TritonRL: Training LLMs to Think and Code Triton Without Cheating

Jiin Woo[1] [*], **Shaowei Zhu**[1], **Allen Nie**[2] [†], **Zhen Jia**[1], **Yida Wang**[1], **Youngsuk Park**[1] [*]
[1]Amazon Web Services [2]Google DeepMind

Abstract

The rapid evolution of Large Language Models (LLMs) has driven a growing demand for automated, high-performance system kernels to accelerate machine learning workloads. We introduce TRITONRL, a domain-specialized 8B-scale LLM for Triton programming, trained via a novel reinforcement learning (RL) framework. While Triton synthesis faces unique challenges, including data scarcity and a high susceptibility to reward hacking, our approach enables robust kernel generation through two primary innovations. First, we implement a multi-layered verification system that provides high-fidelity reward signals, ensuring that generated kernels are both syntactically and functionally valid. Second, we propose Hierarchical Reward Decomposition (HRD), which decouples reinforcement for high-level reasoning and low-level implementation to improve credit assignment in plan–code structured generation. Comprehensive evaluations on KernelBench and TritonBench-T demonstrate that TRITONRL achieves state-of-the-art correctness and runtime speedup, outperforming concurrent Triton-specific models and matching the performance of frontier models with over 100B parameters. Our results highlight the effectiveness of hardware-aware RL paradigms in specialized domain adaptation.

1 Introduction

The exponential growth in demand for GPU computing resources has driven the need for highly optimized GPU kernels that improve computational efficiency, yet with the emergence of numerous GPU variants featuring diverse hardware specifications and the corresponding variety of optimization kernels required for each, developing optimized kernels has become an extremely time-consuming and challenging task. In response to this need, there is growing interest in leveraging large language models (LLMs) for automated kernel generation. While there have been attempts introducing inference frameworks that utilize general-purpose models, such as OpenAI models and DeepSeek, for generating kernels (Ouyang et al., 2025; Lange et al., 2025; NVIDIA Developer Blog, 2025; Muhamed et al., 2023), they often struggle with even basic kernel implementations, thereby highlighting the critical need for domain-specific models specifically tailored for kernel synthesis.

As the need for specialized models for kernel generation has emerged, several works have focused on fine-tuning LLMs for CUDA or Triton. In the CUDA domain, recent RL-based approaches include Kevin-32B Baronio et al. (2025), which progressively improves kernels using execution feedback as reward signals, and CUDA-L1, which applies contrastive RL to DeepSeek-V3 Li et al. (2025c). While these large models (32B-671B parameters) achieve strong CUDA performance, their training costs remain prohibitively expensive. Other research has focused on Triton, a domain-specific language that provides a higher-level abstraction than CUDA for writing efficient GPU kernels. Unlike CUDA, which benefits from decades of development and abundant training data, Triton is newer and has fewer high-quality kernel examples, making it more difficult to train specialized models. Recent works have trained LLMs for Triton kernel generation via supervised fine-tuning (SFT) on

[*]Correspondence to: Jiin Woo <woojiin@amazon.com> and Youngsuk Park <pyoungsu@amazon.com>.
[†]Work done at AWS.

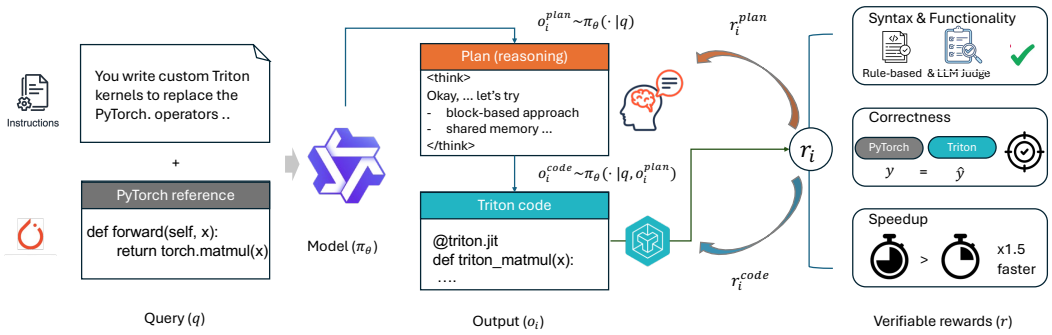


Figure 1: RL training workflow of TRITONRL. The framework illustrates how kernel generation flows through fine-grained verification layers and hierarchical reward decomposition, providing robust and targeted feedback separately to reasoning traces (plan tokens) and to Triton code (code tokens) conditioned on those plans.

torch compiler–generated code (Fisches et al., 2025) or via LLM distillation followed by RL with execution feedback (Li et al., 2025b), typically at the 8B parameter scale. Although these smaller models improve over their base models, there remains substantial room to improve efficiency and correctness.

Furthermore, there is a common issue reported across kernel generation works, reward hacking (Baronio et al., 2025; Li et al., 2025b). Due to the scarcity of high-quality kernel examples compared to other programming languages, most approaches rely on RL training using runtime measurements and correctness rewards from unit tests after kernel execution. However, models frequently learn to exploit unit test loopholes, such as direct use of high-level PyTorch modules, rather than generating proper code, and this phenomenon is particularly prevalent in smaller models (8B and below) (Baronio et al., 2025). This issue fundamentally undermines the core objective of developing more efficient custom kernels to replace existing pre-optimized libraries, while current approaches predominantly rely on simple rule-based syntax verification whose effectiveness remains uncertain. Further discussion of related works is provided in Appendix A.

In this paper, we present TRITONRL, an 8B-scale LLM specialized for Triton programming that achieves state-of-the-art performance in both functional correctness and execution speed. Our training pipeline enables 8B-scale models to robustly surpass much larger frontiers through the following contributions:

- **Robust Verification Framework**: We introduce a multi-layered verification system designed specifically to mitigate Triton-specific reward hacking. By identifying failure modes where models bypass core custom kernel logic, we implement fine-grained verifiers, utilizing both rule-based heuristics and LLM-based judges, that ensure reward signals are grounded in genuine functional and syntactic integrity.
- **Hierarchical Reward Decomposition (HRD)**: We propose a novel RL optimization objective that addresses credit assignment in plan–code structured generation by coupling two distinct verifiable reward functions to the reasoning (planning) and implementation (coding) segments. This approach outperforms uniform sequence-level rewards in both accuracy and speedup.
- **Comprehensive Evaluation and State-of-the-Art Performance**: We provide a rigorous evaluation of open-source and proprietary models using our verification framework, revealing critical performance gaps in functional validity. Our results demonstrate that TRITONRL sets a new state-of-the-art for 8B-scale models and performs comparably to or better than frontier models with over 100B parameters.[1]

[1]Codes and datasets will be available.

2 TRITONRL

In this section, we introduce TRITONRL, a comprehensive reinforcement learning (RL) framework designed to optimize language models for high performance Triton kernel generation. Our approach addresses the dual challenge of achieving functional correctness and hardware specific efficiency, optimizing for execution latency on target accelerators. To this end, our methodology is built upon three core components: (1) base model priming and task curation, (2) reward formulation with robust verification, and (3) policy optimization with hierarchical reward decomposition.

2.1 Base Model Priming and Task Curation

To effectively optimize Triton kernels via RL, we first prime a base model (Qwen3-8B (Team, 2025)) via supervised fine-tuning (SFT) and curate an optimal task distribution. We summarize the notations and pipeline below, deferring detailed dataset construction to Appendix E.1.

Triton Knowledge Distillation. We sample 11K tasks $\mathcal{G}_{KB} = \{g_i\}$ from KernelBook (Paliskara & Saroufim, 2025). For each task query $q_i = q(g_i)$, a teacher model (either DeepSeek R1 (Guo et al., 2025) or GPT OSS 120B (OpenAI, 2025)) generates multiple outputs $o_{i,j}$ comprising reasoning traces and Triton code. This yields the dataset $\mathcal{D}_{SFT} = \{(q_i, o_{i,j})\}$, on which the base model is fine-tuned via standard log-likelihood maximization.

Data Curation for RL. For the RL phase, we refine $\mathcal{G}_{KB}$ via *input augmentation* and difficulty-aware mixing. We augment tasks with randomized input shapes to construct $\tilde{\mathcal{G}}_{KB}$ (training variants using this augmented set are denoted as "+ IA"). The performance benefits of this augmentation are validated in our ablation study (Table 1 and Appendix F.3), where we observe that models trained with input augmentation demonstrate improved robustness to a wider range of kernel shapes, leading to higher correctness and speedup on KernelBench tasks compared to models trained without augmentation.

Additionally, we classify tasks into difficulty levels $d \in \{1, 2\}$ (Ouyang et al., 2025) to form subsets $\mathcal{G}_{KB}^{(d)}$. The RL dataset is sampled according to mixture probabilities $\boldsymbol{p} = (p^{(1)}, p^{(2)})$:

$$\mathcal{D}_{RL}(\boldsymbol{p}) = \{(g_i, q_i)\}_{d\sim\boldsymbol{p}, g_i\sim\mathcal{G}_{KB}^{(d)}, q_i=q(g_i)}$$

Throughout this paper, we default to $\boldsymbol{p} = (1.0, 0.0)$ to focus on Level 1 tasks, omitting $\boldsymbol{p}$ for notational simplicity. As detailed in our ablation study (Appendix F.4), we empirically find that training exclusively on fundamental single-kernel tasks provides more robust skill transfer to complex fusion scenarios than mixing difficulty levels.

Building on the model equipped with essential Triton skills, π_θ and a curated RL dataset $\mathcal{D}_{RL}$, we now focus on the core of our RL framework. Before diving into the specifics, we outline the overall RL training workflow of TRITONRL. For each task (g_i, q_i) sampled from the RL training dataset $\mathcal{D}_{RL}$, the model generates a group of outputs $\{o_{i,j}\}_{j=1}^{G}$ for group size G, where each output $o_{i,j} = \{o_{i,j}^{\text{plan}}, o_{i,j}^{\text{code}}\}$ consists of reasoning traces planning on how to optimize Triton kernels ($o_{i,j}^{\text{plan}}$) followed by Triton code ($o_{i,j}^{\text{code}}$). After executing the generated Triton code, the model receives reward feedback $r_{i,j}$ based on the quality of the output, which it uses to update its parameters θ to maximize expected rewards over time. We illustrate the overall RL training workflow of TRITONRL in Figure 1.

2.2 Reward Formulation with Robust Verification

In reinforcement learning, reward design is critical, as poorly aligned signals can lead to reward hacking, where the model exploits loopholes rather than achieving the intended objectives. We identify specific Triton failure modes, such as ignoring core kernel functionality, that are often missed by conventional reward designs based solely on unit tests. To mitigate

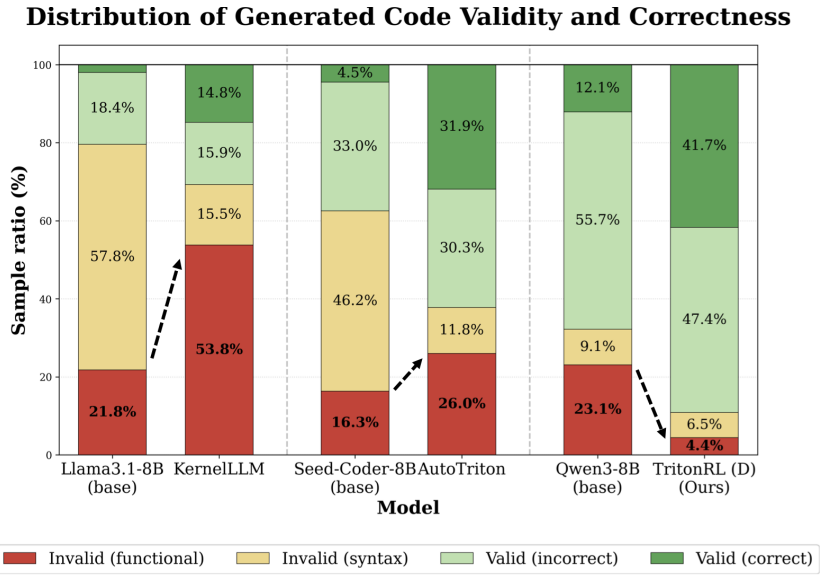


Figure 2: Side-by-side comparison of fine-tuned models for Triton programming (KernelLLM and AutoTriton) and their base models. For each metric, the numbers represent the ratio of corresponding samples among 10 generated samples for each of the 100 tasks in KernelBench Level 1. TRITONRL (D) denotes our model trained from SFT model distilled from DeepSeek-R1. KernelLLM and AutoTriton show increased functional invalidity (red) after fine-tuning, while TRITONRL significantly reduces such errors.

this, we develop fine grained verification layers that rigorously assess code quality across multiple dimensions.

For a generated output o and a corresponding reference PyTorch code g, we introduce verifiers that evaluate the following aspects:

- `valid`: A binary verifier to check output code is syntactically and functionally valid Triton code. It is defined as $\texttt{valid}(o) = \texttt{syntax}(o) \cdot \texttt{func}(o)$, where:
  - `syntax` : A binary verifier that assesses whether code o^{code} is valid Triton syntax. We use a rule-based linter to verify the presence of Triton kernels annotated with @triton.jit.
  - `func`: A binary functionality verifier to detect whether o^{code} constitutes a valid Triton kernel. Syntax checks alone are insufficient, since models may output code that superficially passes verification but defers computations to high-level PyTorch modules (e.g., torch.nn, @) or hardcodes constants, as in Figure 3 (b). To address this, we combine a rule-based linter, which detects actual calls of Triton kernels and flags reliance on PyTorch modules, combined with an LLM-based judge (Qwen3-235B-Instruct (Team, 2025)) that evaluates semantic correctness against task specifications.

  See Figure 3 for the illustration of invalid Triton codes that fail syntax and functionality checks. In Appendix H, we provide additional details on the LLM judge used for functionality verification.
- `compiled`: A binary verifier that checks whether a Triton code can be successfully compiled without errors.
- `correct`: A binary verifier that evaluates whether a Triton code produces correct outputs by compiling and comparing its results against those of the reference code g for five random test inputs $\{x_l\}_{l=1}^{5}$.

$$\texttt{correct}(g,o) \;=\; \texttt{compiled}(o) \cdot \prod_{l=1}^{5} \mathbb{1}\Big[\, o^{\text{code}}(x_l) == g(x_l) \,\Big]$$

- speedup: A scalar score that quantifies the execution time improvement of the generated Triton code o^{code} relative to the reference code g. For a test input x,

$$\texttt{speedup}(g, o) = \frac{\tau(g, x)}{\tau(o^{\text{code}}, x)} \cdot \texttt{correct}(g, o),$$

  where $\tau(\cdot, x)$ is the average runtime of a given code for input x measured over ten runs.

In Figure 2, we analyze the validity of Triton codes generated by existing specialized models, such as AutoTriton (Li et al., 2025b) and KernelLLM (Fisches et al., 2025), and observe a high frequency of functionality violations (red). Interestingly, their base models exhibit lower error rates prior to fine-tuning. This suggests that fine-tuning processes that lack rigorous functionality checks can inadvertently incentivize models to adopt invalid shortcuts, superficially satisfying syntax requirements while failing to implement the core kernel operations.

To mitigate these failure modes, we define two reward functions that deliver robust, accurate signals for correctness and efficiency as follows:

$$R^{\text{correct}}(g, o) = \texttt{valid}(o) \cdot \texttt{correct}(g, o),\ R^{\text{speedup}}(g, o) = \texttt{valid}(o) \cdot \text{clip}\left(\texttt{speedup}(g, o), 2\right). \tag{1}$$

The syntax and functionality checks both serve as necessary conditions for both reward signals to prevent the model from being incentivized to exploit shortcuts. Also, we clip the speedup at 2 to prevent extreme values from destabilizing training.

2.3 Policy Optimization with Hierarchical Reward Decomposition

Building upon the robust reward designs, we introduce a policy optimization framework designed to leverage these signals effectively. While the verification layers ensure high-fidelity feedback, training language models with long reasoning traces remains challenging due to the credit assignment problem. When a single, monolithic reward is applied to a lengthy response, it fails to distinguish the contribution of the high-level plan from that of the low-level code, conflating a brilliant optimization strategy with a minor implementation error.

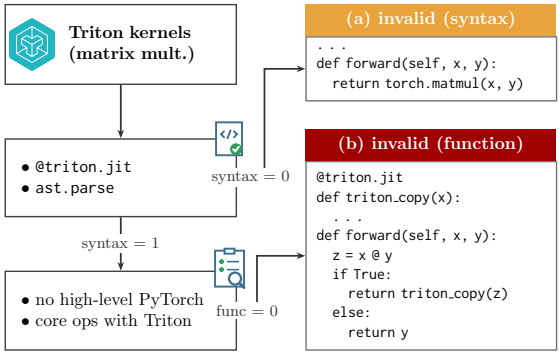


Figure 3: Illustration of the flow of our robust verifier incorporating syntax and functionality checkers and the examples of invalid Triton codes. (a) invalid syntax: the code lacks any Triton blocks with @triton.jit. (b) invalid functionality: the code include dummy Triton code that just copies data without meaningful operation delegating core operation (matrix multiplication) to PyTorch modules (torch.matmul).

This issue is particularly pronounced in Triton kernel code generation, where a model may propose a promising memory orchestration plan in its reasoning trace, yet fail the entire task due to a low level indexing error in the subsequent code. Uniformly penalizing such a response conflates high quality reasoning with poor execution (Qu et al., 2025), preventing the model from internalizing effective optimization strategies.

GRPO with Hierarchical Reward Decomposition (HRD). To address this, we propose an optimization approach based on Group Relative Policy Optimization (GRPO) (Shao et al., 2024), augmented with Hierarchical Reward Decomposition (HRD). GRPO is an actor-only RL algorithm that optimizes the policy by comparing the rewards of multiple outputs within a group generated for the same query, eliminating the need for a separate critic model.

Our key innovation is to segment the long output sequence into two hierarchical actions: high-level planning (o^{plan}) and low-level coding (o^{code}). By assigning distinct reward credits

to these segments (Figure 1), we better align the Triton kernel optimization plan with the final implementation. We decompose the objective into two components and optimize them jointly:

$$\mathcal{J}(\theta) = \mathbb{E}_{q_i \sim \mathcal{D}_{RL},\ \{o_{i,j}\}_{j=1}^{G} \sim \pi_{\theta_{old}}(\cdot|q_i)} \left[\alpha \mathcal{J}_i^{\text{plan}}(\theta) + \mathcal{J}_i^{\text{code}}(\theta) \right]$$

where $\alpha \in [0, 1]$ is a weighting factor that balances the update rates between the model's planning and coding actions. For each action class $c \in \{\text{plan}, \text{code}\}$, the class-specific objective $\mathcal{J}_i^c(\theta)$ is defined as:

$$\mathcal{J}_i^c(\theta) = \frac{1}{G} \sum_{j=1}^{G} \frac{1}{|o_{i,j}|} \sum_{t \in T_{i,j}^c} \min \left[A_{i,j}^c \rho_{i,j,t}^c(\theta), A_{i,j}^c \text{clip} \left(\rho_{i,j,t}^c(\theta), 1-\epsilon, 1+\epsilon \right) \right],$$

where $T_{i,j}^c$ represents the set of token indices for class c in output $o_{i,j}$ and the token-wise probability ratios are given by

$$\rho_{i,j,t}^{\text{plan}}(\theta) = \frac{\pi_\theta(o_{i,j,t}^{\text{plan}} | q_i, o_{i,j,<t}^{\text{plan}})}{\pi_{\theta_{old}}(o_{i,j,t}^{\text{plan}} | q_i, o_{i,j,<t}^{\text{plan}})}, \quad \rho_{i,j,t}^{\text{code}}(\theta) = \frac{\pi_\theta(o_{i,j,t}^{\text{code}} | q_i, o_{i,j}^{\text{plan}}, o_{i,j,<t}^{\text{code}})}{\pi_{\theta_{old}}(o_{i,j,t}^{\text{code}} | q_i, o_{i,j}^{\text{plan}}, o_{i,j,<t}^{\text{code}})}.$$

Each class is normalized by the total response length $|o_{i,j}|$ (standard GRPO normalization), so that the plan- and code-segment gradients scale with their token counts and α sets the balance between them. The group-wise advantages $A_{i,j}^c$ are computed as $A_{i,j}^c = r_{i,j}^c - \frac{1}{G} \sum_{j=1}^{G} r_{i,j}^c$, where we assign different reinforcement signal for each token class as follows:

$$r_{i,j}^{\text{plan}} = R^{\text{speedup}}(g_i, o_{i,j}), \quad r_{i,j}^{\text{code}} = R^{\text{correct}}(g_i, o_{i,j}), \tag{2}$$

where the reward functions are defined in (1).

Hierarchical vs Uniform Rewards. Unlike uniform reward designs (Li et al., 2025b; Baronio et al., 2025) that apply the same signal to all tokens, our HRD provides targeted feedback. In (2), we assign speedup-based rewards to plan tokens to encourage the discovery of efficient optimization strategies, while assigning correctness-based rewards to code tokens. This separation ensures that reasoning traces are encouraged to propose optimization strategies that yield efficient kernels, while code generation is guided to produce valid implementations that faithfully realize these plans. We position HRD relative to other non-uniform credit-assignment methods in Appendix A. Assigning correctness-based rewards to code tokens is deliberate because the implementation is conditioned on the plan, penalizing code for low speedup would unfairly punish a correct implementation of an inherently suboptimal plan. Correctness rewards for code tokens decouple this confound, allowing the code policy to be reinforced purely for faithful realization of a given plan. To support our rationale, we validate this design choice in Appendix F.7 by sweeping all four combinations of the two reward signals across the two token classes. Our assignment (speedup on plan, correctness on code) attains the best performance among the four on both KernelBench levels; in particular, holding the plan reward fixed to speedup, correctness-based rewards for code tokens lead to superior performance over speedup-based rewards.

Balancing Planning and Coding Updates. The weighting factor α manages the co-evolution of the planning and coding policies. To effectively master both, the model requires a stable "practice environment." If $\alpha = 1.0$, the planning distribution shifts too rapidly for the coding policy to stabilize. By setting $\alpha = 0.1$, we ensure the planning distribution evolves gradually, giving the coding policy sufficient "time" to learn correct implementations for a given set of plans. This avoids the premature rejection of promising optimization strategies due to transient implementation errors in early training phases. To validate our choice of α, we conduct an ablation study in Appendix F.6, demonstrating that $\alpha = 0.1$ achieves the best balance, leading to superior performance across all metrics.

3 Experiments

This section provides the detailed recipe of training and evaluation of TRITONRL, followed by the main results and ablation studies.

Model	#Params	LEVEL1 (ROBUST VERIFIER)				LEVEL1 (W/O ROBUST VERIFIER)
		valid	compiled / correct	$fast_1$ / $fast_2$	mean speedup	compiled / correct
Qwen3 (base)	8B	99.0	99.0 / 23.0	9.0 / 5.0	0.41	100.0 / 25.0
Qwen3	14B	100.0	99.0 / 35.0	9.0 / 7.0	0.48	100.0 / 35.0
Qwen3	32B	100.0	100.0 / 62.0	39.0 / 16.0	0.93	100.0 / 67.0
KernelLLM	8B	42.0	40.0 / 20.0	0.0 / 0.0	0.05	99.0 / 34.0
AutoTriton	8B	97.0	92.0 / 57.0	25.0 / 10.0	0.95	100.0 / 87.0
TRITONRL (D)	8B	100.0	100.0 / 78.0	36.0 / 13.0	1.10	100.0 / 81.0
TRITONRL (D) + IA	8B	100.0	99.0 / 78.0	34.0 / 15.0	1.05	100.0 / 81.0
SFT w. DeepSeek-R1 (w.o. RL)	8B	100.0	100.0 / 54.0	28.0 / 12.0	0.88	100.0 / 60.0
TRITONRL (G)	8B	100.0	100.0 / 76.0	35.0 / 17.0	1.09	100.0 / 78.0
TRITONRL (G) + IA	8B	100.0	100.0 / **88.0**	**41.0** / **22.0**	**1.26**	100.0 / 88.0
SFT w. GPT-OSS 120B (w.o. RL)	8B	100.0	100.0 / 64.0	26.0 / 18.0	1.03	100.0 / 73.0
Claude-3.7	-	100.0	100.0 / 63.0	29.0 / 16.0	1.06	100.0 / 73.0
DeepSeek-R1	685B	100.0	100.0 / 69.0	28.0 / 13.0	1.04	100.0 / 74.0
GPT-OSS	120B	100.0	100.0 / 81.0	33.0 / **22.0**	1.20	100.0 / 87.0

Model	#Params	LEVEL2 (ROBUST VERIFIER)				LEVEL2 (W/O ROBUST VERIFIER)
		valid	compiled / correct	$fast_1$ / $fast_2$	mean speedup	compiled / correct
Qwen3 (base)	8B	38.0	37.0 / 0.0	0.0 / 0.0	0.00	100.0 / 56.0
Qwen3	14B	40.0	38.0 / 1.0	0.0 / 0.0	0.00	100.0 / 77.0
Qwen3	32B	36.0	35.0 / 2.0	2.0 / 1.0	0.10	100.0 / 68.0
KernelLLM	8B	3.0	0.0 / 0.0	0.0 / 0.0	0.00	100.0 / 4.0
AutoTriton	8B	11.0	10.0 / 1.0	0.0 / 0.0	0.00	100.0 / 94.0
TRITONRL (D)	8B	63.0	63.0 / 23.0	12.0 / 3.0	0.32	100.0 / 77.0
TRITONRL (D) + IA	8B	52.0	51.0 / 24.0	16.0 / 7.0	0.31	100.0 / 79.0
SFT w. DeepSeek-R1 (w.o. RL)	8B	55.0	51.0 / 13.0	10.0 / 3.0	0.31	100.0 / 67.0
TRITONRL (G)	8B	91.0	91.0 / 23.0	15.0 / 7.0	0.29	98.0 / 30.0
TRITONRL (G) + IA	8B	**93.0**	**93.0** / **28.0**	**19.0** / **10.0**	**0.48**	96.0 / 37.0
SFT w. GPT-OSS 120B (w.o. RL)	8B	88.0	88.0 / 13.0	9.0 / 3.0	0.21	97.0 / 25.0
Claude-3.7	-	34.0	34.0 / 14.0	4.0 / 1.0	0.10	100.0 / 65.0
DeepSeek-R1	685B	31.0	31.0 / 14.0	9.0 / 4.0	0.24	100.0 / 79.0
GPT-OSS	120B	39.0	39.0 / 15.0	12.0 / 4.0	0.25	100.0 / 81.0

Table 1: Main results on KernelBench Level 1 and Level 2. All metrics are reported as pass@10 (%). In each column, the best value across all models is shown in **bold** for the robust-verifier metrics. The left block reports evaluation with the robust verifier (syntax + functionality). The right block (w/o robust verifier) lacks functionality checks, leading to misleading correctness estimates. We denote our model trained from SFT with DeepSeek-R1 (resp. GPT-OSS 120B) as TRITONRL (D) (resp. (G)) and report the results of SFT-only variants without RL for reference (See Section 2.3). We additionally denote our models trained with input augmentation (IA) as + IA, which uses RL datasets where diverse input shapes are augmented (See Section 2.3).

3.1 Training and Evaluation Setups

Training Configuration. We implement the training pipeline using the VeRL framework (Sheng et al., 2025) and initialize all experiments from Qwen3-8B (Team, 2025). Unless noted otherwise, RL training starts from the SFT checkpoint distilled from DeepSeek-R1 (denoted by TRITONRL (D)) and uses the reward r in (2) with $\alpha^* = 0.1$. SFT is performed with batch size 16 and learning rate 1×10^{-5}; RL uses batch size 32 and learning rate 1×10^{-6}. We denote the model fine-tuned from the GPT-OSS 120B SFT checkpoint as TRITONRL (G); most training hyperparameters were kept the same (see Appendix E.2 for details).

Evaluation Benchmarks. We evaluate TRITONRL on KernelBench (Li et al., 2025a)[2]. KernelBench offers an evaluation framework covering 250 tasks, divided into Level 1 (100 single-kernel tasks, such as convolution), Level 2 (100 simple fusion tasks, such as conv+bias+ReLU), and Level 3 (50 full architecture tasks, such as MobileNet), to assess LLM proficiency in generating efficient CUDA kernels. The prompts used for these benchmarks are provided in Appendix I.1. We additionally evaluate on TritonBench-T (Li et al., 2025a) for cross-benchmark generalization (Appendix F.9), and we verify that the training (KernelBook) and evaluation (KernelBench) sets do not overlap at the task level (Appendix E.1.3).

Metrics. We evaluate the performance of LLMs for generating Triton code in terms of (1) Validity (syntax and functionality); (2) Correctness (compilation and correct output); (3) Speedup (relative execution time improvement). We report $\texttt{fast}_1$ and $\texttt{fast}_2$ to indicate

[2]We use the Triton backend version of KernelBench from https://github.com/ScalingIntelligence/KernelBench/pull/35.

the model's ability to generate Triton code that is at least as fast as or twice as fast as the reference PyTorch implementation, respectively. The formal definition of metrics is provided in Appendix C. We measure the pass@k metrics for each aspect, which indicates the ratio of generating at least one successful solution among k sampled attempts. We use $k = 10$ as a default unless specified. We test both Triton codes and reference PyTorch codes on an NVIDIA L40S. We follow the default KernelBench measurement protocol; the full specification (tolerances, warmup, and timed-run counts) and a per-task runtime-variance analysis are provided in Appendix C.1.

Baselines. We compare TRITONRL with several baselines, including KernelLLM (Fisches et al., 2025) and AutoTriton (Li et al., 2025b), which are fine-tuned LLMs specifically for Triton programming. We also include our base model Qwen3-8B (Team, 2025) without any fine-tuning, fine-tuned Qwen3-8B only after SFT, and larger Qwen3 models (e.g., Qwen3-14B and Qwen3-32B). Additionally, we evaluate Claude-3.7 (Anthropic, 2025a) with unknown model size and large model classes beyond 100B (e.g., GPT-OSS 120B (OpenAI, 2025), DeepSeek-R1-0528 (Guo et al., 2025)) as a reference. All baselines are evaluated under an identical configuration: the same one-shot prompt template (Appendix I.1), the same sampling temperature, `top_p`, `max_tokens`, and execution timeout, the same budget $k = 10$, and the same NVIDIA L40S hardware. The full configuration is listed in Appendix E.2.

3.2 Main Experiment Results

Correctness and Speedup Evaluation on KernelBench. The left side of Table 1 shows the pass@10 results using the robust verifier (syntax and functionality) on KernelBench Level 1 (single kernel) and Level 2 (fusion) tasks. TRITONRL consistently outperforms other baselines, including Triton-specific models (KernelLLM and AutoTriton) with fewer than 32B parameters, in terms of validity, correctness, and speedup on both task levels. In particular, TRITONRL outperforms AutoTriton, which also employs RL, by delivering higher correctness and greater speedup, demonstrating the effectiveness of our RL framework with its robust verifiers and hierarchical reward decomposition. Importantly, RL provides significant gains beyond knowledge distillation from large models (DeepSeek-R1 and GPT-OSS 120B), boosting correctness by over 20% and increasing $\texttt{fast}_1$ by up to ∼15%. Furthermore, TRITONRL achieves on-par or better performance even compared to frontier models, underscoring the effectiveness of our approach in enabling smaller models to excel in specialized code generation tasks.

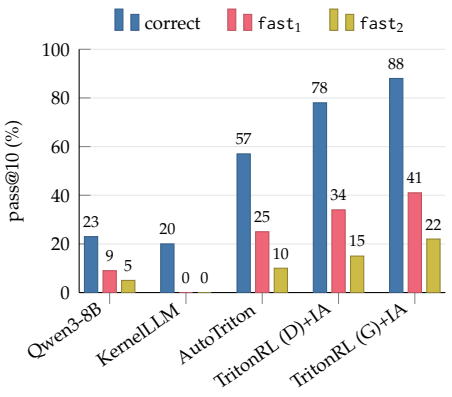


Figure 4: Headline metrics on KernelBench Level 1 (pass@10) across 8B-scale models. TRITONRL substantially outperforms the base model and prior Triton-specialized models (KernelLLM, AutoTriton) on correctness and both speedup.

Beyond our RL framework, we find that careful task curation is critical for achieving these state-of-the-art results. Specifically, input augmentation (IA) further enhances both correctness and speedup (Table 1), highlighting the importance of diversity in training tasks. Comprehensive ablation studies detailing the impact of input augmentation and difficulty-based data mixtures are provided in Appendices F.3 and F.4.

We further verify that these gains extend to harder settings and beyond KernelBench. On the more demanding KernelBench Level 3 (full-architecture tasks, held out of RL training; Appendix F.8), correctness remains near zero for all models, but TRITONRL still produces more valid and compilable kernels than AutoTriton, marking full-architecture generation as a direction for future improvement. On TritonBench-T (Appendix F.9), a different benchmark family, TRITONRL outperforms the fine-tuned Triton-specialized baselines (KernelLLM,

AutoTriton) across validity, correctness, and speedup, with the largest margin on speedup (roughly a 4× higher $\texttt{fast}_1$ than AutoTriton).

Robust Verification Prevents Reward Hacking. In Figure 2, we analyze the validity of Triton codes generated by fine-tuned models to understand the types of errors each model is prone to. Although TRITONRL and other fine-tuned baselines (KernelLLM and AutoTriton) learn to generate more valid codes after fine-tuning, a more detailed breakdown reveals that they exhibit a much higher proportion of functionally invalid codes (red). In contrast, TRITONRL generates significantly fewer invalid codes in terms of both syntax and functionality after fine-tuning, demonstrating the effectiveness of our robust verification in enhancing code quality without reward hacking.

Moreover, Table 1 highlights how heavily prior models relied on cheating shortcuts. Without functionality verification (i.e., w/o the robust verifier), AutoTriton's correctness jumps from 57% to 87%, revealing its tendency to exploit loopholes by superficially meeting syntax checks rather than producing truly functional Triton codes. In contrast, TRITONRL shows only a slight increase ($\leq 3\%$), suggesting it learns to generate genuine code without relying on shortcuts.

Reliability of the LLM Judge. To ensure the integrity of our reward signal, we validate the LLM judge's reliability in Appendix H.1. When evaluated against reference labels from a *training-independent* frontier judge (Opus 4.5 (Anthropic, 2025b)) and human experts, our verifier demonstrates exceptionally low error rates, achieving 2.0% for both false positives and false negatives against the frontier model, and single-digit rates for both against human evaluation. As a direct test of judge dependence, re-evaluating all models on KernelBench Level 1 with Opus 4.5 (not used at any point during training) as the functionality judge preserves TRITONRL's lead over AutoTriton and the Qwen3 base, with correctness matching Table 1 within 1–3 points (Appendix H.2, Table 16). This confirms that our verification framework provides a trustworthy, unbiased foundation for RL training and effectively mitigates cheating behaviors.

Effectiveness of HRD: Hierarchical vs. Uniform. With the integrity of our reward signals established, we now compare it with a *uniform* reward assignment approach, where a single reward is uniformly applied to all tokens without distinguishing between plan and code tokens, which is the reward design chosen by Li et al. (2025b); Baronio et al. (2025). To generalize the reward function used in Li et al. (2025b) and Baronio et al. (2025), we define a uniform reward as a weighted combination of correctness and speedup conditioned on validity, which can be expressed as

$$r_{i,j} = \beta \cdot R^{\text{correct}}(g_i, o_{i,j}) + (1-\beta) \cdot R^{\text{speedup}}(g_i, o_{i,j}), \tag{3}$$

where $\beta \in [0,1]$ is a hyperparameter that controls the ratio of correctness in the reward mixture. In Li et al. (2025b), the reward is computed only based on correctness, which corresponds to $\beta = 1.0$, while in Baronio et al. (2025), the reward is defined as the sum of correctness and speedup with some fixed weights. We defer the detailed GRPO formulation for uniform reward to Appendix D. We compare TRITONRL using hierarchical reward decomposition against models trained with uniform reward designs for various β values, keeping all other RL training settings (GRPO algorithm, hyperparameters, and pre-RL fine-tuning) consistent.

Table 2 demonstrates that hierarchical reward decomposition consistently yields higher correctness and speedup than any uniform reward configuration. While prior works (Li et al., 2025b; Baronio et al., 2025) explored single reward functions with various mixtures of correctness and speedup, our results demonstrate that such uniform designs have limitations in achieving the best model performance. This suggests that decoupling rewards by token class provides more targeted learning signals, better aligning with the distinct roles of planning and coding and leading to higher-quality Triton code generation.

To further validate our design, Appendix F.1 shows that HRD yields more stable training curves with consistent improvements across metrics compared to uniform designs.

Reward Type	β	valid	compiled / correct	fast_1 / fast_2
Hierarchical (α^*)	-	100.0	100.0 / **78.0**	**36.0** / 13.0
Uniform	0.0	100.0	100.0 / 65.0	33.0 / **15.0**
Uniform	0.3	100.0	100.0 / 68.0	29.0 / 14.0
Uniform	0.5	100.0	100.0 / 68.0	35.0 / 13.0
Uniform	0.7	100.0	100.0 / 70.0	34.0 / 12.0
Uniform	0.9	100.0	100.0 / 75.0	31.0 / 11.0
Uniform	1.0	100.0	100.0 / 70.0	35.0 / 12.0

Table 2: Comparison of hierarchical versus uniform reward designs on KernelBench Level 1 tasks. All metrics are reported as pass@10 (%). Reward type specifies whether the single reward is assigned uniformly across all tokens (Uniform) as in (3), or the reward signal is decomposed by token class (Hierarchical) as in (2). The hyperparameter β controls the proportion of correctness in the single reward function applied to all tokens. The models are trained from the same SFT checkpoint distilled from DeepSeek-R1.

Additionally, an ablation study on the hyperparameter α in Appendix F.6 confirms that updating plan tokens more slowly than code tokens yields the optimal performance, further solidifying the robustness of the HRD framework.

4 Conclusion and Limitations

In this work, we introduce TRITONRL, a specialized LLM for Triton code generation trained via a novel RL framework with verifiable and hierarchical rewards, achieving state-of-the-art performance on KernelBench. While generating complex fusion kernels remains an open challenge due to sparse reward signals and model reliance on high-level APIs, TRITONRL still consistently outperforms much larger frontier models. Ultimately, our robust verification and HRD mechanisms provide an essential foundation for hardware-aware RL. Future work will explore extending the verifier with formal guarantees, evaluating across diverse accelerators, and scaling TRITONRL to larger foundation models.

Limitations. Realizing this foundation more broadly, however, requires being explicit about where the present study stops; we accordingly delimit the scope of our claims along four axes. *(i) Model scale.* All results are obtained at the 8B scale. Appendix F.5 shows that the framework transfers across base models at this scale (it improves all metrics when applied to the Seed-Coder-8B-Reasoning base used by AutoTriton), but generalization beyond 8B is not established empirically here. *(ii) Task regime.* The framework targets single-operator (Level 1) and simple-fusion (Level 2) kernel generation, though suppressing functional shortcuts is harder on Level 2, where several operators must be fused into one kernel (Table 1). Full-architecture (Level 3) synthesis lies outside this regime, and the Level-3 numbers in Appendix F.8 confirm that correctness at that scale remains unsolved for all compared models. *(iii) Hardware.* Because RL with execution feedback uses runtime measurements from the target accelerator, the trained policy reflects the performance characteristics of the NVIDIA L40S; cross-hardware transfer is not evaluated in this work. *(iv) Validation scope.* The reported ablations are single training runs, and the human component of our judge-reliability audit covers 22 kernels. Addressing these limitations forms the primary direction of our future work: scaling beyond 8B and toward full-architecture (Level 3) synthesis, evaluating cross-hardware transfer, and broader experimental validation.

References

Anthropic. Claude 3.7 sonnet system card, 2025a. URL https://www.anthropic.com/claude-3-7-sonnet-system-card.

Anthropic. Claude opus 4.5. Large Language Model, 2025b. URL https://claude.ai.

Carlo Baronio, Pietro Marsella, Ben Pan, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.

Mayee F Chen, Michael Y Hu, Nicholas Lourie, Kyunghyun Cho, and Christopher Ré. Aioli: A unified optimization framework for language model data mixing. *arXiv preprint arXiv:2411.05735*, 2024.

Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acereason-nemotron: Advancing math and code reasoning through reinforcement learning. *arXiv preprint arXiv:2505.16400*, 2025.

Zacharias Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm, 5 2025. URL https://huggingface.co/facebook/KernelLLM. Corresponding authors: Aram Markosyan, Mark Saroufim.

Jiaxuan Gao, Shusheng Xu, Wenjie Ye, Weilin Liu, Chuyi He, Wei Fu, Zhiyu Mei, Guangju Wang, and Yi Wu. On designing effective rl reward at training time for llm reasoning. *arXiv preprint arXiv:2410.15115*, 2024.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv preprint*, abs/2501.12948, 2025. URL https://arxiv.org/abs/2501.12948.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025.

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

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*, 2022. URL https://ar5iv.labs.arxiv.org/html/2207.01780.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. *arXiv preprint arXiv:2502.14752*, 2025a.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025b. URL https://arxiv.org/pdf/2507.05687.

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*, 2025c.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. URL https://www.researchgate.net/publication/366137000_Competition-level_code_generation_with_AlphaCode.

Aashiq Muhamed, Christian Bock, Rahul Solanki, Youngsuk Park, Yida Wang, and Jun Huan. Training large-scale foundation models on emerging ai chips. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5821–5822, 2023.

NVIDIA Developer Blog. Automating gpu kernel generation with deepseek-r1 and inference time scaling. NVIDIA Developer Blog, February 2025. URL https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/. Accessed on May 20, 2025.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

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.

Sahan Paliskara and Mark Saroufim. Kernelbook, 5 2025. URL https://huggingface.co/datasets/GPUMODE/KernelBook.

Yuxiao Qu, Anikait Singh, Yoonho Lee, Amrith Setlur, Ruslan Salakhutdinov, Chelsea Finn, and Aviral Kumar. Rlad: Training llms to discover abstractions for solving reasoning problems. *arXiv preprint arXiv:2510.02263*, 2025.

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

Archit Sharma, Sedrick Keh, Eric Mitchell, Chelsea Finn, Kushal Arora, and Thomas Kollar. A critical evaluation of ai feedback for aligning large language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 29166–29190. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/33870b3e099880cd8e705cd07173ac27-Paper-Conference.pdf.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 9460–9471. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/3d719fee332caa23d5038b8a90e81796-Paper-Conference.pdf.

Hongze Tan, Zihan Wang, Jianfei Pan, Jinghao Lin, Hao Wang, Yifan Wu, Tao Chen, Zhihang Zheng, Zhihao Tang, and Haihua Yang. Gtpo and grpo-s: Token and sequence-level reward shaping with policy entropy. *arXiv preprint arXiv:2508.04349*, 2025.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia

Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Weixin Xu, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, Zonghan Yang, and Zongyu Lin. Kimi k1.5: Scaling reinforcement learning with llms, 2025.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Haozhe Wang, Qixin Xu, Che Liu, Junhong Wu, Fangzhen Lin, and Wenhu Chen. Emergent hierarchical reasoning in llms through reinforcement learning. *arXiv preprint arXiv:2509.03646*, 2025.

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 preprint arXiv:2503.14476*, 2025.

Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

A Related Work

A.1 LLM for Kernel Generation

The exponential growth in demand for GPU computing resources has driven the need for highly optimized GPU kernels that improves computational efficiency. However, writing efficient GPU kernels is a complex and time-consuming task that requires specialized knowledge of GPU architectures and programming models. This has spurred significant interest in leveraging Large Language Models (LLMs), for automated kernel generation, especially for CUDA and Triton (Shao et al., 2024; Ouyang et al., 2025; Li et al., 2025a; NVIDIA Developer Blog, 2025). While these general-purpose models excel at a variety of programming tasks, they often struggle with custom kernel generation, achieving low success rates on specialized gpu programming tasks (Ouyang et al., 2025), highlighting the need for domain-specific models tailored to kernel synthesis.

For CUDA kernel generation, Ouyang et al. (2025) introduced KERNELBENCH, an open-source framework for evaluating LMs' ability to write fast and correct kernels on a suite of 250 carefully selected PyTorch ML workloads. Furthermore, Lange et al. (2025) presented an agentic framework, which leverages LLMs to translate PyTorch code into CUDA kernels and iteratively optimize them using performance feedback. Additionally, several works have focused on fine-tuning LLMs tailored for CUDA kernel generation. For example, Kevin-32B (Baronio et al., 2025) is a 32B parameter model fine-tuned via multi-turn RL to enhance kernel generation through self-refinement, and CUDA-L1 (Li et al., 2025c) applies contrastive reinforcement learning to DeepSeek-V3-671B, achieving notable speedup improvements in CUDA optimization tasks.

Another line of research focuses on Triton kernel generation. Li et al. (2025a) introduced TRITONBENCH, providing evaluations of LLMs on Triton programming tasks and highlighting the challenges of Triton's domain-specific language and GPU programming complexity. To further enhance LLMs' capabilities in Triton programming, Fisches et al. (2025) have

introduced KernelLLM, a fine-tuned model of Llama3.1-8B-Instruct via supervised fine-tuning with Pytorch and Triton code pairs in KernelBook Paliskara & Saroufim (2025), but its performance is limited by the quality of training data. Similarly, Li et al. (2025b) introduced AutoTriton, a model fine-tuned specifically for Triton programming from Seed-Coder-8B-Reasoning Zhang et al. (2025), which achieves improved performance via SFT and RL with verifiable rewards based on correctness and rule-based Triton syntax verification, which may have limited improvement in runtime efficiency due to correctness-focused rewards. Both KernelLLM and AutoTriton are concurrent works developed alongside our work, and we provide a detailed comparison in Section 3.2.

A.2 Reinforcement Learning with Verifiable Rewards

Reinforcement Learning (RL) has become a key technique for training Large Language Models (LLMs), especially in domains where verifiable reward signals are available. Unlike supervised fine-tuning (SFT), which relies on curated examples, RL enables models to learn through trial and error, guided solely by reward feedback. This makes the design of accurate reward functions critical, as the model's behavior is shaped entirely by the reward signal. As a result, RL with verifiable rewards (RLVR) (Lambert et al., 2025; Team et al., 2025; Guo et al., 2025) has gained significant traction in applications like mathematics and code generation (Shao et al., 2024; Li et al., 2022), where external verification is feasible through solution correctness or unit test outcomes.

In math and coding applications, the reward can be directly computed solely based on the final outcomes when ground-truth answers or unit tests are available. For tasks where validation is not available or noisy, rule-based verification or LLM-based judges can be employed to verify the quality of generated content (Guha et al., 2025; Guo et al., 2025). For coding tasks, unit tests are commonly used to measure whether generated code meets the specified requirements (Le et al., 2022; Ouyang et al., 2025). However, unit tests often fail to cover edge cases or fully capture the problem requirements, leading to potential "reward hacking" (Skalse et al., 2022) where the model generates code that passes the tests but does not genuinely solve the task (Sharma et al., 2024; Gao et al., 2024). Such reward hacking has been observed in kernel generation tasks, where models produce superficially correct codes passing unit tests by using high-level Pytorch modules instead of implementing custom kernels. To address this, some works Li et al. (2025b); Baronio et al. (2025) have introduced rule-based verification, which checks kernel syntax or use of specific high-level modules.

A.3 Credit Assignment in RLVR

The observation that a single sequence-level reward assigns credit agnostically across tokens has recently motivated several non-uniform credit-assignment designs. **HICRA** (Wang et al., 2025) (hierarchy-aware credit assignment) amplifies a single reward on high-impact planning tokens to concentrate optimization pressure on strategic content. **GTPO** and **GRPO-S** (Tan et al., 2025) reweight a single reward through entropy-based dynamic weighting, at the token level and the sequence level respectively. **RLAD** (Qu et al., 2025) targets credit for high-level strategic content by training a separate abstraction generator and solution generator as a two-player game. No prior method couples multiple different reward signals to multiple token classes within a single-policy.

B Notations

The following notations will be used throughout this paper. For notational simplicity, we denote any function $f(q, o_i)$ as f_i when the context is clear.

- q: prompt given to the model, defining a task to implement in Triton
- o: output sequence generated by the model, which includes both reasoning trace and Triton code
- π_θ: policy model with parameters θ

- G: group size for GRPO
- $o_{i,j}$: j-th sample in the group G for the i-th prompt, which includes a reasoning trace that provides the "plan" for Triton code optimization and implementation and the final "Triton code", i.e. $o_{i,j} = \{o_{i,j}^{\text{plan}}, o_{i,j}^{\text{code}}\}$
- $T_{i,j}^{c}$: set of token indices corresponding to token class $c \in \{\text{plan}, \text{code}\}$ in the j-th sample of the i-th prompt
- $r_{i,j}^{c}$: reward function for token class $c \in \{\text{plan}, \text{code}\}$.
- $A_{i,j}^{c} = r_{i,j}^{c} - \frac{1}{G}\sum_{j=1}^{G} r_{i,j}^{c}$: token-level advantage of the j-th sample for prompt q_i belonging to token class $c \in \{\text{plan}, \text{code}\}$.

C Metrics

We provide the formal definitions of the evaluation metrics used in this paper. Given a set of N tasks $\{g_n\}_{n=1}^{N}$ and k samples $\{o_{n,i}\}_{i=1}^{k}$ generated by the model for each task, we define the following metrics:

$$
\begin{aligned}
\text{valid} &= \frac{1}{N}\sum_{n=1}^{N}\max_{i\in[k]}\mathbb{1}(\texttt{syntax}(g_n, o_{n,i})\cdot\texttt{func}(g_n, o_{n,i}) = 1)\\
\text{compiled} &= \frac{1}{N}\sum_{n=1}^{N}\max_{i\in[k]}\mathbb{1}(\texttt{syntax}(g_n, o_{n,i})\cdot\texttt{func}(g_n, o_{n,i})\cdot\texttt{compiled}(g_n, o_{n,i}) = 1)\\
\text{correct} &= \frac{1}{N}\sum_{n=1}^{N}\max_{i\in[k]}\mathbb{1}(\texttt{syntax}(g_n, o_{n,i})\cdot\texttt{func}(g_n, o_{n,i})\cdot\texttt{correct}(g_n, o_{n,i}) = 1)\\
\text{fast}_p &= \frac{1}{N}\sum_{n=1}^{N}\max_{i\in[k]}\mathbb{1}(\texttt{syntax}(g_n, o_{n,i})\cdot\texttt{func}(g_n, o_{n,i})\cdot\texttt{correct}(g_n, o_{n,i})\cdot\texttt{speedup}(g_n, o_{n,i}) > p)\\
\text{mean\_speedup} &= \frac{1}{N}\sum_{n=1}^{N}\max_{i\in[k]}(\texttt{syntax}(g_n, o_{n,i})\cdot\texttt{func}(g_n, o_{n,i})\cdot\texttt{correct}(g_n, o_{n,i})\cdot\texttt{speedup}(g_n, o_{n,i}))
\end{aligned}
\tag{4}
$$

C.1 Measurement Protocol and Runtime Stability

We follow the default KernelBench evaluation protocol and make its specifics explicit here.

Correctness. Each generated kernel is compared against the reference PyTorch implementation on five randomly generated inputs with $\texttt{atol} = \texttt{rtol} = 1 \times 10^{-2}$ (the KernelBench default), using the dtype specified by each reference task (predominantly `float32`). Robustness to varied input shapes is evaluated separately in Appendix F.3, where each of the 100 tasks per level is run under five distinct input shapes.

Speedup and runtime variance. For each kernel we measure runtime over 10 timed runs preceded by 3 warmup iterations on a single NVIDIA L40S, and report speedup as the ratio of the reference mean runtime to the generated-kernel mean runtime. To summarize measurement stability we report the coefficient of variation (CV = std / mean) of the per-kernel runtime: per task we take the best correct generation's runtime CV and pair it with that task's PyTorch baseline CV (Table 3). The median per-task CV stays at or below 0.31% for the custom kernel and 0.25% for the baseline across all three models, giving a propagated speedup-ratio CV of at most 0.40%, far smaller than the performance differences reported in Table 1.

Table 3: Per-task runtime CV (%) on KernelBench Level 1, using the best correct generation per task. *custom* denotes the generated Triton kernel and *baseline* the PyTorch reference. N is the number of tasks passing the robust verifier, that is, tasks where both executed successfully, outputs matched, and no cheating, syntax, or functionality flag was raised.

Model	Source	N	median	p90
AutoTriton	custom	56	0.16	0.69
AutoTriton	baseline	56	0.16	2.96
TRITONRL (D) + IA	custom	78	0.31	2.73
TRITONRL (D) + IA	baseline	78	0.25	4.25
TRITONRL (G) + IA	custom	87	0.25	2.62
TRITONRL (G) + IA	baseline	87	0.24	4.30

D GRPO with uniform reward assignment

Here we provide the detailed formulation of GRPO with uniform reward assignment, which is used in our experiments (Section 3.2) to compare with our proposed hierarchical reward assignment. The GRPO objective with uniform reward assignment is defined as

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q_i \sim \mathcal{D}_{RL},\ \{o_{i,j}\}_{j=1}^{G} \sim \pi_{\theta_{old}}(\cdot|q_i)} \left[\mathcal{L}_{\text{GRPO}}(\theta, i) \right] \tag{5}$$

where π_θ and $\pi_{\theta_{old}}$ are the policy model and reference model, q_i denotes a prompt given to the model, defining a task to implement in Triton, and $o_{i,j}$ represents i-th response generated by the model for q_i in the group G. The GRPO losses $\mathcal{L}(\theta, i)$ is computed as

$$\mathcal{L}_{\text{GRPO}}(\theta, i) = \frac{1}{G} \sum_{j=1}^{G} \frac{1}{|o_{i,j}|} \sum_{t=1}^{|o_{i,j}|} A_{i,j} \cdot \min \left\{ \frac{\pi_\theta(o_{i,j,t}|q, o_{i,j,<t})}{\pi_{\theta_{old}}(o_{i,j,t}|q, o_{i,j,<t})}, \text{clip} \left(\frac{\pi_\theta(o_{i,j,t}|q, o_{i,j,<t})}{\pi_{\theta_{old}}(o_{i,j,t}|q, o_{i,j,<t})}, 1-\epsilon, 1+\epsilon \right) \right\}, \tag{6}$$

where $A_{i,j}$ denotes the group-wise advantage uniformly applied to all tokens, computed as $A_{i,j} = r_{i,j} - \frac{1}{G} \sum_{j=1}^{G} r_{i,j}$, with $r_{i,j}$ being the reward for the i-th sample. In the uniform reward assignment, we define the reward as a weighted combination of correctness and speedup conditioned on validity, which can be expressed as

$$r_{i,j} = \beta \cdot R^{\text{correct}}(g_i, o_{i,j}) + (1-\beta) \cdot R^{\text{speedup}}(g_i, o_{i,j}), \tag{7}$$

where $\beta \in [0, 1]$ is a hyperparameter that the ratio of correctness in the reward mixture.

E Training Details

E.1 Base Model Priming and Task Curation

For a model to effectively optimize Triton kernels via RL, it must first possess a foundational understanding of Triton syntax and be exposed to a high quality task distribution during RL. We address these requirements through two efforts: (1) base model priming via knowledge distillation to instill fundamental programming skills and (2) task curation via data augmentation and difficulty aware data mixing to construct an optimal training distribution for RL.

E.1.1 Triton Knowledge Distillation via SFT

Recent studies indicate that pretrained language models, particularly at the 8B scale, exhibit limited proficiency in Triton programming, often struggling with low level syntax and hardware specific optimization (Fisches et al., 2025; Li et al., 2025b). To bridge the gap, we perform supervised fine-tuning (SFT) on a base model to distill expertise from frontier models into a more efficient base. In this paper, we utilize Qwen3-8B (Team, 2025) as our base model.

Our primary data source is KernelBook (Paliskara & Saroufim, 2025), which contains a rich collection of Triton kernel generation tasks paired with reference PyTorch implementations. We define $\mathcal{G}_{KB} = \{g_i\}_{i\in\{1,\ldots,11K\}}$ as a set of 11K tasks sampled from KernelBook. For each task query $q_i = q(g_i)$, wrapped with an instruction to generate a Triton kernel for task g_i, we prompt a teacher model to generate Triton kernels multiple times, denoted by $m_i \in [1, 10]$. Each resulting output $o_{i,j}$ (where $j \in \{1, \ldots, m_i\}$) consists of a synthesized reasoning trace followed by the corresponding Triton implementation. For a given teacher model π_{teacher}, this process yields an SFT dataset:

$$\mathcal{D}_{SFT} = \left\{(q_i, o_{i,j})\right\}_{g_i\in\mathcal{G}_{KB}, q_i=q(g_i), \{o_{i,j}\}_{j=1}^{m_i}\sim\pi_{\text{teacher}}(\cdot|q_i)}.$$

With the SFT dataset, we train our base model parameterized by θ, π_θ, to maximize the log-likelihood:

$$\max_\theta \mathbb{E}_{(q_i,o_{i,j})\sim\mathcal{D}_{SFT}}\left[\log \pi_\theta(o_{i,j}|q_i)\right].$$

Through SFT, it learns to internalize the underlying logic of kernel optimization rather than simply memorizing syntax. To explore the impact of different teacher expertise, we construct two separate SFT datasets (approximately 60K samples each) using DeepSeek R1 (Guo et al., 2025) and GPT OSS 120B (OpenAI, 2025) respectively, resulting in two distinct SFT models. Examples of task queries and outputs are provided in Appendix G.5, with additional training details in Appendix E.2.

E.1.2 Data Augmentation and Difficulty-Aware Data Mixing

The selection of training data is crucial for effective RL post-training, and many works have shown that selective sampling by leveraging additional information, such as difficulty or interaction between data points, can significantly improve model performance (Yu et al., 2025; Chen et al., 2025; 2024). While KernelBook provides a large-scale and diverse set of kernels, naive uniform sampling may lead to suboptimal efficiency. Consequently, we augment each task and apply a difficulty aware mixing strategy to refine the RL training set.

Diverse Input Augmentation. We start from the KernelBook task set $\mathcal{G}_{KB} = \{g_i\}_{i\in\{1,\ldots,11K\}}$, used to construct the SFT datasets. To increase diversity and robustness, we augment each task g_i with additional input-generating functions using GPT-OSS 120B, producing randomized input shapes. After augmentation, we validate five input shapes for each task by execution. We denote the resulting set of augmented tasks as $\tilde{\mathcal{G}}_{KB} = \{\tilde{g}_i\}_{i\in\{1,\ldots,|\mathcal{G}_{KB}|\}}$. By default we use the original task set $\mathcal{G}_{KB}$; when RL training uses the augmented set $\tilde{\mathcal{G}}_{KB}$ we indicate this with "+ IA". In Section 3.2, we analyze the impact of input augmentation on final performance.

We now detail the augmentation mechanism, distinguishing it from the separate evaluation-time procedure of Appendix F.3. *Training-time augmentation (KernelBook, used by "+IA").* The original KernelBook tasks ship with deterministic `get_inputs()` functions returning tensors of a single fixed shape (often a small default such as `[4, 4, 4, 4]`). For each task, GPT-OSS 120B is prompted to rewrite `get_inputs()` into a function that samples input shapes randomly on every call, under hard rules that (a) randomize only data-dependent dimensions (batch, sequence, graph size, and the like), (b) preserve parameter-dependent and semantically fixed dimensions, and (c) keep shared semantic axes consistent across inputs, so that for instance an adjacency matrix remains square with the same node count as the corresponding feature tensor. The rewritten function is validated with 5 random seeds and accepted only if `forward` executes successfully under all of them. The accepted function is inserted into the reference code in the training prompt, so during RL *each rollout observes a freshly sampled shape* rather than the single fixed shape of the original task. After validation, the augmented training set contains 2,605 Level-1 tasks and 7,262 Level-2 tasks, of which 99.2% (Level 1) and 99.9% (Level 2) randomize the input shape itself; the remainder are tasks whose `forward` takes no input tensor (`return []`), for which shape augmentation does not apply. *Evaluation-time augmentation (KernelBench, Appendix F.3).* KernelBench's standard evaluation uses a single fixed input shape per task. As a separate multi-shape robustness

diagnostic we materialize 5 distinct shapes per task and run each kernel on all of them ($5 \times 100 = 500$ evaluations per level). This procedure is evaluation-only and never feeds back into training.

The following illustrates what the augmenter produces:

```
# Original KernelBook get_inputs(): single fixed shape
def get_inputs():
    return [torch.rand(4, 4, 4, 4), torch.rand(4, 4, 4, 4)]

# After +IA: shape randomized per call, shared axes kept consistent
def get_inputs():
    rank = torch.randint(1, 5, (1,)).item()
    shape = [torch.randint(1, 11, (1,)).item() for _ in range(rank)]
    return [torch.rand(*shape), torch.rand(*shape)]
```

Each call yields a new shape; the two tensors share a shape so that broadcasting-sensitive operations stay valid; and only data-dependent dimensions vary, by construction of the hard rules. The 5-seed execution check rejects any rewritten function whose sampled shapes can break `forward`, so the variation covers shared-axis, batch, sequence, and graph-size variation without degenerating into either pure numeric reshaping or `forward`-breaking shapes.

Difficulty-Aware Data Mixing. Then, following the difficulty levels defined in KernelBench (Ouyang et al., 2025), we classify tasks into two levels, where Level 1 represents single-kernel tasks, such as convolution, and Level 2 represents fusion tasks, such as conv+bias+ReLU. To label each task g_i, we utilize an LLM based labeler (Qwen3-235B-Instruct (Team, 2025)) and let the difficulty label as $d_i = d(g_i) \in \{1, 2\}$. Focusing on Level 1 and Level 2 tasks, we denote the subset of tasks at difficulty level d as $\mathcal{G}_{KB}^{(d)} = \{g_i \in \mathcal{G}_{KB} | d(g_i) = d\}$ for $d \in \{1, 2\}$ and construct RL training datasets according to various mixture probabilities $\boldsymbol{p}$ as follows:

$$\mathcal{D}_{RL}(\boldsymbol{p}) = \{(g_i, q_i)\}_{d \sim \boldsymbol{p}, g_i \sim \mathcal{G}_{KB}^{(d)}, q_i = q(g_i)},$$

where $\boldsymbol{p} = (p^{(1)}, p^{(2)}) \in [0, 1]^2$ where $p^{(d)}$ denotes the sampling probabilities for difficulty level d. In our experiments (Section 3.2), we explore different training data mixtures and evaluate their effectiveness during RL post-training to identify the optimal configuration. Throughout this paper, we set $\boldsymbol{p} = (1.0, 0.0)$ unless otherwise specified and omit $\boldsymbol{p}$ for notational simplicity. Additional details regarding the labeling prompt is provided in Appendix G.1.

E.1.3 Train/Evaluation Overlap Analysis

Since the RL tasks above are drawn from KernelBook while evaluation uses KernelBench, we verify that the two do not overlap at the task level, quantifying potential contamination along three dimensions.

(1) Reference PyTorch code. KernelBook reference code is collected from public GitHub repositories (Paliskara & Saroufim, 2025), while KernelBench is independently curated by Ouyang et al. (2025) for benchmark evaluation; the two are independently sourced. To quantify structural similarity, we parse each PyTorch reference and pass it through a normalizer that anonymizes identifier names and collapses numeric literals, so that two tasks differing only in shape constants or numeric hyperparameters yield identical normalized ASTs. Normalized ASTs are compared with `difflib.SequenceMatcher`, which returns a similarity in $[0, 1]$, and we report the fraction of KernelBench tasks whose best match in the comparison set reaches similarity ≥ 0.8 (Table 4). Overlap with KernelBook is at most 1% at every level, against a within-KernelBench self-baseline of 38–87%.

(2) Operation structure. For each reference implementation we extract the set of `torch.*`, `nn.*`, and `F.*` API calls together with binary tensor operators (`@`, `+`, `*`, and so on) invoked

Table 4: Fraction (%) of KernelBench tasks whose nearest neighbor by normalized-AST similarity exceeds 0.8.

Comparison set	Level 1	Level 2	Level 3
KernelBook → KernelBench	1.0	0.0	0.0
KernelBench self-baseline	87.0	59.0	38.0

inside the `Model` class's `forward` and `__init__`, and measure operator overlap by the Jaccard similarity of these sets. For each KernelBench task we record the highest Jaccard against the comparison set, and report the fraction clearing 0.8 in Table 5. KernelBook→KernelBench overlap is comparable to (Levels 1 and 2) or below (Level 3) the within-KernelBench self-baseline, so no KernelBook task is operationally more similar to a KernelBench task than KernelBench tasks are to one another. The residual overlap reflects shared ML primitives, which is expected.

Table 5: Fraction (%) of KernelBench tasks whose nearest neighbor by operator-set Jaccard similarity exceeds 0.8.

Comparison set	Level 1	Level 2	Level 3
KernelBook → KernelBench	57.0	7.0	14.0
KernelBench self-baseline	51.0	6.0	38.0

(3) Input-generation functions. Both KernelBook and KernelBench draw input *values* randomly at evaluation time, and our "+ IA" variants further randomize input *shapes* during training (Appendix E.1.2). The probability that a training instance matches a specific KernelBench input is therefore near zero by construction.

E.2 Hyperparameters and Training Setup

For generating reasoning and code traces for SFT, we use temperature=0.6, top_p=0.95 for DeepSeek-R1 (Guo et al., 2025) and temperature=1.0, top_p=1.0 for GPT-OSS 120B (OpenAI, 2025). To label difficulty, we further label tasks into three difficulty levels using Qwen3-235B-Instruct (Team, 2025) (temperature=0.7, top_p=0.8).

For SFT, we train for 2 epochs on DeepSeek-R1-generated data with a maximum sequence length of 12,288 tokens and 3 epochs on GPT-OSS 120B-generated data with a maximum sequence length of 16,384 tokens, all with a batch size of 16 and a learning rate of 1×10^{-5}.

For RL, we train the model for 2 epochs using the VeRL framework (Sheng et al., 2025) with a batch size of 32, a learning rate of 1×10^{-6}, a clip ratio of $\epsilon = 0.2$ for the GRPO objectives, the maximum prompt length 2,048, and the maximum response length 16,384. We use 8 NVIDIA A100 80GB GPUs for both SFT and RL training.

Note that TRITONRL (G), a model trained with SFT checkpoint distilled from GPT-OSS 120B, was trained with 8 NVIDIA H200 GPUs due to resource constraints. Also, we use KernelBook (Paliskara & Saroufim, 2025) tasks where input generating functions are augmented using GPT-OSS 120B for RL training.

Seeds. Unless stated otherwise, every number reported in this paper comes from a single training run. Evaluation draws $k = 10$ samples per task, so the reported pass@10 metrics already average over decoding randomness; the single-run caveat concerns training randomness only.

Baseline Evaluation Configuration. All baselines and all TRITONRL variants are evaluated under an identical configuration, so that the comparisons in Table 1 differ only in the model being evaluated. We use the same decoding parameters as during training, including the maximum response length, temperature, and `top_p`, together with the one-shot prompt

template of Appendix I.1 and a budget of $k = 10$ samples per task. Rollouts are generated on H200 gpus, while kernel compilation and runtime measurement are performed on L40S.

F Additional Experiment Results

F.1 Training Stability.

We analyze the training stability of TRITONRL by tracking the evolution of validity, correctness, and speedup metrics across 166 RL training iterations (2 epochs), applying a 10-iteration smoothing window to highlight underlying trends. As shown in Figure 5, TRITONRL with Hierarchical Reward Decomposition (HRD) demonstrates highly stable training dynamics, exhibiting steady improvements across all evaluated metrics without significant performance collapse. Furthermore, in Figure 6, when compared to baseline models trained with uniform reward designs, the uniform approaches, particularly those relying solely on speedup, exhibit more volatile training trajectories and performance fluctuations. In contrast, TRITONRL with HRD maintains more stable improvements across all metrics, demonstrating that the hierarchical reward structure provides more consistent learning signals, leading to more stable training.

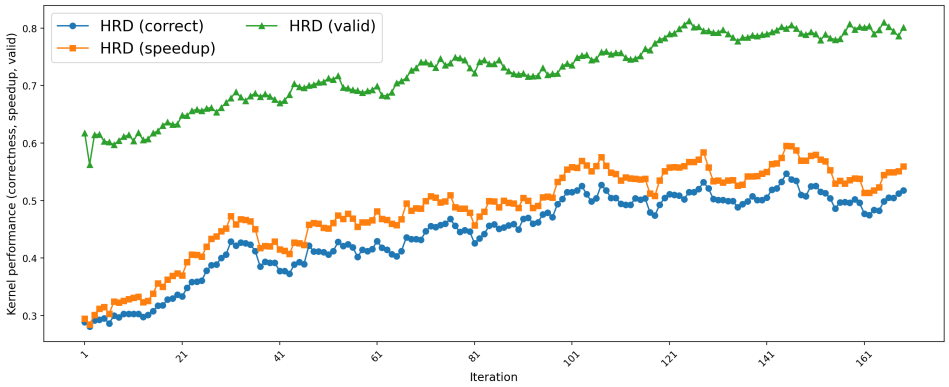


Figure 5: Training stability analysis of TRITONRL with hierarchical reward decomposition (HRD). The curves show the trends of validity, correctness, speedup for training data across RL training iterations (2 epochs).

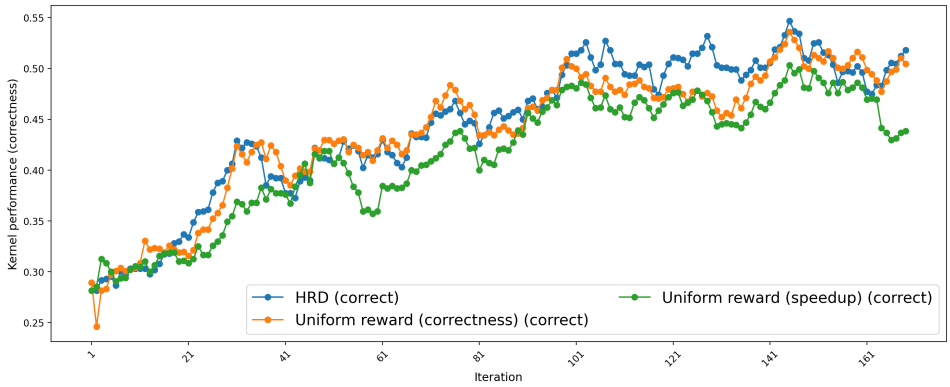
F.2 Pass@k Results

We evaluated inference scaling by varying the number of sampled attempts ($k = 1, 5, 10$), as illustrated in Figure 7 and Figure 8. The correctness and runtime efficiency of TRITONRL increases with more samples, whereas KernelLLM and Qwen3-8B show limited improvement, suggesting that TRITONRL generates a more diverse set of codes and benefits from additional sampling during inference.

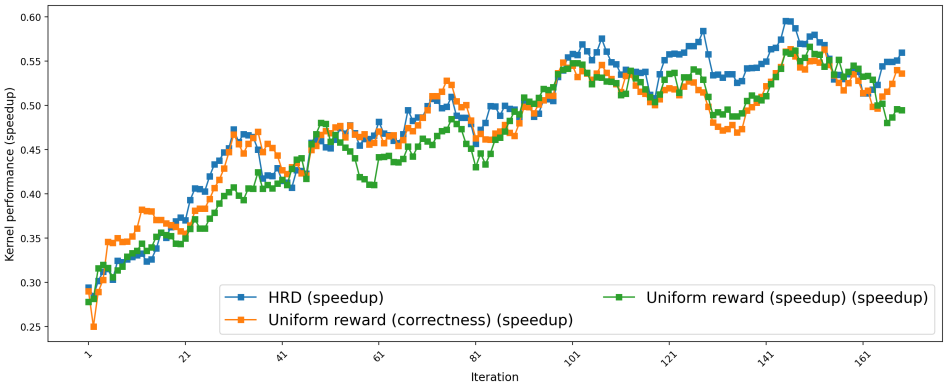
We also report pass@1 and pass@5 results for KernelBench Level 1 and Level 2 tasks in Table 6 and Table 7. These results further demonstrate the strong performance of TRITONRL in generating valid, correct, and efficient Triton code across various pass@k metrics.

F.3 Robustness gains from diverse input augmentation (IA)

To demonstrate the effectiveness of our input augmentation (IA) strategy in enhancing model robustness to diverse input shapes, we conducted additional experiments on input-augmented KernelBench Level 1 and Level 2 tasks, where five distinct input shapes were used for each of the 100 tasks per level (500 total evaluations per level). The results, presented in Table 8, indicate IA improves correctness and speedup for small k (pass@1 and pass@5), showing its role in enhancing robustness to input variations on initial attempts. As k increases to 10, the performance gap narrows or even reverses, suggesting that with sufficient sampling models can produce high-quality outputs regardless of exposure to



(a) Performance metric (Correctness).



(b) Performance metric (Speedup).

Figure 6: Training curves for TRITONRL and models with uniform reward designs (correctness and speedup).

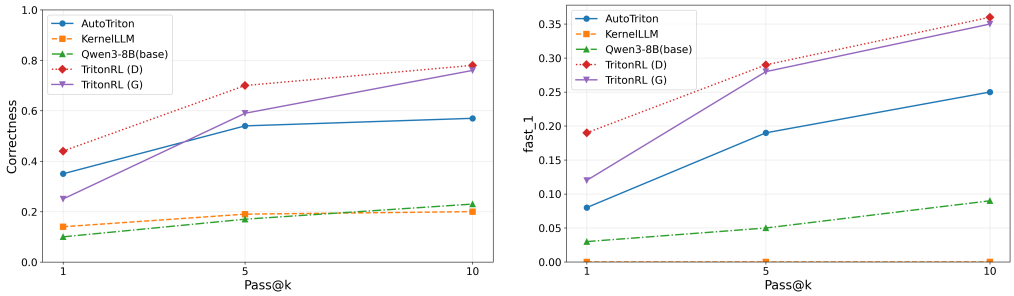


Figure 7: Pass@k correctness and fast$_1$ for $k = 1, 5, 10$ on KernelBench Level 1 tasks. The figures show how performance (correctness on the left, fast$_1$ on the right) of TRITONRL and baseline models scales as the number of attempts increases.

diverse input shapes during training. Nonetheless, IA remains beneficial for robustness, particularly in low-sample scenarios.

F.4 Ablation on Difficulty-Based Data Mixtures.

We explore different data mixtures for RL training of TRITONRL to understand how training data composition affects model performance. Our training dataset consists of two levels of tasks, with the ratio controlled by a data mixture probability vector $\boldsymbol{p} = [p_1, p_2]$, where p_1 and p_2 represent the probabilities of sampling Level 1 and Level 2 tasks, respectively. We evaluate TRITONRL on KernelBench Level 1 and 2 tasks using three data mixing strategies

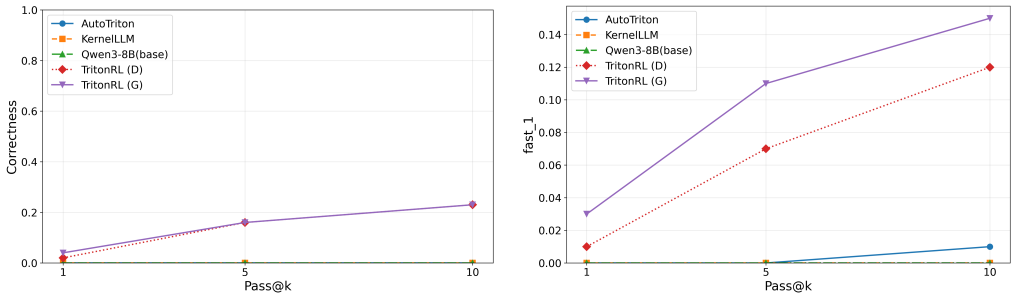


Figure 8: Pass@k correctness and fast_1 for $k = 1, 5, 10$ on KernelBench Level 2 tasks. The figures show how performance (correctness on the left, fast_1 on the right) of TRITONRL and baseline models scales as the number of attempts increases.

Table 6: Pass@k performance comparison for $k = 1, 5, 10$ on KernelBench Level 1 tasks.

Model	PASS@1			PASS@5			PASS@10		
	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2
Qwen3 (8B)	70.0	65.0 / 10.0	3.0 / 2.0	96.0	96.0 / 17.0	5.0 / 4.0	99.0	99.0 / 23.0	9.0 / 5.0
Qwen3 (14B)	80.0	73.0 / 17.0	4.0 / 2.0	99.0	96.0 / 23.0	7.0 / 6.0	100.0	99.0 / 35.0	9.0 / 7.0
Qwen3 (32B)	81.0	73.0 / 27.0	7.0 / 5.0	99.0	96.0 / 53.0	32.0 / 11.0	100.0	100.0 / 62.0	39.0 / 16.0
KernelLLM	32.0	29.0 / 14.0	0.0 / 0.0	41.0	39.0 / 19.0	0.0 / 0.0	42.0	40.0 / 20.0	0.0 / 0.0
AutoTriton	65.0	57.0 / 35.0	8.0 / 4.0	86.0	80.0 / 54.0	19.0 / 7.0	97.0	92.0 / 57.0	25.0 / 10.0
TRITONRL (D)	88.0	84.0 / 44.0	19.0 / 8.0	100.0	100.0 / 70.0	29.0 / 12.0	100.0	100.0 / 78.0	36.0 / 13.0
TRITONRL (D) + IA	88.0	83.0 / 37.0	12.0 / 9.0	100.0	99.0 / 69.0	28.0 / 14.0	100.0	99.0 / 78.0	34.0 / 15.0
SFT w. DeepSeek-R1 (w/o RL)	75.0	70.0 /31.0	13.0 / 7.0	99.0	98.0 / 49.0	25.0 /10.0	100.0	100.0 / 54.0	28.0 / 12.0
TRITONRL (G)	69.0	64.0 / 25.0	12.0 / 9.0	100.0	98.0 / 59.0	28.0 / 15.0	100.0	100.0 / 76.0	35.0 /17.0
TRITONRL (G) + IA	72.0	72.0 / 29.0	13.0 / 10.0	100.0	100.0 / 77.0	36.0 / 22.0	100.0	100.0 / 88.0	41.0 / 22.0
SFT w. GPT-OSS 120B (w/o RL)	43.0	43.0 / 15.0	6.0 / 5.0	94.0	92.0 / 48.0	19.0 / 15.0	100.0	100.0 / 64.0	26.0 / 18.0
Claude-3.7	79.0	74.0 / 31.0	10.0 / 5.0	100.0	100.0 / 51.0	21.0 / 11.0	100.0	100.0 / 63.0	29.0 / 16.0
DeepSeek-R1 (685B)	83.0	81.0 / 32.0	13.0 / 5.0	99.0	98.0 / 56.0	25.0 / 12.0	100.0	100.0 / 69.0	28.0 / 13.0
GPT-oss (120B)	79.0	78.0 / 46.0	18.0 / 10.0	100.0	100.0 / 69.0	30.0 / 18.0	100.0	100.0 / 81.0	33.0 / 22.0

Table 7: Pass@k performance comparison for $k = 1, 5, 10$ on KernelBench Level 2 tasks.

Model	PASS@1			PASS@5			PASS@10		
	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2
Qwen3 (8B)	7.0	7.0 / 0.0	0.0 / 0.0	29.0	29.0 / 0.0	0.0 / 0.0	38.0	37.0 / 1.0	0.0 / 0.0
Qwen3 (14B)	5.0	5.0 / 0.0	0.0 / 0.0	26.0	24.0 / 0.0	0.0 / 0.0	40.0	38.0 / 1.0	0.0 / 0.0
Qwen3 (32B)	10.0	10.0 / 1.0	1.0 / 1.0	28.0	26.0 / 2.0	2.0 / 1.0	36.0	35.0 / 2.0	2.0 / 1.0
KernelLLM	0.0	0.0 / 0.0	0.0 / 0.0	1.0	0.0 / 0.0	0.0 / 0.0	3.0	0.0 / 0.0	0.0 / 0.0
AutoTriton	1.0	1.0 / 0.0	0.0 / 0.0	5.0	5.0 / 0.0	0.0 / 0.0	11.0	10.0 / 1.0	0.0 / 0.0
TRITONRL (D)	28.0	27.0 / 2.0	1.0 / 0.0	55.0	55.0 / 16.0	7.0 / 1.0	63.0	63.0 / 23.0	12.0 / 3.0
TRITONRL (D) + IA	24.0	24.0 / 3.0	3.0 / 2.0	42.0	41.0 / 14.0	10.0 / 5.0	52.0	51.0 / 24.0	16.0 / 7.0
SFT w. DeepSeek-R1 (w/o RL)	22.0	19.0 / 4.0	4.0 / 0.0	50.0	45.0 / 10.0	9.0 / 2.0	55.0	51.0 / 13.0	10.0 / 3.0
TRITONRL (G)	49.0	47.0 / 4.0	3.0 / 1.0	88.0	87.0 / 16.0	11.0 / 2.0	91.0	91.0 / 23.0	15.0 / 7.0
TRITONRL (G) + IA	44.0	44.0 / 6.0	5.0 / 2.0	83.0	82.0 / 18.0	14.0 / 6.0	93.0	93.0 / 28.0	19.0 / 10.0
SFT w. GPT-OSS 120B (w/o RL)	41.0	40.0 / 4.0	4.0 / 0.0	80.0	79.0 / 10.0	7.0 / 1.0	88.0	88.0 / 13.0	9.0 / 3.0
Claude-3.7	16.0	13.0 / 2.0	0.0 / 0.0	32.0	31.0 / 12.0	3.0 / 0.0	34.0	34.0 / 14.0	4.0 / 1.0
DeepSeek-R1	7.0	5.0 / 1.0	0.0 / 0.0	24.0	23.0 / 7.0	4.0 / 3.0	31.0	31.0 / 14.0	9.0 / 4.0
GPT-oss (120B)	9.0	9.0 / 2.0	1.0 / 1.0	30.0	29.0 / 9.0	6.0 / 2.0	39.0	39.0 / 15.0	12.0 / 4.0

Table 8: Pass@k results (k = 1, 5, 10) on input-augmented KernelBench Level 1 and 2. Each of 100 tasks for each level was evaluated with five distinct input shapes (500 total for each level).

Model	Level 1								
	PASS@1			PASS@5			PASS@10		
	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2
TRITONRL (D)	88.0	85.0 / 37.0	12.0 / 6.0	100.0	100.0 /64.0	27.0 / 10.0	100.0	100.0 / 75.0	33.0 / 12.0
TRITONRL (D) + IA	85.0	81.0 / 39.0	14.0 / 7.0	100.0	99.0 / 65.0	27.0 / 11.0	100.0	99.0 / 72.0	32.0 / 13.0
TRITONRL (G)	64.0	62.0 / 30.0	12.0 / 7.0	99.0	99.0 / 67.0	26.0 / 17.0	100.0	100.0 / 80.0	31.0 / 19.0
TRITONRL (G) + IA	77.0	77.0 / 38.0	14.0 / 8.0	99.0	99.0 / 73.0	29.0 / 18.0	100.0	100.0 / 83.0	36.0 / 20.0
Model	**Level 2**								
	PASS@1			PASS@5			PASS@10		
	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2
TRITONRL (D)	31.0	29.0 / 5.0	2.0 / 1.0	56.0	54.0 / 18.0	9.0 / 3.0	66.0	65.0 / 27.0	15.0 / 5.0
TRITONRL (D) + IA	25.0	24.0 / 5.0	3.0 / 1.0	43.0	42.0 / 13.0	8.0 / 3.0	52.0	51.0 / 20.0	13.0 / 4.0
TRITONRL (G)	44.0	42.0 / 5.0	3.0 / 1.0	85.0	84.0 / 15.0	11.0 / 5.0	94.0	93.0 / 23.0	14.0 / 6.0
TRITONRL (G) + IA	56.0	54.0 / 8.0	6.0 / 3.0	88.0	86.0 / 22.0	14.0 / 7.0	94.0	93.0 / 29.0	19.0 / 10.0

for RL training: training exclusively on Level 1 tasks ($\boldsymbol{p} = [1, 0]$), exclusively on Level 2 tasks ($\boldsymbol{p} = [0, 1]$), and a balanced mixture of both ($\boldsymbol{p} = [0.5, 0.5]$), as summarized in Table 9.

Interestingly, training exclusively on Level 1 tasks ($\boldsymbol{p} = [1, 0]$) yields the best correctness and fast_1 on both Level 1 and Level 2 evaluations. Including Level 2 tasks during RL training provides only marginal improvements or worsens performance on Level 2 tasks. This may be because Level 2 tasks are inherently more complex, and reward signals from those tasks are very sparse, making it difficult for the model to learn effectively. Consequently, it suggests that focusing on fundamental single-operation tasks builds more robust Triton block generation skills that transfer effectively to both simple and complex fusion scenarios, rather than directly training on the more challenging fusion problems themselves.

Table 9: Ablation study on data mixture for RL training of TRITONRL, where the performance is evaluated on KernelBench level 1 and level 2 tasks. The models are trained from the same SFT checkpoint distilled from DeepSeek-R1.

Train Data Mixture	Mixing Prob. p	LEVEL1			LEVEL2		
		valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2
Level 1	$[1, 0]$	100.0	**100.0** / **78.0**	**36.0** / 13.0	**63.0**	**63.0** / **23.0**	**12.0** / 3.0
Level 1+2	$[0.5, 0.5]$	100.0	99.0 / 73.0	33.0 / **14.0**	**63.0**	**63.0** / 22.0	9.0 / 3.0
Level 2	$[0, 1]$	100.0	**100.0** / 70.0	30.0 / 10.0	50.0	50.0 / 16.0	9.0 / **4.0**

F.5 Ablation on Base Model Choices

In this section, we investigate the impact of different base models for fine-tuning with our RL framework. We compare two base models: Qwen3-8B, which is our main base model used throughout this paper, and Seed-Coder-8B-Reasoning (Zhang et al., 2025), which is the base model of AutoTriton (Li et al., 2025b). As shown in Table 10, TRITONRL fine-tuned from Seed-Coder-8B-Reasoning outperforms AutoTriton, which is also fine-tuned from the same base model, across all metrics on KernelBench Level 1 and Level 2 tasks. This demonstrates the effectiveness of our RL fine-tuning approach regardless of the base model. Furthermore, when comparing TRITONRL fine-tuned from Qwen3-8B and from Seed-Coder-8B-Reasoning, we observe that the former achieves significantly higher correctness and speedup on both Level 1 and Level 2 tasks, indicating that Qwen3-8B is a more effective foundation for our RL fine-tuning.

Table 10: Comparison of TRITONRL fine-tuned from different base models (Qwen3-8B and Seed-Coder-8B-Reasoning) on KernelBench Level 1 and Level 2. All metrics are reported as pass@10 (%). TRITONRL (G) indicates our model initialized from the GPT-OSS 120B SFT checkpoint; + IA denotes input augmentation. "Qwen3-8B + TRITONRL (G) + IA" and "Seed-Coder-8B-Reasoning + TRITONRL (G) + IA" refer to models fine-tuned with our RL framework from the respective base models under identical settings. AutoTriton uses Seed-Coder-8B-Reasoning as its base model and can be directly compared to "Seed-Coder-8B-Reasoning + TRITONRL (G) + IA".

Model	#Params	LEVEL1			LEVEL2		
		valid	compiled / correct	fast_1 / fast_2	valid	compiled / correct	fast_1 / fast_2
Seed-Coder-8B-Reasoning (base)	8B	91.0	82.0 / 19.0	7.0 / 6.0	49.0	12.0 / 1.0	1.0 / 0.0
AutoTriton	8B	97.0	92.0 / 57.0	25.0 / 10.0	11.0	10.0 / 1.0	0.0 / 0.0
Seed-Coder-8B-Reasoning + TRITONRL (G) + IA	8B	**99.0**	**99.0** / **61.0**	**27.0** / **17.0**	**86.0**	**85.0** / **17.0**	**11.0** / **5.0**
Qwen3-8B (base)	8B	99.0	99.0 / 23.0	9.0 / 5.0	38.0	37.0 / 0.0	0.0 / 0.0
Qwen3-8B + TRITONRL (G) + IA	8B	100.0	100.0 / 88.0	41.0 / 22.0	93.0	93.0 / 28.0	19.0 / 10.0

F.6 Ablation Study on Plan-to-Code Update Ratio (α)

We also analyze the effect of the hyperparameter α, which controls the relative update rate of planning versus coding actions during RL training. Smaller α values slow down plan token updates, allowing coding actions to adapt to more stable planning distributions. As shown in Table 11, $\alpha = 0.1$ achieves the best overall performance, while higher values (e.g., $\alpha = 1.0$) lead to instability and reduce correctness and speedup. Conversely, setting $\alpha = 0.0$, disabling plan updates, also degrades results, indicating that some plan adaptation is necessary. These findings highlight the importance of balancing plan and code updates to avoid premature convergence and maximize Triton code quality.

The per-response token-length statistics in Table 12 give this hyperparameter a precise interpretation. Plan tokens constitute 88–93% of a response across the base model and both TRITONRL variants, aggregated over KernelBench tasks with 10 rollouts per task. Consequently, at equal per-token weight the plan segment would contribute roughly an order of magnitude more gradient mass than the code segment; setting $\alpha = 0.1$ approximately *rebalances* the two segments' gradient contributions, letting the coding policy evolve at a rate comparable to the planning policy instead of being overwhelmed by it.

Table 11: Ablation study of the hyperparameter α in TRITONRL on KernelBench Level 1 tasks. All metrics are reported as pass@10 (%). α is the weighting factor, determining how quickly plan tokens are updated relative to code tokens during training. The default configuration uses $\alpha^* = 0.1$ and the models are trained from the same SFT checkpoint distilled from DeepSeek-R1.

α	valid	compiled / correct	$\texttt{fast}_1$ / $\texttt{fast}_2$
0.0	100.0	100.0 /72.0	32.0 / 12.0
0.1 (α^*)	100.0	100.0 / **78.0**	**36.0** / **13.0**
0.5	100.0	100.0 / 75.0	35.0 / 12.0
0.8	100.0	99.0 / 70.0	33.0 / 10.0
1.0	100.0	100.0 / 71.0	31.0 / 11.0

Table 12: Average plan / code token-length ratio per response, aggregated across KernelBench tasks (10 rollouts per task, then averaged over tasks).

Model	Level	Plan ratio	Code ratio
Qwen3-8B (base)	1	0.91	0.09
TRITONRL (D) + IA	1	0.89	0.11
TRITONRL (G) + IA	1	0.88	0.12
Qwen3-8B (base)	2	0.93	0.07
TRITONRL (D) + IA	2	0.91	0.09
TRITONRL (G) + IA	2	0.91	0.09

F.7 Comparison of Token-Class Reward Assignments

In GRPO, the advantage for the "code" component is computed relative to other codes sampled in the same group, but those codes may be paired with different prior reasoning traces (plans). Penalizing a correct implementation solely because it was conditioned on a weak plan is undesirable: it risks discouraging valid Triton implementations and degrading the base model's already limited Triton skills. Thus, we assign correctness-based rewards for code tokens to avoid such unwarranted penalties and to preserve model's capability to implement correct Triton code.

To validate this design and, more broadly, to map the full space of token-class reward assignments, we sweep all four combinations of the two reward signals $\{R^{\text{speedup}}, R^{\text{correct}}\}$ over the two token classes $\{\text{plan}, \text{code}\}$, all trained for the same $\alpha = 0.1$ starting from the same SFT checkpoint distilled from DeepSeek-R1. Our assignment ((2)), speedup on plan and correctness on code, is the first row of each block in Table 13.

Our assignment attains the best correctness and $\texttt{fast}_1$ on Level 1, and its margin is considerably larger on Level 2, where the two configurations that place the correctness signal on plan tokens lose 9–24 points of validity; this indicates that the speedup signal belongs on the planning segment. Holding the plan reward fixed to speedup, assigning correctness rather than speedup to the code tokens raises $\texttt{fast}_1$ from 33 to 36 on Level 1, consistent with the rationale above: correctness rewards prevent accurate implementations conditioned on suboptimal plans from being unfairly penalized, ultimately helping the model generate more efficient kernels.

Table 13: Four-corner reward-assignment ablation on KernelBench Level 1 and Level 2. All metrics are reported as pass@10 (%). All models are trained with $\alpha = 0.1$ from the same SFT checkpoint distilled from DeepSeek-R1.

LEVEL 1						
Plan reward	**Code reward**	**valid**	**compiled**	**correct**	**fast$_1$**	**fast$_2$**
speedup	correct (HRD)	100	100	**78**	**36**	13
speedup	speedup	100	100	**78**	33	13
correct	correct	100	100	69	31	12
correct	speedup	100	100	77	35	12
LEVEL 2						
Plan reward	**Code reward**	**valid**	**compiled**	**correct**	**fast$_1$**	**fast$_2$**
speedup	correct (HRD)	**63**	**63**	**23**	**12**	3
speedup	speedup	39	38	14	10	3
correct	correct	39	36	12	8	3
correct	speedup	54	52	20	**12**	3

F.8 Evaluation on KernelBench Level 3

Level 3 consists of full-architecture tasks and falls outside the single-operator and simple-fusion regime this work targets; in particular, Level-3 tasks were held out of the RL task pool, so the policy received no execution feedback on full-architecture kernels during training. Among fine-tuned 8B-scale models, TRITONRL produces more syntactically valid and compilable kernels than the Triton-specialized baselines KernelLLM and AutoTriton, but correctness is 0 for every compared model, which indicates how far Triton-specialized models at this scale currently stand from full-architecture kernel generation.

Table 14: KernelBench Level 3 (50 full-architecture tasks), pass@10 (%) across fine-tuned 8B-scale models. Level-3 tasks were excluded from the RL task pool.

Model	**valid**	**compiled**	**correct**	**fast$_1$**	**fast$_2$**
KernelLLM	0.0	0.0	0.0	0.0	0.0
AutoTriton	4.0	0.0	0.0	0.0	0.0
TRITONRL (D) + IA	22.0	20.0	0.0	0.0	0.0
TRITONRL (G) + IA	**32.0**	**30.0**	0.0	0.0	0.0

F.9 Cross-Benchmark Evaluation on TritonBench

To test whether the gains transfer across benchmark families, we additionally evaluate on TritonBench (Li et al., 2025a). We use the TritonBench-T tasks, providing PyTorch-aligned operator-generation tasks. We converted the TritonBench-T problems that take tensors and produce tensors (excluding Adam, SGD, random number generators, autocast, and `quantize_dynamic`) into the KernelBench task format expected by our evaluation infrastructure, and evaluated 8-B scale fine-tuned models (KernelLLM, AutoTriton, TRITONRL) under the identical protocol of Appendix E.2. As shown in Table 15, among fine-tuned 8B-scale models, both TRITONRL variants outperform the Triton-specialized baselines KernelLLM and AutoTriton on every metric, with the largest margins on the speed metrics.

Table 15: TritonBench-T pass@10 (%) across fine-tuned 8B-scale models.

Model	**valid**	**compiled / correct**	**fast$_1$ / fast$_2$**	**mean speedup**
KernelLLM	87.0	83.9 / 16.8	16.8 / 3.1	0.29
AutoTriton	68.3	68.3 / 42.9	5.6 / 1.2	0.39
TRITONRL (D) + IA	**95.7**	**95.7** / **56.5**	21.7 / **10.6**	**0.78**
TRITONRL (G) + IA	92.5	91.9 / 52.2	**23.0** / 6.8	0.63

G Data curation and examples

G.1 Data Mixing Subset Creation

We labeled difficulty level of 11k PyTorch reference codes in KernelBook based on the complexity of kernel implementation using Qwen3-235B-Instruct (Team, 2025). For each given PyTorch reference code, we prompt Qwen3-235B-Instruct (temperature=0.7, top_p=0.8) to label the difficulty level of replacing the PyTorch reference with Triton code as follows:

Instruction (input) example

‘‘‘ <PyTorch reference code> ‘‘‘
Assign a kernel implementation complexity level (1, 2, or 3) of the provided reference PyTorch architecture according to the criteria below:
• Level 1: Single primitive operation. This level includes the foundational building blocks of AI (e.g. convolutions, matrix-vector and matrix-matrix multiplications, losses, activations, and layer normalizations). Since PyTorch makes calls to several well-optimized and often closed-source kernels under-the-hood, it can be challenging for LMs to outperform the baseline for these primitive operations. However, if an LM succeeds, the open-source kernels could be an impactful alternative to the closed-source (e.g., CuBLAS [27]) kernels.
• Level 2: Operator sequences. This level includes AI workloads containing multiple primitive operations, which can be fused into a single kernel for improved performance (e.g., a combination of a convolution, ReLU, and bias). Since compiler-based tools such as the PyTorch compiler are effective at fusion, it can be challenging for LMs to outperform them. However, LMs may propose more complex algorithms compared to compiler rules.
• Level 3: This level includes architectures that power popular AI models, such as AlexNet and MiniGPT, collected from popular PyTorch repositories on GitHub.

G.2 Example of invalid Triton codes in KernelBook

The following is an example of invalid Triton code from KernelBook. The intended objective is to implement a convolution layer with bias addition using Triton. However, the generated code fails to correctly realize the convolution operation and bias addition within the Triton kernel, resulting in incorrect functionality. Specifically, this is not a valid Triton implementation of conv2d because the actual convolution is performed outside the Triton kernel using PyTorch modules. The Triton kernel itself only adds a bias term to the convolution output by loading and adding per-channel bias values; it does not perform any of the core convolution computations, such as sliding window access, dot products, or handling stride, padding, or dilation. Thus, while the Triton code is syntactically valid and functionally correct for bias addition, it does not implement conv2d and cannot be considered a genuine Triton-based convolution.

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

from torch._inductor.select_algorithm import extern_kernels
from torch._inductor.runtime.triton_heuristics import grid
from torch._C import _cuda_getCurrentRawStream as get_raw_stream
from torch._C._dynamo.guards import assert_size_stride


@triton.jit
def triton_poi_fused_convolution_0(in_out_ptr0, in_ptr0, xnumel, XBLOCK: tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
```

```
    tl.full([XBLOCK], True, tl.int1)

    x3 = xindex
    x1 = xindex // 4096 %

    tmp0 = tl.load(in_out_ptr0 + x3, None)
    tmp1 = tl.load(in_ptr0 + x1, None, eviction_policy='evict_last')

    tmp2 = tmp0 + tmp1

    tl.store(in_out_ptr0 + x3, tmp2, None)


def call(args):
    primals_1, primals_2, primals_3 = args
    args.clear()

    assert_size_stride(primals_1, (2, 3, 1, 1), (3, 1, 1, 1))
    assert_size_stride(primals_2, (2,), (1,))
    assert_size_stride(primals_3, (4, 3, 64, 64), (12288, 4096, 64, 1))

    with torch.cuda._DeviceGuard(0):
        torch.cuda.set_device(0)

        buf0 = extern_kernels.convolution(
            primals_3,
            primals_1,
            stride=(1, 1),
            padding=(0, 0),
            dilation=(1, 1),
            transposed=False,
            output_padding=(0, 0),
            groups=1,
            bias=None
        )

        assert_size_stride(buf0, (4, 2, 64, 64), (8192, 4096, 64, 1))

        buf1 = buf0
        del buf0

        get_raw_stream(0)

        triton_poi_fused_convolution_0[grid(32768)](
            buf1, primals_2, 32768,
            XBLOCK=128,
            num_warps=4,
            num_stages=1
        )

        del primals_2

    return buf1, primals_1, primals_3


class ModelNew(nn.Module):
    def __init__(self, num_classes=2, in_channels=3, is_deconv=False, is_batchnorm=False,
          *args, **kwargs):
        super(ModelNew, self).__init__()
        self.is_deconv = is_deconv
        self.in_channels = in_channels
        self.is_batchnorm = is_batchnorm

        self.final = nn.Conv2d(self.in_channels, num_classes, kernel_size=1)
```

```
    def forward(self, input_0):
        primals_1 = self.final.weight
        primals_2 = self.final.bias
        primals_3 = input_0

        output = call([primals_1, primals_2, primals_3])
        return output[0]
```

G.3 Example of invalid Triton codes for Level 2 tasks

The following is an example of invalid Triton code generated during RL training of TRITONRL. This level 2 task aims to implement a two-branch linear transformation with message passing, structured like a Graph Neural Network (GNN) layer. However, while the code implement linear transformations using a custom Triton kernel, it fails to fully realize the intended GNN functionality with only Triton kernels by using torch.bmm.

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

@triton.jit
def _linear_triton_kernel(
    input_ptr, weight_ptr, bias_ptr, output_ptr,
    B, M, N, K,
    stride_input_b, stride_input_m, stride_input_k,
    stride_weight_n, stride_weight_k,
    stride_output_b, stride_output_m, stride_output_n,
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
):
    pid_b = tl.program_id(0)
    pid_m = tl.program_id(1)
    pid_n = tl.program_id(2)

    pid_m = tl.multiple_of(pid_m * BLOCK_SIZE_M, BLOCK_SIZE_M)
    pid_n = tl.multiple_of(pid_n * BLOCK_SIZE_N, BLOCK_SIZE_N)

    input_ptr += pid_b * stride_input_b
    output_ptr += pid_b * stride_output_b

    offs_m = pid_m + tl.arange(0, BLOCK_SIZE_M)
    offs_n = pid_n + tl.arange(0, BLOCK_SIZE_N)
    offs_k = tl.arange(0, BLOCK_SIZE_K)

    input_ptrs = input_ptr + (offs_m[:, None] * stride_input_m + offs_k[None, :] *
        stride_input_k)
    weight_ptrs = weight_ptr + (offs_n[:, None] * stride_weight_n + offs_k[None, :] *
        stride_weight_k)

    acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)

    for k in range(0, K, BLOCK_SIZE_K):
        mask_k = (offs_k[None, :] < K - k)
        mask_m = (offs_m[:, None] < M)
        mask_n = (offs_n[None, :] < N)

        a = tl.load(input_ptrs, mask=mask_k & mask_m, other=0.0)
        b = tl.load(weight_ptrs, mask=mask_k & mask_n, other=0.0)

        b = tl.trans(b)
        acc += tl.dot(a, b)

        input_ptrs += BLOCK_SIZE_K * stride_input_k
```

```
        weight_ptrs += BLOCK_SIZE_K * stride_weight_k

    bias_ptrs = bias_ptr + offs_n
    bias = tl.load(bias_ptrs, mask=offs_n < N, other=0.0)
    acc += bias[None, :]

    offs_out_m = pid_m + tl.arange(0, BLOCK_SIZE_M)
    offs_out_n = pid_n + tl.arange(0, BLOCK_SIZE_N)
    out_ptrs = output_ptr + (offs_out_m[:, None] * stride_output_m + offs_out_n[None, :]
         * stride_output_n)

    mask = (offs_out_m[:, None] < M) & (offs_out_n[None, :] < N)
    tl.store(out_ptrs, acc, mask=mask)


def linear_triton(input, weight, bias):
    B, M, K = input.shape
    N = weight.shape[0]
    output = torch.empty((B, M, N), device=input.device, dtype=input.dtype)

    grid = (B, triton.cdiv(M, 64), triton.cdiv(N, 64))

    _linear_triton_kernel[grid](
        input, weight, bias, output,
        B, M, N, K,
        input.stride(0), input.stride(1), input.stride(2),
        weight.stride(0), weight.stride(1),
        output.stride(0), output.stride(1), output.stride(2),
        BLOCK_SIZE_M=64, BLOCK_SIZE_N=64, BLOCK_SIZE_K=64
    )

    return output


class ModelNew(nn.Module):
    def __init__(self, state_dim, name='', out_state_dim=None):
        super().__init__()
        self.state_dim = state_dim
        self.out_state_dim = out_state_dim if out_state_dim is not None else state_dim
        self.fc1 = nn.Linear(in_features=self.state_dim, out_features=self.out_state_dim)
        self.fc2 = nn.Linear(in_features=self.state_dim, out_features=self.out_state_dim)
        self.name = name

    def forward(self, input, adj):
        state_in = linear_triton(input, self.fc1.weight, self.fc1.bias)
        temp = torch.bmm(adj, input)
        forward_input = linear_triton(temp, self.fc2.weight, self.fc2.bias)
        return state_in + forward_input
```

G.4 Invalid Triton code examples

Here, we provide some examples of invalid Triton code generated by the baseline models.

Example 1: Triton syntax failure (syntax = 0) – no Triton kernel. This is not a Triton kernel because it does not include any custom `@triton.jit` kernel.

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

# To replace with Triton, a custom kernel would be needed (not provided here)
```

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

    def forward(self, x):
        return F.avg_pool2d(x, kernel_size=3, stride=None, padding=0)

def get_inputs():
    a = torch.randn(1, 128).cuda()
    b = torch.randn(1, 128).cuda()
    return [a, b]

def get_init_inputs():
    return []
```

Example 2: Functional failure (syntax = 1, func = 0) – use of high-level PyTorch module. This implementation contains a Triton kernel, but the main computation (convolution) is still handled by the PyTorch `nn.Conv3d` module. The Triton kernel is only used for a simple addition, rather than implementing the full intended operation.

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


@triton.jit
def add_bias_kernel(
    out_ptr,
    bias_ptr,
    n_elements,
    out_C,
    out_D,
    out_W,
    out_H,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    out = tl.load(out_ptr + offsets, mask=mask, other=0.0)

    stride_channel = out_D * out_W * out_H
    channel_idx = (offsets // stride_channel) %
    bias = tl.load(bias_ptr + channel_idx, mask=mask, other=0.0)

    out = out + bias
    tl.store(out_ptr + offsets, out, mask=mask)


def triton_add_bias(out: torch.Tensor, bias: torch.Tensor):
    assert out.is_cuda and bias.is_cuda, "Tensors must be on CUDA."

    out = out.contiguous()
    bias = bias.contiguous()
    n_elements = out.numel()
    BLOCK_SIZE = 128

    grid = lambda meta: (
        (n_elements + meta["BLOCK_SIZE"] - 1) // meta["BLOCK_SIZE"],
    )
```

```
    batch_size, out_channels, D, W, H = out.shape

    add_bias_kernel[grid](
        out,
        bias,
        n_elements,
        out_channels,
        D,
        W,
        H,
        BLOCK_SIZE=BLOCK_SIZE
    )

    return out


class ModelNew(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        kernel_size: int,
        stride: int = 1,
        padding: int = 0,
        dilation: int = 1,
        groups: int = 1,
        bias: bool = False
    ):
        super(ModelNew, self).__init__()
        self.conv3d = nn.Conv3d(
            in_channels,
            out_channels,
            (kernel_size, kernel_size, kernel_size),
            stride=stride,
            padding=padding,
            dilation=dilation,
            groups=groups,
            bias=bias
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        out = self.conv3d(x)
        if self.conv3d.bias is not None:
            out = triton_add_bias(out, self.conv3d.bias)
        return out
```

Example 3: Functional failure (syntax = 1, func = 0) – hardcoded output and no meaningful computation. While the Triton kernel is syntactically correct, but it doesn't actually implement the intended operation (Group Normalization). The kernel doesn't compute mean or variance, which are essential for GroupNorm. To implement real GroupNorm, you'd need to compute per-group statistics and normalize accordingly. Also, it only loads the input tensor and writes it back unchanged.

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


@triton.jit
def groupnorm_kernel(
    x_ptr, # Pointer to x tensor
    y_ptr, # Pointer to y tensor (not used here)
```

```
    out_ptr, # Pointer to output tensor
    n_elements, # Total number of elements
    BLOCK_SIZE: tl.constexpr,
):
    # Each program handles a contiguous block of data of size BLOCK_SIZE
    block_start = tl.program_id(0) * BLOCK_SIZE
    # Create a range of offsets [0..BLOCK_SIZE-1]
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Mask to ensure we don't go out of bounds
    mask = offsets < n_elements
    # Load input value
    x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
    # Compute x squared
    x_squared = x * x
    # Store the result
    tl.store(out_ptr + offsets, x, mask=mask)


def triton_groupnorm(x: torch.Tensor, y: torch.Tensor):
    assert x.is_cuda and y.is_cuda, "Tensors must be on CUDA."
    x = x.contiguous()
    y = y.contiguous()

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

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

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

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


class ModelNew(nn.Module):
    def __init__(self, num_features: int, num_groups: int) -> None:
        super().__init__()
        self.num_features = num_features
        self.num_groups = num_groups

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Use Triton kernel for elementwise operations
        x_triton = triton_groupnorm(x, x)
        # Manually compute mean and variance (as Triton kernel only handles x)
        # Actual GroupNorm logic would go here
        # For this example, we return the Triton processed tensor
        return x_triton
```

G.5 SFT and RL Dataset construction with KernelBook

To synthesize SFT dataset, we extract 11,621 PyTorch reference codes from KernelBook, executable without errors, such as

```
import torch
import torch.nn as nn


class Model(nn.Module):

def __init__(self):
```

```
super(Model, self).__init__()

def forward(self, neighbor):
return torch.sum(neighbor, dim=1)


def get_inputs():
return [torch.rand([4, 4, 4, 4])]


def get_init_inputs():
return [[], {}]
```

For each given PyTorch reference code, we construct an instruction for a teacher model to generate CoTs and Triton kernels as:

Instruction (input) example

Your task is to write custom Triton kernels to replace as many PyTorch operators as possible in the given architecture, aiming for maximum speedup. You may implement multiple custom kernels, explore operator fusion (such as combining matmul and relu), or introduce algorithmic improvements (like online softmax). You are only limited by your imagination.
You are given the following architecture:
‘‘‘ <PyTorch reference code> ‘‘‘
You have to optimize the architecture named Model with custom Triton kernels. Optimize the architecture named Model with custom Triton kernels! Name your optimized output architecture ModelNew. Output the new code in codeblocks. Please generate real code, NOT pseudocode, make sure the code compiles and is fully functional. Just output the new model code, no other text, and NO testing code! Before writing a code, reflect on your idea to make sure that the implementation is correct and optimal.

Given the instruction for each PyTorch reference code, we collect (CoT, Triton kernel code) pairs from DeepSeek-R1 and construct outputs for SFT by concatenating the pairs as follows:

Triton kernel with CoT (output) example

```
 <think>
CoT
</think>
‘‘‘
<Triton kernel code>
‘‘‘
```

For RL training, we use the same instruction input as a prompt for the same set of Pytorch reference codes in KernelBook, without the output synthesized by DeepSeek-R1 because RL training only requires reward feedback, which can be directly obtained from executing the generated Triton code.

H LLM judge for functional validity verification

To further enhance the functional validity verifier beyond unit tests, we employ an LLM-based judge to identify potential cheating behaviors in the generated Triton code, which may pass unit tests and syntax checks but fail to implement the intended functionality. We utilize Qwen3-235B-Instruct (Team, 2025) as the judge model, prompted to determine if the Triton code genuinely implements the same functionality without relying on high-level PyTorch constructs or hardcoded constants. We design the following prompt that details the functional requirements of Triton kernels for the judge model:

Judge Instruction for Functional Validity Verification

Detect whether the following Triton kernel implementation of an operation is cheating or not. A kernel is considered 'cheating' according to the following rules:

1. **All Core Computation Logic Must Be Written in Triton**
- All core computation logic must be implemented in Triton and must not delegate any part of the computation to high-level PyTorch APIs, such as torch.nn or torch.nn.functional.
- Wrapping high-level torch APIs in a Triton kernel wrapper without expressing the computation logic in Triton is not acceptable.
- Examples:
(1). When implementing matrix multiplication in Triton, the use of `torch.matmul`, `@`, `torch.nn.Linear`, or similar high-order APIs is forbidden.
(2). When implementing layer normalization in Triton, the use of `torch.nn.functional.layer_norm` is forbidden.
(3). When implementing softmax in Triton, the use of `torch.softmax` is forbidden.
(4). When implementing convolution or convolution transpose, the use of `torch.nn.functional.conv2d` or `torch.nn.ConvTranspose3d` is forbidden.
(5). When implementing attentions in Triton, the use of transformer library or other high-level APIs is forbidden.

2. **No Trivial Identity or No-Op Computation**
- Any part of the Triton kernel should always apply a mathematically-nontrivial operation to the input tensors like computation, reduction, or data layout change.
- Examples:
(1). Kernels that merely copy input to output (i.e., 'output_val = input_val') without performing meaningful computation are not allowed.
(2). Kernels that does 'output_val = input_val + 0' or '* 1.0' are not allowed.

3. **No Omission of Necessary Computations**
- When implementing an operator that involves multiple components, the kernel must implement **all** required parts. Skipping any required mathematical component of the algorithm is considered cheating.
- Examples:
1. When implementing LayerNorm, the kernel must include computations of
- Mean and variance computation
- Normalization
- Scaling and bias if applicable
2. When implementing matmul followed by a transpose, the kernel must not omit the transpose of the matmul results.

4. **The Kernel Must Address a Known Performance Bottleneck**
- The operation should be **non-trivial** enough that writing a custom kernel could yield meaningful performance benefits using PyTorch API
- Examples:
- Fused element-wise ops (e.g., GELU + dropout + bias)
- Reductions (e.g., softmax, LayerNorm, RMSNorm)
- Operations that can benefit from different memory layout of intermediate tensors

5. **The kernel Should Make Efficient Use of Available Parallelism in GPU**
- Kernels should make use of block/thread-level parallelism ('program_id', 'arange', 'BLOCK_SIZE', etc.)
- No scalar-only computations or logic that doesn't scale with input size

H.1 Reliability of the LLM Judge in Validity Assessment

The integrity of the reward signal in our RL framework hinges on the ability to discern whether a generated kernel is genuinely valid or merely exploiting test-suite loopholes. Since the combination of a rule-based verifier and an LLM judge (Qwen3-235B) constitutes the core of our reward mechanism, ensuring that the LLM judge remains unbiased and accurately detects "cheating" is critical for stable policy optimization. To rigorously evaluate the reliability of this verification layer, we analyzed its error rates against ground-truth labels provided by both a frontier LLM (Opus 4.5 (Anthropic, 2025b)) and human experts.

For this assessment, we utilized a dataset of 4,000 kernels generated for KernelBench Level 1 tasks (100 problems, 10 samples per problem) by various Triton-specialized models, including KernelLLM, AutoTriton, and our TritonRL variants. We extracted functional validity labels from Opus 4.5, using the same prompt provided to Qwen3-235B (see Appendix H), and treated them as pseudo-ground truth to calculate False Positive (FP) and False Negative (FN) rates. Furthermore, we conducted a human evaluation on a random sample of 22 kernels that passed unit tests, ensuring an even mix of valid and invalid implementations as labeled by the frontier model.

As summarized in Table 17, the results demonstrate the robustness of our verification system:

- **Consistency with Frontier Models:** When the rule-based verifier (which checks basic syntax, mandatory Triton API usage, and exclusion of high-level PyTorch modules) is integrated with the LLM judge, the FP and FN rates are remarkably low at 2.0% across the 4,000 samples.
- **Alignment with Human Judgment:** In the human-labeled subset, the combined verification layer showed an FP rate of 5.0% and an FN rate of 9.0%.

This analysis confirms that the output of our LLM judge is highly consistent with both expert human intuition and frontier model reasoning. By providing a reward signal that is free from significant bias and resistant to shortcuts, our multi-layered verification framework establishes a trustworthy foundation for hardware-aware reinforcement learning.

We stress that Opus 4.5 is used here as a *training-independent reference judge*, not as ground truth. The purpose of the comparison is to test whether our training-time judge's assessments are specific to one model family, not to establish an absolute standard of validity.

H.2 Re-evaluation under a Training-Independent Judge

A distinct concern from judge *reliability* is judge *overfitting*: whether the reported gains reflect genuine kernel quality or exploitation of quirks specific to the Qwen3-235B-Instruct judge used to compute rewards during RL. To test this, we re-evaluated the key comparison models on the full KernelBench Level 1 set using Opus 4.5 (a model not used at any point during training, neither as a teacher, a labeler, nor a judge) as the functionality judge. Table 16 reports the result. TRITONRL retains its lead over both AutoTriton and the Qwen3 base under this independent judge, and absolute correctness agrees with Table 1 within 1–3 points across all rows. Had the RL policy overfitted to the training-time judge, a judge with a different training distribution would have ranked TRITONRL worse; we observe no such degradation.

I Evaluation and examples

I.1 Evaluation with KernelBench

To evaluate the trained models, we construct prompts for 250 tasks in KernelBench. Similar to KernelBook, KernelBench provides a reference PyTorch code for each task. For each given reference PyTorch code, we construct a prompt with one simple example pair of (PyTorch code, Triton kernel code), similarly to the one-shot prompting format in KernelBench. Here, we use the following PyTorch and Triton codes for a simple add operation as an example:

Table 16: KernelBench Level 1 pass@10 (%) with Opus 4.5 as the functionality judge, which is independent of training. Qwen3-235B-Instruct is the judge used at training time and reported in Table 1.

Model	#Params	valid	compiled / correct	$fast_1$ / $fast_2$
Qwen3 (base)	8B	97.0	96.0 / 24.0	8.0 / 5.0
AutoTriton	8B	97.0	90.0 / 54.0	22.0 / 9.0
TRITONRL (D)	8B	100.0	100.0 / 78.0	36.0 / 13.0
TRITONRL (D) + IA	8B	100.0	99.0 / 77.0	34.0 / 15.0
TRITONRL (G)	8B	100.0	100.0 / 76.0	35.0 / 17.0
TRITONRL (G) + IA	8B	100.0	100.0 / **88.0**	**41.0** / **22.0**

Table 17: Comparison of error rates based on ground-truth sources and kernel sampling methods. FP and FN denote False Positive and False Negative rates, respectively.

Ground-truth source	Kernel sampling condition	Total samples	FP (%)	FN (%)
Opus 4.5	All	4,000	2.0	2.0
Human	Correct & even mix of valid/invalid kernels[†]	22	5.0	9.0

[†]Labeled by Opus 4.5.

```
### PyTorch reference code ###
import torch
import torch.nn as nn
import torch.nn.functional as F

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

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

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

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

### Triton kernel code ###

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

@triton.jit
def add_kernel(
    x_ptr, # Pointer to first input
    y_ptr, # Pointer to second input
    out_ptr, # Pointer to output
    n_elements, # Total number of elements in input/output
    BLOCK_SIZE: tl.constexpr,
):
    # Each program handles a contiguous block of data of size BLOCK_SIZE
    block_start = tl.program_id(0) * BLOCK_SIZE
    # Create a range of offsets [0..BLOCK_SIZE-1]
```

```
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Mask to ensure we don't go out of bounds
    mask = offsets < n_elements
    # Load input values
    x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
    y = tl.load(y_ptr + offsets, mask=mask, other=0.0)
    # Perform the elementwise addition
    out = x + y
    # Store the result
    tl.store(out_ptr + offsets, out, mask=mask)

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

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

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

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

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

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

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

Prompt (input) example

You write custom Triton kernels to replace the pytorch operators in the given architecture to get speedups.
You have complete freedom to choose the set of operators you want to replace. You may make the decision to replace some operators with custom Triton kernels and leave others unchanged. You may replace multiple operators with custom implementations, consider operator fusion opportunities (combining multiple operators into a single kernel, for example, combining matmul+relu), or algorithmic changes (such as online softmax). You are only limited by your imagination.
Here's an example to show you the syntax of inline embedding custom Triton kernels in torch. The example given architecture is:
‘‘‘ <Example PyTorch code> ‘‘‘
The example new architecture with custom Triton kernels looks like this:
‘‘‘ <Example Triton kernel code> ‘‘‘
You are given the following architecture:
‘‘‘ <PyTorch reference code> ‘‘‘
Optimize the architecture named Model with custom Triton kernels! Name your optimized output architecture ModelNew. Output the new code in codeblocks. Please generate real code, NOT pseudocode, make sure the code compiles and is fully functional. Just output the new model code, no other text, and NO testing code!

I.2 Example of Triton codes generated by TRITONRL (D) with speedup > 1

Problem 12 in KernelBench Level 1: diagonal matrix multiplication with x11 speedup.

```
import torch
import triton
import triton.language as tl

@triton.jit
def fused_diag_matmul_kernel(
  vec_a_ptr, # Pointer to the diagonal vector A (N,)
  mat_b_ptr, # Pointer to the input dense matrix B (N, M)
  output_ptr, # Pointer to the output matrix (N, M)
  # --- Matrix dimensions ---
  N, # The size of the diagonal vector A
  M, # The number of columns in matrix B
  TOTAL_ELEMENTS, # Total number of elements in the output matrix (N * M)
  # --- Tuning parameters ---
  BLOCK_SIZE: tl.constexpr, # Number of elements each Triton program will handle
):
  """
  Computes C = diag(A) * B, where A is a vector representing the diagonal.
  This kernel treats the operation as a simple element-wise multiplication
  where each row of B is scaled by the corresponding element of A.
  """
  # 1. CALCULATE OFFSETS: Determine which elements this program instance will process.
  pid = tl.program_id(axis=0)
  block_start_offset = pid * BLOCK_SIZE
  offsets = block_start_offset + tl.arange(0, BLOCK_SIZE)

  # 2. CREATE BOUNDARY-CHECK MASK: Prevent out-of-bounds memory access.
  mask = offsets < TOTAL_ELEMENTS

  # 3. COMPUTE 2D INDICES FROM 1D OFFSETS: Find the row for vector A.
  row_indices = offsets // M

  # 4. LOAD DATA EFFICIENTLY: Read from global memory into registers.
  vec_a_elements = tl.load(vec_a_ptr + row_indices, mask=mask, other=0.0)
  mat_b_elements = tl.load(mat_b_ptr + offsets, mask=mask, other=0.0)
```

```
  # 5. PERFORM THE CORE COMPUTATION: The actual fused operation.
  output = vec_a_elements * mat_b_elements

  # 6. STORE THE RESULT: Write the computed values back to global memory.
  tl.store(output_ptr + offsets, output, mask=mask)
```

Problem 94 in KernelBench Level 1: MSE (Mean Squared Error) with x3 speedup.

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

@triton.jit
def mse_kernel(
    pred_ptr,
    target_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    # 1. Block Setup
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    # 2. Masked Load with Zero-Padding
    # CRITICAL: other=0.0 ensures that elements outside the valid range
    # contribute 0 to the sum, preventing calculation errors.
    pred_vals = tl.load(pred_ptr + offsets, mask=mask, other=0.0)
    target_vals = tl.load(target_ptr + offsets, mask=mask, other=0.0)

    # 3. Compute Squared Error
    # This happens in parallel across all threads in the block
    diff = pred_vals - target_vals
    sq_diff = diff * diff

    # 4. Intra-Block Reduction
    # tl.sum collapses the 1024 values in this block into a single scalar.
    # This is extremely fast as it uses shared memory/registers, avoiding VRAM.
    block_sum = tl.sum(sq_diff, axis=0)

    # 5. Global Accumulation
    # We use an atomic add to aggregate the partial sums from each block
    # into the single global output location.
    tl.atomic_add(output_ptr, block_sum)

def triton_mse(pred: torch.Tensor, target: torch.Tensor) -> torch.Tensor:
    # Ensure data layout is contiguous for coalesced memory access
    if not pred.is_contiguous(): pred = pred.contiguous()
    if not target.is_contiguous(): target = target.contiguous()

    n_elements = pred.numel()
    if n_elements == 0:
        return torch.tensor(0.0, device=pred.device)

    # Output must be initialized to zero because atomic_add accumulates on top of it
    output = torch.zeros(1, device=pred.device, dtype=pred.dtype)

    BLOCK_SIZE = 1024
    # Standard grid calculation
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
```

```
    mse_kernel[grid](pred, target, output, n_elements, BLOCK_SIZE=BLOCK_SIZE)

    # Final division by N happens on Python side (negligible cost)
    return output[0] / n_elements

class ModelNew(nn.Module):
    def forward(self, predictions, targets):
        return triton_mse(predictions, targets)
```

Problem 99 in KernelBench Level 1: Triplet Margin Loss with x4 speedup.

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

@triton.jit
def triplet_margin_loss_kernel(
    anchor_ptr,
    positive_ptr,
    negative_ptr,
    output_ptr,
    margin,
    # FEATURE_SIZE as constexpr allows the compiler to optimize the loop unrolling
    FEATURE_SIZE: tl.constexpr,
    BLOCK_SIZE: tl.constexpr,
):
    # 1. Parallelize by Batch (Row)
    # Each program instance (block) handles exactly one triplet sample from the batch.
    pid = tl.program_id(axis=0)

    # Calculate starting memory addresses for this specific row
    offset_start = pid * FEATURE_SIZE
    anchor_row = anchor_ptr + offset_start
    positive_row = positive_ptr + offset_start
    negative_row = negative_ptr + offset_start

    # Registers to accumulate squared distances across the loop
    ap_sum = 0.0
    an_sum = 0.0

    # 2. Tiled Processing of Features
    # We loop over the feature dimension in chunks (tiles) of BLOCK_SIZE.
    # This keeps register pressure low and allows handling features > 1024 dims.
    for j in range(0, FEATURE_SIZE, BLOCK_SIZE):
        # Generate offsets for the current tile
        offsets = j + tl.arange(0, BLOCK_SIZE)
        mask = offsets < FEATURE_SIZE

        # Load data chunks (coalesced reads)
        a = tl.load(anchor_row + offsets, mask=mask, other=0.0)
        p = tl.load(positive_row + offsets, mask=mask, other=0.0)
        n = tl.load(negative_row + offsets, mask=mask, other=0.0)

        # Vectorized math on the chunk
        diff_ap = a - p
        sq_ap = diff_ap * diff_ap

        diff_an = a - n
        sq_an = diff_an * diff_an

        # 3. Partial Reduction
        # Reduce the chunk to a scalar and add to the running total.
        # tl.sum is highly optimized for intra-block communication.
        ap_sum += tl.sum(sq_ap, axis=0)
```

```
        an_sum += tl.sum(sq_an, axis=0)

    # 4. Final Loss Calculation (per row)
    d_ap = tl.sqrt(ap_sum)
    d_an = tl.sqrt(an_sum)

    loss_i = d_ap - d_an + margin
    loss_i = tl.where(loss_i > 0, loss_i, 0.0) # ReLU

    # 5. Global Accumulation
    # Aggregate this row's loss into the total batch loss.
    # Note: For massive batch sizes, this atomic_add can become a contention hotspot.
    tl.atomic_add(output_ptr, loss_i)

class ModelNew(nn.Module):
    def __init__(self, margin=1.0):
        super().__init__()
        self.margin = margin

    def forward(self, anchor, positive, negative):
        # Ensure contiguous memory for safe pointer arithmetic
        anchor = anchor.contiguous()
        positive = positive.contiguous()
        negative = negative.contiguous()

        batch_size, n_features = anchor.shape

        # Allocate output tensor (scalar)
        total_loss = torch.zeros(1, device=anchor.device, dtype=anchor.dtype)

        # Grid Size = Batch Size (one block per sample)
        grid = (batch_size,)

        triplet_margin_loss_kernel[grid](
            anchor,
            positive,
            negative,
            total_loss,
            self.margin,
            FEATURE_SIZE=n_features, # Pass actual dimension dynamically
            BLOCK_SIZE=1024, # 1024 is standard for max occupancy
        )

        return total_loss[0] / batch_size
```