COMPACTION-AWARE REINFORCEMENT LEARNING FOR ASCEND TRITON KERNEL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Automatically generating high-performance kernels with LLM-based agents promises substantial gains in development efficiency, but two obstacles remain. First, while increasing multi-turn interaction with execution environments is crucial for refining generation quality, continuous context accumulation rapidly exceeds the model's effective context window. Second, training long-horizon tasks via GRPO inherently incurs sparse rewards due to its reliance on terminal outcome evaluation, severely hindering step-level credit assignment; To address these obstacles, we propose ATRL (Ascend Triton Reinforcement Learning), a post-training method that refines GRPO into a compaction-aware, segment-level objective for long-horizon kernel generation. We also design a lightweight agent harness providing cascaded compilation, correctness, and profiling feedback, alongside an Ascend-specific knowledge base injected during agentic RL training and evaluation to compensate for the base model's missing platform expertise. Evaluations on NPUKernelBench and KernelBench, both executed on Ascend hardware, show that ATRL consistently surpasses all baselines, including GRPO, in both correctness and speedup.

1 INTRODUCTION

High-performance kernels are critically important in LLMs infrastructure, as they directly accelerate both the training and inference pipelines of LLMs. Recent advances in large language models (LLMs) have opened a promising path toward automated kernel generation. A growing body of work trains models to translate high-level PyTorch reference implementations into optimized kernels in domain-specific languages (DSLs) such as Triton Tillet et al. (2019), whose tile-based programming model simplifies kernel development and substantially improves developer productivity Guo et al. (2026); Liu et al. (2026). To further improve generation quality, researchers have adopted an agentic paradigm: instead of producing a kernel through a single-round chat interaction, an LLM-based agent interacts with an execution environment across multiple turns–generating code, receiving compilation and correctness feedback, profiling performance, and iteratively refining its output Dai et al. (2026); Du et al. (2026b); Lange et al. (2025). This multi-turn, tool-augmented workflow has proven substantially more effective than single-round generation, as it enables the model to learn from execution feedback and explore a broader optimization landscape.

Despite this progress, two fundamental challenges remain unresolved. First, the multi-turn agentic workflow inherently causes context to grow continuously: each turn appends tool results like view files, steadily inflating the input length. As recent workload characterizations have shown, this unbounded context growth not only degrades the quality of model inference, but also risks exceeding the effective context window, terminating the optimization loop prematurely Yuan et al. (2026); Li et al. (2026). Existing kernel-generation agents Dai et al. (2026); Liu et al. (2026) either ignore this problem entirely or apply ad hoc truncation that discards potentially valuable debugging history, yielding no principled solution. Second, training long-horizon kernel generation with GRPO relies on a terminal outcome reward. Because credit is assigned only after the full trajectory finishes, intermediate edits receive little supervision, the learning signal remains sparse, and step-level credit assignment becomes unreliable. Standard GRPO therefore struggles to distinguish a useful repair from a later regression across many edit–verify turns.

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.

To address these challenges, we propose **ATRL** (Ascend Triton Reinforcement Learning), a post-training method that refines GRPO for long-horizon kernel generation. ATRL decomposes a long agent trajectory into segments at verifiable boundaries and optimizes the policy with segment-level rewards built on three ideas: (i) *structured trajectory compaction*, which condenses the accumulated history into a compact checkpoint once the context grows too long; (ii) *mixed segment reward*, which combines the terminal task success with the verifiable progress measured at each segment's boundaries; and (iii) *task-relative segment advantage*, which normalizes segment advantages across rollouts of the same task and weights them segment-wise in the loss. The compaction checkpoint itself participates in training, so the model learns both to preserve state and to continue from it. Throughout, a lightweight agent harness supplies compilation, correctness, and profiling feedback, and an on-demand knowledge base provides Ascend-specific optimization guidance.

2 Related Work

2.1 LLM-Based Kernel Generation

Training-free agentic generation. Several frameworks employ LLM agents that iteratively refine kernels through execution feedback without modifying model weights. AdaExplore Du et al. (2026b) introduces failure-driven adaptation, synthesizing recurring errors into reusable validity rules, combined with diversity-preserving tree search to escape local optima. The robust-kbench framework Lange et al. (2025) proposes an evolutionary meta-generation procedure with LLM-based verifiers for correctness filtering, while also exposing exploitable benchmark loopholes. MKEvolve Yoo et al. (2026) adopts a modular decomposition strategy, iteratively co-evolving sub-kernel decompositions and their implementations to improve interpretability and reduce token consumption. While effective at test time, these approaches do not improve the model's intrinsic generation ability and incur substantial inference cost per task.

Training-based generation. DRTriton Guo et al. (2026) proposes a synthetic data pipeline (CSP-DAG) with curriculum RL to train models for PyTorch-to-Triton conversion, achieving strong generalization despite training only on synthetic data. Dr. Kernel Liu et al. (2026) introduces KernelGYM, a distributed GPU environment with reward-hacking detection, and identifies a biased policy-gradient issue in GRPO, proposing Turn-level Reinforce-Leave-One-Out (TRLOO) for unbiased multi-turn advantage estimation. MusaCoder Cheng et al. (2026) presents an end-to-end alignment pipeline for Moore Threads MUSA kernels. It proceeds from SFT through diversity-preserving RFT to two-stage RL, and introduces PrimeEcho, MirrorPop, and Buffered Dynamic Retry to stabilize training. Kernel-Smith Du et al. (2026a) combines an evolutionary agent with post-training RL, converting long-horizon trajectories into step-centric supervision, and validates cross-backend transfer between NVIDIA Triton and MetaX MACA. CUDA-Agent Dai et al. (2026) scales agentic RL with a skill-augmented environment and automated verification/profiling, achieving state-of-the-art results on KernelBench. Compiler-R1 Pan et al. (2025) applies RL to compiler pass-sequence auto-tuning, demonstrating that RL-trained LLMs can optimize IR-level metrics.

Despite this rapid progress, existing work is overwhelmingly centered on NVIDIA CUDA/Triton or, more recently, other GPU backends (MetaX MACA, Moore Threads MUSA). *None of these methods address Ascend kernel generation*, where the Da Vinci architecture introduces fundamentally different compute and memory semantics. Moreover, most approaches treat the multi-turn execution environment as a black box for reward computation, without exploiting the transparency of intermediate verification steps for fine-grained credit assignment. Our work fills both gaps by targeting the Ascend Triton ecosystem and adopting a white-box RL formulation.

2.2 Agentic RL

Agentic Reinforcement Learning. Agentic RL trains language models to improve through multi-turn interaction with tools and stateful environments, rather than optimizing isolated responses. ArCHer establishes a hierarchical actor-critic formulation that combines turn-level value learning with token-level policy optimization (Zhou et al., 2024). WebRL further addresses task scarcity, sparse feedback, and policy drift in web environments using a self-evolving curriculum and outcome-supervised rewards (Qi et al., 2025). For long-horizon digital agents, LOOP eliminates the learned critic through a leave-one-out baseline, enabling memory-efficient policy optimization di-

rectly in target environments (Chen et al., 2025). RAGEN instead studies the dynamics of multi-turn RL and identifies the Echo Trap, in which declining reward variance and policy entropy destabilize learning, motivating its stabilized StarPO-S algorithm (Wang et al., 2025).

Recent studies address delayed credit assignment and unstable exploration through hierarchical or step-aware policy optimization. GiGPO combines episode-level advantages with step-level comparisons among actions from equivalent states (Feng et al., 2025). Agent Lightning decomposes arbitrary agent executions into trainable transitions through hierarchical credit assignment (Luo et al., 2025), while AgentGym-RL progressively increases interaction horizons to stabilize long-horizon training (Xi et al., 2025). However, these methods are not readily applicable to Ascend kernel generation, where code states rarely recur and rewards are inherently sparse. In contrast, our framework leverages compiler, correctness, and performance feedback to optimize long-horizon kernel-generation trajectories.

3 PRELIMINARY

3.1 AGENTIC ASCEND TRITON KERNEL GENERATION

An Ascend Triton kernel-generation task starts from a reference PyTorch implementation and a set of test inputs covering the required shapes and data types. The goal is to produce a Triton kernel that matches the reference outputs while running efficiently on the Ascend platform. Triton expresses the kernel as a collection of parallel tile programs Tillet et al. (2019), which the Ascend backend compiles into an executable kernel.

Producing a correct and efficient kernel usually requires several attempts. The agent writes or modifies the kernel, invokes a unified evaluation tool, and revises the implementation based on the structured report it returns—compiler diagnostics, correctness results, and performance measurements. This consolidated feedback makes the environment white-box, allowing training to distinguish compilation failures, numerical errors, and performance bottlenecks rather than relying only on final success; the full verification cascade is detailed in Section 4.1 and Appendix B.

Multi-turn optimization also creates an increasingly long interaction history. Source-code snapshots, compiler logs, correctness reports, and profiling results accumulate even after they are no longer useful. A long history increases inference cost, distracts the model with stale information, and may eventually exhaust its context window. Effective context management is therefore necessary to sustain long-horizon kernel optimization.

3.2 GROUP RELATIVE POLICY OPTIMIZATION

Group Relative Policy Optimization (GRPO) Shao et al. (2024) is an online policy optimization algorithm that avoids training a separate value model. For each task x, it samples a group of G trajectories $\{\tau_i\}_{i=1}^{G}$ from the old policy. Let $R_i = R(x, \tau_i)$ be the scalar reward of trajectory τ_i. Its group-relative advantage is

$$\widehat{A}_i = \frac{R_i - \mu_R}{\sigma_R}, \qquad \mu_R = \frac{1}{G}\sum_{j=1}^{G} R_j, \tag{1}$$

where σ_R is the standard deviation of the group rewards. This group-relative baseline removes the critic required by conventional actor–critic methods and directly compares candidate trajectories generated for the same task.

For each generated token, let $\rho_{i,t}$ be its importance ratio and define the clipped ratio as $\bar{\rho}_{i,t} = \mathrm{clip}(\rho_{i,t}, 1-\varepsilon, 1+\varepsilon)$. The GRPO objective is

$$\mathcal{J}_{\mathrm{GRPO}}(\theta) = \mathbb{E}_{i,t}\left[\min\left(\rho_{i,t}\widehat{A}_i, \bar{\rho}_{i,t}\widehat{A}_i\right) - \beta D_{i,t}^{\mathrm{KL}}\right]. \tag{2}$$

where $D_{i,t}^{\mathrm{KL}}$ measures the divergence between the trainable policy π_θ and a frozen reference policy π_{ref}, ε is the clipping coefficient, and β controls the KL regularization strength. Following Shao et al. (2024), the divergence is estimated per token with the unbiased and nonnegative $k3$ estimator

$$D_{i,t}^{\mathrm{KL}} \approx \frac{\pi_{\mathrm{ref}}(v_{i,t})}{\pi_\theta(v_{i,t})} - \log\frac{\pi_{\mathrm{ref}}(v_{i,t})}{\pi_\theta(v_{i,t})} - 1, \tag{3}$$

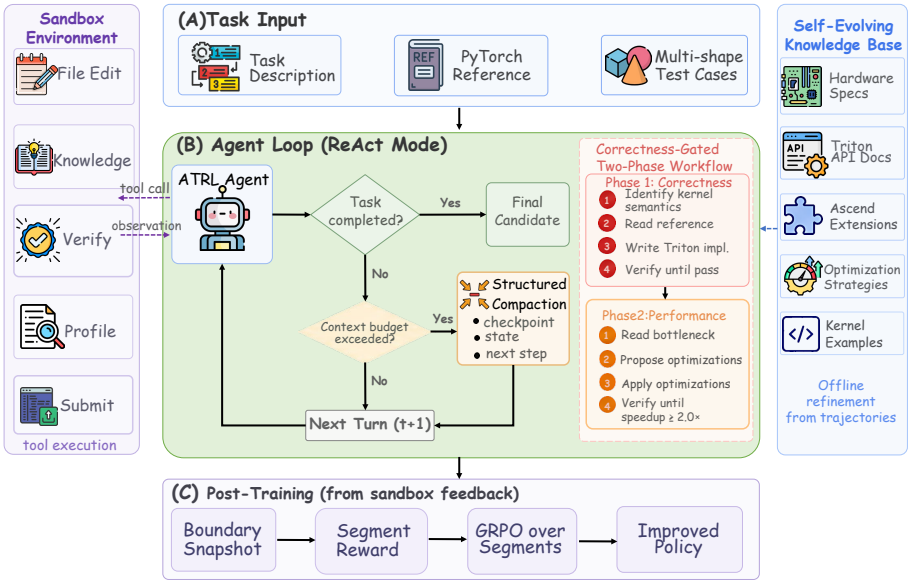

Figure 1: Method overview. The agent iteratively edits and verifies a Triton kernel in a sandbox. Long interactions are compacted into structured checkpoints, producing segments that receive both trajectory-level and incremental rewards.

where $v_{i,t}$ is the token generated at step t of trajectory τ_i and the token probabilities are conditioned on the trajectory prefix. The expectation is taken over sampled tasks, trajectories, and policy-generated tokens.

Recent variants relax the symmetric clipping interval of GRPO. In particular, DAPO Yu et al. (2025) clips the importance ratio asymmetrically, $\mathrm{clip}(\rho_{i,t}, 1-\varepsilon_l, 1+\varepsilon_h)$ with $\varepsilon_h > \varepsilon_l$. This *clip-higher* scheme enlarges the upper bound so that low-probability tokens can still raise their probability during optimization, counteracting the entropy collapse commonly observed in long reinforcement learning runs and thereby encouraging exploration. ATRL combines this asymmetric clipping with the KL-regularized GRPO objective; Section 4.3 develops the resulting segment-level formulation.

Standard outcome-supervised GRPO broadcasts the same trajectory-level advantage $\widehat{A}_i$ to every generated token in τ_i. This is problematic for long-horizon kernel generation: unsuccessful early attempts, useful diagnostic actions, and the final successful repair all receive the same credit. Moreover, if most trajectories in a group fail compilation or correctness checks, their rewards become identical or nearly identical, producing little learning signal after group normalization. These limitations motivate the trajectory decomposition and fine-grained reward assignment introduced in Section 4.

4 METHOD

We formulate Ascend Triton kernel generation as a long-horizon interaction between a language-model policy and an executable Ascend environment. Starting from a PyTorch reference implementation, the policy generates a kernel implementation, invokes verification tools, and revises the implementation using compiler, correctness, and profiling feedback. Two properties of this setting shape the design of ATRL. First, trajectories are long and grow without bound: a single kernel can require many edit–verify cycles, and the accumulated tool history eventually exceeds the model's context window. Second, the only naturally available scalar signal is the terminal outcome, which is too coarse to distinguish a useful repair from a later regression. ATRL addresses the first with a correctness-gated optimization workflow (Section 4.1) and structured trajectory compaction (Section 4.2), and the second with a compaction-aware RL objective that assigns credit to the resulting trajectory segments (Section 4.3). Throughout, an on-demand knowledge base supplies Ascend-specific programming and optimization guidance (Section 4.1).

4.1 AGENTIC KERNEL OPTIMIZATION

Interaction loop. The agent operates in an isolated sandbox and alternates between language-model generation and tool execution. Its tools support file inspection and editing, knowledge retrieval, verification, and final submission; the full tool suite and sandbox setup are given in Appendix A, and the corresponding prompt templates in Appendix E. Tool outputs are returned as observations in the next model context, allowing the policy to repair compilation errors, correct numerical mismatches, and optimize measured latency without requiring a complete kernel in a single response.

Correctness-gated optimization. We decouple correctness establishment from performance tuning by gating access to optimization guidance. During *implementation*, the agent identifies the kernel semantics from the PyTorch reference, retrieves relevant API documentation, and iterates until the candidate passes verification. During *optimization*, the agent uses profiling signals to select transformations such as tiling, multi-core partitioning, or improved memory access, while re-running correctness tests after every change. The gate prevents the policy from trading correctness for apparent speed. Near the turn limit, the harness may inject a brief reminder to create or verify an implementation; the reminder is also written to the rollout buffer so that inference and training observe the same sequence.

On-demand knowledge retrieval. The sandbox exposes a structured knowledge base of Ascend-specific guidance covering hardware constraints, Triton APIs and their Ascend extensions, example kernels, and optimization strategies, retrieved through a *judge–retrieve–apply* chain that bounds token consumption. Before writing the kernel, the policy consults hardware constraints, examples, and API documentation to ground the initial implementation; during optimization, profiling signals guide the choice among transformations. Profiling reports diagnose symptoms—for example, low vector utilization or memory-bound execution—but do not prescribe an optimization, leaving the policy to connect each observation to an appropriate transformation. The retrieval interface and corpus maintenance are described in Appendix A.3.

Cascaded verification. Each verification call executes a four-stage cascade—static analysis, compilation, numerical correctness, and hardware benchmarking—and stops at the first failed stage, so invalid candidates never reach the accelerator. The ordering is itself a safeguard: submissions that bypass the Triton kernel are rejected before any execution, and benchmarking runs only after every correctness case passes. Each stage emits a structured report that doubles as the environment observation. Stage definitions, precision criteria, and integrity safeguards are given in Appendix B.

Outcome reward. The verifier produces a scalar score that tracks progress from syntactic validity to execution efficiency:

$$R = w_{\text{ast}}\mathbf{1}_{\text{ast}} + w_{\text{cmp}}\mathbf{1}_{\text{cmp}} + w_{\text{cor}}p_{\text{cor}} + w_{\text{all}}\mathbf{1}_{\text{all}} + w_{\text{spd}}\min\Big(\frac{s}{s^\star}, 1\Big), \tag{4}$$

where $\mathbf{1}_{\text{ast}}$ and $\mathbf{1}_{\text{cmp}}$ indicate that the candidate passes static analysis and compiles, respectively, $\mathbf{1}_{\text{all}}$ indicates that all correctness cases pass, p_{cor} is the fraction of correctness cases passed, s is the measured speedup, and $s^\star$ is the target speedup. The speed term is defined only when all correctness cases pass and is zero otherwise. We use $(w_{\text{ast}}, w_{\text{cmp}}, w_{\text{cor}}, w_{\text{all}}, w_{\text{spd}}) = (0.02, 0.15, 0.60, 0.13, 0.40)$ and $s^\star = 2.0$. The maximum reward is 1.30; a fully correct implementation with no speedup receives 0.90, and fully correct rollouts therefore occupy $(0.90, 1.30]$ as speedup increases to the target. Thus, an incorrect but fast implementation cannot receive performance credit.

4.2 STRUCTURED TRAJECTORY COMPACTION

The iterative workflow above can produce trajectories that exceed the model's context window, making an explicit mechanism for preserving execution state necessary. Truncating old messages discards precisely the failures and constraints needed for subsequent repair. Instead, after a verification call and once context usage crosses a threshold, we ask the rollout model to replace the accumulated history with a structured checkpoint. The checkpoint records seven types of state: attempted

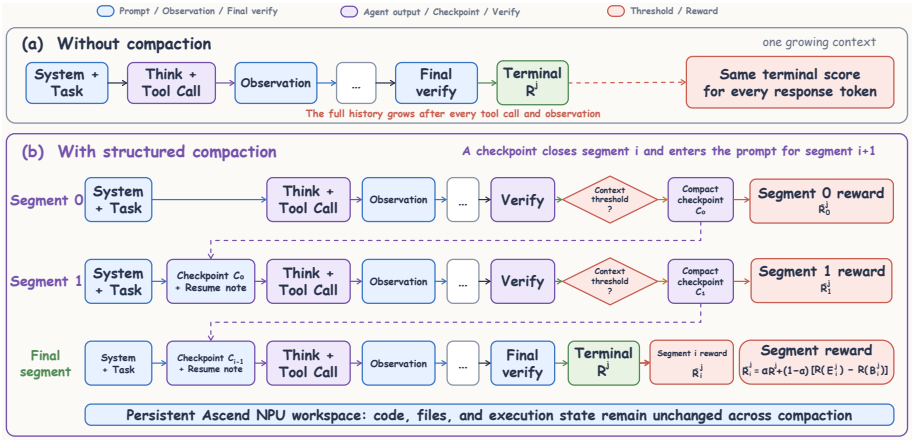

Figure 2: Compaction-aware segmentation. A checkpoint closes one training segment and initializes the prompt of the next; the sandbox workspace persists across the boundary.

approaches, discovered constraints, design decisions, errors and fixes, the current implementation, the latest verification result, and one immediate next action.

After compaction, the next context contains the unchanged system instruction and task, followed by the checkpoint and a short continuation instruction. The sandbox workspace and implementation files remain unchanged, so execution continues from the same environment state. Compaction is attempted only under conservative trigger conditions, and a checkpoint is accepted only if it is non-empty and shorter than the history it replaces.

Compaction partitions rollout j into $I_j + 1$ segments, $\tau^j = \{\tau_1^j, \ldots, \tau_{I_j+1}^j\}$. At each boundary we store the code, verification status, and performance measurements. These snapshots allow us to measure what changed within a segment. We deliberately place the generated checkpoint in two positions: it is part of the preceding segment's response, where RL can supervise its generation, and part of the following segment's prompt, where it conditions continued execution. This dual role makes checkpoint quality itself optimizable while teaching the policy to preserve the state required for later kernel revisions. Trigger thresholds, prompt format, and failure-handling details appear in Appendix C.

4.3 Compaction-Aware Reinforcement Learning

Compaction changes the unit of training from one uninterrupted trajectory to several state-dependent segments. Standard trajectory-level GRPO assigns one terminal advantage to every generated token and cannot distinguish a useful repair from a later regression. We therefore combine global task success with the measured progress of each segment.

Segment reward. Let R^j denote the score of the final verified implementation in rollout j. For segment i, let B_i^j and E_i^j denote its boundary states and define

$$r_i^j = R(E_i^j) - R(B_i^j), \qquad \widetilde{R}_i^j = \alpha R^j + (1-\alpha) r_i^j, \tag{5}$$

with $\alpha = 0.5$ and $R(B_1^j) = 0$. The first term preserves the end-to-end objective. The second attributes local progress: fixing compilation or correctness increases r_i^j, whereas breaking a previously correct kernel or reducing its speed makes r_i^j negative. Both terms use Equation 4, so they have the same scale. Setting $\alpha = 1$ makes every segment receive the same terminal reward R^j, reducing the signal to the terminal outcome without per-segment progress; $\alpha = 0$ retains only the local progress term.

Task-relative advantage. For one kernel task, we sample a group g of rollouts and pool all of their segments. If $N_g = \sum_{j \in g}(I_j + 1)$, we compute

$$\mu_g = \frac{1}{N_g} \sum_{j \in g} \sum_{i} \widetilde{R}_i^j, \quad \sigma_g^2 = \frac{1}{N_g} \sum_{j \in g} \sum_{i} (\widetilde{R}_i^j - \mu_g)^2, \quad \widehat{A}_i^j = \frac{\widetilde{R}_i^j - \mu_g}{\sigma_g}. \tag{6}$$

Normalizing within a task avoids comparing reward increments from kernels with different difficulty and optimization headroom. The segment advantage is broadcast only to response tokens in that segment.

Policy objective. We optimize a clipped policy-gradient objective:

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G} \sum_{j=1}^{G} \frac{1}{I_j + 1} \sum_{i=1}^{I_j+1} \frac{1}{|a_i^j|} \sum_{t=1}^{|a_i^j|} \left[\min\Big(\rho_t \widehat{A}_i^j,\ \mathrm{clip}(\rho_t, 1-\varepsilon_l, 1+\varepsilon_h) \widehat{A}_i^j \Big) - \beta D_t^{\mathrm{KL}} \right] \right], \tag{7}$$

where $\rho_t = \pi_\theta(v_t \mid x_t) / \pi_{\mathrm{old}}(v_t \mid x_t)$, ε_l and ε_h are asymmetric clipping coefficients following the clip-higher schedule of DAPO Yu et al. (2025) (values in Appendix D), and D_t^{KL} denotes the per-token divergence from the frozen reference policy π_{ref}, estimated with the $k3$ estimator of Equation 3 and weighted by β as in GRPO Shao et al. (2024). Averaging first over tokens, then segments, and finally rollouts prevents long responses or frequently compacted trajectories from receiving disproportionate weight; we keep this nested normalization rather than the flat token-level aggregation of DAPO because it preserves segment-level credit assignment under trajectory compaction.

5 Experiments

5.1 Setup

Model and Datasets. We experiment with Qwen3.6-35B. Our training dataset is curated from the open-source cuda-llm[1] dataset, from which we select a subset of samples for RL training.

Baselines. We compare: (1) untuned Qwen3.6-35B; (2) Qwen3.6-35B-SFT; (3) Dr.Kernel Liu et al. (2026), using the open-source 14B[2] checkpoint (Drkernel-14B). We also train Qwen3-30B with this method, listed as Drkernel-30B; (4) GRPO, for which we train Qwen3.6-35B. The harness ablation additionally includes Qwen3.8-27B. Entries marked "–" are not evaluated.

Benchmarks. We evaluate correctness and speedup on KernelBench (Ouyang et al., 2024) and NPUKernelBench[3]. All performance measurements run on the Ascend platform.

Evaluation Metrics. We assess results using two key metrics: (1) Pass@1, the percentage of tasks where the agent generates a kernel that successfully compiles and passes functional correctness checks; We found that existing open-source evaluation standards are overly lenient, using atol=1e-2, rtol=1e-2, which permits some incorrect kernel implementations to pass precision validation. We instead adopt stricter thresholds: for fp32, atol=10^{-3} and rtol=2^{-13}; for bf16, atol=10^{-1} and rtol=2^{-7}. (2)Fast@1 and Fast@1.2, the fraction of generated kernels whose measured speedup ratio over the reference reaches at least 1.0 or 1.2.

5.2 Main Results

Tables 1 and 2 report the performance of ATRL and baselines on KernelBench and NPUKernelBench, respectively.

On KernelBench, ATRL achieves the best Pass@1 among reported entries on all three levels, reaching 73 of 100 on Level 1, 35 of 100 on Level 2, and 10 of 50 on Level 3. Level 3 is compared only with GRPO (2 of 50 Pass@1); the other rows are not evaluated on this split. On Level 1, GRPO attains the highest Fast@1 and Fast@1.2, at 26% and 23%, with ATRL at 23% and 22%. On Level

[1] https://github.com/ByteDance-Seed/cudaLLM

[2] https://huggingface.co/hkust-nlp/drkernel-14b

[3] https://github.com/Just-it/AscendOpGenAgent.

Table 1: Main Results on KernelBench. We report Pass@1 and Fast@1/1.2. Metrics are reported relative to torch eager baseline. Each entry is the number of problems that meet the metric, out of 100, 100, and 50 problems on Levels 1–3. **Bold** indicates the best result.

Method	Level 1			Level 2			Level 3		
	Pass@1	Fast@1	Fast@1.2	Pass@1	Fast@1	Fast@1.2	Pass@1	Fast@1	Fast@1.2
Drkernel-14B	15	–	–	1	–	–	–	–	–
Drkernel-30B	37	23	19	12	4	**3**	–	–	–
Qwen3.6-35B	20	8	7	1	0	0	–	–	–
Qwen3.6-35B-SFT	33	12	11	4	2	2	–	–	–
GRPO	66	**26**	**23**	19	2	2	2	0	0
ATRL(ours)	**73**	23	22	**35**	**7**	**3**	**10**	0	0

Table 2: Main Results on NPUKernelBench. We report Pass@1 and Fast@1/1.2. Metrics are reported relative to torch eager baseline. Each entry is the number of problems that meet the metric, out of 20, 33, 31, and 11 problems on Levels 0–3. **Bold** indicates the best result.

Method	L0			L1			L2			L3		
	Pass@1	Fast@1	Fast@1.2	Pass@1	Fast@1	Fast@1.2	Pass@1	Fast@1	Fast@1.2	Pass@1	Fast@1	Fast@1.2
Qwen3.6-35B	12	7	5	7	2	1	3	3	3	3	3	2
GRPO	11	5	4	9	1	0	6	1	0	3	1	0
ATRL(ours)	**18**	**12**	**8**	**23**	**4**	**3**	**18**	**8**	**8**	**9**	**4**	**2**

2, ATRL attains the highest Fast@1 at 7%, and ties Drkernel-30B for the highest Fast@1.2 at 3%; SFT and GRPO are both at 2% and 2%. On Level 3, Fast@1 and Fast@1.2 are 0% for both GRPO and ATRL. The Pass@1 gain is consistent across levels. On Level 1 it does not raise Fast@1 or Fast@1.2 above GRPO. On Level 2 it coincides with higher Fast@1 (7% versus GRPO's 2%) and Fast@1.2 (3% versus 2%). On Level 3 both speedup rates stay at 0%.

On NPUKernelBench, ATRL achieves the best Pass@1 and Fast@1 on all four levels. The Pass@1 margin over GRPO is largest on Level 1 (23 versus 9 of 33) and Level 2 (18 versus 6 of 31). On Level 3, the most challenging split, ATRL obtains 9 Pass@1, 4 Fast@1, and 2 Fast@1.2 out of 11, compared with 3, 1, and 0 for GRPO. This shows that ATRL is particularly effective in long-horizon, hardware-specific settings where correct kernel generation requires accumulating and reusing domain knowledge across multiple optimization turns.

5.3 Ablation Study

We ablate the two components that address the obstacles stated in the introduction: (i) the lightweight agent harness of Section 4.1, which turns the sandbox into a white-box environment, and (ii) the mixed segment reward of Section 4.3, which replaces terminal GRPO credit assignment. Table 3 reports Pass@1 and Fast@1.

Effect of the agent harness. We compare the same multi-turn loop with versus without the harness: cascaded static analysis, compilation, numerical correctness, and benchmarking; correctness-gated access to profiling; and on-demand retrieval from the Ascend knowledge base through the judge–retrieve–apply chain. On NPUKernelBench the largest relative gain is on Level 2 (2 to 13 of 31 Pass@1), where operators are both fused and hardware-specific; Qwen3.8-27B, a stronger starting point, still improves on Levels 0, 2, and 3, while Level 1 remains comparable. On KernelBench, enabling the harness raises frozen Qwen3.6-35B from 49% to 71% Pass@1 on Level 1 and from 7% to 34% on Level 2, and raises Fast@1 on Level 1 from 19% to 32%. On Level 3, Pass@1 falls from 4 to 2 of 50 and Fast@1 from 1 to 0. This multi-turn comparison is separate from the Qwen3.6-35B row in Table 1. This supports the claim that the harness, including retrieval from the Ascend knowledge base, supplies checks and Triton-Ascend APIs that the base model does not apply on its own.

Table 3: Ablation of the agent harness and the mixed segment reward. Each entry is the number of problems that meet the metric. KernelBench Levels 1–3 contain 100, 100, and 50 problems; NPUKernelBench Levels 0–3 contain 20, 33, 31, and 11.

(a) KernelBench

Method	L1		L2		L3	
	Pass@1	Fast@1	Pass@1	Fast@1	Pass@1	Fast@1
Qwen3.6-35B$_{\text{w/o our harness}}$	49	19	7	0	4	**1**
Qwen3.6-35B$_{\text{w our harness}}$	71	32	34	6	2	0

(b) NPUKernelBench

Method	L0		L1		L2		L3	
	Pass@1	Fast@1	Pass@1	Fast@1	Pass@1	Fast@1	Pass@1	Fast@1
Qwen3.6-35B$_{\text{w/o our harness}}$	12	7	7	2	3	3	3	3
Qwen3.6-35B$_{\text{w our harness}}$	18	12	16	5	13	7	4	4
Qwen3.8-27B$_{\text{w/o our harness}}$	15	10	16	9	14	10	5	1
Qwen3.8-27B$_{\text{w our harness}}$	19	**13**	15	**10**	17	**11**	6	3

6 Conclusion

We introduced ATRL, a compaction-aware reinforcement learning method for long-horizon Ascend Triton kernel generation. ATRL preserves execution state through structured checkpoints and incorporates checkpoint generation into policy training. By combining terminal outcomes with verifiable progress at segment boundaries, it provides finer-grained credit assignment than outcome-supervised GRPO. A lightweight agent harness and an Ascend-specific knowledge base support iterative implementation, verification, and optimization. On KernelBench, ATRL improves Pass@1 over GRPO from 66% to 73% on Level 1 and from 19% to 35% on Level 2, with additional correctness gains on NPUKernelBench. Although performance gains vary across benchmark levels, these results support jointly designing context management and reward assignment for long-horizon kernel-generation agents. Future work will investigate stronger performance-oriented credit assignment and extend the approach to additional hardware backends.

References

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krahenbuhl. Reinforcement learning for long-horizon interactive LLM agents. *ArXiv*, abs/2502.01600, 2025. URL `https://api.semanticscholar.org/CorpusID:276106993`.

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 GPUs. *ArXiv*, abs/2606.04847, 2026.

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

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, Yining Li, Jiaxing Xie, Huanan Dong, Yaguang Wu, Xiangjun Huang, Jian Yang, Hui Wang, Bowen Zhou, Bowen Li, Qipeng Guo, and Kai Chen. KernelSmith: A unified recipe for evolutionary kernel optimization. *ArXiv*, abs/2603.28342, 2026a.

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

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Siqi Guo, Ming Lin, and Tianbao Yang. DRTriton: Large-scale synthetic data driven reinforcement learning for Triton kernel generation. *ArXiv*, abs/2603.21465, 2026.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *ArXiv*, abs/2509.14279, 2025.

Xiaozhe Li, Tianyi Lyu, Yizhao Yang, Liang Shan, Siyi Yang, Ligao Zhang, Zhuoyi Huang, Qingwen Liu, and Yang Li. Escaping the context bottleneck: Active context curation for LLM agents via reinforcement learning. *ArXiv*, abs/2604.11462, 2026.

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*, abs/2602.05885, 2026.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K. Qiu, and Yuqing Yang. Agent lightning: Train ANY AI agents with reinforcement learning. *ArXiv*, abs/2508.03680, 2025. URL https://api.semanticscholar.org/CorpusID:280526917.

Anne Ouyang, Simon Guo, and Azalia Mirhoseini. KernelBench: Can LLMs write GPU kernels? https://huggingface.co/datasets/ScalingIntelligence/KernelBench, 2024. Dataset v0.1.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. *ArXiv*, abs/2506.15701, 2025.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, Tianjie Zhang, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300, 2024.

Philippe Tillet, H.T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Fei-Fei Li, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *ArXiv*, abs/2504.20073, 2025. URL https://api.semanticscholar.org/CorpusID:278170861.

Zhiheng Xi, Jixuan Huang, Chenyang Liao, Baodai Huang, Honglin Guo, Jiaqi Liu, Rui Zheng, Junjie Ye, Jiazheng Zhang, Wenxiang Chen, Wei He, Yiwen Ding, Guanyu Li, Zehui Chen, Zhengyin Du, Xuesong Yao, Yufei Xu, Jiecao Chen, Tao Gui, Zuxuan Wu, Qi Zhang, Xuanjing Huang, and Yu-Gang Jiang. AgentGym-RL: Training LLM agents for long-horizon decision making through multi-turn reinforcement learning. *ArXiv*, abs/2509.08755, 2025.

Jason Yoo, Rajarshi Saha, Shaowei Zhu, Tao Yu, Wei Tang, and Youngsuk Park. MKEvolve: A modular multi-agent framework for kernel code generation. In *2nd Workshop on Compositional Learning at ICML*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *ArXiv*, abs/2503.14476, 2025.

Yichao Yuan, Ankita Nayak, Souvik Kundu, and Nishil Talati. Agentic AI workload characteristics. *ArXiv*, abs/2605.26297, 2026.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL. In *International Conference on Machine Learning*, 2024. URL https://api.semanticscholar.org/CorpusID:268091206.

A Agent Harness Details

A.1 Tool Suite and Sandbox Execution

The agent has file-viewing and incremental-editing tools, a knowledge catalog and document reader, a parameter-free verifier, and a submission tool. Tools are packaged as executable scripts, copied into the sandbox, and added to `PATH`.

A.2 Termination and Final Scoring

A rollout ends upon explicit submission, exhaustion of the turn or token budget, or sandbox timeout. If the code changes after the last verification, the harness performs one final verification before scoring. Whenever speedup improves, the harness saves the corresponding implementation and performance record as a historical-best snapshot. At each compaction boundary, the current verified state and the historical-best metadata are stored together, while segment rewards are computed from the current state.

A.3 Knowledge Base

The knowledge base contains five modules:

- hardware specifications, including core counts, on-chip memory, alignment, and transfer paths;
- a latency-optimization library with indexed strategies and operator-to-strategy mappings;
- Triton API documentation;
- Ascend-specific extensions for buffers, address spaces, and device functions; and
- complete kernel examples.

Documents are exposed at four levels of granularity: catalog overview, section, individual reference, and full text. Retrieval therefore proceeds as a *judge–retrieve–apply* procedure: the policy first diagnoses what it needs, then retrieves only the targeted evidence, instead of loading whole documents into the context. The corpus is refined offline by aggregating frequent errors and performance bottlenecks from collected trajectories, then updating documentation, prompts, and tool diagnostics. These revisions modify the environment resources between training runs; the knowledge base itself is not updated by the policy during a rollout.

B Verifier Details

B.1 Pipeline Structure

The verifier is a fail-fast cascade of four stages: static analysis, compilation, numerical correctness, and hardware benchmarking. Each stage runs only if its predecessor passed, and each writes a structured report that doubles as the observation returned to the agent. Compilation is exercised by importing and running the candidate in an isolated subprocess with a fixed time budget; import or launch failures are reported as compile errors. A stage passes only if every test case passes, and benchmarking additionally refuses to start unless the correctness report records a full pass, so no hardware time is spent on invalid candidates. Each stage terminates with a dedicated exit code (failure, correctness-gate refusal, missing baseline anchor, or tampered baseline), which the harness maps onto the indicator terms of Equation 4. To keep observations compact, runtime errors are grouped by type, and each group retains one representative traceback after normalizing temporary paths and generated-source line numbers.

B.2 Static Analysis

The first stage parses the submitted source into an abstract syntax tree, without executing it, and rejects four classes of degenerate submissions: (i) no Triton kernel at all, or kernels with empty bodies; (ii) a kernel that is defined but never invoked from `ModelNew.forward`, where invocations

are traced through same-file helper functions so that an unused kernel cannot be hidden behind an indirection; (iii) a kernel that is invoked while part of the computation still runs through PyTorch, where `forward` may only call the kernel plus a whitelist of allocation and shape-manipulation utilities, and compute-heavy torch functions are rejected both in the forward call chain and at module scope; and (iv) placeholder kernels, where a kernel counts as a placeholder if it uses no computational `tl.*` API and every value it stores is a variable whose only assignment is a `tl.load`, and a submission is rejected when all launched kernels are placeholders. The stage also blocks evasion techniques: dynamic attribute access into the `torch`/`nn` namespaces via `getattr`, direct calls to fused device operators, host-side evaluation of transcendental functions that precompute kernel input tables, and host–device synchronization in the launch path. Every rejection carries the regression class and a concrete repair suggestion that the agent can act on directly.

B.3 Numerical Verification

Seeding protocol. Inputs are generated, and both the reference and the candidate are instantiated, under identical seeds, so randomly initialized parameters agree bitwise.

Hidden-state alignment. For reference models whose output depends on internal state (randomly initialized weights, buffers, caches), a probe runs once before the first forward pass. Same-named parameters and buffers must match bitwise; the scan recurses into instance dictionaries and module-level caches, so weights hidden in plain containers are still found, and a transposed two-dimensional storage layout is accepted. The probe separates this failure mode—which no kernel repair can fix—from ordinary numerical mismatch and reports it before accuracy comparison.

Comparison criteria. Outputs must match in count and shape, and the positions (and for infinities, the signs) of non-finite values must match exactly. The remaining comparison dispatches on dtype: bool outputs require exact equality; integer outputs require exact equality for integer inputs but allow $|\Delta| \leq 1$ for float inputs (quantization); float outputs must satisfy the three-part criterion in Table 4; and non-computational operators, flagged per task, must produce bitwise-identical outputs of the same dtype, verified through integer reinterpretation so that even NaN payload differences are caught. Optional outputs that the reference returns as `None` must also be `None` in the candidate. Each test case runs in isolation with its own try/except, and failures are persisted with structured per-case metrics.

Table 4: Float-precision thresholds by output dtype. A float output passes if (1) every element satisfies $|\hat{y} - y| \leq \mathrm{atol} + \mathrm{rtol}\,|y|$; (2) the fraction of matched elements is at least 0.9, where an element matches if $|y| < s_{\mathrm{thr}}$ implies $|\hat{y} - y| \leq e_{\max}$, and otherwise $|\hat{y} - y|/(|y| + 10^{-7}) \leq r_{\mathrm{thr}}$; and (3) the mean relative error $|\hat{y} - y|/(|y| + 10^{-7})$ over all finite elements is below r_{thr}. All quantities are computed in fp32.

dtype	atol	rtol	s_{thr}	$e_{\max}$	r_{thr}
fp32	10^{-3}	2^{-13}	2^{-14}	2^{-30}	2^{-13}
fp16	9×10^{-2}	2^{-10}	2^{-11}	2^{-16}	2^{-10}
bf16	10^{-1}	2^{-7}	2^{-8}	2^{-16}	2^{-7}

B.4 Benchmarking Methodology

Benchmarking measures each verified shape separately. Both models are instantiated under the seeding protocol above, and latency is collected with the profiler: after five warm-up iterations, five iterations are recorded, and only these enter the statistics. Per-shape latency is aggregated at the kernel level: durations are grouped by kernel name, per-kernel launch counts are recovered from row multiplicity so that multi-launch kernels are not underestimated, outlier samples are discarded, and the shape latency is the sum of per-kernel per-call averages. The per-shape speedup is the ratio of reference to candidate latency, and the overall speedup is the geometric mean over shapes with valid measurements. Profiling failures are retried up to three times per shape. Besides speedup, the stage reports peak memory and per-kernel utilization of the compute and data-movement pipes (vector, cube, and memory-transfer bandwidth); the agent reads these ratios to diagnose bottlenecks such as vector under-utilization or memory-bound execution and, guided by the knowledge base, to select transformations. Device frequency is locked during measurement.

B.5 Integrity Safeguards

When a task is frozen, the reference implementation and its test inputs are hashed; verification and benchmarking re-check these hashes and abort with a dedicated error if anything changed, so the agent cannot edit the reference, the inputs, or the scoring pipeline. Reference files are read-only in the sandbox. Fixed seeds, locked frequencies, and the per-stage subprocess timeout make verification outcomes reproducible.

C Context-Compaction Details

C.1 Trigger and Back-off

The harness evaluates compaction after each agent turn. A compaction request is issued only when (i) compaction is enabled, (ii) at least six non-system/task messages are present, (iii) context usage exceeds a configured fraction of the model window, (iv) the most recent action was verification, and (v) the harness is not in a failure back-off period. We use exact usage reported by the inference service when available and otherwise estimate usage from message content plus role and JSON-serialization overhead. Training uses a lower threshold than evaluation to expose the policy to sufficient checkpoint-writing examples. Following a failed attempt, compaction is disabled for three interaction steps.

C.2 Checkpoint Schema

The compaction prompt requests the following sections in a fixed order:

1. **Attempt history**: attempted approaches and their outcomes, merging repeated failures;
2. **Learned constraints**: API, interface, AST, and hardware restrictions discovered so far;
3. **Design decisions**: the rationale for the current implementation;
4. **Errors and fixes**: observed failures and their resolutions;
5. **Files and code**: relevant paths and the latest implementation;
6. **Current state**: verification status, pass rate, speedup, and latest verifier output;
7. **Next step**: one concrete continuation action.

All sections must appear, with `none` used for empty fields. The model returns bullet-style text inside designated tags, may not call tools, and must merge valid information from any earlier checkpoint rather than stack summaries.

C.3 Validation and Context Rebuilding

We extract only text inside the designated tags and remove accidental code fences. A checkpoint is accepted only if it is non-empty and shorter than the replaced history. Compaction is attempted at most twice. Upon success, the context is rebuilt as

$$[\text{system},\ \text{task},\ \text{checkpoint},\ \text{resume instruction}],$$

where the resume instruction includes the checkpoint's next action and explicitly asks the model to continue from the existing workspace. On failure, the original conversation remains intact.

At each accepted boundary, the harness stores the messages and actions in the completed segment, together with token counts, compaction latency, current code, verification output, performance record, and historical-best speedup. It then opens a new rollout buffer. A rollout compacted I times consequently yields $I + 1$ rows in the training queue.

D TRAINING DETAILS

Boundary rewards. Each segment's start state reuses the snapshot taken at the preceding compaction boundary, and segment-end scores are computed from the same verifier state used by the terminal reward. Therefore, improvements and regressions in AST validity, compilation, correctness, and speed are measured in a common unit.

Degenerate groups. All segments from rollouts of the same operator task participate in the statistics in Equation 6. If the group contains only one segment, its standard deviation and advantage are set to zero because no relative comparison is available.

Loss masking. Environment observations, system messages, task instructions, and checkpoint text reused as a later prompt are excluded from the response loss. A checkpoint is optimized only when it is first generated as the response that closes the preceding segment. Token-level importance ratios are computed against the rollout policy, and the segment advantage is broadcast over the corresponding response mask.

Optimization hyperparameters. We use $\alpha = 0.5$ for the mixed segment reward and asymmetric ratio clipping with $\varepsilon_l = 0.20$ and $\varepsilon_h = 0.28$. The KL penalty in Equation 7 applies the per-token $k3$ estimator of Equation 3 against the frozen reference policy, weighted by the coefficient β.

E PROMPT TEMPLATE

Figure 3 presents the prompt templates used by the Ascend Triton Kernel Generation Agent.

ATRL Prompt Templates — Ascend Triton Kernel Generation Agent

① System Prompt

```
You are a Triton Ascend NPU operator implementation expert.

Your workspace is `{workspace_dir}`. ALL file paths must start with `{workspace_dir}/`.

Available tools:
- list_skills: list all available skills with name/description/path.
- read_skill: read a skill's SKILL.md (directory + first sentence per heading by default,
  --section for a chapter, --full for entire file, --file for a reference file).
- str_replace_editor: view, create, and edit files.
- run_verify: run AST check + correctness verify + performance benchmark
  (speedup_vs_torch). No arguments needed - call it after writing code or after any
  optimization. If AST fails or any case fails, benchmark is skipped.
- submit: finish the task. Call after verify passes AND you have optimized for speedup
  (target: {speedup_target}x). When speedup reaches the target, try pushing it further
  before submitting; submit once gains stall.

## Phase 1: Coding & Verification (steps 1-6)

1. Call `list_skills` to see available skills and their purposes.
2. Read the reference implementation (provided in task) to understand the operator type
   (elementwise / reduce / matmul / attention / ...).
3. Use `read_skill` to gather knowledge ON DEMAND - read only what's relevant to this
   operator, not entire skills. Don't over-explore; each read consumes context and leaves
   less room for code and fixes.
   - `read_skill --skill hardware-specs --file references/hw-{arch}.md`: read hardware
     specs for the target architecture.
   - `read_skill --skill op-design-guide`: see operator category design guides. Read the
     section matching your operator type for design constraints, algorithm skeletons, and
     key implementation patterns.
   - Use `list_skills` to discover other skills (api-reference, ascend-extension, examples)
     and `read_skill` to explore them.
4. Write the implementation using `str_replace_editor create` at
   `{workspace_dir}/src/{op_name}_triton_ascend_impl.py`.

   ## Implementation Contract (REQUIRED - the verifier's AST check enforces this)

   Your impl file MUST define a class named exactly **`ModelNew`** with a
   **`forward(self, ...)`** method. Mirror the reference `Model` class in
   `{workspace_dir}/src/{op_name}.py`:

   - `ModelNew` must accept the SAME constructor arguments as `Model` (use
     `get_init_inputs()` for the signature) and expose a `forward()` with the SAME
     arguments and return shape as `Model.forward()`.
   - Inside `forward`, prepare inputs/outputs and launch your `@triton.jit` kernel(s) via
     `kernel[grid](...)`.
   - Define your `@triton.jit` kernel(s) at module level.
   - Do NOT name the class `Model` or anything other than `ModelNew`, and do NOT omit
     `forward`. Copying the reference `class Model` into the impl file will fail AST check.
   - Do NOT name the class anything other than `ModelNew`, and do NOT omit `forward`.

   ### forward() body rules (the AST check enforces a strict whitelist)

   `forward()` is a thin host wrapper. It MUST launch EVERY `@triton.jit` kernel via
   `kernel_name[grid](args...)` - a defined kernel that `forward()` doesn't launch is
   rejected. Inside `forward()` you may ONLY:
     - allocate outputs: `torch.empty` / `empty_like` / `zeros` / `ones` / `full` /
       `full_like`
     - read metadata: `.shape` / `.size()` / `.stride()` / `.numel()` / `.dtype` /
       `.device`
     - reshape/layout: `.contiguous()` / `.to()` / `.view()` / `.reshape()` / `.permute()`
       / `.transpose()` / `.expand()` / `.clone()`
     - compute the grid: `triton.cdiv` / `triton.next_power_of_2`
   `forward()` must NOT:
     - call any torch compute op - `max`, `min`, `sum`, `mean`, `matmul`, `mm`, `exp`,
       `log`, `sqrt`, `norm`, `layer_norm`, `softmax`, `relu`, etc. - or
       `torch.nn.functional` / `F.*`
     - use Python `for` or `while` loops
     - do ANY numeric work. Put 100% of the computation in the `@triton.jit` kernel;
       `forward()` only preps pointers/shapes and launches.

5. Call `run_verify` to check AST + correctness + speedup. The tool outputs a structured
   summary: ast_valid, passed_cases/total_cases, pass_rate, error_groups, and
   speedup_vs_torch (when all cases pass).
6. If errors exist, fix them with `str_replace_editor str_replace`, then call `run_verify`
   again. Repeat until all cases pass.

## Phase 2: Performance Optimization (step 7, ONLY after verified_success: true)

7. After `run_verify` reports `verified_success: true` (all cases passed), your
   implementation is CORRECT. Now optimize for performance:
   a. Route the structured profiler feedback by field presence BEFORE writing any code:
      when `profile_summaries` contains paths, read those summaries and call `read_skill`
      on `msprof-analysis`; when `kernel_metrics` is present, call `read_skill` on
      `torch-profiler-analysis`. If neither field is present, continue with the full
      `latency-optimizer` index instead of waiting for or inventing profiler data.
   b. From the analysis skill, identify the bottleneck type (VEC / MTE2 / CUBE / SCALAR
      bound) for your kernel, then use its mapping table to find candidate optimization
      points. Also check the `latency-optimizer` skill's full index for additional points
      that fit.
   c. Apply ONE optimization at a time, then call `run_verify` to confirm correctness is
      maintained and check the updated speedup_vs_torch.
   d. When speedup reaches {speedup_target}x, try pushing it further - keep applying
      optimization points (from msprof or latency-optimizer) to see if speedup improves -
      then call submit once gains stall. (The target is enough to submit, but pushing
      further is encouraged; don't loop forever.)

## Critical Rules

- Do NOT read `latency-optimizer`, `msprof-analysis`, or `torch-profiler-analysis` until
  `run_verify` reports `verified_success: true`. They are optimization-phase skills.
- Pass tensors directly to Triton kernels. Do not use .data_ptr().
- Prefer module-level `@triton.jit` kernels launched as `kernel[grid](...)`.
- NEVER read files under tools/ - they are infrastructure scripts.
- Call `run_verify` after EVERY code change - do NOT assume correctness.
- Do NOT loop on thinking. Write code, verify, fix, verify, optimize, verify, submit.
- Each optimization iteration: read ONE optimization point -> apply -> run_verify.
- When speedup reaches the {speedup_target}x target, try pushing it further before calling
  `submit`; submit once gains stall.
```

② Task Prompt

```
# Task: Implement {op_name}
Target architecture: {arch}

## Reference Implementation
```python
{task_code}
```

Test Cases
{test_case_summary}

File Paths
Workspace: `{workspace_dir}`
Implementation file: `{workspace_dir}/src/{op_name}_triton_ascend_impl.py`
Reference file: `{workspace_dir}/src/{op_name}.py`

The reference uses `class Model`. The impl file MUST use `class ModelNew` (same `__init__`
/ `forward` signature). Do not copy the reference class name.
```

③ Compaction Instruction

```
You are now acting as a compaction engine for this AI coding assistant. Condense the
conversation ABOVE into a structured checkpoint that lets you resume the operator
implementation work with no loss of essential context. The system prompt and the original
task message remain available after compaction - do NOT restate them; capture only the work
state.

Output EXACTLY the Markdown structure below: keep every section, in order. Use terse
bullets, not prose paragraphs. Write "(none)" for an empty section - never drop a section.
Do not wrap the output in code fences.

## Attempts History
- Each line:`Attempt N-M: [approach] -> [outcome]`.
- Only record approach-level changes, not micro fixes. Merge repeated failures of the same
  kind into one line.

## Constraints Learned
- Rules discovered while working: AST-check bans, API limits, hardware constraints,
  verifier interface contracts (e.g. ModelNew signature).
- Deduplicate; these must survive every future compaction.

## Design Decisions
- Why the current implementation approach was chosen (one line each).

## Errors and Fixes
- error -> how it was resolved (or why it is still unresolved).

## Files and Code
- Exact file paths, and the CURRENT full content of the implementation file (final version
  only; drop intermediate snapshots). If the file is very long, keep the kernel signatures,
  class/forward structure, and the parts most relevant to the next fix.

## Current State
- Verification progress: ast_passed / compile_passed / correctness passed, pass_rate,
  speedup_vs_torch so far, number of verify runs.
- The COMPLETE output of the LAST run_verify (or "(none)" if never run).

## Next Step
- The single next action to continue the task.

Rules:
- Write concise engineering prose. Preserve exact file paths, commands, error strings,
  identifiers, numeric values, and function signatures.
- Completely DROP: skill/reference document contents read via tools (they live on disk and
  can be re-read), intermediate code snapshots, repeated skill listings, and verbose tool
  outputs not listed above.
- If the conversation already contains a prior checkpoint summary, it is a PRIOR
  compaction: do not copy it forward verbatim - preserve still-true facts, drop stale ones,
  and merge newer information into this single consolidated summary.
- Do NOT mention this summarization request or that the context was compacted.
- Wrap the ENTIRE checkpoint output in <compact_result>...</compact_result> tags. The
  content between these tags will be extracted as the compacted checkpoint; anything
  outside the tags (including any thinking, reasoning, or analysis) will be discarded and
  NOT passed to the next conversation turn. Output only the tags and the checkpoint, do not
  call any tool.
```

④ Compaction Resume Note

```
[Context was compacted. The checkpoint above is your own record of all work so far on this
task: attempts, constraints learned, the current implementation file, and the latest verify
result. Do NOT restart the task, re-read skills, or rewrite the implementation from
scratch. Continue directly from the checkpoint's Current State and execute the Next Step
now.

Next Step from checkpoint:
{next_step}
```

Figure 3: Prompt templates and context compaction instructions for the Ascend Triton Kernel Generation Agent.