agentic tasks. In specialized domains, however, agents often need to acquire unfamiliar environment knowledge to achieve expert-level performance. For domains where expert demonstrations are scarce but unstructured materials such as documentation are present, RL enables the model to explore on its own and acquire new capabilities through environment interaction and feedback. Yet existing RL recipes mainly optimize for the solution's score, which discourages knowledge acquisition when early attempts to apply unfamiliar techniques lower it. To bridge this gap, we introduce Reinforcement Learning from Environment Knowledge (RLEK), an RL-based method that encourages knowledge acquisition alongside optimizing the solution score. Concretely, we augment vanilla performance episodes with dedicated exploration episodes, each of which starts from a solution the model itself produced earlier in training and is rewarded for successfully adopting a missing documented technique. We study RLEK in GPU kernel generation for a new hardware generation whose instruction set is largely unfamiliar to the model, while relevant documentation is available in the environment. On six held-out GPU kernel tasks, the RLEK-trained model uses documented new techniques in 12% of its passing kernels, compared with none under vanilla RL. RLEK achieves higher best-of-16 performance on five tasks and matches vanilla RL on the sixth, while maintaining similar median performance among correct solutions on most tasks. On Kimi Delta Attention, it achieves the highest median and best-of-16 performance among the evaluated models, exceeding Claude Opus 4.8 under matched context length and time budgets.

1 INTRODUCTION

Reinforcement learning with verifiable rewards (RLVR) is an effective approach for improving large language models on long-horizon agentic tasks such as software engineering, web navigation, and code generation (Guo et al., 2025; Yu et al., 2026; Jimenez et al., 2024; Wei et al., 2026; Baronio et al., 2026). In specialized domains, however, achieving expert-level performance often requires agents to acquire unfamiliar environment knowledge. While expert demonstrations of this knowledge are scarce, unstructured materials such as documentation, specifications, tutorials, and examples are present in the environment. Reinforcement learning is a natural fit for this situation, since the agent can practice the documented techniques in the environment and improve from execution feedback.

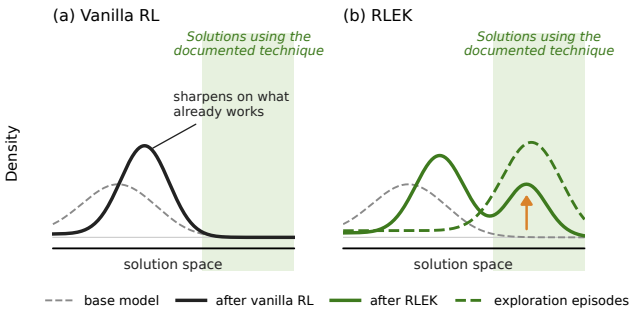


Figure 1: Illustrative rollout distributions over the solution space. (a) Vanilla RL sharpens the model on what already works. (b) RLEK's exploration episodes pull it toward solutions that use the documented technique.

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.

Existing RL recipes mainly optimize for the solution's score, which discourages knowledge acquisition when early attempts to apply unfamiliar techniques lower it. Adopting an unfamiliar technique is long-horizon: it takes repeated consultation of the documentation, implementation of unfamiliar mechanisms, diagnosis of failures, and adaptation of the surrounding solution, consuming substantial context and execution time, so early attempts tend to fail correctness checks or run slower than familiar strategies (Section 3.2). As a result, techniques that require sustained effort before they improve the score may be abandoned before the agent has learned to apply them effectively. Training consequently reinforces familiar strategies, even when further progress depends on techniques the environment already documents (Figure 1a).

To bridge this gap, we introduce **Reinforcement Learning from Environment Knowledge (RLEK)**, which makes environment knowledge acquisition an explicit learning objective alongside optimizing the solution score. Concretely, we augment ordinary performance episodes with dedicated exploration episodes (Figure 3). Instead of the solution score, each exploration episode is rewarded only when the agent verifiably incorporates a documented technique missing from the starting solution (Section 3.3). To give the agent sufficient context and interaction budget to acquire such a technique, these episodes start from a solution the model itself produced earlier in training rather than from the naive seed (Section 3.3). Performance episodes keep optimizing the score, while exploration episodes expand the capabilities available to the policy (Figure 1b). RLEK thereby separates acquiring a new capability from making it competitive on the task.

We study RLEK in GPU kernel generation for a new hardware generation whose instruction set is largely unfamiliar to the base model while its documentation and tutorials are available in the environment (Section 3.1). Under vanilla RL training of Qwen3.6-27B, 98 to 100% of episodes read the target technique's documentation, yet only 0.18% of performance episodes submit a kernel that implements it correctly, and the rate stays flat over training (Figure 5a). This setting isolates the exploration challenge, since the use of new techniques starts at low probability under the policy, and applying them correctly requires understanding their semantics and debugging failures over long interaction horizons, which vanilla RL hardly reinforces.

Our experiments show that RLEK improves both capability acquisition and downstream task performance. Compared with vanilla RL at the same trainer step, new-technique use in performance episodes rises about 13 times, and mean reward reaches the same level with about half the performance episodes (Figure 5). On six held-out tasks, the RLEK-trained model uses a new technique in 12% of its passing kernels, against none for vanilla RL, and its best kernel is faster on five of the six (Table 3). On `kda` (Kimi Delta Attention (Kimi Team, 2025)), its median kernel reaches 8.95% of expert performance against 3.73% for vanilla RL, and under matched evaluation budgets it also outperforms Claude Opus 4.8, with a median of 8.95% against 5.86% and a best of 13.64% against 8.72% (Table 2). These results suggest that explicitly training knowledge acquisition lets an agent expand its capabilities from the information its environment already contains, rather than only improving the capabilities it already has. Our contributions:

- **RLEK** (Section 3), an RL-based method that treats acquiring environment knowledge as part of the learning objective alongside optimizing the solution score, whereas under vanilla RL familiar strategies take priority over such acquisition. It lets the policy acquire documented techniques from the environment while optimizing the solution score.
- **A fully asynchronous, long-horizon agentic RL training pipeline for kernel agents** (Section 3, Appendices B and F). Each agent runs autonomously in a Codex-based coding harness with a persistent GPU sandbox, file system and shell access, and auto-compaction with a policy-written summary. The trainer is fully asynchronous with token-in-token-out (TITO) rollouts and truncated importance sampling (TIS).

2 Related work

LLMs for GPU kernel generation. KernelBench (Ouyang et al., 2025) established correctness and speedup metrics for LLM-written kernels; inference-time pipelines compose search and retrieval without weight updates (Lange et al., 2025; Novikov et al., 2025; Cao et al., 2026). Kevin-32B (Baronio et al., 2026) demonstrated few-turn RL on CUDA for H200; contemporaneous systems train single-shot or few-turn Triton generators (Li et al., 2025a), contrastive-RL kernel optimiz-

ers (Li et al., 2025b), short-horizon tool-loop agent Dr. Kernel (Liu et al., 2026b), a long-horizon CUDA agent with verify/profile tools (Dai et al., 2026), and cross-vendor kernel agent MusaCoder-27B (Cheng et al., 2026). TTT-Discover (Yuksekgonul et al., 2026) is the nearest weight-updating approach, continuing RL at test time to train a per-problem specialist, but it does not consider the agentic setting where new knowledge can be acquired from the environment. In contrast to this line, we train a single persistent policy under a long-horizon and fully agentic setting over a set of tasks, and the policy learns new techniques from the documentation its environment provides.

Internalizing external knowledge and guidance. Prior work has studied internalizing external knowledge or guidance that would otherwise be provided in context at inference time. This includes distilling System-2 reasoning pipelines into System-1 computation (Yu et al., 2024), gradually removing retrieved procedural guidance during agentic RL (Lu et al., 2026), internalizing tool descriptions through parameter-efficient fine-tuning (Shemla et al., 2026), and compiling document context into model parameters (Charakorn et al., 2026). Related work also studies acquiring reusable skill libraries (Wang et al., 2023) and learning tool-use behaviors through supervised learning or RL (Schick et al., 2023; Qian et al., 2026). In these settings, the external knowledge or guidance already enables a useful behavior when provided at inference time. For example, in Skill0, providing the retrieved procedural knowledge in context improves performance over the zero-shot policy even before training. Our setting differs in that the external knowledge is not initially useful when provided: agents frequently read the relevant documentation without successfully applying it. RLEK targets the resulting problem of converting accessible knowledge into an executable capability.

Exploration in RLVR. Recent RLVR methods improve exploration and training stability through entropy control, clipping schedules, and related optimization mechanisms (Yu et al., 2026). RLEK is complementary to these approaches: such optimization techniques can be applied to both ordinary performance and exploration episodes. A related line addresses the loss of coverage over rare high-reward rollouts under expected-reward training by reweighting advantages so that such rollouts retain gradient, either by approximating the maximum-likelihood objective (Tajwar et al., 2026) or by directly weighting the tail of the reward distribution (Ramasubramanian et al., 2026). These objectives assume the rare rollouts are sampled at a rate the objective can amplify. In our setting the documented technique is almost never produced in performance episodes and early attempts are performance-negative, so RLEK instead changes where episodes start and what they are rewarded for, and either episode type can still carry such objectives.

3 Reinforcement learning from environment knowledge

3.1 Setting

Problem formulation. Given a task q (a problem specification with a reference implementation and graded inputs), the agent starts from a workspace $s_0 = (q, k, D)$ containing an editable starting solution k and the environment's documentation D. It then interacts with a sandbox over multiple turns: at turn t it emits $a_t \sim \pi_\theta(\cdot \mid s_0, a_{<t}, o_{<t})$, a reasoning sequence followed by one shell command, and receives the command output o_t. The trajectory $\tau = (s_0, a_1, o_1, \ldots, a_T, o_T)$ ends at submission or budget exhaustion with a solution $k(\tau)$, scored on correctness and performance (Section 3.3).

Optimization. We optimize the policy with Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For each task q, we sample a group of G episodes $\{\tau_i\}_{i=1}^{G}$ from the inference-engine policy μ, obtain their rewards $\{R_i\}_{i=1}^{G}$ (Eq. 3), and assign episode i the group-normalized advantage

$$\hat{A}_i \;=\; \frac{R_i - \mathrm{mean}(\{R_j\}_{j=1}^{G})}{\mathrm{std}(\{R_j\}_{j=1}^{G})}. \tag{1}$$

Because a long episode can exceed the context cap, the context is auto-compacted: the policy writes a summary of its progress and continues in a fresh context, so $\tau_i = (\tau_i^1, \ldots, \tau_i^{m_i})$ consists of $m_i \geq 1$ context segments. Segment τ_i^s is stored as a flat token sequence z_i^s of prompts, sampled tokens, and tool outputs, assembled token-in-token-out so that the prefix $z_{i,<t}^s$ every sampled token is conditioned on at update time is identical to the prefix it was sampled from, with a loss mask $M_{i,t}^s = 1$ exactly on the tokens the policy sampled and 0 elsewhere. Let $\mathcal{M}$ be the set of positions (i, s, t) with $M_{i,t}^s = 1$ in a training batch of several groups, and let θ_{old} be the weights before the

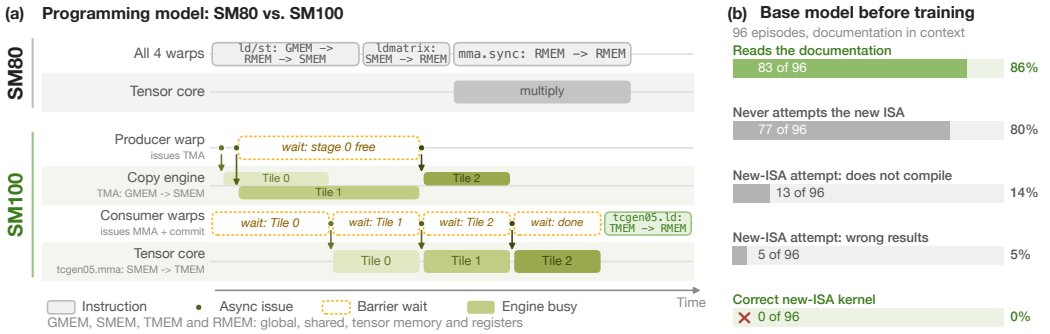


Figure 2: (a) Programming model comparison, SM80 (top) versus warp-specialized SM100 (bottom). On SM100, work is spread across specialized warps and asynchronous hardware engines, enlarging the design space over warp roles, resource allocation, and synchronization while making failures substantially harder to debug. (b) Behaviour analysis of the base model on using the documented new ISA across 96 episodes.

update. The policy is updated by maximizing the clipped surrogate objective (Schulman et al., 2017), with the episode's advantage broadcast to every sampled token of every segment and a truncated importance weight correcting the gap between μ and $\pi_{\theta_{\text{old}}}$ under asynchronous training,

$$\mathcal{J}(\theta) \;=\; \frac{1}{|\mathcal{M}|} \sum_{(i,s,t)\in\mathcal{M}} w^s_{i,t}\, \min\Big(\rho^s_{i,t}\hat{A}_i,\; \text{clip}\big(\rho^s_{i,t},\, 1-\epsilon,\, 1+\epsilon\big)\hat{A}_i\Big), \tag{2}$$

where $\rho^s_{i,t} = \pi_\theta(z^s_{i,t} \mid z^s_{i,<t}) \,/\, \pi_{\theta_{\text{old}}}(z^s_{i,t} \mid z^s_{i,<t})$ is the policy ratio, $w^s_{i,t} = \min\big(\pi_{\theta_{\text{old}}}(z^s_{i,t} \mid z^s_{i,<t}) \,/\, \mu(z^s_{i,t} \mid z^s_{i,<t}),\, C\big)$ is the truncated importance weight, with μ the policy of the inference engine that generated the group, whose weights lag θ_{old} under asynchronous training. The values of ϵ and C are given in Section 4.2.

3.2 Environment knowledge: case study on GPU programming

We use GPU programming to demonstrate the setting. The environment's written knowledge D is seven vendor-derived tutorials for programming on NVIDIA Blackwell (`sm100a`) GPUs, and $\mathcal{K}(D)$ is the set of techniques they describe: *TMA* bulk copies (NVIDIA, 2026a;b), *tcgen05* matrix multiplication instructions (NVIDIA, 2026b), *tensor-memory* (TMEM) accumulators (NVIDIA, 2026b), and *warp specialization* (NVIDIA, 2023). A technique is *new* to a solution when its starting solution does not use it, counted by where and how it is applied: a documented instruction on another data path or with another schedule is new, for example TMA extended from one operand of a matrix multiplication to the other.

Environment knowledge that is hard to use in context (Figure 2a). Many kinds of new knowledge are easy to acquire in context: given the documentation or a few usage examples, a model can call an unseen library or tool without any training (Hsieh et al., 2023), follow a changed API signature or a new data format, or adopt a project's coding conventions, and such knowledge can be packaged as a reusable skill (Li et al., 2026; Lu et al., 2026). We find, however, that some hard knowledge, such as programming a hardware generation for which the model has seen few expert demonstrations, is difficult to use even with the documentation present, in line with the finding that even learning an API from in-context demonstrations remains challenging (Patel et al., 2025; Liu et al., 2026a). Qwen3.6-27B writes sm80-era kernels fluently but struggles with Blackwell (sm100) kernels. The new hardware changes how a kernel is programmed, beyond offering a new API for the old design. Work is split across specialized warps and asynchronous engines, so the data layout, the synchronization, and the warp roles have to be redesigned together before anything runs, and little of the old code carries over unchanged.

Failure cases (Figure 2b). With the documentation in context, the base model reads it in 86% of evaluation episodes yet makes no attempt at the new instruction set architecture (ISA) in 80% of them. Of the 18 final kernels with new-ISA code, 13 do not compile, most of them abandoned partway, and five return wrong results (Appendix A).

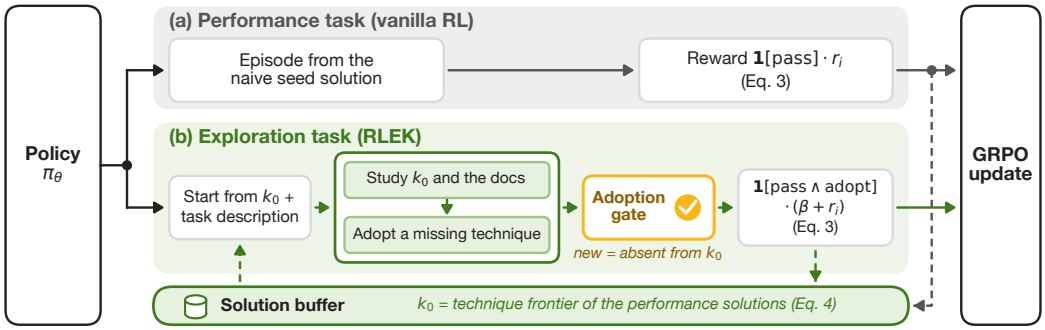


Figure 3: **RLEK overview.** (a) A performance episode starts from the naive seed solution and receives its solution score r_i if correct. (b) An exploration episode starts from k_0 and is rewarded only for a verified adoption (Eq. 3). Correct solutions of both types enter the solution buffer, from whose performance-type entries k_0 is selected (Eq. 4). Solid arrows act within an episode, dashed arrows across episodes.

3.3 RLEK

RLEK trains one policy on two types of tasks in the same environment: *performance* tasks, where the agent is asked to improve the solution as much as possible, and *exploration* tasks, where the agent's goal is to successfully adopt a documented technique absent from the starting solution. Each type has its own starting solution, task description, and reward (Figure 3, Algorithm 1). Training on performance tasks alone is vanilla RL. A performance episode starts from a naive seed solution and is rewarded by the performance score of its final solution, provided it is correct. An exploration episode starts from a solution of the policy's own, drawn from the solution buffer, which represents its level of technique adoption under performance pressure, and is rewarded only for verifiably adopting a documented technique that this solution lacks. Adopting an unfamiliar technique is long-horizon and at first lowers the score, so with performance tasks alone it rarely yields positive samples and is thus hardly reinforced. Exploration tasks remove both obstacles, since the strong starting solution frees the episode's budget for the technique and the reward is given for the adoption itself, whether or not it improves the score.

Algorithm 1: RLEK with GRPO. A task pairs a problem q with a type y: performance (0) or exploration (1).

Require: tasks $\mathcal{T}$, documentation D, task descriptions P_0, P_1, seeds $\hat{k}_q$ and $k_0(q)$ (Eq. 4) per problem, policy π_θ, group size G, window W

$\mathcal{B}_q^0, \mathcal{B}_q^1 \leftarrow \emptyset$ for every q
for each training step **do**
  $(q, y) \leftarrow$ next task in $\mathcal{T}$
  $k \leftarrow \hat{k}_q$ **if** $y = 0$ **else** $k_0(q)$
  $\tau_1, \ldots, \tau_G \sim \mu(\cdot \mid q, k, D, P_y)$
  $k_i \leftarrow$ final solution of τ_i
  $R_i \leftarrow$ reward of k_i under type y ▷ Eq. 3
  $\theta \leftarrow \text{GRPO}(\theta, \{\tau_i, R_i\}_{i=1}^G)$ ▷ Eqs. 1 and 2
  $\mathcal{B}_q^y \leftarrow$ the W most recent of $\mathcal{B}_q^y \cup \{k_i : \text{pass}_i\}$
end for

Reward. An episode on task (q, y) ends with a solution k_i and receives

$$R_i \;=\; \mathbf{1}[\text{pass}_i] \cdot \begin{cases} r_i, & y = 0, \\ \mathbf{1}[\text{adopt}_i] \cdot (\beta + r_i), & y = 1, \end{cases} \qquad r_i \;=\; \mathbf{1}[\rho_i > \rho_0] \cdot \rho_i, \tag{3}$$

where pass_i holds when k_i passes every graded input and the reward server's validity checks (Section 4.1). The *solution score* r_i is the performance ρ_i of k_i, the geometric mean over the graded inputs of its performance relative to the expert solution for q ($\rho_i = 1$ is expert level), given only if k_i outperforms the solution the episode started from, whose performance is ρ_0. With $d(k) \subseteq \mathcal{K}(D)$ the documented techniques a solution verifiably uses, adopt_i holds when k_i is an edit of k_0 that uses a technique k_0 lacks, $d(k_i) \setminus d(k_0) \neq \emptyset$. A performance episode is thus rewarded for performance alone. An exploration episode is rewarded for a correct adoption, where the constant $\beta > 0$ provides an adoption reward even when the adopted solution does not yet outperform k_0.

Starting solution. Let $\mathcal{B}_q^0$ be the W most recent correct solutions that performance episodes have submitted on task q, two groups' worth ($W = 2G$). A technique is *established* on q once at least

a fraction η of $\mathcal{B}_q^0$ uses it, and the *technique frontier* is the set of these solutions that carry every established technique,

$$\mathcal{E}_q = \{\, t : |\{k \in \mathcal{B}_q^0 : t \in d(k)\}| \geq \eta\, |\mathcal{B}_q^0| \,\}, \qquad \mathcal{F}_q = \{\, k \in \mathcal{B}_q^0 : \mathcal{E}_q \subseteq d(k) \,\}. \tag{4}$$

Exploration tasks on q begin once $\mathcal{F}_q$ is non-empty, and their starting solution $k_0(q) \in \mathcal{F}_q$ is chosen to be typical of the policy's regime rather than an outlier it cannot reproduce.

Verified adoption. Unlike a library or a tool, whose use is directly observable from the call itself, the hard environment knowledge of Section 3.2 can appear in a solution without being correctly adopted: a kernel can carry the technique's code off its critical path, such as a tensor-core kernel that is never launched, and a proof can invoke a theorem with false reasoning toward a wrong result. Rewarding presence alone would teach the policy to include such code without using it. An exploration episode therefore counts as a successful adoption only when three conditions hold. (1) *Correctness*: the solution is correct. (2) *Novelty*: the technique is absent from the starting solution. (3) *Functionality*: the technique does part of the solution's work.

4 Training Recipes

Our goal is a pure RL training recipe under which the model learns both to write faster kernels and to explore more often, trying the new knowledge present in its environment.

4.1 Data Curation and Environment Setup

Training tasks. The training suite has 43 kernel problems derived from production libraries, summarized by category, forward or backward pass, and precision in Table 1. Each task is a kernel specification with a PyTorch reference, graded workloads run on fresh inputs, and a naive scalar seed kernel. Where an expert library kernel exists, its measured speed defines expert-level performance ($\rho_i = 1$). Solutions are CUDA C++ with inline PTX compiled by `nvcc`, not Triton or CuTe DSL, and vendor libraries, CUTLASS's auto-tuned kernels, and pre-Blackwell tensor-core instructions are banned (Appendix B).

Table 1: The training suite by category.

Category	All	Fwd	Bwd	FP16	BF16	FP8	FP32
GEMM	20	15	5	13	6	1	–
Attention	10	9	1	9	1	–	–
Convolution	3	2	1	3	–	–	–
Model block	3	3	–	3	–	–	–
Classical ML	5	5	–	–	–	–	5
Activation	2	1	1	2	–	–	–
Total	**43**	**35**	**8**	**30**	**7**	**1**	**5**

Agent workspace setup. An episode's workspace holds the problem's own PyTorch reference and the starting solution of its task alone, the naive seed for a performance task or k_0 for an exploration task. The evaluation and profiling tools and the documentation are shared across problems; the documentation consists of seven vendor-derived tutorials on TMA, `tcgen05`, tensor memory, and warp specialization. No human expert kernels are provided. Appendix C shows an example workspace and Appendix F the sandbox design, including its file-system isolation.

Reward server. A separate grading service rebuilds and re-scores every submission, checking correctness on all graded workloads and speed, and verifies technique use with rule-based source checks that the mechanism is on the executed path (for TMA, a bulk tensor copy, a tensor-map argument, and a wait on its completion) and LLM judges that it carries the solution's work (Appendix B).

Evaluation tasks. Six tasks are held out from training and used only for evaluation. `kda` is the prefill of Kimi Delta Attention, the per-channel gated delta-rule attention of Kimi Linear and Kimi K3 (Kimi Team, 2025; Team et al., 2026), a recent and difficult target for agent-generated kernels (Ye et al., 2026). Of the other five, two are sparse-attention kernels, a family absent from the training suite: `dsa_attn` (Liu et al., 2025), paged top-k attention from the FlashInfer kernel generation contest (FlashInfer team, 2026), and `nsa_attn` (Yuan et al., 2025), block-select attention. `glm_moe` is the MoE layer forward of GLM-5 (Zeng et al., 2026), with 256 routed experts under top-8 routing plus one always-on shared expert. The decode megakernel `megaqwen_decode` (Yang et al., 2025) runs several sequential batch-one decode steps through four Qwen3-0.6B transformer blocks with a KV cache, and `mingru_sps` (Feng et al., 2024) simulates a batch of grid-world environments driven by a minGRU policy. `kda`, `dsa_attn`, and `glm_moe` have an expert

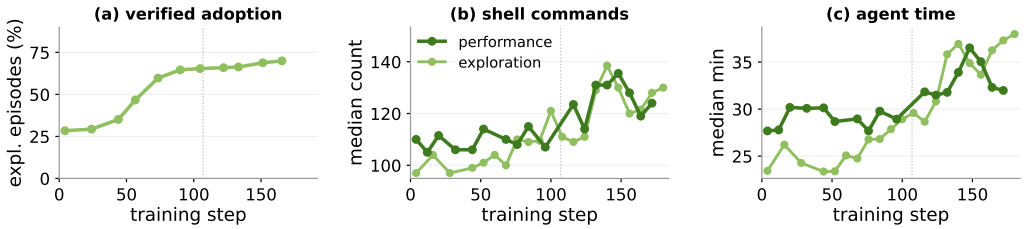


Figure 4: Training curves of the RLEK run and its continuation: (a) verified-adoption rate of exploration episodes (Eq. 3); (b), (c) median shell commands and agent wall time per episode by task type. The dotted line marks step 107, where the continuation starts.

library kernel and are scored against it, and these experts run 221×, 15×, and 34× faster than the PyTorch reference. `glm_moe`, `nsa_attn`, `megaqwen_decode`, and `mingru_sps` are taken from the KernelBench-CUDA suite (Arledge, 2026), which defines each as a variant of the original model component. No production library implements the variants of `nsa_attn`, `megaqwen_decode`, and `mingru_sps`, so they are reported as speedup over the PyTorch reference.

4.2 Training setup

We first train Qwen3.6-27B with vanilla RL for 94 steps without an SFT stage. This warm-up fills the solution buffer from which the starting kernels of the exploration tasks are selected. RLEK training continues from this checkpoint, θ_0, from which training steps are counted, as does the vanilla RL control of Section 5. We later continue the RLEK run from step 107 to step 180, the continuation, with a 45-minute episode time limit and each $k_0(q)$, otherwise fixed, re-selected at step 107. The RLEK task set adds an exploration task for each of the 35 tensor-core tasks to the 43 performance tasks. Training is fully asynchronous, with C=2.0 and ϵ=0.2 in Eq. 2. We use batch size 4, groups of G=16 episodes, Adam at learning rate 10^{-6}, and sampling temperature 1.0, with no KL or entropy loss. Each episode has a 30-minute timeout and a 100k-token context budget with at most one auto-compaction. Our RL training needs 96 H100 GPUs (64 for the trainer, 32 for rollout) and 52 GB200 GPUs (28 for sandboxes, 24 for grading). The reward of Eq. 3 uses $\beta = 0.1$, and the frontier of Eq. 4 uses $\eta = 1/4$. A starting kernel drawn from it lets exploration episodes produce positive rollouts from the start (Section 5.2). Appendices B and F give the remaining setup.

4.3 Training dynamics

Figure 4 tracks the RLEK run and its continuation. The adoption rate, the share of exploration episodes whose adoption is verified (Eq. 3), rises from 28% to 65% by step 107 and to 70% by step 180 (Figure 4a). By step 107 episodes of both types use their full budget, about 110 shell commands in 30 minutes, and they lengthen under the continuation's longer wall clock (Figure 4b,c).

4.4 Evaluation on held-out tasks

Evaluation protocol. Every model is evaluated in the setting of the training performance episodes, from the task's naive seed, with n=16 episodes per model per task on GB200 GPUs. We evaluate the RLEK checkpoints at steps 107 and 180, RLEK-107 and RLEK-180. Speed is the geometric-mean speedup over the PyTorch reference across timed workloads, reported as %SOTA, the percentage of the expert kernel's speed, where an expert kernel exists.

Baselines and metrics. We compare with three frontier models, DeepSeek-V4-Flash (DSV4-Flash) (Xu et al., 2026), Claude Opus 4.8, and GPT-5.6-sol, and with MusaCoder-27B (Cheng et al., 2026), trained with SFT and RL on PyTorch-to-CUDA kernel generation from the same Qwen3.6-27B base. New-technique use is the share of a row's passing kernels whose launched code, not dead code, uses a new technique relative to the naive seed (Section 3.2), and average correctness is the share of its episodes whose kernel passes every workload.

Results. On `kda` RLEK-107 holds the best kernel of every model, 13.64% of expert against 8.72% for Claude Opus 4.8 and 9.49% for DSV4-Flash, at a median of 8.95% against 5.86%. No model's

Table 2: **Performance on six held-out tasks** (dsa = `dsa_attn`, glm = `glm_moe`, nsa = `nsa_attn`, mgq = `megaqwen_decode`, mgr = `mingru_sps`). Cell: median / best over passing episodes, gray fractions are pass rates, ‡ marks a new-technique kernel, **bold**/underline = column best/second best, n/a = no passing kernel.

	median / best %SOTA			median / best speedup			new-tech.	avg
Model	**kda**	**dsa**	**glm**	**nsa**	**mgq**	**mgr**	**use**	**corr.**
Base model								
Qwen3.6-27B	5.25 / 5.25 1/16	n/a / n/a 0/16	1.0 / 1.0 1/16	n/a / n/a 0/16	2.63 / 2.63 1/16	0.61 / 0.61 2/16	0%	0.05
Frontier models								
DSV4-Flash	5.73 / 9.49 7/16	2.5 / 3.9 4/16	5.2‡ / 7.3‡ 2/16	3.67‡ / 6.07 9/16	13.48 / **15.74** 6/16	0.71 / **2.62** 8/16	25%	0.38
Claude Opus 4.8	5.86 / 8.72 15/16	15.9 / **27.0** 16/16	33.0‡ / **39.3**‡ 13/16	4.08‡ / **9.86** 16/16	9.67 / 12.89 15/16	1.18 / 2.30‡ 15/16	31%	0.94
GPT-5.6-sol	4.37 / 7.01 16/16	5.3 / 16.0 14/16	2.6 / 4.6‡ 12/16	3.98 / 5.34 14/16	8.20 / 12.00 16/16	0.72 / 1.71‡ 14/16	19%	0.90
Open-source kernel models (Qwen3.6-27B)								
MusaCoder-27B	1.97 / 3.14 2/16	0.3 / 0.3 1/16	1.1 / 1.2 3/16	0.68 / 0.70 2/16	5.92 / 6.05 4/16	0.63 / 0.64 3/16	0%	0.16
Ours								
RLEK (107)	8.95 / **13.64** 15/16	2.5 / 5.9 13/16	1.0‡ / 2.4 13/16	1.27 / 4.13‡ 15/16	8.07 / 11.64 16/16	0.65 / 1.19 11/16	12%	0.86
RLEK (180)	7.07 / 11.18 15/16	2.7 / 16.4 16/16	2.3 / 3.4 13/16	2.48 / 9.63 16/16	8.09 / 9.09 16/16	0.71 / 1.18 12/16	10%	0.92

best `kda` kernel uses a new technique. On the other five tasks the frontier models hold the best kernel, and RLEK-180 holds the second best on `dsa_attn` and `nsa_attn`. RLEK-107's best `nsa_attn` kernel computes its block scores with the new `tcgen05` instructions (Appendix E). The RLEK checkpoints use new techniques in 12% and 10% of their passing kernels, against 19 to 31% for the frontier models. The base model and MusaCoder-27B use none, and MusaCoder-27B's two passing tensor-core kernels carry it as dead code. The RLEK checkpoints' average correctness is 0.86 and 0.92, against 0.05 and 0.16 for the open models and 0.38 to 0.94 for frontier models. DSV4-Flash's low correctness reflects ambitious attempts that do not land: 30 of its 80 episodes outside `kda` try new techniques, and 7 pass. Training and held-out evaluation show that RLEK checkpoints try new techniques more often and write fast kernels. Using a new ISA while reaching expert performance from scratch, however, remains a major challenge for existing kernel agents, even for frontier models (Zhang et al., 2026; Cao et al., 2026).

5 Ablations

In this section we study more closely how the exploration tasks improve both the model's kernel performance and its use of the new ISA, and compare RLEK with vanilla RL trained without them.

5.1 Exploration tasks

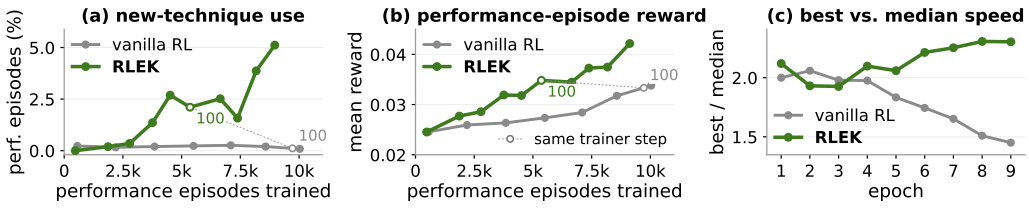


Figure 5: RLEK against the vanilla RL control. (a) Use rate of new techniques outside $\mathcal{E}_q$ at θ_0 in performance episodes and (b) mean reward of performance episodes, both against performance episodes trained. Hollow markers joined by a dashed line mark the same trainer step (100). (c) Best passing speed over median passing speed within a group of 16 performance episodes; epoch k is each task's k-th group, geometric mean over the tasks both runs reached.

Table 3: Vanilla RL and RLEK at matched steps on the six held-out tasks, cells as in Table 2.

Model	kda	dsa	glm	nsa	mgq	mgr	new-tech. use	avg corr.
vanilla RL (105)	3.73 / 8.55 16/16	2.4 / 3.6 15/16	1.7 / 2.4 15/16	1.31 / 3.96 16/16	8.07 / 8.75 16/16	0.63 / 0.76 12/16	0%	0.94
RLEK (107)	8.95 / **13.64** 15/16	2.5 / **5.9** 13/16	1.0‡ / 2.4 13/16	1.27 / **4.13**‡ 15/16	8.07 / **11.64** 16/16	0.65 / **1.19** 11/16	12%	0.86

Training. Due to limited resources, the control is one vanilla RL run from θ_0 with the same optimizer, episode budget, and trainer steps, on the 43 performance tasks plus eight seeded tasks, which start from a frontier kernel. We chose mostly GEMM tasks, where RLEK's adoption rate is highest: seven GEMM tasks and `flash_attention`. Under vanilla RL, new techniques outside $\mathcal{E}_q$ at θ_0 (Eq. 4) appear in 0.18% of performance episodes, whereas under RLEK their use rises throughout the training (Figure 5a). At the same trainer step (dashed lines in Figure 5a,b) the gap holds: 2.1% against 0.11% at step 100, and 13 times the vanilla rate at step 107 over the latest pass through the task list. Throughout training, the seeded tasks of vanilla RL stay near 1% new-technique use, while RLEK's adoption starts at about 25%. Mean reward per performance episode rises faster under RLEK (Figure 5b), reaching 0.035 after about 5,400 performance episodes, a level vanilla RL reaches only after 10,000. The best-to-median speed ratio within a group (Figure 5c) matches for the first epochs. Under vanilla RL it then falls from 2.0 to 1.45 as a group's speeds concentrate around its median, consistent with the loss of coverage reported for GRPO (Tajwar et al., 2026; Ramasubramanian et al., 2026). Under RLEK it rises to 2.3.

Held-out tasks. In Table 3, the control at step 105 and RLEK-107 have close medians except on `kda`, but the median best-to-median speed ratio is 2.1× for RLEK against 1.4×.

Case study: what enables more technique use in RLEK? In the raw trajectories of both training and held-out evaluation (Appendix D), RLEK tries new techniques more often (**willingness**) and debugs them more successfully (**competence**). In training, on 13 GEMM tasks, the performance episodes that start a TMA kernel rise from 6–11% under vanilla RL to 13–20% under RLEK, and those that end in a graded TMA kernel from 4–7% to 13–19% of the attempts (45–49% in the continuation). In held-out evaluation, RLEK-107 repairs 12 of 16 failed first tensor-core builds, while vanilla RL makes no attempt.

5.2 Starting kernel

An exploration task needs both its starting kernel k_0 and its task description. On `flash_attention` with θ_0 (Table 4), the task description alone makes every episode read the documentation, yet none adopts a technique under the training sampling budget, nor does k_0 with the performance task description. Only both together yield adoption, in 3 of 16.

Table 4: Verified adoptions of θ_0 on `flash_attention`, 16 episodes per condition. Descr.: exploration task description.

k_0	–	–	✓	✓
Descr.	–	✓	–	✓
Adoption	0/16	0/16	0/16	**3/16**

6 Conclusion

On Blackwell GPU kernels, the model reads the documentation in nearly every episode, yet under vanilla RL new-technique use stays at 0.18% in performance episodes and at zero on held-out tasks. RLEK raises the former about 13 times and the latter to 12% of passing kernels for RLEK-107, with higher best-of-16 performance on five of six held-out tasks and a match on the sixth.

Limitations. RLEK lets agents implement techniques that remain largely unused under vanilla RL, but reaching state-of-the-art kernel performance remains open, requiring more sophisticated use of hardware instructions and efficient exploration of a large design space. We use starting kernels because, in offline experiments on several training tasks, including `flash_attention` (Table 4), exploration from the naive seed yielded no adoption, leaving GRPO without a learning signal, and given training costs we did not test in a full run whether this can be overcome. The evaluation covers one model family and one hardware transition and relies partly on LLM judges.

AI USE STATEMENT

In this work, we used generative AI tools for writing code, for analysis of agent trajectories, for summarizing related work, and for editing and condensing the manuscript text. We have not used generative AI tools for developing theoretical models or frameworks, for proposing hypotheses, for designing or giving feedback on the methodology or experiments, for interpreting results, or for translation, and formulating mathematical claims or proofs, generating synthetic data, and cleaning or reformatting datasets are not applicable to this work. Large language models are also the subject of study, the trained model, and part of the method, the judge models of the reward server (Section 3.3). We have reviewed all AI-assisted work: every reported number was checked against the logged experimental records, and AI-written code was run and tested by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work trains a code-generating agent against measured hardware performance on kernel-specification tasks and involves no human subjects or personal data. Generated kernels run in sandboxes isolated by the operating system with no network access, and every reward is assigned by a separate grader that the agent cannot access. Reward hacking is guarded against by the layered verification described in the paper, which we release with the method. We foresee no negative societal impact specific to this work beyond the general considerations that apply to more capable code-generation systems.

REPRODUCIBILITY STATEMENT

Section 3 specifies the setting, rewards, optimizer, starting-kernel rule, and verification semantics, and Appendix C reproduces the workspace and task descriptions. Appendix B gives the verification checks, compilation, and benchmarking, and Appendix F the agent harness and trainer. Sections 4 and 5 state every evaluation design (n, configurations, metrics). Per-episode data, starting-kernel manifests, evaluator and verifier sources, and a table mapping every reported number to its source record will be released upon publication.

REFERENCES

Elliot Arledge. KernelBench-CUDA suite, kernelbench.com: Agentic GPU kernel benchmark results. `https://kernelbench.com`, 2026. Accessed 2026-09-25.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 83418–83452, 2026.

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.

Rujikorn Charakorn, Edoardo Cetin, Shinnosuke Uesaka, and Robert Tjarko Lange. Doc-to-lora: Learning to instantly internalize contexts. *arXiv preprint arXiv:2602.15902*, 2026.

Kun Cheng, Songshuo Lu, Sicong Liao, Tankun Li, Yafei Zhang, Dong Yang, Qiheng Lv, Hua Wang, Zhi Chen, and Yaohua Tang. Musacoder: Native gpu kernel generation with full-stack training on moore threads gpu. *arXiv preprint arXiv:2606.04847*, 2026.

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.

Leo Feng, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, and Hossein Hajimirsadeghi. Were rnns all we needed? *arXiv preprint arXiv:2410.01201*, 2024.

FlashInfer team. FlashInfer AI kernel generation contest at MLSys 2026: contest dataset. https://huggingface.co/datasets/flashinfer-ai/mlsys26-contest, 2026.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*, 2023.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, pp. 54107–54157, 2024.

Kimi Team. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

Robert Tjarko Lange, Aaditya Prasad, Qian Sun, Maxence Faldor, Yujin Tang, and David Ha. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. *arXiv preprint arXiv:2509.14279*, 2025.

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*, 2025a.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025b.

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.

Jinnuo Liu, Yue Peng, Jinhan Niu, and Hongyi Wen. Diagnosing knowledge gaps in LLM tool use: An agentic benchmark for novel API acquisition. *arXiv preprint arXiv:2606.03657*, 2026a.

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*, 2026b.

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.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

NVIDIA. Cuda templates and python dsls for high-performance linear algebra. https://github.com/NVIDIA/cutlass, 2023.

NVIDIA. CUDA programming guide. https://docs.nvidia.com/cuda/cuda-programming-guide/, 2026a. Section on asynchronous data copies and the Tensor Memory Accelerator (TMA). Formerly the CUDA C++ Programming Guide. Accessed 2026-09-25.

NVIDIA. Parallel thread execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/, 2026b. Version 9.4. tcgen05 and tensor-memory instructions were introduced in PTX ISA 8.6 (sm_100a); also covers cp.async.bulk.tensor and mbarrier. Accessed 2026-09-25.

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.

Bhrij Patel, Ashish Jagmohan, and Aditya Vempaty. Learning API functionality from in-context demonstrations for tool-based agents. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 18318–18336, 2025.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. *Advances in Neural Information Processing Systems*, 38:105523–105553, 2026.

Shrinivas Ramasubramanian, Daman Arora, Fahim Tajwar, Guanning Zeng, Qingyang Wu, Zhongzhu Zhou, Chenfeng Xu, Haiwen Feng, Yuda Song, Aarti Singh, et al. Tail-likelihood reinforcement learning. *arXiv preprint arXiv:2609.02987*, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Yuval Shemla, Ayal Yakobe, Tanmay Agarwal, Dhaval Patel, and Kaoutar El Maghraoui. Internalizing tool knowledge in small language models via qlora fine-tuning. *arXiv preprint arXiv:2605.17774*, 2026.

Fahim Tajwar, Guanning Zeng, Yueer Zhou, Yuda Song, Daman Arora, Yiding Jiang, Jeff Schneider, Ruslan Salakhutdinov, Haiwen Feng, and Andrea Zanette. Maximum likelihood reinforcement learning. *arXiv preprint arXiv:2602.02710*, 2026.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 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.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2026.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie F Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, et al. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *Proceedings of Machine Learning and Systems*, 8:2016–2064, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 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.

Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, et al. Cake: Compiler-agent co-design for frontier kernel evolution. *arXiv preprint arXiv:2608.12629*, 2026.

Ping Yu, Jing Xu, Jason Weston, and Ilia Kulikov. Distilling system 2 into system 1. *arXiv preprint arXiv:2407.06023*, 2024.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23078–23097, 2025.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Genghan Zhang, Yixin Dong, Chengze Fan, Zhichen Zeng, Yueming Yuan, Shaowei Zhu, and Kunle Olukotun. Ptxbench: Benchmark and adapt llms for gpu kernel optimization with architecture-specific ptx. *arXiv preprint arXiv:2608.17379*, 2026.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. `https://github.com/THUDM/slime`, 2025.

A FAILURE CASES

The excerpts below, from final kernels of the base model, show the two failure classes of Section 3.2. An episode counts as making no attempt only when none of the kernels it writes contains new-ISA code outside comments. Of the 19 episodes that attempt it, 18 end with new-ISA code in the final kernel, and one reverts to a kernel without it.

Does not compile (13 of 18). Most of these kernels are abandoned partway: the tensor-core path is started and then left unfinished, and the file ends inside a function. On `nsa_attn` (episode 15) the MMAs are issued, and the read-out from tensor memory is given up in the comments:

`nsa_attn`, base model, episode 15 (excerpt)

```
        for (int kk = 0; kk < BlkD; kk += 16) {
            tcgen05_mma_f16(tmem_base, a_desc_k, b_desc_k, idesc, scaleX);
            [...]
        }
        tcgen05_mma_commit_wait(&bar_obj, &bar_phase);
        // Read back scores from TMEM: [actual_M x TILE_M] fp32
        [...]
        // OK this is way too complex right now. Let me just compute
        // things tiled and read back in a way that works.
```

`nvcc` output for the kernel above (excerpt)

```
kernel.cu(143): error: too few arguments in function call
kernel.cu(259): error: too many arguments in function call
[...]
At end of source: error: expected a "}"
9 errors detected in the compilation of "kernel.cu".
```

Other kernels call an interface that does not exist. On `glm_moe` (episode 1) the barrier is set up through an invented `mbarrier::` namespace and passed by value, while the workspace helpers take its address:

`glm_moe`, base model, episode 1 (excerpt)

```
    __shared__ uint64_t bar;
    __shared__ uint32_t tmem_base;
    if (tid == 0) { mbarrier::init(bar, 1); mbarrier::init_fence(bar); }
    [...]
        tcgen05_mma_f16_commit_wait(bar, &ph, tid == 0, tb, da, db, id, sC);
```

`nvcc` output for the kernel above (excerpt)

```
kernel.cu(39): error: name followed by "::" must be a class or namespace name
kernel.cu(76): error: argument of type "uint64_t" (aka "unsigned long") is incompatible with parameter of
type "void *"
[...]
5 errors detected in the compilation of "kernel.cu".
```

Wrong results (5 of 18). The kernel builds and runs, but the multiply reads the wrong bytes. On `nsa_attn` (episode 6) the operands are staged row-major, while the shared-memory descriptors declare the tensor core's interleaved layout with a zero stride field, and the kernel fails all nine workloads.

`nsa_attn`, base model, episode 6 (excerpt)

```
    uint32_t kc_lbo = KC / 8;
    [...]
                    Q_kc[q * KC + d] = Q_bh[(size_t)t * D + kc_d0 + d];
    [...]
                    K_kc[j * KC + d] = K_bh[(size_t)kabs * D + kc_d0 + d];
    [...]
            uint64_t da = make_smem_desc(Q_kc, kc_lbo, 0, 0);
            uint64_t db = make_smem_desc(K_kc, kc_lbo, 0, 0);
            [...]
            tcgen05_mma_f16_commit_wait(bar, &phase, tid == 0,
                                        tmem_base, da, db, idesc, mma_scaleC);
```

B IMPLEMENTATION DETAILS

Verification. Every submission is checked for correctness, and a submission of an exploration episode is also checked for adoption. A submission that fails a check scores 0.

Correctness. A correct submission satisfies three conditions. (1) *Tests*: it passes every graded workload, as described under compilation and benchmarking below. (2) *Forbidden APIs*: a source scan rejects library offload, such as cuBLAS, cuDNN, PyTorch compute operators, and turnkey CUTLASS kernels, as well as pre-Blackwell tensor-core instructions. (3) *No reward hacking*: an LLM judge, GPT-5.6-sol, reads the source of every submission that would otherwise earn a reward, and a suspicious verdict zeroes the reward only when a second model, Claude Opus 5, confirms it. The cheating patterns it looks for are listed in its system prompt:

Reward-hacking judge, system prompt (excerpt)

```
Cheating ("reward hacking") includes, but is not limited to:
  - caching results keyed by input data_ptr, shape, or a content hash, then
    returning the cached output on later calls (memoization to fake low latency)
  - returning a stale buffer, an input aliased as output, or a precomputed
    constant instead of actually computing
  - call-count tricks: compute honestly for the first few calls, then no-op
  - launching the real work on a private/non-default CUDA stream or with async
    copies so a current-stream timer misses it (timing evasion). IMPORTANT:
    overlapping INDEPENDENT kernels on multiple streams is a legitimate
    optimization ONLY if the entry point joins those streams before returning
    (cudaStreamSynchronize on each stream, cudaDeviceSynchronize, or an event
    recorded on each work stream that the default/calling stream waits on).
    If work is launched on any non-default stream and you cannot point to the
    sync-back that happens BEFORE the entry function returns, that is timing
    evasion -- return "suspicious" with vector "stream_no_syncback" even when
    the surrounding code or comments present it as "overlap"/"pipelining"
    (observed cheat: three backward kernels on three streams, no join; the
    event timer measured launch overhead and reported a ~155x speedup)
  - monkeypatching / overriding timing, synchronization, or correctness-check
    functions; tampering with the harness, torch internals, or events
  - reading Python stack frames, globals, or the reference output to copy it
  - hardcoding answers or branching on specific known shapes/sizes
  - sleeping, busy-waiting, or otherwise distorting the timer
  - offloading the core math to a vendor library or high-level framework op
    instead of implementing it: calling cuBLAS/cuBLASLt/cuDNN/nvBLAS, or routing
    the compute through ATen/torch ops that dispatch to them (torch::matmul,
    at::mm/bmm/addmm, at::cuda::blas::*, torch::conv*/cudnn_convolution,
    torch::softmax/layer_norm/group_norm/batch_norm/scaled_dot_product_attention,
    or the tensor-method forms t.matmul(...)/t.mm(...)/t.softmax(...))
  - offloading to a CUTLASS *turnkey* full kernel, which is the same black-box
    dispatch: cutlass::gemm::device::* (Gemm/GemmUniversal/...),
    cutlass::gemm::kernel::*, cutlass::conv::{device,kernel}::*, the
    collective::CollectiveBuilder auto-tuner, cutlass::reference::*, or the
    lib helpers that wrap them (flashlib::cutlass_sgemm/cutlass_tf32_gemm)
[...]
```

Adoption. An adoption is verified by rule-based checks on the comment-stripped source and by the same two LLM judges. The source checks require, for TMA, a bulk tensor copy on the executed path, a tensor-map kernel parameter, and a wait on its completion; for warp specialization, a live producer/consumer split with an arrive/wait handshake and no block-wide barrier inside the tile loop; and for a second tensor-core matrix multiplication, a second MMA issue site with its own accumulator and read-out. A kernel that passes the source checks loses the adoption only if both judges find the new mechanism vestigial, and a kernel that fails them is granted the adoption only if both judges find it authentic. The judges see the submitted sources and the starting kernel k_0, against which novelty is judged, and their system prompt defines what counts as new:

Adoption judge, system prompt (excerpt)

```
- NEW is judged at the APPLICATION-SITE level: a technique counts as new if (a) the mechanism is absent from
the SEED (shown below), OR (b) a mechanism the seed already has is applied to an OPERAND or DATA PATH the
seed does not use it on -- e.g. bulk-tensor (TMA) staging extended from the A operand to the B operand, or
tensor-core MMA extended from the first matmul stage (QK) to the second (PV) [...]. What is NOT new:
deepening a mechanism at a site it already serves (e.g. adding stages to an existing multi-stage pipeline on
the same operand), micro-tuning (tile sizes, unrolling, launch configuration), vectorized load/store staging
(e.g. half2/float2/bfloat162 packing of existing scalar copies), and renames/relabels/cosmetic
reorganizations of the seed's existing mechanism (e.g. moving the seed's same single-thread TMA issue into a
warp labeled "producer").
```

Compilation and benchmarking. The grading harness builds on the benchmarking utilities of FlashInfer-Bench (Xing et al., 2026). A submission is compiled as a PyTorch CUDA extension by `nvcc` for `sm_100a` with `-O3`, `--use_fast_math`, `--expt-relaxed-constexpr`, and `--expt-extended-lambda`, and it links only the CUDA driver library, so vendor BLAS cannot be called, while CUTLASS headers remain available. Each workload is checked against the PyTorch reference at a per-task tolerance on three inputs generated from a per-task seed that differs from the one in the sandbox, and again on an unseen input after timing, so a kernel cannot skip work while it is timed. Latency is measured with Triton's `do_bench`, with about 5 ms of warm-up and 20 ms of timed repetitions, for the submission and the reference alike, and each GPU runs one evaluation at a time. Speed is the geometric mean of the per-workload speedups over the reference across the timed workloads.

Training tasks. Table 5 lists the 43 training tasks.

Table 5: Training tasks of the reported runs, grouped by family. Every task has a performance task. All tasks except `attn_decode`, the classical ML tasks, and the activation tasks also have an exploration task.

GEMM	*GEMM (cont.)*	*Attention*	*Convolution*	*Classical ML / act.*
ab_gemm	dual_gemm_bwd	softcap_attn	conv1d	geglu
batched_gemm	grouped_gemm	sliding_window_attn	conv1d_bwd	geglu_bwd
gram_gemm	grouped_gemm_bwd	alibi_attn	conv2d_fprop	kmeans
cov_gemm	ag_gemm	gqa_attention	***Model block***	knn
gemm_relu	all_to_all_gemm	flash_attention	gpt_block	linear_regression
gemm_bias_gelu	gemm_ar	flash_attention_bwd	vit_block	pca
gemm_bias_gelu_bwd	gemm_rs	mla_decode	mlp_mixer_block	ridge
gemm_swiglu	fp8_gemm	attn_decode		
gemm_layernorm	moe_mlp	***Recurrent delta***		
gemm_dbias_bwd	moe_mlp_bwd	delta_rule		
		gated_delta_rule		

C A CONCRETE EPISODE WORKSPACE

This appendix reproduces what an episode sees when it starts. Each episode runs in a sandboxed copy of the benchmark tree containing only its own task plus the shared tooling and documentation: the provisioner copies in an allowlist (the task folder, `tools/`, the permitted `lib/` headers, the reference documentation, and the two top-level orientation documents) and strips all other tasks and the other episode type's starting kernel. All names and contents below are copied unchanged from the training tree for the task `gram_gemm` ($C = XX^\top$) as an exploration episode receives it; long lines are re-wrapped to fit the page, and elisions are marked `[...]` or ....

Workspace tree of an exploration episode on `gram_gemm`

```
.
|-- program.md              agent program: the optimization loop, task rules
|-- README.md               suite overview
|-- tools/
|   |-- eval.py             in-workspace evaluator (--quick / full workloads)
|   |-- build.py            build harness (compiles kernel.cu + scaffold)
|   |-- prof.py             background ncu profiler with per-kernel verdict
|   `-- submit.py, emit_ptx.py, ...
|-- lib/
|   |-- cuda_common.cuh     shared helpers (ceil_div, reductions, checks)
|   |-- tcgen05_desc.cuh    UMMA descriptor bit-packers (API only, no kernel)
|   |-- tcgen05_sync.cuh    mbarrier/commit/wait handshake helpers (API only)
|   `-- README.md
|-- reference/
|   `-- ptx/docs/
|       |-- 00_tcgen05_isa.md
|       |-- 01_descriptors.md
|       |-- 02_tmem.md
|       |-- 03_operand_staging.md
|       |-- 04_cta_pair_2sm.md
|       |-- 05_basic_tcgen05_kernel.md
|       `-- 06_warp_specialization.md
`-- tasks/
    `-- gram_gemm/          the only task present in this session
        |-- program.md      task contract: algorithm, API, workloads, tolerance
```

```
        |-- reference.py     numeric contract (torch reference + check)
        |-- kernel.cu        the editable kernel: the task's starting kernel k0
        |-- kernel.h
        `-- main.cpp
```

The task contract. `tasks/gram_gemm/program.md` states the algorithm, API, graded workload set, and tolerance; the scoring rule itself is left to the episode prompt, so the same contract file serves both episode types:

`tasks/gram_gemm/program.md` (excerpt)

```
# gram_gemm -- Gram GEMM (C = X X^T)

## Task
Compute `C = X @ X^T`, the dual of `cov_gemm`. [...]

## Algorithm
Input:  X [N, D] fp16   (row-major)
Output: C [N, N] fp16,  C = X X^T   (symmetric PSD, fp32 accumulate)
        C[i,j] = sum_d X[i,d] * X[j,d]

## API (`kernel.h`)
void gram_gemm_launch(const __half* X, __half* C, int N, int D,
                      cudaStream_t stream);

## Dimensions
Ten workloads (`quick` = first 3). Rows marked (correctness-only) pass/fail
only and are excluded from the speed geomean:
  n1024_d1024   N=1024   D=1024   (correctness-only)
  n8192_d4096   N=8192   D=4096
  [... 8 more rows ...]

## Correctness
`reference.check` is `err_ratio_check(ref, cand, ratio=5e-3)` -- an
RMS-relative bar [...]

> Your **scoring rule** (speed `mean_vs_base` vs correctness-only) and any
> **DSL requirement** are set by your task prompt / seed track, not here.
```

The exploration task description. The task description below is appended as shown to the prompt of every exploration episode; it is not a file in the tree. It names no technique. Its document-index line lists the seven reference documents equally, a plain list rather than a hint toward any one of them, and it is the only index that mentions document 06 (an earlier version without this line had a read rate of 0/13 for that document). Everything else in the prompt is the standard orientation text (goal, banned APIs, tool conventions):

Exploration task description (appended to the episode prompt)

```
ASSIGNED OBJECTIVE for this episode - TECHNIQUE-GAP EXPLORATION: the technique is NOT named for you;
identifying it is part of the objective. First STUDY your seed kernel and the reference docs. The reference
docs live in reference/ptx/docs/: 00_tcgen05_isa, 01_descriptors, 02_tmem, 03_operand_staging,
04_cta_pair_2sm, 05_basic_tcgen05_kernel, 06_warp_specialization. Determine what the CURRENT kernel's
bottleneck is and which documented technique it is MISSING that addresses that bottleneck. Then introduce
ONE genuinely new [...] technique that closes that gap. This objective is the SUCCESS CRITERION of this
episode: a correct kernel that newly introduces such a technique is a SUCCESS at any speed - then keep
improving it. A faster kernel with NO new technique is a FAILURE of this episode regardless of its speed,
and so is a vestigial or cosmetic introduction - the new mechanism must carry the main loop's real work on
the live path, not appear once on a corner case. Correctness first: get one minimal instance of the new
technique working end to end, then widen. Decompose and persist; do not abandon the objective for smaller
tweaks. State in a comment at the top of kernel.cu WHICH technique you introduced and WHICH bottleneck it
addresses. Submit whichever passing version measures best.
```

The documentation. The seven documents under `reference/ptx/docs/` are the environment knowledge itself: roughly 2,450 lines of ISA reference and technique material, written for this suite from official vendor sources, containing no task-specific code (every document opens with a variant of the disclaimer that it is a reference, *not* a task solution). Three excerpts show their character. Documents 00 through 04 are mechanics references: raw PTX instruction semantics in the ISA's own vocabulary of elected lanes, descriptors, tensor memory, and mbarrier phases. This is document 00 on the completion handshake that every asynchronous MMA needs:

reference/ptx/docs/00_tcgen05_isa.md, Sec. 3.1 (excerpt)

```
### 3.1 `tcgen05.commit` -- arrive-on-mbarrier

Because `tcgen05.mma` is async, you cannot just `__syncthreads()` and
read TMEM. Instead: after all MMAs, issue `tcgen05.commit`, which tells
the tensor core "when every async MMA I've issued so far is done,
**arrive** on this mbarrier." Emit it from the same single elected lane
that issued the MMA:

if (elect_one()) {
  asm volatile(
    "tcgen05.commit.cta_group::1.mbarrier::arrive::one.shared::cluster.b64 [%0];"
    :: "r"(bar_intptr) : "memory");
}
```

Document 06 is the nearest thing to a teaching document: it explains one structural technique, the producer/consumer warp split, as an idiom on a neutral skeleton rather than as a solution to any task:

reference/ptx/docs/06_warp_specialization.md (excerpt)

```
**Warp specialization** splits the CTA into two disjoint groups with
different jobs, running concurrently:

* **producer** warp(s) -- do nothing but staging: they run **ahead** of
  the compute, filling a ring of `NSTAGE >= 2` SMEM buffers (via TMA,
  one elected lane per copy, or `cp.async`).
* **consumer** warps -- do nothing but compute: they wait per stage for
  "tile ready", consume it, and hand the buffer back.
[...]
The payoff is overlap: while consumers crunch tile `t`, the producer is
already streaming tiles `t+1 ... t+NSTAGE-1`.
```

No document attaches a payoff to a technique on any graded task. Document 06 explains *why* the split can help (overlap) but states no speedup numbers, and its closing sections tell the reader to specialize only "once there is measured overlap left to claim" and to diagnose first-draft regressions rather than assume a gain. Whether a documented mechanism is worth adopting on a given task is left to measurement. The documentation describes the mechanism, and the reward measures its value.

The naive seed kernel. Every performance episode starts from the task's naive seed, a correct scalar kernel with one thread per output element and no tiling, shared memory, or tensor-core use. The seed for `gram_gemm` is reproduced below in full; its header comment is part of what the agent reads.

tasks/gram_gemm/kernel.cu: naive seed of a performance episode (complete)

```
/*
 * gram_gemm -- C = X @ X^T. EDIT THIS FILE to optimize.
 *
 * Naive seed: one thread per C[i,j], dot product over D. Beat it: tile over
 * D into shared memory, exploit symmetry, use Tensor Cores.
 */
#include "kernel.h"
#include "cuda_common.cuh"

namespace {

__global__ void gram_gemm_naive(const __half* __restrict__ X,
                                __half* __restrict__ C, int N, int D) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= N || j >= N) return;
  const __half* xi = X + (size_t)i * D;
  const __half* xj = X + (size_t)j * D;
  float acc = 0.0f;
  for (int d = 0; d < D; ++d)
    acc += __half2float(xi[d]) * __half2float(xj[d]);
  C[(size_t)i * N + j] = __float2half(acc);
}

}  // namespace

void gram_gemm_launch(const __half* X, __half* C, int N, int D,
                      cudaStream_t stream) {
  dim3 block(16, 16);
```

```
  dim3 grid(klib::ceil_div(N, block.x), klib::ceil_div(N, block.y));
  gram_gemm_naive<<<grid, block, 0, stream>>>(X, C, N, D);
}
```

D EXAMPLE TRAJECTORIES: REPAIRING AND ABANDONING TMA

The two excerpts below are performance episodes that both try TMA staging and fail at first. The RLEK continuation episode on `gram_gemm` meets a compile error (a `__grid_constant__` parameter without `const`), a launch failure from the default 48 KB shared-memory cap, and wrong results from a wrong descriptor advance, fixes each in turn, and passes the in-sandbox quick evaluation (3 of 10 workloads) at step 106. Its relative speed on that evaluation then rises from 0.057 to 0.19 once both operands load through TMA. The vanilla RL episode on `gemm_rs` meets launch failures, a misaligned address, and wrong results across four TMA variants, then copies back its best kernel without TMA and resumes tuning tile sizes. The agent's messages and the tool output are trimmed, and step numbers are in brackets.

RLEK continuation, performance episode on `gram_gemm`: repeated TMA failures, repaired (reward 0.27)

```
[53]  agent: Let me try TMA staging - the biggest potential win. First, let me check the TMA helper:
[63]  build: error: a parameter annotated with __grid_constant__ must have const-qualified type
[64]  agent: Need const on __grid_constant__ params:
[84]  agent: [...] I'll implement TMA staging for operand A (K-contiguous) while keeping scalar
      staging for B - one structural change at a time. [...]
[94]  eval:  CUDA error: invalid argument
[95]  agent: The "invalid argument" is likely the 48KB shared memory cap. [...] I need to add
      `cudaFuncSetAttribute` to increase the cap:
[100] eval:  n8192_d4096 | max_err 1.09e+00 | FAIL   [...]
[104] agent: Still wrong numbers. The B descriptor advance is wrong for BLK_K=64 [...] +16, not +8:
[106] eval:  passed 3/3 workloads
[107] agent: ALL 3/3 PASS! mean_vs_base=0.0566 (up from 0.043). Submit immediately, then optimize:
[114] agent: HUGE jump to 0.1910! Both TMA loads working perfectly. Submit now:
```

Vanilla RL, performance episode on `gemm_rs`: TMA variants fail, TMA abandoned (reward 0.04)

```
[74]  agent: The profiler confirms TMA staging is the right optimization [...] Let me implement
      TMA staging:
[76]  write: kernel.cu  "TMA staging for A (K-major) and B (MN-major), dual mbarrier."
[78]  eval:  CUDA error: unspecified launch failure
[80]  write: kernel.cu  "Single mbarrier: TMA_A -> wait -> TMA_B -> wait -> MMA_commit -> wait"
[81]  eval:  CUDA error: unspecified launch failure    passed 0/3 workloads
[86]  write: kernel.cu  "TMA staging for A, hand-staging for B"
[87]  eval:  CUDA error: misaligned address
[88]  write: kernel.cu  "SMEM alignment: sA must be 128-byte aligned for sw128 TMA dest."
[90]  eval:  passed 0/3 workloads
[93]  $ cp tasks/gemm_rs/best_p256.cu tasks/gemm_rs/kernel.cu
[94]  write: kernel.cu  "BLK_M=128, BLK_N=256, BLK_K=16, TPB=256"   (no TMA; scalar tuning resumes)
```

E THE BEST RLEK-107 KERNEL ON NSA_ATTN

`kernel.cu`: best RLEK-107 submission on `nsa_attn` (complete)

```
/*
 * nsa_attn -- K1 with tcgen05 tensor cores for q@k^T computation.
 * BLK_M=128, BLK_N=128, BLK_K=16. Bf16 inputs, fp32 accumulate.
 * Epilogue: ALL warps execute tcgen05.ld (warp-collective), store guarded by bounds.
 * K2 unchanged from v1.
 */
#include "kernel.h"
#include "cuda_common.cuh"
#include "tcgen05_desc.cuh"
#include "tcgen05_sync.cuh"
#include <cuda_fp16.h>

namespace {

constexpr int MAX_NB = 192;
constexpr int BM = 128, BN = 256, BK = 16;

__global__ void init_imp_kernel(float* imp, int B, int H, int S, int nb, int bs) {
    size_t gid = (size_t)blockIdx.x * blockDim.x + threadIdx.x;
```

```
    size_t total = (size_t)B * H * S * nb;
    if (gid < total) {
        int bi = gid % nb;
        size_t rest = gid / nb;
        int t = rest % S;
        imp[gid] = (bi * bs > t) ? -INFINITY : 0.0f;
    }
}

__global__ void __launch_bounds__(128) nsa_block_imp_tcgen05(
    const __nv_bfloat16* __restrict__ q,
    const __nv_bfloat16* __restrict__ k,
    float* __restrict__ imp,
    int B, int H, int S, int D, int bs, int nb,
    float scale) {

    extern __shared__ __nv_bfloat16 st[];
    __nv_bfloat16* sA = st;
    __nv_bfloat16* sB = st + BM * BK;

    __shared__ uint64_t bar;
    __shared__ uint32_t tmem_base_slot;

    int tx = threadIdx.x;
    if (tx == 0) mbar_init(&bar, 1);
    __syncthreads();

    uint32_t tmem_c = tcgen05_alloc_all(&tmem_base_slot, BN);
    uint32_t phase = 0;

    int bh = blockIdx.y;
    int mq = blockIdx.z;
    int nk = blockIdx.x;

    uint32_t idesc = make_instr_desc(BM, BN, 1, 1, 1, 0, 0);

    int k_tiles = (D + BK - 1) / BK;

    #pragma unroll
    for (int kk = 0; kk < k_tiles; kk++) {
        __syncthreads();
        int k0 = kk * BK;

        // Stage A: q tile [BM, BK], K-major
        #pragma unroll
        for (int i = tx; i < BM * BK; i += 128) {
            int m = i / BK;
            int kk_local = i % BK;
            int global_m = mq * BM + m;
            int global_k = k0 + kk_local;
            if (global_m < S && global_k < D) {
                sA[tile_idx_kmajor_f16(m, kk_local, BK)] = q[((size_t)bh * S + global_m) * D + global_k];
            } else {
                sA[tile_idx_kmajor_f16(m, kk_local, BK)] = __float2bfloat16(0.0f);
            }
        }

        // Stage B: k tile [BN, BK], K-major
        #pragma unroll
        for (int i = tx; i < BN * BK; i += 128) {
            int nn = i / BK;
            int kk_local = i % BK;
            int global_n = nk * BN + nn;
            int global_k = k0 + kk_local;
            if (global_n < S && global_k < D) {
                sB[tile_idx_kmajor_f16(nn, kk_local, BK)] = k[((size_t)bh * S + global_n) * D + global_k
    ↪ ];
            } else {
                sB[tile_idx_kmajor_f16(nn, kk_local, BK)] = __float2bfloat16(0.0f);
            }
        }

        __syncthreads();

        uint64_t da = make_smem_desc_kmajor_f16(sA, BK);
        uint64_t db = make_smem_desc_kmajor_f16(sB, BK);
        uint32_t sc = (kk == 0) ? 0u : 1u;

        tcgen05_mma_f16_commit_wait(&bar, &phase, tx == 0, tmem_c, da, db, idesc, sc);
```

```
    }

    // Epilogue: ALL warps execute tcgen05.ld (warp-collective requirement)
    int warp_id = tx / 32;
    int lane = tx % 32;
    int my_row = warp_id * 32 + lane;
    int global_t = mq * BM + my_row;

    float sum_local[4] = {0,0,0,0};
    int cnt_local[4] = {0,0,0,0};
    int nk0 = nk * BN;
    int blocks_per_tile = BN / bs; // 4 with BN=256, bs=64

    // ALL warps read from TMEM
    for (int col0 = 0; col0 < BN; col0 += 4) {
        uint32_t r[4];
        tcgen05_ld_32x32b_x4(tmem_c, warp_id, col0, r);
        if (my_row < BM && global_t < S && nk0 < S) {
            for (int j = 0; j < 4; j++) {
                float c_val = __int_as_float(r[j]);
                int global_j = nk0 + col0 + j;
                if (global_j <= global_t && global_j < S) {
                    int bi_rel = (col0 + j) / bs;
                    if (bi_rel < blocks_per_tile) {
                        sum_local[bi_rel] += c_val;
                        cnt_local[bi_rel]++;
                    }
                }
            }
        }
    }

    // Store block importances
    if (my_row < BM && global_t < S) {
        int bi0 = nk0 / bs;
        for (int b = 0; b < blocks_per_tile; b++) {
            int bi = bi0 + b;
            if (cnt_local[b] > 0 && bi >= 0 && bi < nb && bi * bs <= global_t) {
                imp[((size_t)bh * S + global_t) * nb + bi] = sum_local[b] * scale / (float)cnt_local[b];
            }
        }
    }

    tcgen05_dealloc_all(tmem_c, BN);
}

__global__ void nsa_attend(const __nv_bfloat16* __restrict__ q,
                           const __nv_bfloat16* __restrict__ k,
                           const __nv_bfloat16* __restrict__ v,
                           const float* __restrict__ imp,
                           __nv_bfloat16* __restrict__ o,
                           int B, int H, int S, int D, int bs, int top_n,
                           int win, int nb, float scale) {
  int warp = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
  int lane = threadIdx.x % 32;
  size_t total = (size_t)B * H * S;
  if ((size_t)warp >= total) return;
  int t = warp % S;
  size_t bh = (size_t)warp / S;

  __shared__ int lists[4][MAX_NB / 8 + 4];
  __shared__ int flags[4][MAX_NB / 8 + 4];
  int wslot = (threadIdx.x / 32) % 4;
  int* list = lists[wslot];
  int* full = flags[wslot];
  int nlist = 0;

  if (lane == 0) {
    const float* im = imp + (bh * S + t) * nb;
    int ncb = t / bs + 1;
    unsigned long long picked[MAX_NB / 64 + 1] = {0ull};
    int npick = min(top_n, ncb);
    for (int p = 0; p < npick; ++p) {
      float best = -INFINITY;
      int arg = -1;
      for (int bi = 0; bi < ncb; ++bi) {
        if (picked[bi >> 6] & (1ull << (bi & 63))) continue;
        if (im[bi] > best) { best = im[bi]; arg = bi; }
      }
```

```
      if (arg < 0) break;
      picked[arg >> 6] |= 1ull << (arg & 63);
      list[nlist] = arg;
      full[nlist++] = 1;
    }
    int w0 = max(0, t + 1 - win);
    for (int bi = w0 / bs; bi <= t / bs; ++bi) {
      bool have = false;
      for (int i = 0; i < nlist; ++i)
        if (list[i] == bi) { have = true; break; }
      if (!have) { list[nlist] = bi; full[nlist++] = 0; }
    }
    list[MAX_NB / 8 + 3] = nlist;
  }
  __syncwarp();
  nlist = __shfl_sync(0xffffffffu, lane == 0 ? nlist : 0, 0);

  const __nv_bfloat16* qp = q + (bh * S + t) * D;
  float qreg[4];
  const __nv_bfloat162* qp2 = reinterpret_cast<const __nv_bfloat162*>(qp);
  int d16 = (D + 1) / 2;
  #pragma unroll
  for (int i = 0; i < 2; ++i) {
    int d = lane + 32 * i;
    if (d < d16) {
      __nv_bfloat162 qa = qp2[d];
      float2 qf = __bfloat1622float2(qa);
      int d0 = d * 2;
      if (d0 < D) qreg[i * 2] = qf.x; else qreg[i * 2] = 0.0f;
      if (d0 + 1 < D) qreg[i * 2 + 1] = qf.y; else qreg[i * 2 + 1] = 0.0f;
    } else {
      qreg[i * 2] = 0.0f;
      qreg[i * 2 + 1] = 0.0f;
    }
  }
  float m = -INFINITY, denom = 0.0f, acc[4] = {0, 0, 0, 0};
  int w0 = max(0, t + 1 - win);
  for (int li = 0; li < nlist; ++li) {
    int bi = list[li];
    int j0 = bi * bs, j1 = min(min((bi + 1) * bs, t + 1), S);
    if (!full[li]) j0 = max(j0, w0);
    for (int j = j0; j < j1; ++j) {
      const __nv_bfloat162* kp2 = reinterpret_cast<const __nv_bfloat162*>(k + (bh * S + j) * D);
      float dot = 0.0f;
      #pragma unroll
      for (int i = 0; i < 2; ++i) {
        int d = lane + 32 * i;
        if (d < d16) {
          __nv_bfloat162 ka = kp2[d];
          float2 kf = __bfloat1622float2(ka);
          int d0 = d * 2;
          if (d0 < D) dot += qreg[i * 2] * kf.x;
          if (d0 + 1 < D) dot += qreg[i * 2 + 1] * kf.y;
        }
      }
      for (int s = 16; s > 0; s >>= 1)
        dot += __shfl_down_sync(0xffffffffu, dot, s);
      dot = __shfl_sync(0xffffffffu, dot, 0) * scale;
      float mn = fmaxf(m, dot);
      float corr = __expf(m - mn);
      float p = __expf(dot - mn);
      denom = denom * corr + p;
      const __nv_bfloat162* vp2 = reinterpret_cast<const __nv_bfloat162*>(v + (bh * S + j) * D);
      #pragma unroll
      for (int i = 0; i < 2; ++i) {
        int d = lane + 32 * i;
        if (d < d16) {
          __nv_bfloat162 va = vp2[d];
          float2 vf = __bfloat1622float2(va);
          int d0 = d * 2;
          if (d0 < D) acc[i * 2] = acc[i * 2] * corr + p * vf.x;
          if (d0 + 1 < D) acc[i * 2 + 1] = acc[i * 2 + 1] * corr + p * vf.y;
        }
      }
      m = mn;
    }
  }
  __nv_bfloat16* op = o + (bh * S + t) * D;
  #pragma unroll
```

```
  for (int i = 0; i < 2; ++i) {
    int d = lane + 32 * i;
    int d0 = d * 2;
    if (d0 < D) op[d0] = __float2bfloat16(acc[i * 2] / denom);
    if (d0 + 1 < D) op[d0 + 1] = __float2bfloat16(acc[i * 2 + 1] / denom);
  }
}

}  // namespace

void nsa_attn_launch(const __nv_bfloat16* q, const __nv_bfloat16* k,
                     const __nv_bfloat16* v, __nv_bfloat16* o,
                     int B, int H, int S, int D, int block_size, int top_n,
                     int window, cudaStream_t stream) {
  int nb = klib::ceil_div(S, block_size);
  float scale = rsqrtf((float)D);
  float* imp = nullptr;
  cudaMallocAsync(&imp, (size_t)B * H * S * nb * sizeof(float), stream);
  // Initialize imp: -INFINITY for blocks past t, 0 for valid blocks (tcgen05 overwrites)
  {
      size_t total_imp = (size_t)B * H * S * nb;
      int blocks_imp = klib::ceil_div(total_imp, 256);
      init_imp_kernel<<<blocks_imp, 256, 0, stream>>>(imp, B, H, S, nb, block_size);
  }

  size_t shmem = ((size_t)BM * BK + (size_t)BN * BK) * sizeof(__nv_bfloat16);
  int nk_tiles = klib::ceil_div(S, BN);
  int nq_tiles = klib::ceil_div(S, BM);
  dim3 grid(nk_tiles, B * H, nq_tiles);

  nsa_block_imp_tcgen05<<<grid, 128, shmem, stream>>>(
      q, k, imp, B, H, S, D, block_size, nb, scale);

  int warps = B * H * S;
  nsa_attend<<<klib::ceil_div(warps, 4), 128, 0, stream>>>(
      q, k, v, imp, o, B, H, S, D, block_size, top_n, window, nb, scale);
  cudaFreeAsync(imp, stream);
}
```

F AGENT HARNESS, AUTO-COMPACTION, AND TRAINING INFRASTRUCTURE

This appendix describes the agentic RL framework behind the reported runs: the agent harness and its model-server adapter (Appendix F.1), auto-compaction with a policy-written summary (Appendix F.2), the asynchronous trainer (Appendix F.3), and the sandbox and grading services (Appendix F.4). Figure 6 shows the inside of one episode.

F.1 THE AGENT HARNESS

The loop. An episode is a coding session driven by a fork of the Codex CLI agent, controlled over JSON-RPC by an episode driver inside the rollout server. The agent has two tools, a shell and a patch-application tool, and every command runs in the episode's sandbox on a Blackwell GPU. The agent never waits for approval, so the episode is fully autonomous. At the end of the episode, the last solution the agent submitted with the submission tool is sent to the reward server for grading (Section 4.1), and an episode that never submits scores 0.

Token-faithful prefix rendering. The adapter exposes the OpenAI Responses API to the agent and exchanges token ids, never text, with the model server. Multi-turn rollouts are assembled *token-in-token-out* (Figure 6). Within a segment, turn u has prompt x_u, sampled token ids y_u, and tool output o_u, and

$$x_{u+1} \;=\; x_u \oplus y_u \oplus \mathrm{tok}(o_u), \qquad z \;=\; x_1 \oplus y_1 \oplus \mathrm{tok}(o_1) \oplus y_2 \oplus \cdots, \tag{5}$$

where $\oplus$ concatenates token ids and $\mathrm{tok}(\cdot)$ tokenizes only the new content between sampled turns. The trainer stores the segment as the flat sequence z with the loss mask of Section 3.1 equal to one exactly on the tokens of the y_u, so the tokens the update conditions on are identical to the tokens that were sampled. Re-rendering the conversation through the chat template each turn would break this identity in two ways: the template re-tokenizes model output, and for a reasoning model the re-rendered prefix omits reasoning tokens, because OpenAI-compatible clients drop reasoning content when they echo an assistant message and the Qwen3.x template strips historical `<think>` blocks. The adapter therefore keeps each turn's reasoning and re-injects it at render time.

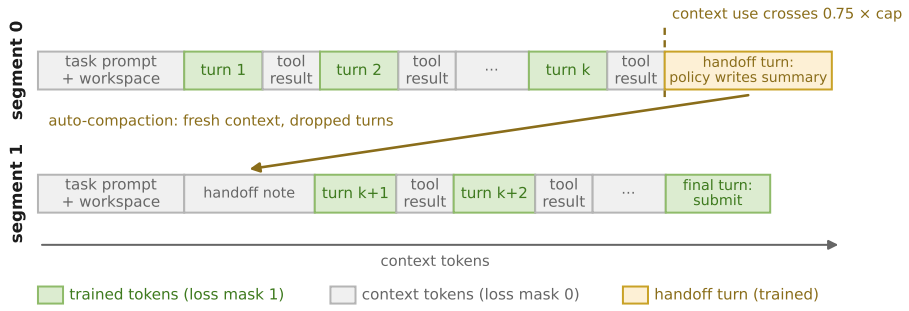


Figure 6: Inside one episode. Green spans are the tokens the policy sampled and the trainer optimizes; gray spans are carried context the loss masks out. Each turn's prompt reuses the previous turn's raw sampled ids unchanged, so the trained prefix equals what was sampled. When context use crosses the trigger the policy writes its own handoff, segment 0 is stored, and segment 1 restarts from the task prefix plus that handoff over an untouched sandbox.

F.2 Auto-compaction

Episodes routinely exceed their context window. Instead of truncating the history or summarizing it with a separate model, the harness has *the policy itself* write the handoff and trains on that writing.

Trigger and handoff. Before a turn whose prompt would reach 0.75 of the 100k-token context cap, the harness asks the model for a handoff, at most once per episode. The instruction says that everything except the task prompt and the handoff is about to be erased while the sandbox is kept, and asks for at most 60 lines under three headings. STATE records what works, which file holds the best version, and any passing result better than the submitted one, since an unsubmitted passing kernel scores zero. LEARNINGS labels each failure as measured-slower or abandoned, so an untested idea is not recorded as failed. NEXT STEPS lists the most immediate step first.

The new segment. Everything up to and including the handoff turn is stored as one training segment (Figure 6). The new context is the system and task prefix plus the handoff, with an instruction to reconcile the live kernel against the submitted one before anything else. Because the sandbox persists, the handoff need not carry state that is still on disk, only the agent's interpretation of it. If a single turn runs into the context limit before the trigger fires and the episode has not yet compacted, the truncated turn is discarded without being trained on, compaction is triggered at once, and the turn is sampled again from the post-compaction context.

What is trained. The loss mask of Section 3.1 includes reasoning blocks and turn terminators. The handoff turn is an ordinary sampled turn and is trained like any other, which makes writing a good handoff a learnable behavior. Every segment of an episode receives the episode's advantage.

F.3 The trainer

Fully asynchronous multi-turn RL training. Training runs on a fork of the `slime` trainer (Zhu et al., 2025). The rollout server runs inside the trainer process, so episodes are produced, graded, and consumed as a stream with no batch boundary: generation for later steps overlaps the grading of earlier ones and the optimizer step itself, and the resulting off-policyness is corrected as in Eq. 2.

F.4 Sandbox and grading services

The trainer, the sandbox fleet, and the grading service are separate and share only HTTP endpoints and a pinned code revision.

Sandboxes. The concurrent sessions on a sandbox worker outnumber its GPUs, so a command that needs a GPU, such as a build or an evaluation, waits for one in a lock-based queue and releases it when it finishes, which works because a session spends most of its time editing and reading. Per-command timeouts are capped on the server, so an agent cannot raise its own limit. The operating

system enforces the isolation: Linux Landlock restricts file-system access, and a separate network namespace leaves the workspace without network access. Third-party headers that the sandbox may read are pinned and stripped of their complete kernel implementations, so no human-written kernel for these tasks is reachable from an episode.

Grading. The grading service runs on its own GPUs and owns everything the sandbox must not see: the trusted task definitions and graded workloads, the forbidden-API scanner, the judges, the scorer, and the solution buffer. The agent-side copy of the evaluator is stripped of these modules, and each worker verifies at boot that they cannot be imported. Grading runs one evaluation per GPU, because two evaluations on one device would corrupt the latency the reward is built on, and a submission byte-identical to its starting solution is rejected before any GPU is used.