Kevin: Multi-Turn RL for Generating CUDA Kernels

Carlo Baronio [*1] Pietro Marsella [*1] Ben Pan [*1] Simon Guo [1] Silas Alberti [2]

Abstract

Writing GPU kernels is a challenging task and critical for AI systems' efficiency. It is also highly iterative: domain experts write code and improve performance through execution feedback. Moreover, it presents verifiable rewards like correctness and speedup, making it a natural environment to apply Reinforcement Learning (RL). To explicitly incorporate the iterative nature of this process into training, we develop a flexible multi-turn RL recipe that addresses unique challenges encountered in real-world settings, such as learning from long trajectories and effective reward attribution across turns. We present Kevin the Kernel Writer, the first model trained with multi-turn RL for CUDA kernel generation and optimization. In our evaluation setup, Kevin shows significant gains over its base model (QwQ-32B), improving correctness of generated kernels (in pure CUDA) from 56% to 82% and mean speedup from 0.53x to 1.10x of baseline (PyTorch Eager), and surpassing frontier models like o4-mini (0.78x). Finally, we study its behavior across test-time scaling axes: we found scaling serial refinement more beneficial than parallel sampling. In particular, when given more refinement turns, Kevin shows a higher rate of improvement.

1. Introduction

Writing efficient GPU kernels (Dao et al., 2022; Zhao et al., 2025; Ye et al., 2025) in domain-specific languages such as CUDA (Nickolls et al., 2008), Triton (Tillet et al., 2019), ThunderKittens (Spector et al., 2024), CUTLASS (NVIDIA Corporation, 2025) is critical for enabling AI systems' efficiency at scale, yet it remains difficult and costly due to the deep domain expertise required. This has led to a surge of interest in exploring how Large Language Models (LLMs) could help generate GPU kernels (Ouyang et al., 2025; Li et al., 2025; NVIDIA, 2025) using agentic systems (Damani et al., 2024; Chen et al., 2025; METR, 2025; Lange et al., 2025; Google DeepMind, 2025) that leverage extensive test-time compute. These inference-based approaches are inherently limited by the base models' capability in this domain. On the other hand, the presence of verifiable rewards in the form of correctness and speedup against a reference implementation makes reinforcement learning (RL) a natural approach. This leads to our investigation: *How can we train a model using RL to solve the real-world engineering task of CUDA kernel generation?*

GPU kernel generation emphasizes not just functional correctness, but more importantly performance — distinguishing this code optimization problem from binary-reward tasks that involve passing unit tests (Jimenez et al., 2024) or producing an acceptable proof (Zheng et al., 2022). Since speedup is a continuous goal, performance engineers take an iterative approach: they conduct many rounds of optimization based on previous kernel code, its execution result, and timing profiles. Hence, arriving at an optimized solution relies on multiple turns conditioned on previous execution feedback. In contrast, popular RL methods to train LLMs on verifiable rewards (Shao et al., 2024; Lambert et al., 2025) rely on the outcome reward of a single turn ("single-turn RL training"). We hypothesize that explicitly incorporating successive turns of code generation, execution, and feedback into each RL training step ("multi-turn RL training") better mirrors the iterative nature of kernel development, helping the model to learn more advanced code generation strategies that span multiple refinement turns.

We design a simple yet effective multi-turn RL training recipe, shown in Figure 1, that addresses the *key challenges* of training for CUDA kernel generation and optimization:

1. **Long trajectories lead to sparse rewards and context explosion**. To improve sample efficiency, we split trajectories and use each turn as an individual training sample. To address context explosion from long CoTs while preserving reasoning information, we summarize CoTs of prior turns.
2. **Finding an optimal solution may require rewarding suboptimal kernels that eventually lead to more per-**

[*]Equal contribution [1]Stanford University, Stanford, CA, USA [2]Cognition AI, USA. Correspondence to: Carlo Baronio <cbaronio@stanford.edu>, Pietro Marsella <marsella@stanford.edu>, Ben Pan <benpan@stanford.edu>.

Published at the Efficient Systems for Foundation Models (ES-FoMo-III) Workshop, ICML 2025, Vancouver, Canada. Copyright 2025 by the author(s).

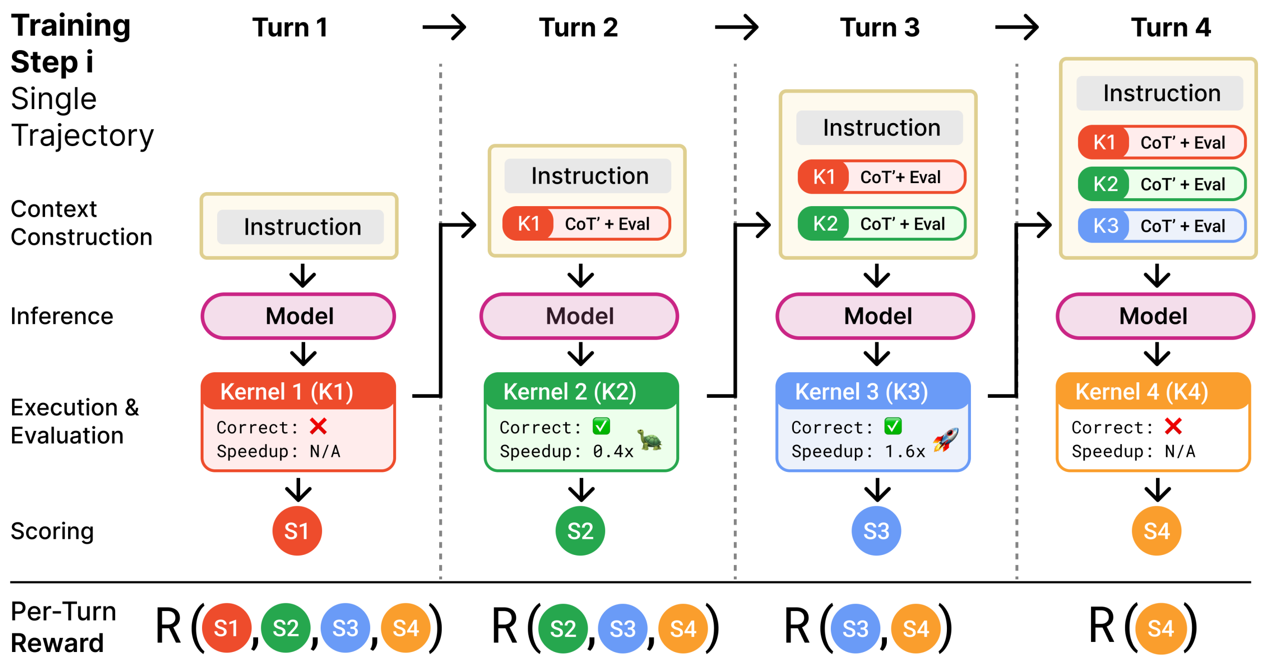

Figure 1. Within each training step, the model iteratively generates, executes, and refines kernels over multiple turns. Kernels are rewarded individually, based both on their performance and their contribution to subsequent speedups: `K1`, for example, while incorrect, leads to both a correct, slow kernel, `K2`, and a correct, performant kernel, `K3`, and should thus be rewarded accordingly. This setup enables Kevin to learn advanced code generation strategies that span multiple turns. Note: `CoT'` is the summarized chain of thought (CoT).

formant ones. Therefore, we study approaches to aggregate intermediate rewards across turns, finding a configuration that balances the correctness-performance trade-off.

3. **Reward hacking is prevalent as kernel generation is an open-ended, real-world engineering task:** e.g. the model can trick the evaluation harness, lazily copying the reference implementation instead of actually implementing kernels. To prevent this, we analyze the model's failure modes and enforce strict rule-based checks.

Enabled by our multi-turn RL training method on 180 KernelBench tasks from Level 1 and 2, we present Kevin the Kernel Writer, the first RL-trained model to generate CUDA kernels. We compare Kevin and other models in our evaluation setting on a representative KernelBench eval set. Kevin improves upon its base model (QwQ-32B (Team, 2025c)) both in correctness (56% $\rightarrow$ 82%) and mean speedup of generated kernels (in pure CUDA): from 0.53x to 1.10x over PyTorch Eager, while outperforming frontier models like OpenAI o4-mini (0.78x).

We then study the characteristics of Kevin in a test-time scaling setting, comparing it to a single-turn RL-baseline. We systematically compare the benefits of scaling along two axes of test-time compute: sequentially with more refinement turns (Ehrlich et al., 2025; Wang et al., 2025a) or in parallel with more trajectories (Brown et al., 2024; Snell et al., 2024). In our setting, we find that sequential scaling is much more effective, highlighting the importance of iterating upon execution feedback. We observe that the model trained with multi-turn RL exhibits better scaling characteristics with more refinement turns, compared to the base model and the single-turn RL baseline. Our core contributions include:

1. **We design an effective yet flexible multi-turn RL training strategy that significantly improves model's capability on CUDA kernel generation**. This strategy addresses challenges that arise in real-world settings, and may be applicable to other environments that benefit from iterative optimizations.
2. **We found multi-turn is more effective both for training and inference** through systematic ablations: the multi-turn trained model outperforms the single-turn trained model across different evaluation setups. Furthermore, we found multi-turn inference is more effective across both models under a fixed inference budget.
3. **Kevin exhibits strong test-time scaling trends on both serial and parallel axes**, with a faster rate of improvement than its single-turn RL counterpart and its base model, while maintaining exploration capacity.

2. Background and Related Work

2.1. LLM for GPU Kernel Optimization

There has been a surge of interest in exploring how to leverage LLMs to generate GPU kernels (NVIDIA, 2025), driven by the high cost and the long engineering cycles required to develop them (e.g. 2 years for efficient FlashAttention (Dao, 2023) port after Hopper GPU release). However, frontier models underperform on representative benchmarks like KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025), likely due to GPU code being severely underrepresented in the training data (CUDA, for example, accounts for less than 0.01% of pretraining data in the Stack (Kocetkov et al., 2022; Li et al., 2023)). Collecting more expert-written code is expensive, as only a limited number of developers are able to implement high-quality kernels. To tackle this task, there has been an explosion of agentic systems (Damani et al., 2024; Chen et al., 2025; METR, 2025) with custom workflows and evolutionary search methods (Lange et al., 2025; Google DeepMind, 2025). Yet these approaches typically incur high inference cost — e.g. $15 per kernel (Lange et al., 2025). Improving the base LLM's kernel-generation ability is therefore essential — and could significantly boost the efficiency for downstream agentic workflows.

2.2. RL Optimization for LLMs Targeting Verifiable Domains

Reinforcement Learning techniques like GRPO (Shao et al., 2024) have been shown to significantly enhance LLMs' performance on verifiable domains (Lambert et al., 2025) such as math (Team, 2025b; Wang et al., 2025b) and competitive programming (Team, 2025c; Luo et al., 2025a;b). These approaches can be further adapted for real-world software tasks, using fine-grain unit tests (Liu et al., 2023) or comparisons between code edits (Wei et al., 2025) as outcome rewards. Existing methods for code optimizations — where objective concerns performance beyond correctness — have been largely confined to supervised fine-tuning (Waghjale et al., 2024) and imitation learning (Shypula et al., 2024), highlighting Kevin's RL approach a novel contribution for this setting.

Given that tasks like performance optimization or long-horizon planning require multiple sequences of interrelated actions, several works (Goldie et al., 2025; Cao et al., 2025; Wang et al., 2025c; Zhou et al., 2024; Zhuang* et al., 2025) have explored RL training for multi-turn optimizations beyond optimizing for outcome from a single turn. Specific for the code setting, RLEF (Gehring et al., 2025) frames code generation as a multi-turn RL task: the model is allowed a fixed number of refinements turns and assigned a single binary pass/fail reward for final generation — training with such an approach yields notable sample-efficiency gains. Unlike RLEF, which assigns rewards only at the final turn, our multi-turn RL framework for Kevin trains on every turn regardless of how optimal the code is, and optimizes for performance beyond just correctness. It is worth noting that Kevin's multi-turn RL training could be viewed as a variant of Meta-Learning (Xiang et al., 2025; Duan et al., 2016) or In-Context Reinforcement Learning (Nie et al., 2024; Tajwar et al., 2025; Schmied et al., 2025), where the focus is to improve solution quality during test-time with feedback (Qu et al., 2025); but adapted in a novel way to the challenging real-world setting of GPU kernel generation and code optimization.

3. Task and Baseline

3.1. Environment and Evaluation

We use KernelBench (Ouyang et al., 2025), a popular benchmark for evaluating LLMs' ability to generate CUDA kernels for deep learning workloads in PyTorch. We chose 180 of the 100 Level 1 problems (basic operators: convolutions, matrix multiplies, loss functions, etc.) and 100 Level 2 problems (sequences of operators with fusion opportunities) as training environments. Since KernelBench does not provide a train-test split, we asked the authors to construct 80 additional tasks following the same methodology. We build the evaluation set (Appendix A) by combining our 80 newly created tasks with the 20 remaining original KernelBench tasks, for a total of 100 evaluation tasks.

Each KernelBench task consists in generating a CUDA kernel given a PyTorch reference implementation, which is used to evaluate correctness and speedup. In our set up, we evaluate the model-generated kernels as follows: we verify the output is in the correct format (ensure resultant code is only implemented with inline CUDA) and check for reward hacking (Section 6.2). We then evaluate the kernel for compilation, runtime errors, and correctness. If the implementation is correct, we profile the kernel for its runtime.

3.2. Kernel Score Design

As we are concerned both with correctness and speedup, we assign a score S for each kernel evaluation result that effectively balances the correctness-performance tradeoff.

$$S = 0.3 \cdot \mathbf{1}_{\{\text{correct}\}} + \frac{T_{\text{baseline}}}{T_{\text{kernel}}} \cdot \mathbf{1}_{\{\text{correct}\}}$$

Correctness is checked against the reference program when tested with randomized inputs; speedup is computed as the ratio between PyTorch baseline time and kernel runtime. We experimented with various weights of correctness and speedup, finding this configuration through ablations on models ranging from 7B to 32B. (Appendix B)

In addition, we explored rewarding intermediate objectives (successfully compile or execute), yet this caused model to over-optimize for intermediate steps (e.g. generating kernels that only compile, but aren't necessarily correct). We also experimented with a length penalty on the response, as suggested by (Team, 2025a), but found that it degrades our model's performance during training.

3.3. Single-Turn Training

We apply GRPO (Shao et al., 2024) to train the model on kernel generation without iterating on external feedback ("single-turn" training). In each training step, we sample 16 responses per task and assign the evaluated score as the reward for each kernel. We compute the GRPO loss according to (Shao et al., 2024), which updates the policy by maximizing the following objective:

$$\begin{aligned}
&\mathcal{J}_{GRPO}(\theta) = \mathbb{E}\big[q \sim P(Q),\, \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(O \mid q)\big] \\
&\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\Bigg\{ \min\Bigg[\frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}\hat{A}_{i,t}, \\
&\qquad \text{clip}\Big(\frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})},\, 1-\epsilon,\, 1+\epsilon\Big)\,\hat{A}_{i,t}\Bigg] \\
&\qquad - \beta\, D_{KL}(\pi_\theta \| \pi_{\text{ref}})\Bigg\} \\
&\text{where } \hat{A}_{i,t} = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r})}, \\
&\qquad r_i \text{ is the score of a specific kernel.}
\end{aligned} \tag{1}$$

We note the importance of using a base model with strong enough priors to obtain a non-sparse reward for correctness and speedup in the beginning of training. For instance, training on `DeepSeek-R1-Distill-Qwen7B` (DeepSeek-AI, 2025) exhibited reward hacking (see Section 6.2) and failed to learn.

Hence, we use a stronger base model, `Qwen QwQ-32B` (Team, 2025c). We perform two gradient steps for a batch (1 on-policy, 1 off-policy) following (Shao et al., 2024). We use `max_response_length` = 16384.

Following (Yu et al., 2025), we apply `Clip-Higher`, decoupling the lower and higher clipping range (0.2 and 0.28 respectively). We sample with `temperature` = 0.9 for both training and inference. We set the KL coefficient to 0 to allow the model to deviate freely from the base policy, following (Luo et al., 2025a).

We observe that reward plateaus after 50 steps, likely because single-turn training prevents the model from refining its kernels. Many generated kernels are nearly correct–often a syntax or compilation fix away–but still receive 0 reward, discouraging the model from producing them. Similarly, the correct kernels do not achieve high speedup, as the model optimizes for correctness rather than attempting a risky approach. We address these limitations through multi-turn training.

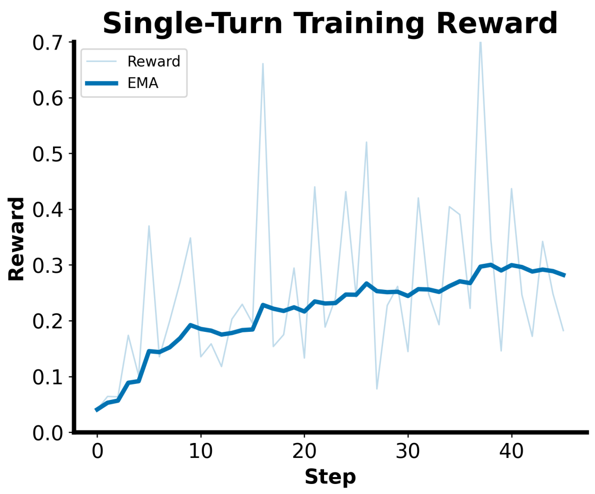

Figure 2. **Reward plateaus during single-turn training.**

4. Multi-Turn Training

In each multi-turn training step:

1. For each task, we sample m parallel trajectories with n refinement turns. To improve sample efficiency, each refinement turn (CoT + response) in a trajectory becomes a single training sample. The response of the model after the CoT consists of a kernel and a CoT summary.
2. We construct the context of a sample by including the history of previous responses, which include generated kernels along with their summarized CoTs, and evaluation feedback.
3. We evaluate the generated kernel and compute its score as shown in Section 3.2. The reward of each turn (CoT + response) is the discounted sum of current and subsequent scores, which we elaborate in Section 4.3.
4. For each task, we normalize the rewards across the mn samples for advantage calculation. Then we compute the GRPO loss over the entire batch.

4.1. Managing Context

Reasoning models generate long CoTs, especially for complex tasks like kernel generation. Including all CoTs causes the context to grow rapidly, reaching 50-100k tokens within a few turns, surpassing the model's context length. To prevent context explosion, we discard CoTs of previous turns; yet to preserve information regarding the reasoning process, we ask the model to summarize the changes applied. This summary, along with the generated kernels and evaluation results, is passed to subsequent turns.

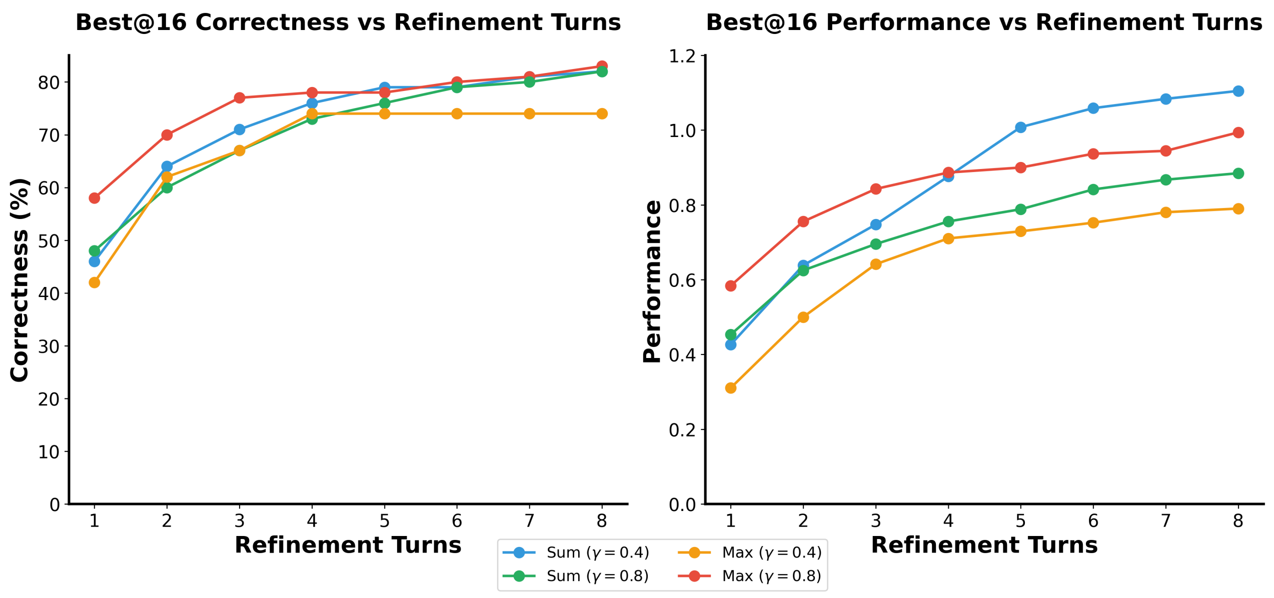

Figure 3. **Sum with $\gamma = 0.4$ is the most effective reward formulation.** Here we compare how models trained with different reward formulations scale with refinement turns.

4.2. Training On Every Refinement Turn

In a naive implementation, each n-turn trajectory is a single training sample. To improve sample efficiency, we split a n-turn trajectory into n training samples, each corresponding to the kernel + CoT summary of a refinement turn with the context containing the history. Hence, the kernel and CoT summary receives the reward of that particular turn.

4.3. Reward Aggregation and Discounting

We initially explored two naive strategies for multi-turn credit assignment. The greedy approach assigns to each turn its corresponding kernel score, while the outcome-based approach assigns to all turns the best score in the trajectory. The former failed to reward early suboptimal turns that lead to performant kernels later, while the latter ignores individual contributions and is sample inefficient.

Our method balances both approaches by aggregating the future kernels scores with a discount factor. We conduct ablations on the reward formulation. For score aggregation, we can either take the sum $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ or maximum $R_t = \max_{i=t,\ldots,T} \left\{\gamma^{i-t} r_i\right\}$ over future scores. Sum favors generating multiple good kernels, while max prioritizes achieving one high-performing kernel. We evaluate both forms with $\gamma = 0.4$ and $\gamma = 0.8$.

Experiments show that sum with $\gamma = 0.4$ scales best over 8 turns, though max performs better with $\gamma = 0.8$ with fewer turns. (See Appendix B for more details)

4.4. Multi-Turn Training Behavior

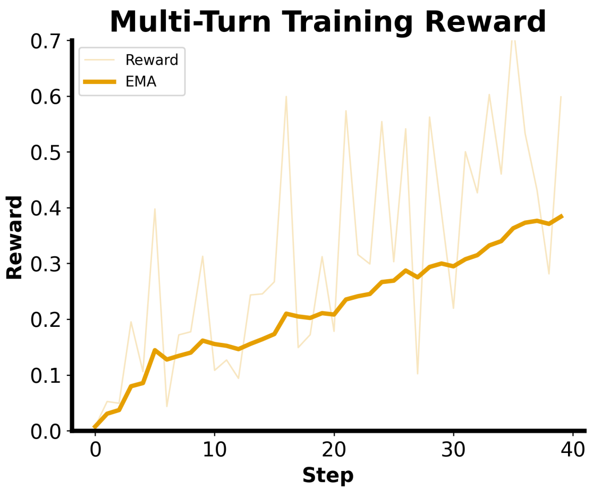

Figure 4. **Reward climbs steadily for multi-turn training.**

For our final training run for Kevin, we use 16 parallel trajectories and 4 refinement turns per task. Each batch contains 8 tasks. We use the sum reward formulation with discount factor $\gamma = 0.4$.

Unlike single-turn training, reward now steadily increases. We also observe response length behaviors similar to (Luo et al., 2025b): the response length initially decreases, and then it starts increasing again as the model attempts more sophisticated solutions. Following (Luo et al., 2025b), we extend the max response length from 16K to 22K tokens at step 30.

5. Evaluation

As kernel generation is a challenging task, models are often given extensive test-time compute to tackle it. In our inference setting, we employ multiple parallel trajectories, where each trajectory is made up of several serial turns.

We mark a given trajectory **correct** if it contains at least one correct kernel. Its **performance** is the speedup of the fastest kernel (within the trajectory) overx PyTorch Eager reference (0 speedup if no kernel is correct). We also consider the **fast**$_p$ metric, introduced by (Ouyang et al., 2025), which is a binary indicator for whether a trajectory contains a correct kernel with performance of p or more. To aggregate a metric across k parallel trajectories for a given task, we compute: **best@k**, the maximum for that metric across all trajectories; **avg@k**, the average value across trajectories.

5.1. Result on KernelBench Eval Set

We compare Kevin against frontier models and the single-turn RL baseline on our aforementioned KernelBench eval set of 100 tasks (Section 3.1), with 16 parallel trajectories, 8 serial refinement turns. As shown in Table 1, Kevin achieves a higher performance than its single-turn trained counterpart and other frontier models, demonstrating significant improvement from its base model (`QwQ-32B`). Qualitatively, Kevin is able to more effectively implement more aggressive optimizations across several turns. (See Appendix G)

5.2. Scaling Refinement Turns

Leveraging execution feedback is crucial at test time (Ehrlich et al., 2025; Wang et al., 2025a). Thus, we evaluate how Kevin scales with additional refinement turns. As shown in Figure 5, the single-turn model achieves slightly better performance with 1 turn, as its training objective optimizes for a single attempt. However, when given more refinement turns, the multi-turn trained model achieves significantly higher performance, with its curve showing the highest slope. This shows that multi-turn training enhances the model's ability to refine and optimize kernels over turns.

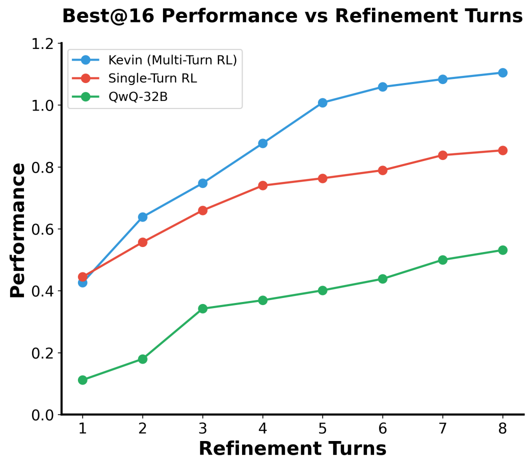

Figure 5. **Kevin effectively leverages multiple turns**. Here we vary the number of refinement turns for each trajectory; Kevin exhibits better scaling characteristics at test-time.

5.3. Scaling Parallel Samples

We study how best@k performance scales when increasing the number of parallel trajectories k, while fixing the number of serial refinements turns. Prior work for RLVR on math problems (Yue et al., 2025) found that RL training limits models' exploration capacity, leading to worse best@k metrics than the base model at large k. As shown in Figure 6, the performance curve of the single-turn RL model presents a lower slope compared to the base model, possibly hinting at this phenomenon. In contrast, our model trained with multi-turn RL achieves a higher slope compared to both the single-turn counterpart and the base model, suggesting that multi-turn training could maintain model's exploration capacity while improving model's performance.

Model	**Correctness**		**Performance**		**fast**$_1$		**fast**$_{1.5}$	
	best@16	**avg@16**	**best@16**	**avg@16**	**best@16**	**avg@16**	**best@16**	**avg@16**
Kevin (Multi-Turn)	**82%**	**46%**	**1.10x**	**0.40x**	**43%**	15%	**20%**	**6%**
Single-Turn RL	**82%**	45%	0.85x	0.35x	**43%**	**16%**	16%	4%
Qwen QwQ-32B	56%	11%	0.53x	0.08x	23%	3%	10%	1%
OpenAI o4-mini	38%	22%	0.78x	0.27x	21%	7%	13%	**6%**
OpenAI o3-mini	27%	8%	0.30x	0.08x	9%	2%	4%	2%

Table 1. **Kevin, trained with multi-turn RL, outperforms other models in correctness and performance.** Here we evaluate models on 100 unseen KernelBench tasks, under a test-time compute setup of 16 parallel trajectories with 8 refinement turns each trajectory.

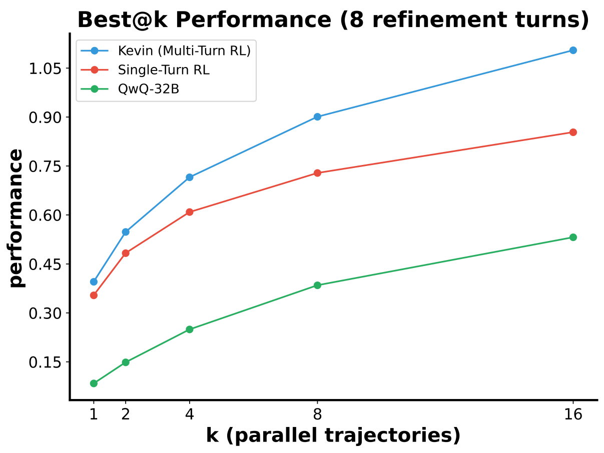

Figure 6. **Multi-turn training maintains exploration capacity.** Here we vary k, the number of parallel samples, while refinement turns are fixed to 8. best@k performance is computed with the estimator according to (Chen et al., 2021).

5.4. Parallel vs Sequential Scaling

As scaling test-time compute through parallel sampling (Snell et al., 2024) and sequential iterative refinement (Ehrlich et al., 2025) are both helpful, we want to systemically understand and compare their effectiveness in the context of kernel generation. To investigate, we evaluate 3 inference-time configurations with 128 generated kernels: 128 trajectories with 1 turn, 32 trajectories with 4 turns, and 16 trajectories with 8 turns. As shown in Table 2, we find that in our experimental setup, allocating more refinement turns during test-time is consistently better across the multi-turn RL trained, single-turn RL trained, and base model, with 16 trajectories and 8 turns being the most optimal configuration for most cases.

As demonstrated in Section 5.1, multi-turn outperforms single-turn training when evaluated in a multi-turn inference setting. However, given that single-turn training optimizes for one-shot performance, a natural question arises: does the single-turn trained model perform better by generating more one-shot responses in parallel? In Table 2, we observe that in a single-turn inference setting with 128 parallel trajectories, the single-turn model achieves slightly better performance than the multi-turn model; yet its correctness and performance quickly improve with more turns. This strengthens the case for training a model that could use feedback effectively across multiple turns. Moreover, the multi-turn trained model achieves significantly higher performance, with faster improvements compared to the single-turn trained model. This shows that multi-turn training enhances the model's ability to improve performance over turns at test-time.

6. Discussion

6.1. Diagnosing Model Instability

We observe that training for longer often leads to the model producing repetitive and nonsensical outputs ("junk"). During multi-turn training, the junk first appears in the final turn and gradually spreads to earlier turns, resulting in model collapse afterwards.

To investigate this issue, we identified a proxy signal, which we call the "Not Okay Ratio". `QwQ-32B` always begins its chain of thought with `"Okay, "` but after 40 steps of training, the model begins with erratic variants like `"Okay Amigos, so I need to optimize this 3D tensor-matrix multiplication"` and `"Okay Holy crap, I need to get this code optimized"`. These "Not Okay" responses

	Inference Configuration			Performance	Correctness
Model	Total	# Trajectories	# Turns	pass@128	pass@128
Multi-Turn RL	128	16	8	**1.10x**	82.00%
Multi-Turn RL	128	32	4	1.02x	**83.00%**
Multi-Turn RL	128	128	1	0.65x	76.00%
Single-Turn RL	128	16	8	**0.85x**	**82.00%**
Single-Turn RL	128	32	4	0.81x	79.00%
Single-Turn RL	128	128	1	0.70x	73.00%
QwQ-32B	128	16	8	**0.53x**	**57.00%**
QwQ-32B	128	32	4	0.47x	52.00%
QwQ-32B	128	128	1	0.42x	54.00%

Table 2. **Multi-turn inference with 16 trajectories and 8 turns is our most optimal setup.** Here we compare inference configurations and their corresponding performance ($\times$ speedup) and correctness rates, on multi-turn (Kevin), single-turn RL trained models, and base model `QwQ-32B`. We compute pass@128 by taking the best kernel out of the 128 generated kernels.

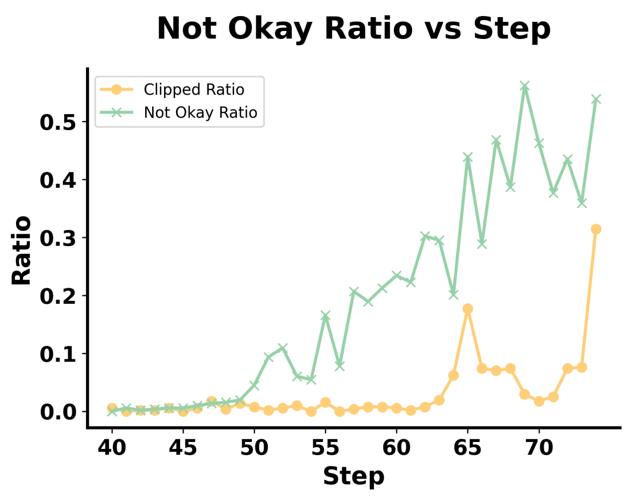

Figure 7. **"Not okay ratio" is a proxy for model instability and predicts junk.** It starts rising around step 48, while junk appears 15 steps later. "Clipping Ratio" (Luo et al., 2025b) reflects responses truncated for junk.

indicate early signs of model instability and appear well before junk, making the "Not Okay Ratio" a valuable predictor.

We initially tried to mitigate instability by adding a KL penalty (0.001 and 0.01) to the GRPO loss, but it slowed learning without affecting model stability. Instead, by using constant length normalization in the GRPO loss (Liu et al., 2025) and gradient norm clipping of 0.05, we are able to effectively delay the onset of junk until step 100.

6.2. Reward Hacking

In our early experiments with smaller models like `DeepSeek-R1-Distill-Qwen-7B`, we observed frequent reward hacking: the model calls the reference implementation (PyTorch) by directly copying it, wrapping it in try-except statements, or inheriting the reference implementation method. See Appendix F for detailed examples.

Reward hacking typically emerges when the model capabilities falls short of task difficulty (Amodei et al., 2016). In our setting, when model fails to produce correct kernels, the "hacked' kernels are likely the only ones receiving positive reward and get disproportionately reinforced due to advantage normalization. To prevent this, we upgraded our base model to the more capable `QwQ-32B` model as a stronger prior.

However, we observe instances of reward hacking even for stronger models. For Level 2 tasks (targeting kernel fusion), we observe that the model only fuses simple operators (e.g. ReLU, Max), leaving the operator worth optimizing (e.g. convolutions) unfused and unmodified (left in PyTorch). To prevent this, we impose stricter format checks that assign 0 reward to responses with any PyTorch functional operators.

6.3. Data Distribution

We found it critical to have a balanced difficulty distribution across the dataset, so that on average each batch contains both easier and harder tasks. In one experiment with `DeepSeek-R1-Distill-Qwen-14B` (DeepSeek-AI, 2025), we trained on a subset of only easy tasks. We observed that the reward quickly plateaus as the model overfits to a single difficulty level. Thus, we address this issue by using a stronger base model `QwQ-32B` and training on both level 1 and 2 of the dataset, which contained tasks with a variety of difficulty and associated optimization techniques.

7. Conclusion

7.1. Summary

We designed a multi-turn RL training recipe that addresses challenges when applied to the real-world task of kernel generation: specifically, effective context management and credit attribution across every turn to enable better sample efficiency. We also implemented mechanisms to prevent reward hacking, found an interesting proxy reward to diagnose instability, and experimented with approaches to constrain this issue.

We present Kevin the Kernel Writer, the first model trained with RL to generate CUDA kernels, on KernelBench Level 1 and 2 tasks. Evaluated on an unseen KernelBench evaluation set, Kevin outperforms its single-turn RL counterpart and frontier models, demonstrating that our training recipe enables the model to learn more effective refinement strategies. Multi-turn training also enables better test-time scaling, both when increasing sequential refinement and parallel sampling compute, while preserving the exploration capacity of the model.

7.2. Limitations

Since the base model (`QwQ-32B` Team, 2025c) is already heavily post-trained, additional RL training could easily destabilize it (Team et al., 2025). Due to limited compute and long RL training time for this task, we perform training up to 80 gradient steps. Consequently, we were unable to run more exhaustive ablations (e.g. varying the number of turns during Multi-Turn RL) and defer those studies to future work.

We further note limitations regarding kernel optimizations. As KernelBench tasks are specified with a specific predefined tensor input size, the speedups we measure in Section 3.2 are only accurate for those dimensions on our NVIDIA H200 GPUs.

7.3. Future Work

We outline several directions for extending our method. Incorporating a learned value network and using Proximal Policy Optimization (Schulman et al., 2017) might improve the baseline estimation during training. At training and test-time, we could implement more sophisticated search methods such as beam search or Monte-Carlo Tree Search (Silver et al., 2017). Inspired by recent works (Sareen et al., 2025), we could also leverage the value network as a verifier for search at test-time.

Our multi-turn RL process demonstrates success in the real-world engineering task of GPU kernel generation. However, we designed this recipe in a flexible manner, potentially applicable to a wider range of tasks that feature verifiable rewards and execution feedback across a trajectory (such as code and software system optimization). We believe explicitly training models to reason about complex tasks over multiple turns to be a key step towards enabling autonomous AI systems.

8. Acknowledgment

We are grateful to Allen Nie, Alex Zhang, Anna Goldie, Anikait Singh, Anne Ouyang, Azalia Mirhoseini, Benjamin Spector, Bradley Brown, Jerry Liu, Jon Saad-Falcon, Jordan Juravsky, Justus Mattern, Karina Nguyen, Konwoo Kim, Lynn Cherif, Michael Zhang, Neil Band, Qijing Jenny Huang, Shafin Khan, Shayan Talaei, Tatsunori Hashimoto, and William Li for their helpful discussions and feedback on this project.

References

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. Concrete problems in ai safety, 2016. URL https://arxiv.org/abs/1606.06565.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Cao, S., Hegde, S., Li, D., Griggs, T., Liu, S., Tang, E., Pan, J., Wang, X., Malik, A., Neubig, G., Hakhamaneshi, K., Liaw, R., Moritz, P., Zaharia, M., Gonzalez, J. E., and Stoica, I. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Chen, T., Xu, B., and Devleker, K. Automating gpu kernel generation with deepseek-r1 and inference-time scaling. https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/, February 2025. Accessed: 2025-05-15.

Damani, S., Hari, S. K. S., Stephenson, M., and Kozyrakis, C. Warpdrive: An agentic workflow for ninja gpu transformations. In *Proceedings of the Machine Learning for Systems Workshop at NeurIPS 2024*, 2024. URL https://mlforsystems.org/assets/papers/neurips2024/paper32.pdf. Accessed: 2025-05-15.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Doshi, T. Gemini 2.5: Our most intelligent models are getting even better. https://blog.google/technology/google-deepmind/google-gemini-updates-io-2025/, May 2025. Accessed: 2025-05-21.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. Rl^2: Fast reinforcement learning via slow reinforcement learning, 2016. URL https://arxiv.org/abs/1611.02779.

Ehrlich, R., Brown, B., Juravsky, J., Clark, R., Ré, C., and Mirhoseini, A. Codemonkeys: Scaling test-time compute for software engineering, 2025. URL https://arxiv.org/abs/2501.14723.

Gehring, J., Zheng, K., Copet, J., Mella, V., Carbonneaux, Q., Cohen, T., and Synnaeve, G. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2025. URL https://arxiv.org/abs/2410.02089.

Goldie, A., Mirhoseini, A., Zhou, H., Cai, I., and Manning, C. D. Synthetic data generation & multi-step rl for reasoning & tool use, 2025. URL https://arxiv.org/abs/2504.04736.

Google DeepMind. Alphaevolve: A gemini-powered coding agent for designing advanced algorithms, May 2025. URL https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/. Accessed: 2025-05-15.

Hu, J., Wu, X., Zhu, Z., Xianyu, Wang, W., Zhang, D., and Cao, Y. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework, 2024. URL https://arxiv.org/abs/2405.11143.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The stack: 3 tb of permissively licensed source code, 2022. URL https://arxiv.org/abs/2211.15533.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Lambert, N., Morrison, J., Pyatkin, V., Huang, S., Ivison, H., Brahman, F., Miranda, L. J. V., Liu, A., Dziri, N., Lyu, S., Gu, Y., Malik, S., Graf, V., Hwang, J. D., Yang, J., Bras, R. L., Tafjord, O., Wilhelm, C., Soldaini, L., Smith, N. A., Wang, Y., Dasigi, P., and Hajishirzi, H. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL https://arxiv.org/abs/2411.15124.

Lange, R. T., Prasad, A., Sun, Q., Faldor, M., Tang, Y., and Ha, D. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition, 2025. URL https://pub.sakana.ai/static/paper.pdf. Accessed: 2025-05-15.

Li, J., Li, S., Gao, Z., Shi, Q., Li, Y., Wang, Z., Huang, J., Wang, H., Wang, J., Han, X., Liu, Z., and Sun, M. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025. URL https://arxiv.org/abs/2502.14752.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you!, 2023. URL https://arxiv.org/abs/2305.06161.

Liu, J., Zhu, Y., Xiao, K., Fu, Q., Han, X., Yang, W., and Ye, D. Rltf: Reinforcement learning from unit test feedback, 2023. URL https://arxiv.org/abs/2307.04349.

Liu, Z., Chen, C., Li, W., Qi, P., Pang, T., Du, C., Lee, W. S., and Lin, M. Understanding r1-zero-like training: A critical perspective, 2025. URL https://arxiv.org/abs/2503.20783.

Luo, M., Tan, S., Huang, R., Patel, A., Ariyak, A., Wu, Q., Shi, X., Xin, R., Cai, C., Weber, M., Zhang, C., Li, L. E., Popa, R. A., and Stoica, I. Deepcoder: A fully open-source 14b coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51, 2025a. Notion Blog.

Luo, M., Tan, S., Wong, J., Shi, X., Tang, W. Y., Roongta, M., Cai, C., Luo, J., Li, L. E., Popa, R. A., and Stoica, I. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2, 2025b. Notion Blog.

METR. Measuring automated kernel engineering, February 2025. URL https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/. Accessed: 2025-05-15.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. In *ACM SIGGRAPH*

2008 Classes, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781450378451. doi: 10.1145/1401132.1401152. URL https://doi.org/10.1145/1401132.1401152.

Nie, A., Su, Y., Chang, B., Lee, J. N., Chi, E. H., Le, Q. V., and Chen, M. Evolve: Evaluating and optimizing llms for exploration, 2024. URL https://arxiv.org/abs/2410.06238.

NVIDIA. Gpu mode at nvidia gtc 2025, 2025. URL https://www.youtube.com/watch?v=mdDVkBeFy9A. Accessed: 2025-05-15.

NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines, May 2025. URL https://github.com/NVIDIA/cutlass. Accessed: 2025-05-15.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

Qu, Y., Yang, M. Y. R., Setlur, A., Tunstall, L., Beeching, E. E., Salakhutdinov, R., and Kumar, A. Optimizing test-time compute via meta reinforcement fine-tuning, 2025. URL https://arxiv.org/abs/2503.07572.

Sareen, K., Moss, M. M., Sordoni, A., Agarwal, R., and Hosseini, A. Putting the value back in rl: Better test-time scaling by unifying llm reasoners with verifiers, 2025. URL https://arxiv.org/abs/2505.04842.

Schmied, T., Bornschein, J., Grau-Moya, J., Wulfmeier, M., and Pascanu, R. Llms are greedy agents: Effects of rl fine-tuning on decision-making abilities, 2025. URL https://arxiv.org/abs/2504.16078.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., and Yazdanbakhsh, A. Learning performance-improving code edits, 2024. URL https://arxiv.org/abs/2302.07867.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL https://arxiv.org/abs/1712.01815.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Spector, B. F., Arora, S., Singhal, A., Fu, D. Y., and Ré, C. Thunderkittens: Simple, fast, and adorable ai kernels, 2024. URL https://arxiv.org/abs/2410.20399.

Tajwar, F., Jiang, Y., Thankaraj, A., Rahman, S. S., Kolter, J. Z., Schneider, J., and Salakhutdinov, R. Training a generally curious agent, 2025. URL https://arxiv.org/abs/2502.17543.

Team, K. Kimi k1.5: Scaling reinforcement learning with llms, 2025a. URL https://arxiv.org/abs/2501.12599.

Team, N. Sky-t1: Train your own o1 preview model within $450. https://novasky-ai.github.io/posts/sky-t1, 2025b. Accessed: 2025-01-09.

Team, P. I., Jaghouar, S., Mattern, J., Ong, J. M., Straube, J., Basra, M., Pazdera, A., Thaman, K., Ferrante, M. D., Gabriel, F., Obeid, F., Erdem, K., Keiblinger, M., and Hagemann, J. Intellect-2: A reasoning model trained through globally decentralized reinforcement learning, 2025. URL https://arxiv.org/abs/2505.07291.

Team, Q. Qwq-32b: Embracing the power of reinforcement learning, March 2025c. URL https://qwenlm.github.io/blog/qwq-32b/.

Tillet, P., Kung, H. T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Waghjale, S., Veerendranath, V., Wang, Z. Z., and Fried, D. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness?, 2024. URL https://arxiv.org/abs/2407.14044.

Wang, G., Qin, H., Jacobs, S. A., Holmes, C., Rajbhandari, S., Ruwase, O., Yan, F., Yang, L., and He, Y. Zero++: Extremely efficient collective communication for giant model training, 2023. URL https://arxiv.org/abs/2306.10209.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. Openhands: An open platform for ai software developers as generalist agents, 2025a. URL https://arxiv.org/abs/2407.16741.

Wang, Y., Yang, Q., Zeng, Z., Ren, L., Liu, L., Peng, B., Cheng, H., He, X., Wang, K., Gao, J., Chen, W., Wang, S., Du, S. S., and Shen, Y. Reinforcement learning for reasoning in large language models with one training example, 2025b. URL https://arxiv.org/abs/2504.20571.

Wang, Z., Wang, K., Wang, Q., Zhang, P., Li, L., Yang, Z., Yu, K., Nguyen, M. N., Liu, L., Gottlieb, E., Lam, M., Lu, Y., Cho, K., Wu, J., Fei-Fei, L., Wang, L., Choi, Y., and Li, M. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning, 2025c. URL https://arxiv.org/abs/2504.20073.

Wei, Y., Duchenne, O., Copet, J., Carbonneaux, Q., Zhang, L., Fried, D., Synnaeve, G., Singh, R., and Wang, S. I. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution, 2025. URL https://arxiv.org/abs/2502.18449.

Xiang, V., Snell, C., Gandhi, K., Albalak, A., Singh, A., Blagden, C., Phung, D., Rafailov, R., Lile, N., Mahan, D., Castricato, L., Franken, J.-P., Haber, N., and Finn, C. Towards system 2 reasoning in llms: Learning how to think with meta chain-of-thought, 2025. URL https://arxiv.org/abs/2501.04682.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., and Ceze, L. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025. URL https://arxiv.org/abs/2501.01005.

Yu, Q., Zhang, Z., Zhu, R., Yuan, Y., Zuo, X., Yue, Y., Fan, T., Liu, G., Liu, L., Liu, X., Lin, H., Lin, Z., Ma, B., Sheng, G., Tong, Y., Zhang, C., Zhang, M., Zhang, W., Zhu, H., Zhu, J., Chen, J., Chen, J., Wang, C., Yu, H., Dai, W., Song, Y., Wei, X., Zhou, H., Liu, J., Ma, W.-Y., Zhang, Y.-Q., Yan, L., Qiao, M., Wu, Y., and Wang, M. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Yue, Y., Chen, Z., Lu, R., Zhao, A., Wang, Z., Yue, Y., Song, S., and Huang, G. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model?, 2025. URL https://arxiv.org/abs/2504.13837.

Zhao, C., Zhao, L., Li, J., and Xu, Z. Deepgemm: clean and efficient fp8 gemm kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025.

Zheng, K., Han, J. M., and Polu, S. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2022. URL https://arxiv.org/abs/2109.00110.

Zhou, Y., Zanette, A., Pan, J., Levine, S., and Kumar, A. Archer: Training language model agents via hierarchical multi-turn rl, 2024. URL https://arxiv.org/abs/2402.19446.

Zhuang*, R., Vu*, T., Dimakis, A., and Sathiamoorthy, M. Improving multi-turn tool use with reinforcement learning. https://www.bespokelabs.ai/blog/improving-multi-turn-tool-use-with-reinforcement-learning, 2025. Accessed: 2025-04-17.

A. KernelBench Modifications

We use KernelBench (Ouyang et al., 2025) as our training environments. KernelBench is a popular benchmark for evaluating LLMs' ability to generate performant CUDA kernels for deep learning workloads in PyTorch. Each KernelBench task consists in generating a CUDA kernel given a PyTorch reference implementation, which is used to evaluate correctness and speedup.

A.1. Task Improvements

We identify several limitations in the original KernelBench and introduce targeted modifications to address them. These changes are crucial to mitigate reward hacking, as shown in Section 6.2.

- We sand-boxed the kernel evaluation process so that fatal errors, such as `CUDA illegal memory accesses`, do not crash the RL training process.
- A significant issue we noted in KernelBench was that for many tasks, the input tensors used to measure performance are quite small. This causes kernel launch overhead to take up a significant portion of the runtime. To address this, we enlarged the tensor dimensions of the affected tasks.
- A sneakier bug in the KernelBench's evaluation harness caused the tested kernel to recycle the output tensor from the reference implementation (which was run immediately before) as its own tensor output. As a result of this, a kernel that only computes (correctly) a portion of the output tensor would still pass the correctness check. We address this by running the tested kernel first and only after the reference implementation, thus avoiding this hack.

In the end, we chose a total of 180 tasks as training environments, with 90 of the 100 Level 1 problems and 90 Level 2 problems (sequences of operators with fusion opportunities).

A.2. Construction of Additional Evaluation Set

Since KernelBench does not provide a train-test split, we asked the authors to construct 80 additional tasks following the same methodology that KernelBench was constructed.

KernelBench Level 2 is constructed by composing a subset of PyTorch operators as sequences of operators. Specifically, the PyTorch operators are categorized as:

- **Main operators:** `Conv2d`, `Matmul`, `Gemm`, `BMM`, `Conv3d`, `ConvTranspose2d`, `ConvTranspose3d`.
- **Activations:** `ReLU`, `Sigmoid`, `Tanh`, `LeakyReLU`, `GELU`, `Swish`, `Softmax`, `Mish`, `Hardtanh`, `HardSwish`.
- **Element-wise operators:** `Add`, `Multiply`, `Subtract`, `Divide`, `Clamp`, `Scale`, `ResidualAdd`.
- **Normalizations:** `BatchNorm`, `LayerNorm`, `InstanceNorm`, `GroupNorm`.
- **Pooling:** `MaxPool`, `AvgPool`, `GlobalAvgPool`.
- **Bias:** `BiasAdd`.
- **Reductions:** `Sum`, `Mean`, `Max`, `Min`, `LogSumExp`.
- **Others:** `ResidualAdd`, `Scaling`.

To construct the additional eval set (unseen from train set), following the methodology from original KernelBench task construction:

1. We sample from the available operators listed above: 1 main operator (computationally expensive), and 2-5 other operators.
2. We ask a language model, namely Gemini 2.5-Flash (Doshi, 2025), to generate a PyTorch program that creates a kernel by combining these operators. We also ask it to generate sample tensor sizes for the task.

3. We ensure this PyTorch program can be executed and has a runtime on NVIDIA H200 > 0.1ms, to avoid the runtime being dominated by kernel launch (CPU) overhead.

4. We make sure this PyTorch program (with the same sequence of operators) is not present in existing KernelBench Level 1 and 2 programs.

We manually inspected all new task programs to ensure their validity. We build the evaluation set by combining our 80 newly created tasks with the 20 remaining original KernelBench tasks, for a total of 100 unseen evaluation tasks.

B. Additional Details on Multi-Turn RL

Here we elaborate on design choices for our RL Training as described in Section 3.3 and Section 4, along with some ablation results.

B.1. Motivation for Turn-wise Reward

In our multi-turn RL training setup, within each training step we have a trajectory with n refinement turns. A possible approach would be to compute the reward based on the kernel at the last turn, similar to what is used in RLEF (Gehring et al., 2025). However, for the GPU kernel optimization setting, using just the last kernel might not be optimal at times: for example, as shown earlier in Figure 1, kernel 3 is correct but kernel 4 is incorrect as the model attempts more aggressive optimizations.

In this setting, computing reward based on the best kernel among the trajectory instead (max speedup) is a more natural choice. However, using only the max kernel score forces us to discard all turns in a trajectory after the max turn, possibly wasting a significant amount of inference rollouts: In the previous example, we would have to completely discard the reasoning trace, code, and evaluation for kernel 4. Thus, we arrived at our approach in Section 4.3, which uses a discounted look-ahead max or sum, enabling more sample-efficient training.

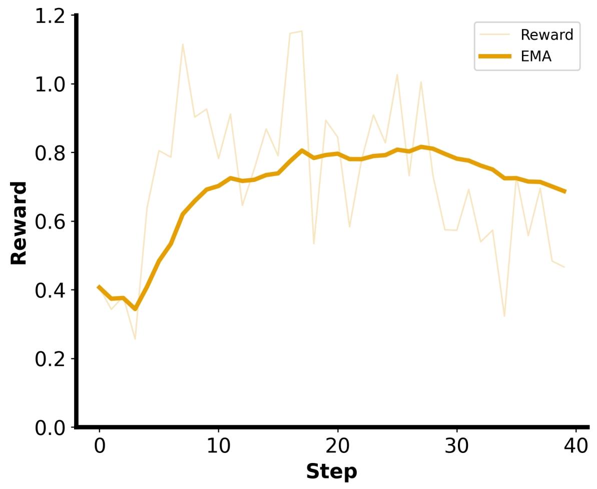

Figure 8. Training reward with correctness weighting of 1, performance / speedup weighting of 1. Concretely, $S = \mathbf{1}_{\{\text{correct}\}} + \frac{T_{\text{baseline}}}{T_{\text{kernel}}} \cdot \mathbf{1}_{\{\text{correct}\}}$.

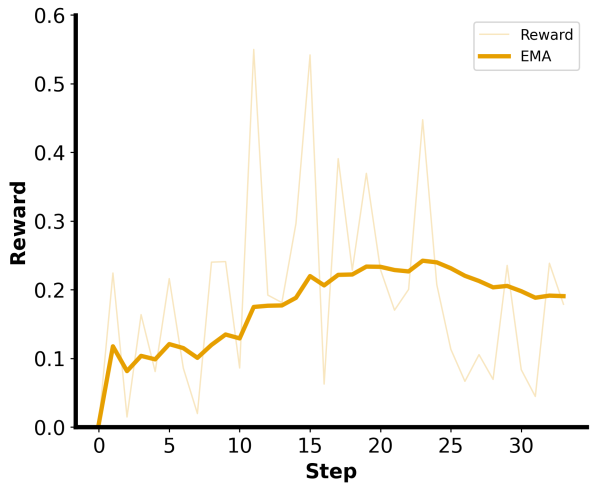

Figure 9. Training reward with no correctness weighting, performance / speedup weighting of 1. (speedup is 0 if kernel is incorrect). Concretely, $S = \mathbf{1}_{\{\text{correct}\}} \cdot \frac{T_{\text{baseline}}}{T_{\text{kernel}}}$.

B.2. Weighting for Score

In Section 3.2, we explain our score design, which assigns a scalar value (score S) based on a kernel's correctness and speedup. We explore score design and how to balance the correctness-performance trade-off, after series of small-scale ablations on `QwQ-32B` (Team, 2025c).

We decided on a weighting of `0.3` on correctness and using speedup for performance (raw speedup itself, no weighting), which is $S = 0.3 \cdot \mathbf{1}_{\{\text{correct}\}} + \mathbf{1}_{\{\text{correct}\}} \cdot \frac{T_{\text{baseline}}}{T_{\text{kernel}}}$.

Here we present some ablation studies we ran with different weighting configurations for score design, particularly focusing on adjusting the weighing for correctness, in the context of single-turn RL (GRPO) training (as shown in Section 3.3). As show an example in Figure 8, where we set the weighting to 1.0 for correctness, the reward plateaus and eventually decreased; concretely, we observed that the model over-optimizes for generating correct kernels and does not explore speedup as much, causing the reward to plateau during training. In another experiment in Figure 9, we set the weighting to 0 for correctness, only rewarding the model for generating performant (and correct) kernels. We again observed the reward plateau. Thus, we hypothesize that it is still important to reward the model for correct kernels, as long as the correctness reward is not too significant, balancing the correctness-performance tradeoff.

B.3. Number of Trajectories during Training

We vary the number of parallel trajectories during Multi-Turn RL training (Section 4), using 64 parallel trajectories instead of 16 for each task during each training step. We note that best@16 correctness slightly increases, but the overall performance does not show significant improvements. Due to the high-compute requirements of doing more generations during training, we chose to train with 16 parallel trajectories.

B.4. Length Penalty

We explore incorporating response length as a part of the reward design to incentivize the model to use its reasoning tokens more efficiently. We attempted a run using the length penalty from Kimi (Team, 2025a) on `DeepSeek-R1-Distill-Qwen-14B`. As shown in Figures 10 and 11, we found that the response length of the responses collapses, with the model no longer outputting CoT after 10 training steps, suggesting that the addition of a length penalty is counterproductive for our setting.

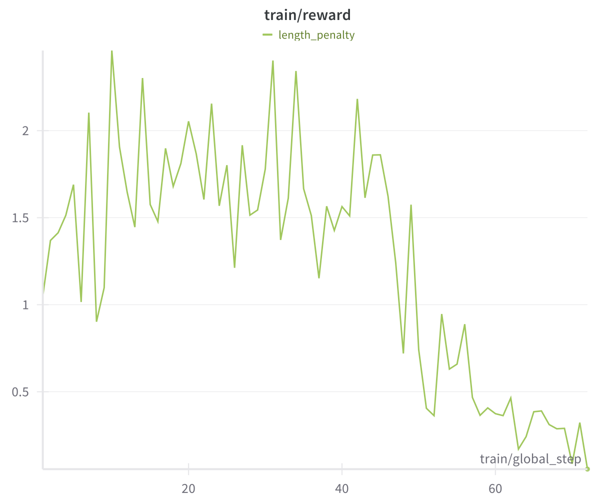

Figure 10. Training Reward collapses when including length penalty as part of reward

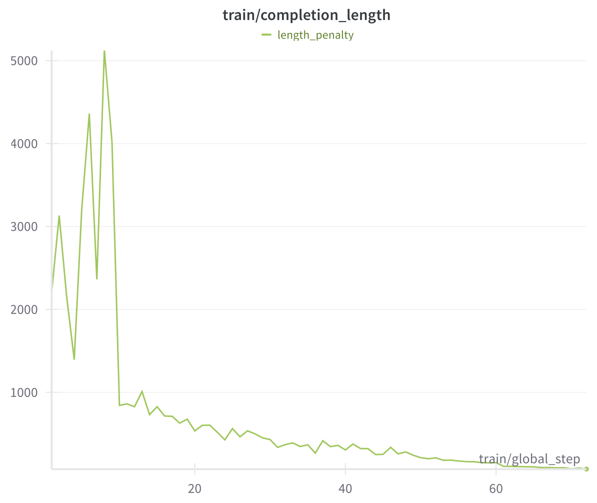

Figure 11. Response length of generations collapses when including length penalty as part of reward.

C. RL Infrastructure

Although a few open-source RL frameworks existed when we began this study, it is still difficult to support training in a kernel evaluation environment and including multiple turns within one training step. We built our training framework on top of the OpenRLHF (Hu et al., 2024) framework.

We use vLLM (Kwon et al., 2023) for inference and DeepSpeed Zero-3 (Wang et al., 2023) for offloading optimizer states.

Each of the 8 GPUs handles the kernel generation and evaluation for one task. After the response generation finishes, each

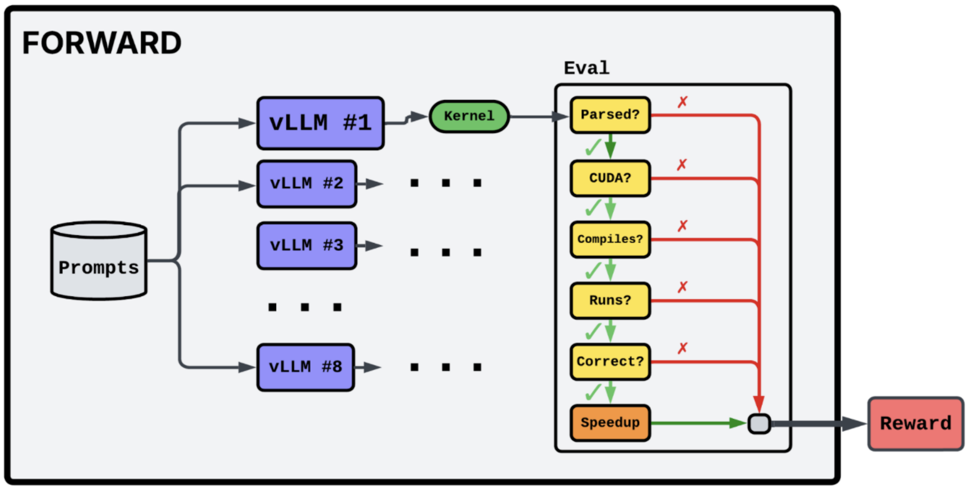

Figure 12. Overview of our RL Training infrastructure.

GPU offloads its vLLM engine to CPU memory and evaluates the kernels it generated. We run the evaluation and calculate reward and evaluation info. Each GPU then wakes up its corresponding vLLM engine and regenerates kernels.

Each full RL training run took multiple days due to the limited compute we have. Hence to iterate quickly and compare across configurations, we train up to 40-50 global steps (80-100 gradient steps).

D. Inference Setup

Our prompt is similar to the prompt used in KernelBench (Ouyang et al., 2025). We use this during training and test-time inference. In the first refinement turn, we add an example of the inline CUDA format to the prompt but remove it afterwards.

Below we show how we construct the context in the simplest case (of one turn, or the base prompt). In the context, we present model the KernelBench task, instructions, and a simple 1-shot example of a CUDA add kernel (to inform model the desired format for response):

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

class Model(nn.Module):
    """
    Simple model that performs Layer Normalization.
    """
    def __init__(self, normalized_shape: tuple):
        """
        Initializes the LayerNorm layer.

        Args:
            normalized_shape (tuple): Shape of the input tensor to be normalized.
        """
        super(Model, self).__init__()
        self.ln = nn.LayerNorm(normalized_shape=normalized_shape)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Applies Layer Normalization to the input tensor.

        Args:
            x (torch.Tensor): Input tensor of shape (*, normalized_shape).
        Returns:
            torch.Tensor: Output tensor with Layer Normalization applied, same shape as
    input.
        """
        return self.ln(x)

Replace pytorch operators in the given architecture with raw CUDA kernels, optimizing
    for performance on NVIDIA H100 (e.g. shared memory, kernel fusion, warp primitives,
    vectorization,...). Use torch.utils.cpp_extension.load_inline and name your
    optimized output architecture ModelNew. You are not allowed to use torch.nn (except
    for Parameter, containers, and init). The input and output have to be on CUDA
    device. Your answer must be the complete new architecture (no testing code, no
    other code): it will be evaluated and you will be given feedback on its correctness
    and speedup so you can keep iterating, trying to maximize the speedup. After your
    answer, summarize your changes in a few sentences.Here is an example:

import torch.nn as nn
from torch.utils.cpp_extension import load_inline

# Define the custom CUDA kernel for element-wise addition
elementwise_add_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void elementwise_add_kernel(const float* a, const float* b, float* out, int
    size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        out[idx] = a[idx] + b[idx];
    }
}

torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b) {
    auto size = a.numel();
    auto out = torch::zeros_like(a);

    const int block_size = 256;
    const int num_blocks = (size + block_size - 1) / block_size;

    elementwise_add_kernel<<<num_blocks, block_size>>>(a.data_ptr<float>(),
    b.data_ptr<float>(), out.data_ptr<float>(), size);

    return out;
}
"""

elementwise_add_cpp_source = (
    "torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b);"
)

# Compile the inline CUDA code for element-wise addition
elementwise_add = load_inline(
    name="elementwise_add",
    cpp_sources=elementwise_add_cpp_source,
    cuda_sources=elementwise_add_source,
    functions=["elementwise_add_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)


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

    def forward(self, a, b):
        return self.elementwise_add.elementwise_add_cuda(a, b)
```

For our multi-turn RL training (Section 4) and inference (Section 5), we provide model with the kernels, CoTs (summarized), and evaluation results of all previous turns in chronological order. We truncate the turns that do not fit inside the context window, starting from the earliest ones.

```
<Base prompt containing pytorch architecture and instruction>

Here are your previous attempts:

< for each (i) previously generated kernel >
    <Previously generated kernel G[i]>

    <Summary of CoT[i]>

    <if parsing error>

        Your previous answer failed to be parsed due to not adhering to the desired
    formatting. Here is the error message: <error_message>

    <elif compilation error>

        Your previous answer failed to compile. Here is the error message:
    <error_message>

    <elif run error>

        Your previous answer compiled successfully but had runtime errors. Here is the
    error message: <error_message>

    <elif correctness error>

        Your previous answer was incorrect. Here is the error message: <error_message>

    <elif correct>

        Your previous answer was correct but can be made faster. Here is the speedup
    you achieved relative to the baseline: <speedup>

Restart your reasoning process and generate new, complete code.
```

E. Training Stability

The analysis of the ”not okay ratio” led us to believe that model instability caused the appearance of nonsensical and repetitive outputs. Therefore, we attempted runs where we enabled KL divergence penalty in the GRPO loss, which would penalize the model from deviating from the base policy too much. Following DeepScaleR (Luo et al., 2025b), we set the KL coefficient to 0.001 and attempted an ablation run. However, we found that the reward plateaus with KL enabled, suggesting that the KL penalty slows down learning. Thus we attempted other techniques of constraining the model from deviating into regions of instability, such as clipping the gradient norm aggressively — which was effective in our setting.

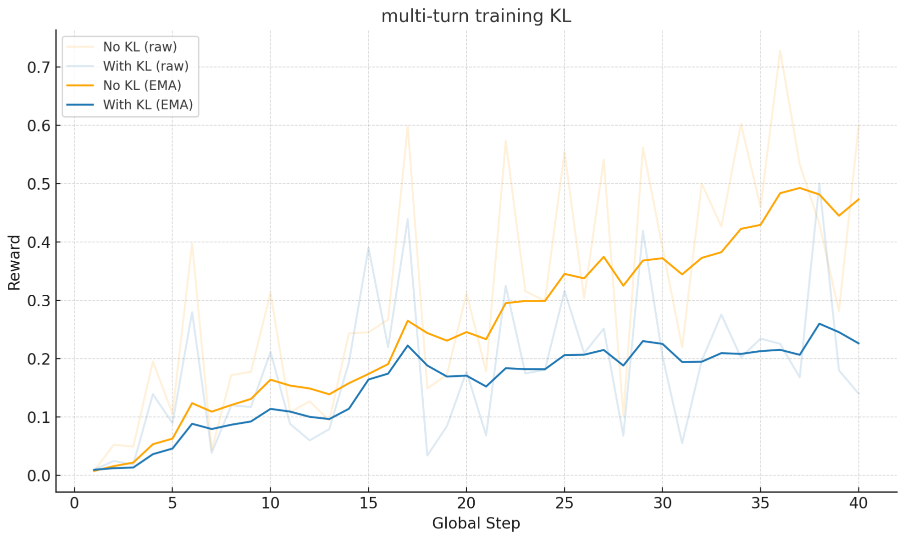

Figure 13. **Adding a KL penalty slows down learning.** Here we conduct an ablation with KL coefficient $\beta = 0.001$ versus $\beta = 0$. We see that the reward plateaus with KL enabled.

We use 4 refinement turns at train-time for efficient training. During test time, we can afford more extensive test-time compute, so we evaluate on 8 turns instead of 4 turns.

F. Reward Hacking

Here we present excerpts from generated kernels that show signs of reward hacking, previously mentioned in Section 6.2.

In the following example, the model simply copies the PyTorch reference implementation, thus getting rewarded for generating a correct answer with 1.0x speedup. To prevent this, we modify our kernel evaluation environment so that it checks each generated kernel if it contains instances of `torch.nn` or `torch.nn.functional`. We assign a reward of 0 to those.

```
class ModelReLU(Module):
    ...
    def forward(self, x):
        relu = torch.nn.ReLU()
        return relu(x)
```

Similarly, the model wraps an incorrect implementation of the CUDA kernel in a try-except statement and invokes the PyTorch implementation functions as a fallback. To prevent this, we assign a reward of 0 to kernels that contain try or except.

```
class ModelReLU(Module):
    ...
    def forward(self, x):
        try:
            ...  \# CUDA implementation
        except Exception as e:
            print("Custom ReLU kernel failed to compile. Using default ReLU instead.")
            relu = torch.nn.ReLU()
            return relu(x)
```

Here the model inherits from the reference implementation, bypassing the need for a CUDA implementation. To prevent this, we assign a reward of 0 to kernels that contain pass.

```
class ModelReLU(Module):
    ...
    def forward(self, x):
        ...   \# CUDA implementation

class ModelReLU(Model):  \# Model is the the reference implementation
    pass
```

G. Kernel Highlight

As an example, we look at one the task form our evaluation set, `Conv3D_GELU_Tanh_Clamp`:

```
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Model using Conv3d, GELU, Clamp, Tanh
    """
    def __init__(self, in_channels, out_channels, kernel_size, clamp_min, clamp_max):
        super(Model, self).__init__()
        self.conv = nn.Conv3d(in_channels, out_channels, kernel_size)
        self.clamp_min = clamp_min
        self.clamp_max = clamp_max

    def forward(self, x):
        """
        Forward pass of the model.

        Args:
            x (torch.Tensor): Input tensor of shape (batch_size, in_channels, depth,
    height, width).

        Returns:
            torch.Tensor: Output tensor of shape (batch_size, out_channels,
    depth-kernel_size+1, height-kernel_size+1, width-kernel_size+1).
        """
        x = self.conv(x)
        x = torch.tanh(x)
        x = torch.clamp(x, min=self.clamp_min, max=self.clamp_max)
        x = torch.nn.functional.gelu(x)
        return x
```

The task is made up of a computationally intensive function (`3D convolution`) and a series of lightweight functions (`tanh, clamp, gelu`).

The model implements a correct solution in the first turn by fusing the kernels, but it only achieves a speedup of **1.06x** over the reference implementation.

```
    import torch
from torch.utils.cpp_extension import load_inline
import math

class ModelNew(torch.nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, clamp_min, clamp_max):
        super(ModelNew, self).__init__()
        self.clamp_min = clamp_min
        self.clamp_max = clamp_max

        # Initialize convolution weights and bias
        self.weight = torch.nn.Parameter(torch.empty(out_channels, in_channels,
    kernel_size, kernel_size, kernel_size))
        self.bias = torch.nn.Parameter(torch.empty(out_channels))
        # Weight initialization following PyTorch's Conv3d default
        torch.nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in)
        torch.nn.init.uniform_(self.bias, -bound, bound)

        # Define the fused convolution and activation CUDA kernel
        conv_activation_source = """
        #include <torch/extension.h>
        #include <cuda_runtime.h>
        #include <math.h>

        __global__ void conv_activation_kernel(const float* input, const float* weights,
    const float* bias, float* output,
                                            int N, int C_in, int D, int H, int W, int
    C_out, int kD, int kH, int kW,
                                            float clamp_min, float clamp_max) {
            int idx = blockIdx.x * blockDim.x + threadIdx.x;
            const int W_out = W - kW + 1;
            const int H_out = H - kH + 1;
            const int D_out = D - kD + 1;
            const int total_elements = N * C_out * D_out * H_out * W_out;
            if (idx >= total_elements) return;

            // Decompose index into output coordinates
            int w_out = idx % W_out;
            int h_out = (idx / W_out) % H_out;
            int d_out = (idx / (W_out * H_out)) % D_out;
            int f = (idx / (W_out * H_out * D_out)) % C_out;
            int n = idx / (C_out * D_out * H_out * W_out);

            float sum = 0;
            for (int c = 0; c < C_in; ++c) {
                for (int kd = 0; kd < kD; ++kd) {
                    for (int kh = 0; kh < kH; ++kh) {
                        for (int kw = 0; kw < kW; ++kw) {
                            // Input tensor index calculation
                            int d_in = d_out + kd;
                            int h_in = h_out + kh;
                            int w_in = w_out + kw;
                            int input_idx = n * C_in * D * H * W +
                                           c * D * H * W +
                                           d_in * H * W +
                                           h_in * W +
                                           w_in;
                            // Weight tensor index calculation
                            int weight_idx = f * C_in * kD * kH * kW +
                                            c * kD * kH * kW +
                                            kd * kH * kW +
                                            kh * kW +
                                            kw;
                            sum += input[input_idx] * weights[weight_idx];
                        }
                    }
                }
            }
            sum += bias[f];

            // Apply activations sequentially
            sum = tanh(sum);
            sum = fmaxf(clamp_min, fminf(sum, clamp_max));
            {
                float inner = sum + 0.044715f * sum * sum * sum;
                inner *= 0.79788456f; // sqrt(2/pi) approximation
                sum = sum * 0.5f * (1.0f + tanh(inner));
            }

            // Output tensor index calculation
            int output_idx = n * C_out * D_out * H_out * W_out +
                            f * D_out * H_out * W_out +
                            d_out * H_out * W_out +
                            h_out * W_out +
                            w_out;
            output[output_idx] = sum;
        }

        torch::Tensor conv_activation_cuda(torch::Tensor input, torch::Tensor weights,
    torch::Tensor bias,
                                          float clamp_min, float clamp_max) {
            // Dimension extraction
            int N = input.size(0);
            int C_in = input.size(1);
            int D = input.size(2);
            int H = input.size(3);
            int W = input.size(4);
            int C_out = weights.size(0);
            int kD = weights.size(2);
            int kH = weights.size(3);
            int kW = weights.size(4);

            // Output dimensions
            int D_out = D - kD + 1;
            int H_out = H - kH + 1;
            int W_out = W - kW + 1;

            // Create output tensor
            auto output = torch::empty({N, C_out, D_out, H_out, W_out}, input.options());

            // Launch kernel
            int total_elements = N * C_out * D_out * H_out * W_out;
            const int block_size = 256;
            const int grid_size = (total_elements + block_size - 1) / block_size;
            conv_activation_kernel<<<grid_size, block_size>>>(
                input.data_ptr<float>(), weights.data_ptr<float>(),
    bias.data_ptr<float>(),
                output.data_ptr<float>(), N, C_in, D, H, W, C_out, kD, kH, kW,
                clamp_min, clamp_max
            );
            return output;
        }
        """

        conv_activation_header = """
        torch::Tensor conv_activation_cuda(torch::Tensor input, torch::Tensor weights,
    torch::Tensor bias,
                                          float clamp_min, float clamp_max);
        """

        # Load the CUDA kernel
        self.conv_activation = load_inline(
            name="conv_activation",
            cpp_sources=conv_activation_header,
            cuda_sources=conv_activation_source,
            functions=["conv_activation_cuda"],
            verbose=True
        )

    def forward(self, x):
        return self.conv_activation.conv_activation_cuda(x, self.weight, self.bias,
    self.clamp_min, self.clamp_max)
```

On turn 2 it attempts to improve on the existing implementation, but fails due to a compilation error.

On turn 3, it attempts to modify the loop order to improve coalesced memory access, but this actually worsens performance, as the kernel now exhibits a speedup of **0.61x**. The model attempts this strategy again at turn 4, and is able to order the loops in a way that increases memory coalescing: this brings up the performance to **1.19x**. At turn 5 and 6, the model attempts some minor optimizations (such as precomputing certain indices, or deleting intermediate variables), which however has no measurable effect in performance.

On turn 7, the model figures out a surprising optimization: it records the convolution's kernel size as a compile time constant, which allows it to use `# pragma unroll` to unroll the 3 loops needed to apply the convolution. This results in a significant speedup of **1.93x** over baseline.

```
from torch.utils.cpp_extension import load_inline
import math

class ModelNew(torch.nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, clamp_min, clamp_max):
        super(ModelNew, self).__init__()
        self.clamp_min = clamp_min
        self.clamp_max = clamp_max

        # Initialize convolution weights and bias
        self.weight = torch.nn.Parameter(torch.empty(out_channels, in_channels,
    kernel_size, kernel_size, kernel_size))
        self.bias = torch.nn.Parameter(torch.empty(out_channels))
        # Weight initialization following PyTorch's Conv3d default
        torch.nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in)
        torch.nn.init.uniform_(self.bias, -bound, bound)

        # Define the fused convolution and activation CUDA kernel with optimizations
        kernel_size_val = kernel_size
        conv_activation_source = f"""
        #include <torch/extension.h>
        #include <cuda_runtime.h>
        #include <math.h>

        #define KERNEL_SIZE {kernel_size_val}
        #define KERNEL_D {kernel_size_val}
        #define KERNEL_H {kernel_size_val}
        #define KERNEL_W {kernel_size_val}

        __global__ void conv_activation_kernel(
            const float* __restrict__ input, const float* __restrict__ weights, const
    float* __restrict__ bias,
            float* __restrict__ output,
            int N, int C_in, int D, int H, int W, int C_out,
            float clamp_min, float clamp_max) {{
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                const int W_out = W - KERNEL_W + 1;
                const int H_out = H - KERNEL_H + 1;
                const int D_out = D - KERNEL_D + 1;
                const int total_elements = N * C_out * D_out * H_out * W_out;
                if (idx >= total_elements) return;

                int w_out = idx % W_out;
                int h_out = (idx / W_out) % H_out;
                int d_out = (idx / (W_out * H_out)) % D_out;
                int f = (idx / (W_out * H_out * D_out)) % C_out;
                int n = idx / (C_out * D_out * H_out * W_out);

                float sum = 0.0;

                // Precompute constants for GELU
                const float sqrt_2_over_pi = 0.79788456f;
                const float approx_coeff = 0.044715f;

                for (int c = 0; c < C_in; ++c) {{
                    // Compute input channel base pointer
                    const int input_c_offset = n * C_in * D * H * W + c * D * H * W;
                    const float* input_c = input + input_c_offset;

                    // Compute center position in the input channel
                    const int center_offset = d_out * H * W + h_out * W + w_out;
                    const float* center = input_c + center_offset;

                    // Weight base for current output channel f and input channel c
                    const int weight_base = f * C_in * KERNEL_D * KERNEL_H * KERNEL_W + c
    * KERNEL_D * KERNEL_H * KERNEL_W;

                    #pragma unroll
                    for (int kd = 0; kd < KERNEL_D; ++kd) {{
                        #pragma unroll
                        for (int kh = 0; kh < KERNEL_H; ++kh) {{
                            #pragma unroll
                            for (int kw = 0; kw < KERNEL_W; ++kw) {{
                                const int offset = kd * H * W + kh * W + kw;
                                const float input_val = center[offset];
                                const int w_off = weight_base + kd * KERNEL_H * KERNEL_W
    + kh * KERNEL_W + kw;
                                const float weight_val = weights[w_off];

                                sum += input_val * weight_val;
                            }}
                        }}
                    }}
                }}

                sum += bias[f];

                // Apply activations with optimized order
                {{
                    float tanh_val = tanhf(sum);
                    float clamped = fmaxf(clamp_min, fminf(tanh_val, clamp_max));
                    float inner = clamped + approx_coeff * clamped * clamped * clamped;
                    inner *= sqrt_2_over_pi;
                    sum = clamped * 0.5f * (1.0f + tanhf(inner));
                }}

                // Output index calculation
                const int output_base = n * C_out * D_out * H_out * W_out +
                                        f * D_out * H_out * W_out +
                                        d_out * H_out * W_out +
                                        h_out * W_out;
                output[output_base + w_out] = sum;
            }}

        torch::Tensor conv_activation_cuda(torch::Tensor input, torch::Tensor weights,
    torch::Tensor bias,
                                           float clamp_min, float clamp_max) {{
            int N = input.size(0);
```

```
            int C_in = input.size(1);
            int D = input.size(2);
            int H = input.size(3);
            int W = input.size(4);
            int C_out = weights.size(0);

            int D_out = D - KERNEL_D + 1;
            int H_out = H - KERNEL_H + 1;
            int W_out = W - KERNEL_W + 1;

            auto output = torch::empty({{N, C_out, D_out, H_out, W_out}},
    input.options());

            int total_elements = N * C_out * D_out * H_out * W_out;
            const int block_size = 256;
            const int grid_size = (total_elements + block_size - 1) / block_size;

            conv_activation_kernel<<<grid_size, block_size>>>(
                input.data_ptr<float>(), weights.data_ptr<float>(),
    bias.data_ptr<float>(),
                output.data_ptr<float>(), N, C_in, D, H, W, C_out,
                clamp_min, clamp_max
            );
            return output;
        }}
        """

        conv_activation_header = """
        torch::Tensor conv_activation_cuda(torch::Tensor input, torch::Tensor weights,
    torch::Tensor bias,
                                        float clamp_min, float clamp_max);
        """

        # Load the CUDA kernel with fast math optimization
        self.conv_activation = load_inline(
            name="conv_activation",
            cpp_sources=conv_activation_header,
            cuda_sources=conv_activation_source,
            functions=["conv_activation_cuda"],
            extra_cuda_cflags=['-use_fast_math'],
            verbose=True
        )

    def forward(self, x):
        return self.conv_activation.conv_activation_cuda(x, self.weight, self.bias,
    self.clamp_min, self.clamp_max)
```

In its final turn, the model attempts a more advanced implementation that further parallelizes the computation across kernels before performing a warp-level reduction. However, it fails to implement the strategy correctly, due to applying the reduction across the wrong axis. We do note the model has shown success in implementing complex warp reductions in several other tasks.