Extended Abstract

Motivation High-performance CUDA kernel generation remains a challenging problem for large language models (LLMs), especially when balancing functional correctness and execution speed. While flagship models like GPT-4 show promise, their generations often fail to compile or run efficiently. We explore whether reinforcement tuning via Group Relative Policy Optimization (GRPO) can improve the reliability and performance of open-source LLMs in this domain.

Dataset We benchmarked and trained our models using the KernelBench dataset (), which consists of 270 cuda kernel generation problems. These difficulties are largely bucketed into 4 levels, which range from level 1 (simple kernels related to neural network building blocks like convolutions, mat mul layers, etc) to level 4 (full huggingface architectures). The generated kernels are evaluated on three dimensions: whether they successfully compile, if they compile, whether they are correct, and finally if they are correct, what runtime do they give relative to the PyTorch baselines. We then train with a multi-task objective, which balances compilability, correctness, and speed.

Method First, we conducted a baseline sweep. We compiled native torch kernels, kernels generated from open source (Llama-8B/70B, Qwen-Coder-32B) and closed source models (GPT-4.1). As expected, we found that GPT-4.1 significantly outperformed other models, while Llama-8B performed the worst. To improve the performance of our base models, adopted a 2 stage approach. First, we SFT'd our model on generations from GPT-4.1. Specifically, we generated rollouts from GPT-4.1 on an uncontaminated split and SFT'd LLama-8B on those rollouts to emulate the performance of a GPT-4.1 sized model using `TRL`'s `SFTTrainer` von Werra et al. (2023).

Next,we tuned our model with group relative policy optimization (GRPO) with the following reward: $r = \alpha\,\mathbf{1}[\text{correct}] + \beta\,\mathbf{1}[\text{compiled}] + (\gamma)\min(t_0/t, r_{\max})$, where t_0 represents the baseline time of the Pytorch implementation and t represents the runtime of our generated kernels. Furthermore, for tuning our 8B model, we noticed that the original reward function was difficult to learn from as all generated kernels would be incorrect, resulting in zero reward signal from the environment. To bootstrap the training process, we therefore added an LLM-as-a-judge in the reward environment, which would examine trajectories from the base model, critique them, and provide extra continuous signals to train off of. Empirically, we found this to be crucial for jump-starting the training process.

Implementation We trained most of our models on a 48GB A6000 node for five SFT epochs (LoRA 8, lr 1e-4) and five GRPO epochs. We used a learning rate of 0.0001 for our SFT tunes and a learning rate of 0.00001 for our RFT tunes. We used gradient accumulation to reduce noise across multiple steps while keeping the training process within GPU bounds. For GRPO, we generated 6 different roll-outs every each update step, with the relative advantages being computed over those 6 trajectories. Finally, for the SFT process we used a LoRA adapter with a rank of 8, which showed us promising gains. The adapter unsuccessfully didn't allow us to train well for GRPO and we had to expand the rank to 32 for the GRPO tunes.

Results GPT-4.1 remains the strongest single model, achieving near-perfect compilability (100%) and high correctness (88%) on our eval suite and generates kernels that were just 1.4x slower than native PyTorch baselines. On the other hand, the vanilla Llama-8B model had 0% correctness, while the 70B variant had 88% correctness but 4x slower runtime than the torch baseline. After training, the Llama-8B model's performance rose to 33% correctness (from 0% before training) and genearted kernels were 2.5x slower than the PyTorch baselines. Thus, the tuning process took our 0 utility baseline model and drastically improved its ability to generation correct kernels.

Discussion Overall, our pipeline shows that post-training via SFT can start to bridge much of the gap between small open-source LLMs and state-of-the-art closed-source models on CUDA kernel generation. We also hope that our training tricks like cold-starting with an LLM-as-a-judge and SFT can generalize to other RFT environments with hard to satisfy, sparse rewards. That said, to see the full-scale of benefits, we likely want to train more intelligent base models to take full advantage of the rich reward signals in the environment. Due to compute limitations, we were only able to train an 8B parameter model, which we would want to scale to larger sizes for seeing stronger boosts. We remain optimistic about the promise of GRPO on these larger models, where learning from the environment can be much more rewarding than what we can do with just SFT'ing.

Reinforcement Tuning Open Source LLMs for Kernel Generation

Aksh Garg
Department of Computer Science
Stanford University
akshgarg@stanford.edu

Jeffrey Heo
Department of Computer Science
Stanford University
jeffheo@stanford.edu

Megan Mou
Department of Computer Science
Stanford University
meganmou@stanford.edu

Abstract

We experiment with several post-training methods to improve open-source LLMs' ability to generate correct and high-performance CUDA kernels. First, we perform supervised fine-tuning (SFT) on rollouts from GPT-4.1 to densify the reward signal. Then, we apply Group Relative Policy Optimization (GRPO) using a composite reward that captures compilability, functional correctness, and relative runtime speed. On the KernelBench benchmark, our posttuned Llama-8B model achieves 25% correctness (up from 0%) and average runtimes within 2.5× of native PyTorch, closing over one-third of the gap to GPT-4.1. We introduce an LLM-as-a-judge mechanism to bootstrap sparse rewards and demonstrate that this densification of the reward can serve as a practical recipe for reliable code generation in low-resource settings, where baseline models are unsuccessful in reaching non-zero terminal rewards.

1 Introduction

Writing high-performance CUDA kernels remains a major bottleneck for modern machine learning. Unfortunately, writing optimal CUDA kernels requires deep expertise in GPU architectures and extensive manual tuning Ouyang et al. (2025). Offloading this time-intensive process to LLMs can unlock significant performance boosts when it comes to training and doing inference on models. Unfortunately, most LLMs struggle at Kernel Generation. More speficically, KernelBench provides a rigorous evaluation suite of 250 real-world PyTorch workloads but finds that even the best LLMs exceed the PyTorch baseline on fewer than 20% of tasks without iterative refinement Ouyang et al. (2025). Some efforts have tried to optimize LLM capability for kernel generation. For example, Sakana AI's "AI CUDA Engineer" automates kernel discovery via an agentic evolutionary loop (which translates problems into initial kernels and then iteratively improves upon previously generated kernels to improve outputs), achieving 10–100× speedups on many workloads. However, they treat the LLM as a fixed black box and does not adapt its parameters Sakana AI (2025).

In contrast, in this paper, we try to determine if we can directly inject the ability to generate compilable kernels into model weights. More concretely, Group Relative Policy Optimization (GRPO) has shown that fine-tuning LLMs on verifiable, oracle-based rewards can dramatically boost performance in reasoning benchmarks without requiring a learned value critic Shao et al. (2024); Mroueh (2025). For example, DeepSeek-R1 used GRPO to double pass@1 rates on mathematical reasoning tasks by leveraging exact correctness oracles Shao et al. (2024); Mroueh (2025). Extensions such as

GRPO-LEAD introduce length-dependent accuracy rewards, explicit penalties for incorrect outputs, and difficulty-aware advantage reweighting to combat reward sparsity Zhang and Zuo (2025), but these remain confined to reasoning and have not been applied to code generation.

In this work, we test two types of approaches for improving the quality of generated kernels: (1) supervised fine-tuning (SFT) on GPT-4.1 rollouts to densify the reward signal, and (2) GRPO using a composite reward that jointly captures compilability, functional correctness, and normalized speedup relative to a PyTorch baseline. We also introduced an LLM-as-a-judge mechanism to bootstrap sparse rewards during the cold-start phase, where otherwise the lack of intelligent kernels generated by baseline LLMs functionally blocks training.

2 Related Work

2.1 KernelBench and Benchmarking LLMs for CUDA

We trained and evaluated our models on KernelBench Ouyang et al. (2025), which has a list of 270 PyTorch problems, along with methods to benchmark correctness and speed of model generated responses. The original KernelBench paper showed that without iterative feedback, both closed- and open-source LLMs only exceed the PyTorch baseline in slightly 20% of tasks, motivating us to try integrate the performance metrics directly into the training loop in hopes to improve this fraction by post-training directly on the reward function.

2.2 Agentic Evolutionary Pipelines vs. Model Adaptation

Sakana AI's "AI CUDA Engineer" wraps a fixed LLM in a translation $\rightarrow$ generation $\rightarrow$ update loop, which they claim enables them to reach up to 100× speedups on common kernels Sakana AI (2025). While their search-based approach demonstrates the power of iterative refinement with profiling feedback, it leaves the model parameters unchanged. By contrast, our method adapts the LLM itself via supervised fine-tuning and GRPO, enabling the model to internalize performance objectives rather than relying solely on external search, which can be computationally and monetarily expensive to arbitrarily scale at run time.

2.3 Reinforcement Tuning with GRPO

There's been several improvements to the model reward tuning process over the past few years. First, in contrast to PPO-based RLHF, which relies on a separate value critic, has a high memory footprint, and often is difficult to tune Schulman et al. (2017), Direct Preference Optimization (DPO) fine-tunes a model directly on a closed-form classification loss with pairwise preference data Sharma et al. (2023), eliminating the need for sampling or an explicit reward model. Moreover, in addition to being simpler to implement, DPO matches or exceeds PPO on human-preference tasks and is more sample-efficient Sharma et al. (2023); Deng et al. (2025). However, the nature of DPO only enables it to learn a single update per data point. This makes it expensive to scale from a data point of view, as we need to invest heavily on the human data side to get enough data for training the model.

Group Relative Policy Optimization (GRPO) overcomes these issues by defining one scoring metric/rubric, which we can then score multiple different trajectories against. This makes it drastically more data-efficient to learn from. Specifically, in GRPO, we compute group-relative baselines over multiple samples. Empirically, Kapoor et. al found that this yields lower-variance advantage estimates and up to 50 % less memory usage compared to PPO's critic Kapoor (2025). Mroueh (2025) further shows that GRPO provides more stable policy updates and greater sample efficiency than PPO across diverse reasoning benchmarks Mroueh (2025), and Li et al. (2025) demonstrate that GRPO can integrate multiple reward signals (e.g., safety, helpfulness) at lower cost than either PPO or DPO Li et al. (2025). Extensions like GRPO-LEAD introduce length-aware accuracy rewards and difficulty-aware reweighting to combat reward sparsity Zhang and Zuo (2025), but none of these works have yet applied GRPO to the domain of high-performance code generation, motivating our study of GRPO for CUDA kernel optimization.

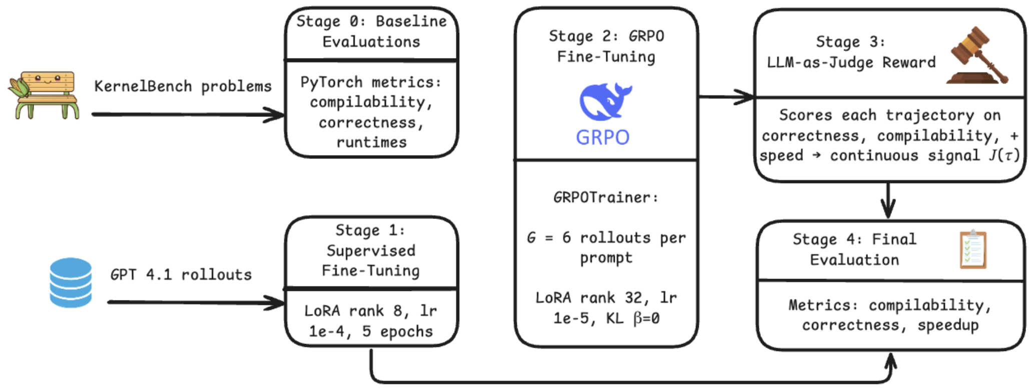

KernelBench problems
Stage 0: Baseline Evaluations
PyTorch metrics: compilability, correctness, runtimes
GPT 4.1 rollouts
Stage 1: Supervised Fine-Tuning
LoRA rank 8, lr 1e-4, 5 epochs
Stage 2: GRPO Fine-Tuning
GRPO
GRPOTrainer:
G = 6 rollouts per prompt
LoRA rank 32, lr 1e-5, KL β=0
Stage 3: LLM-as-Judge Reward
Scores each trajectory on correctness, compilability, + speed → continuous signal J(τ)
Stage 4: Final Evaluation
Metrics: compilability, correctness, speedup

Figure 1: Method Overview.

2.4 GRPO Extensions and Our Contributions

There are some recent improvements like GRPO-LEAD's length-dependent accuracy rewards, explicit penalties for incorrect outputs, and difficulty-aware advantage reweighting, which attempt to address reward sparsity and conciseness in reasoning tasks Zhang and Zuo (2025). We extend these ideas by introducing an LLM-as-a-judge to bootstrap sparse correctness signals in the cold-start stage. We find that this is empirically extremely useful in the initial stages of the training process, where the model is struggling to generate any kernels that are correct and get a reward.

3 Method

3.1 Benchmarking and Baseline Configuration

We evaluated five model classes on the KernelBench validation set Ouyang et al. (2025): (1) native PyTorch kernels; (2) Llama-8B and Llama-70B using Hugging Face's Transformers; (3) GPT-4.1 via the OpenAI API; and (4) our fine-tuned variants. For each prompt, we record:

- **Compilability**: fraction of kernels that compile on an Ampere GPU without errors
- **Correctness**: Whether the outputs from our generated kernels align with the outputs from the torch baselines that KernelBench provides.
- **Speedup**: ratio t_0/t, where t_0 is the PyTorch baseline runtime and t is the generated kernel's runtime.

We then combined these individual metrics into a composite reward as shown in equation (Eq. 1). (Note: that we originally faced issues with correctness due to differences in tiling, number of SMs, and register sizes. Adjusting compile style to the specific GPUs we were able to rent from AWS helped address those initial hiccups.) Figure 1 presents an integrated overview of our methods and how they relate to each other.

3.2 Composite Reward Function

We slightly tweak the reward from KernelBench, which just rewarded the correctness and speed of kernels. We found that especially with the base models that even compilability wasn't great - therefore, we added an extra loss for the ability of the kernels to compile. The remaining rewards were conditional on the steps before. i.e. if a kernel didn't compile, it could never be correct, and if it wasn't correct, we would never consider its speed.

$$r(\tau) = \alpha\,\mathbf{1}[\text{compiled}(\tau)] + \beta\,\mathbf{1}[\text{correct}(\tau)] + \gamma\,\min\big(t_0/t(\tau),\, r_{\max}\big)\cdot\mathbf{1}[\text{correct}], \tag{1}$$

where α, β, γ control each component's contribution and $r_{\max}$ caps extreme speedups to improve stability.

3.3 Supervised Fine-Tuning Warm Start

We first distilled GPT-4.1 into our base model via supervised fine-tuning (SFT), following the approach in Shao et al. (2024). For each training example, we constructed an input sequence x, which contained (i) a natural language description of the kernel we wanted to generate, (ii) function prototypes of the expected output from the function, and (iii) a target sequence $y = (y_1, \ldots, y_T)$, which represented the output from a larger model (GPT-4.1 in our case) when run on the same input. We then minimize the token-level cross-entropy loss:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{t=1}^{T} \log \pi_\theta\big(y_t \mid y_{<t},\, x\big),$$

where π_θ was the conditional distribution over the next token of the model we were trying to tune. During training, only the parameters of a LoRA adapter (rank $r = 8$) were updated, while the original pretrained weights remained frozen following the original design in Hu et al. (2021). As we desired, only tuning the LoRA modules allowed us to significantly reduce our memory footprint (to around 1.5% of the original model parameters) while empirically still allowing the model to improve its ability to generate valid kernels. We trained for 5 epochs with a learning rate of 1×10^{-4} and batch size 16. We only tuned the learning rate until arriving at a smooth loss trajectory but future experiments might want to try running a hyperparameter sweep over more LoRA parameters.

3.4 Group Relative Policy Optimization (GRPO) Fine-Tuning

We train directly with the HF `GRPOTrainer`, which by default implements group-relative advantage estimation without clipping (μ = 1) and with no KL penalty (β = 0). It also adds per-token normalization across the batch as described Shao et al. (2024) and in the TRL documentation von Werra et al. (2023). For each prompt x, the model generates G trajectories $\{\tau_i\}_{i=1}^{G}$, each of length $|o_i|$. If we denote the policy ratio at token t of trajectory i as

$$r_{i,t}(\theta) = \frac{\pi_\theta\big(o_{i,t} \mid x,\, o_{i,<t}\big)}{\pi_{\text{ref}}\big(o_{i,t} \mid x,\, o_{i,<t}\big)}.$$

, then we can compute the group-relative advantage at the sequence level,

$$A^{i,t} = r'(\tau_i) - \frac{1}{G}\sum_{j=1}^{G} r'(\tau_j),$$

using our reward r' (NOTE: we modify the reward slightly from the one described above. See sec. 3.5 for more details). The GRPO loss is then given by

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\frac{\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\Big[r_{i,t}(\theta)\, A^{i,t} \;-\; \beta\, D_{\text{KL}}\big[\pi_\theta \| \pi_{\text{ref}}\big]\Big]}{\sum_{i=1}^{G}|o_i|},$$

where

1. $\beta = 0$ by default (no KL penalty),
2. the denominator enforces token-level normalization, and
3. $\mu = 1$ - therefore, no clipping is applied to the policy ratio.

3.5 Aux Reward through an LLM-as-a-Judge

Early in RL the 8B model produced zero correct kernels. This resulted in a zero reward across all our rollouts and correspondingly a 0 advantage and gradient. To bootstrap learning, we added an LLM-as-a-judge J (GPT-4.1) that provides a continuous critique of each trajectory τ. To get these scores, we prompt GPT-4.1 to score each generated kernel on three dimensions—correctness, compilability, and speed—returning a JSON with three floats in $[0, 1]$. We share the exact prompt used in the implementation details in appendix C. Overall, our extra reward then becomes

$$J(\tau) = \frac{\text{correctness} + \text{compilability} + \text{speed}}{3}.$$

And the full RL reward is augmented to be

$$r'(\tau) = r(\tau) + \lambda\, J(\tau),$$

where $r(\tau)$ is the kernel reward from before (Eq. 1) and $\lambda = 0.5$ balances the judge's continuous signal against the binary oracle. By supplying continuous feedback on partial successes, this mechanism jump-starts training, enabling the policy to escape the zero-reward plateau and learn progressively better kernels.

4 Experimental Setup

4.1 Hardware and Software Environment

All experiments were conducted on a single NVIDIA A6000 GPU (48 GB). We ran PyTorch 2.1, Hugging Face Transformers 4.30, and TRL 0.4. All random seeds were fixed to 42 to ensure reproducibility.

4.2 Training Hyperparameters

We warm-start via supervised fine-tuning (SFT) on GPT-4.1 rollouts, then apply GRPO fine-tuning with G=6 rollouts per each prompt. We only tuned learning rate for our models due to compute budgets and in the case of GRPO tested LoRA ranks 8 and 32.

Table 1: Supervised Fine-Tuning (SFT) Configuration

Parameter	**Value**
Epochs	5
LoRA Rank	8
Learning Rate	1×10^{-4}
Batch Size	16
Weight Decay	0.01
Max Sequence Length	2048
Gradient Accumulation	1

Table 2: GRPO Fine-Tuning Configuration

Parameter	**Value**
Epochs	2.5
LoRA Rank	32
Learning Rate	1×10^{-5}
Batch Size	1
Samples per Prompt G	6
KL Penalty β	0 (default)
Reward Cap $r_{\max}$	3.0
Gradient Accumulation	1

4.3 Inference and Metrics

At inference time, we generate one completion per prompt ($G = 1$) with temperature 0. For each generated kernel, we evaluate whether it compiles once and then run 100 generations to benchmark correctness and runtime with random inputs.

5 Results

5.1 Dataset and Evaluation Setup

We focus our evaluation on Level 1 of the KernelBench dataset, which contains fundamental neural network operations such as matrix multiplications, convolutions, and activation functions. KernelBench consists of four difficulty levels with 100/100/50/20 problems respectively, totaling 270 CUDA kernel generation tasks. For our scope, we perform an 80:20 random train/test split on the complete Level 1 dataset (100 problems), using 80 problems for training and 20 for evaluation. This subset provides a robust foundation for assessing improvements in basic GPU kernel generation while still being computational feasible for our experimental setup. Our results are as shown below, with the SFT loss curves displayed in Figure 2

5.2 Model Selection

We evaluate several off-the-shelf open-source language models as baselines to establish the current state-of-the-art in CUDA kernel generation. Our baseline models include:

- **DeepSeek-R1-Distill-Llama-8B**: A distilled version of the DeepSeek R1 model with 8 billion parameters
- **DeepSeek-R1-Distill-Qwen-7B**: A 7 billion parameter model based on the Qwen architecture
- **Qwen2.5-Coder series**: Specialized coding models in 3B, 7B, and 14B parameter configurations
- **Qwen3-14B**: The latest generation Qwen model with 14 billion parameters

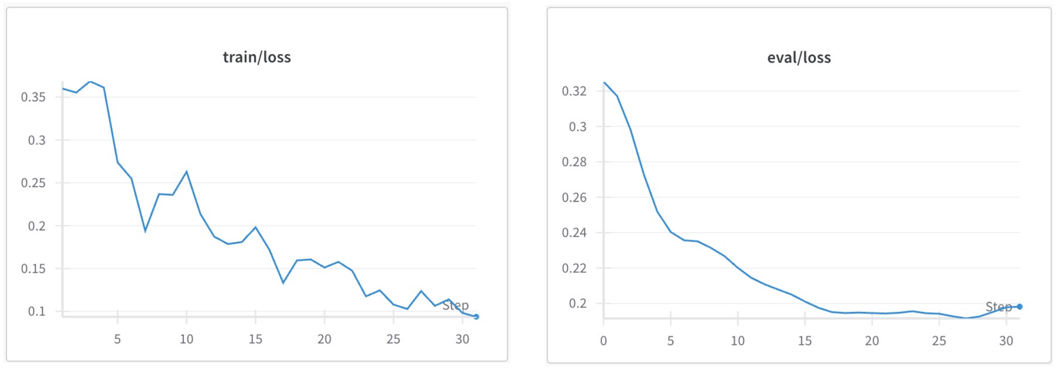
These models represent a diverse range of architectures and parameter scales, providing comprehensive coverage of current open-source capabilities in code generation tasks.

5.3 Quantitative Evaluation

Table 3 presents the performance comparison between baseline and fine-tuned models across three key metrics: compilation success rate, functional correctness, and average runtime. Most interestingly, we see that SFT'ing our model lifts baseline performance of the Llama-8B model series from 0% correctness all the way to 25% correctness. Interestingly, for a stronger model like Qwen3-Coder-32B we didn't see as strong of a lift with SFT - in fact, we observed a minor slowdown in runtime performance down to 58.5ms, which is consistent with other published findings which show that SFT doesn't generalize to reasoning capabilities. We think these gains were more noticeable in the case of the Llama-8B model; however, as the baseline had really low performance and consequently had much more room to climb. For more larger scale models, we anticipate that RFT will be the stronger source of lift.

Table 3: Performance Comparison on KernelBench Level 1 (20 test problems)

Model	**Compiled (#/20)**	**Correct (#/20)**	**Avg. Runtime (milliseconds)**	**Correctness Rate (%)**
Baseline Models				
DeepSeek-R1-Distill-Llama-8B	4/20	0/20	-	0%
DeepSeek-R1-Distill-Qwen-7B	0/20	0/20	-	0%
Qwen2.5-Coder-3B	17/20	1/20	1.19	5%
Qwen2.5-Coder-7B	17/20	8/20	9.81	40%
Qwen2.5-Coder-14B	17/20	12/20	11.84	60%
Qwen3-Coder-32B	20/20	18/20	57.4	100%
GPT-4.1	20/20	19/20	8.6	100%
SFT Fine-tuned Models				
DeepSeek-R1-Distill-Llama-8B	16/20	5/20	13.2	25%
Qwen3-Coder-32B	20/20	18/20	58.5	100%
GRPO Fine-tuned Models				
DeepSeek-R1-Distill-Llama-8B	4/20	0/20	-	0%
DeepSeek-R1-Distill-Qwen-7B	0/20	0/20	-	0%
Qwen2.5-Coder-3B	17/20	1/20	1.08	5%
Qwen2.5-Coder-7B	17/20	8/20	8.61	40%
Qwen2.5-Coder-14B	20/20	12/20	10.61	60%
Qwen3-14B	1/20	1/20	0.013	5%



SFT Training Loss SFT Evaluation Loss

Figure 2: Supervised fine-tuning cross-entropy loss over 5 epochs (32 steps): (left) evaluation loss steadily declines from 0.32 to 0.19 by step 20 before plateauing, indicating stable generalization on held-out GPT-4.1 rollouts; (right) training loss starts near 0.36 with minor early fluctuations and converges to 0.10 by the final step, demonstrating effective convergence of the LoRA-rank 8 adapter at a learning rate of 1×10^{-4}.

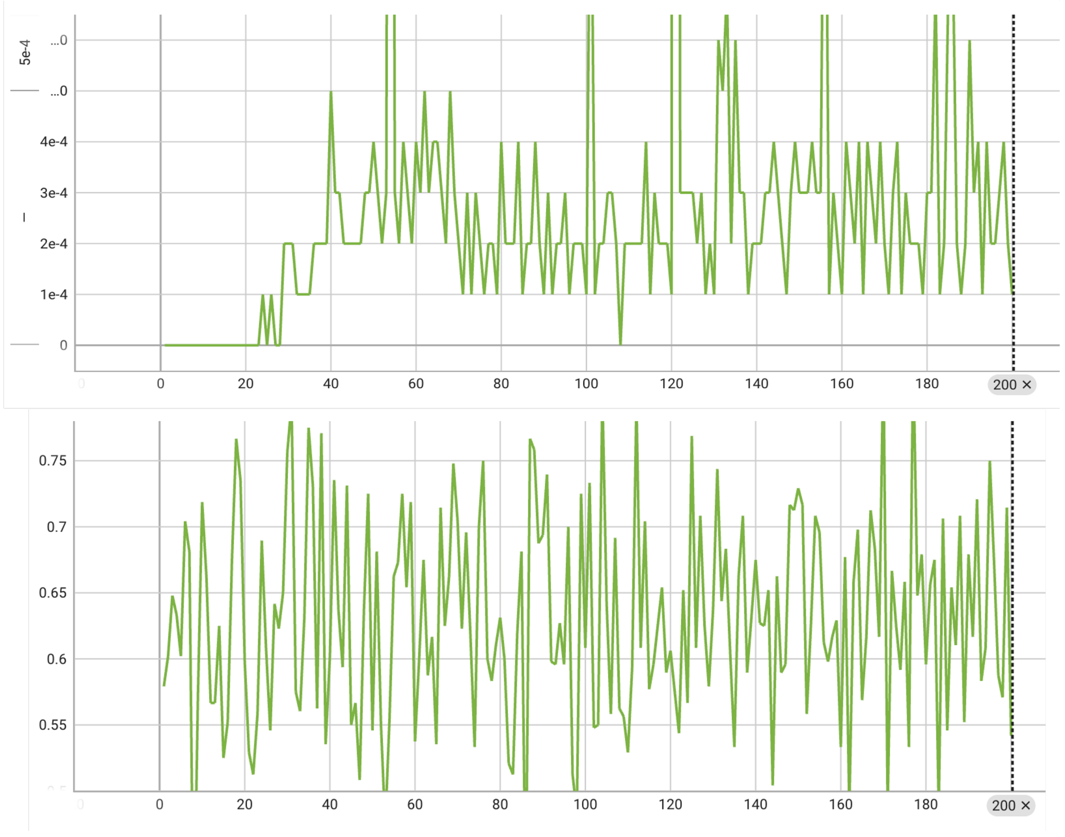


Figure 3: Training dynamics during GRPO fine-tuning over 200 steps on the 80-problem Level 1 training set, with loss (top) and reward (bottom) trajectories. The limited number of epochs (2.5 epochs with batch size 1) as well as the sparseness and low granularity of reward signals prevent full convergence, resulting in high variance and incomplete learning dynamics across both metrics.

6 Discussion

Our experiments show that supervised fine-tuning (SFT) and GRPO reinforcement tuning each improve an 8B-parameter model's CUDA kernel performance, but SFT delivers far larger gains in practice.

Importantly, we hypothesize that because SFT trains on dense, token-level cross-entropy against high-quality GPT-4 rollouts, it provides a low-variance learning signal at every step. The model quickly masters valid, compilable kernels, rising from 0 % to over 25 % correctness. In contrast, GRPO fails to improve the fundamental capabilities required for CUDA kernel generation. Across all model variants tested, compilation rates remain identical between baseline and GRPO-tuned models (e.g., Qwen2.5-Coder-7B: 17/20 compiled for both baseline and GRPO). Similarly, correctness rates show no improvement. This lack of improvement in compilation and correctness rates indicates that GRPO with LLM-as-a-Judge struggles with code generation tasks. Unlike natural language generation where partial improvements can be incrementally rewarded, CUDA kernel generation requires precise syntax, correct memory management, and proper threading patterns. We empirically show that our sparse reward signal (0.5 for compilation, 1.0 for correctness, 1.5 for optimal) fails to adequately capture improvements.

To strengthen GRPO's impact under limited compute and parameter budgets, we believe that future work could:

- **Learned Reward Models.** Train a small neural reward network on human-annotated kernels to provide denser, lower-variance feedback tailored to our composite objective.
- **Model & Adapter Scaling.** Apply GRPO to larger base models (30 B+ parameters) or use higher-rank LoRA adapters, increasing the chance of correct seeds and boosting capacity to learn from sparse rewards.

Although our study focused on CUDA kernel synthesis, the contrast between SFT's more fine-grained imitation and GRPO's coarse feedback is common across sparse-reward domains. Whenever expert demonstrations or high-quality rollouts are available, a combination of supervised imitation for rapid competence and targeted reinforcement tuning for task-specific metrics could yield improved performance.

7 Conclusion

We evaluated supervised fine-tuning (SFT) and Group Relative Policy Optimization (GRPO) on CUDA kernel generation using the KernelBench benchmark. SFT on GPT-4.1 rollouts yielded rapid gains—boosting correctness from 0% to over 25%—by providing dense, low-variance token-level supervision. In contrast, GRPO's sparse, sequence-level rewards failed to improve compilability or correctness under our compute and adapter constraints. Future work should explore learned reward models, curriculum strategies, and scaling to larger models or higher-rank adapters to better harness reinforcement tuning. More broadly, our findings underscore the value of combining fine-grained imitation with targeted RL for sparse-reward tasks across domains.

8 Team Contributions

- **Aksh Garg:** Baseline evaluations, SFT tunes + evaluation, unsuccessful GRPO tunes $\rightarrow$ dense reward formulation
- **Jeffrey Heo:** GRPO Fine-tuning, Baseline + Post-GRPO Evaluation
- **Megan Mou:** Baseline evaluations, reward formulation + variants

Changes from Proposal No significant changes

References

Xun Deng, Han Zhong, Rui Ai, Fuli Feng, Zheng Wang, and Xiangnan He. 2025. Less is More: Improving LLM Alignment via Preference Data Selection. *arXiv preprint arXiv:2502.14560* (2025). `https://arxiv.org/abs/2502.14560`

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685* (2021).

Dr. Amit Kapoor. 2025. An In-Depth Look at Group Relative Policy Optimization (GRPO). *Dramita Kapoor's AI Blog* (2025). `https://dramitakapoor.com/2025/02/24/an-in-depth-look-at-group-relative-policy-optimization-grpo/`

Xuying Li, Zhuo Li, Yuji Kosuga, and Victor Bian. 2025. Optimizing Safe and Aligned Language Generation: A Multi-Objective GRPO Approach. *arXiv preprint arXiv:2503.21819* (2025). `https://arxiv.org/abs/2503.21819`

Youssef Mroueh. 2025. Reinforcement Learning with Verifiable Rewards: GRPO's Effective Loss, Dynamics, and Success Amplification. *arXiv preprint arXiv:2503.06639* (2025). `https://arxiv.org/abs/2503.06639`

Anne L. Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? ArXiv preprint arXiv:2502.10517.

Sakana AI. 2025. The AI CUDA Engineer: Agentic Evolutionary Framework for High-Performance Kernel Discovery. `https://sakana.ai/ai-cuda-engineer/`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Yang Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300* (2024).

Archit Sharma, Daniel Lee, and Sara Rafique. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *arXiv preprint arXiv:2305.18290*. `https://arxiv.org/abs/2305.18290`

Lukas von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2023. TRL: Transformer Reinforcement Learning. GitHub repository, `https://github.com/huggingface/trl`.

Jixiao Zhang and Chunsheng Zuo. 2025. GRPO-LEAD: A Difficulty-Aware Reinforcement Learning Approach for Concise Mathematical Reasoning in Language Models. *arXiv preprint arXiv:2504.09696* (2025). `https://arxiv.org/abs/2504.09696`

A Additional Analysis

Figure 4 shows the baseline run times of the torch kernels on different GPU types and different problem difficulties. This gives us a rough sense of the ranges that we expected our generated kernels to fall into.

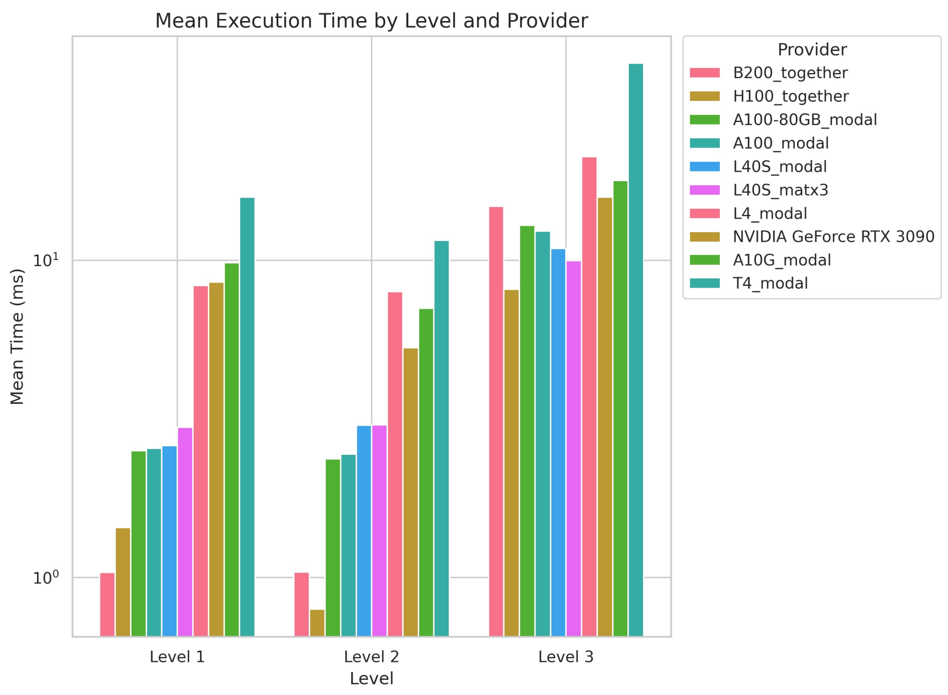


Figure 4: Mean execution time of the PyTorch baseline (in milliseconds, log scale) on KernelBench Levels 1–3 across 10 GPU configurations. On Levels 1–2, budget cards `B200_together` (1 ms) and `H100_together` (1.2 ms) are fastest, while `L40S_modal` takes around 3 ms. On Level 3, legacy `T4_modal` exhibits the highest latency (18 ms), followed by `B200_together` (12 ms). Mid-range GPUs (e.g., RTX 3090, A100 variants) lie between these extremes, illustrating the baseline performance range used for normalizing relative speedups.

B LLM As a Judge Prompts for GRPO

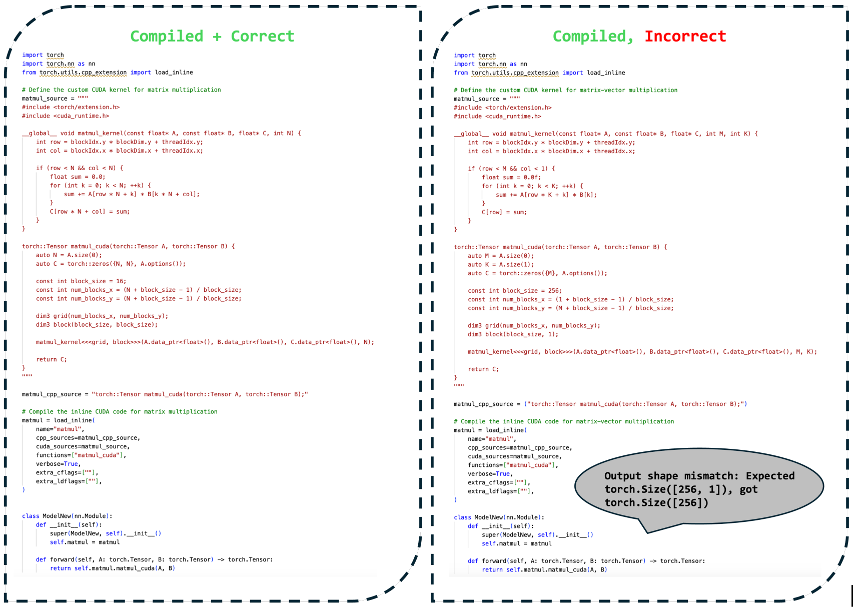
Listing 1: LLM Reward for KernelBench

```python
def kernelbench_llm_reward_func(prompts, completions, original_src, **kwargs) -> list[float]:

    # Get API key from environment
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        raise ValueError("OPENAI_API_KEY environment variable not set")

    client = OpenAI(api_key=api_key)

    rewards: list[float] = []

    for i, (comp, src) in enumerate(zip(completions, original_src)):
        response_text = comp[0]["content"]
        prompt = prompts[i]
        response_text = llm_code_extractor(prompt, response_text, ["python", "cpp"])
        custom_cuda_code = extract_first_code(response_text, ["python", "cpp"])

        # Fail fast if code extraction failed
        if custom_cuda_code is None:
            print(f"Code extraction failed for prompt {prompt}")
            rewards.append(0.0)
```

```
                continue

        # Construct evaluation prompt
        evaluation_prompt = f"""This function evaluates the quality of
             a CUDA kernel implementation based on multiple criteria:

        1. Correctness (0.0-1.0):
        - Does the kernel implement the required functionality from
            the prompt?
        - Is the mathematical logic correct?
        - Are the input/output tensor operations handled properly?

        2. Compilability (0.0-1.0):
        - Will the code compile without errors?
        - Are all CUDA functions and types properly defined?
        - Are memory allocations and deallocations handled correctly?

        3. Speed/Optimization (0.0-1.0):
        - How well is the kernel optimized for performance?
        - Are memory access patterns efficient?
        - Is thread utilization optimal?
        - Are there any obvious performance bottlenecks?

        Original PyTorch implementation:
        {src}

        Generated CUDA kernel:
        {custom_cuda_code}

        Return your answer as a JSON with the following fields:
        {{
            "correctness": float,
            "compilability": float,
            "speed": float,
            "reasoning": str
        }}
        """

        try:
            # Get LLM evaluation
            response = client.chat.completions.create(
                model="gpt-4.1-nano",
                messages=[{"role": "user", "content":
                    evaluation_prompt}],
                response_format={"type": "json_object"}
            )

            # Parse response
            eval_result = ResponseFormat.parse_raw(response.choices
                [0].message.content)

            # Calculate overall score (normalized to 0-1 range)
            overall_score = (eval_result.correctness + eval_result.
                compilability + eval_result.speed) / 3.0

            rewards.append(overall_score)

        except Exception as e:
            print(f"LLM evaluation failed: {e}")
            rewards.append(0.0)

    return rewards
```

Figure 5 shows generated kernels from our fine tuned models. On the left, we show a kernel generated that successfully compiles and is correct, while on the right, we show a kernel that does compile but produces the wrong output.

C Generated Kernels



Compiled + Correct

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

# Define the custom CUDA kernel for matrix multiplication
matmul_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void matmul_kernel(const float* A, const float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float sum = 0.0;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}

torch::Tensor matmul_cuda(torch::Tensor A, torch::Tensor B) {
    auto N = A.size(0);
    auto C = torch::zeros({N, N}, A.options());

    const int block_size = 16;
    const int num_blocks_x = (N + block_size - 1) / block_size;
    const int num_blocks_y = (N + block_size - 1) / block_size;

    dim3 grid(num_blocks_x, num_blocks_y);
    dim3 block(block_size, block_size);

    matmul_kernel<<<grid, block>>>(A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), N);

    return C;
}
"""

matmul_cpp_source = "torch::Tensor matmul_cuda(torch::Tensor A, torch::Tensor B);"

# Compile the inline CUDA code for matrix multiplication
matmul = load_inline(
    name="matmul",
    cpp_sources=matmul_cpp_source,
    cuda_sources=matmul_source,
    functions=["matmul_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)


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

    def forward(self, A: torch.Tensor, B: torch.Tensor) -> torch.Tensor:
        return self.matmul.matmul_cuda(A, B)
```

Compiled, Incorrect

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

# Define the custom CUDA kernel for matrix-vector multiplication
matmul_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void matmul_kernel(const float* A, const float* B, float* C, int M, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < 1) {
        float sum = 0.0f;
        for (int k = 0; k < K; ++k) {
            sum += A[row * K + k] * B[k];
        }
        C[row] = sum;
    }
}

torch::Tensor matmul_cuda(torch::Tensor A, torch::Tensor B) {
    auto M = A.size(0);
    auto K = A.size(1);
    auto C = torch::zeros({M}, A.options());

    const int block_size = 256;
    const int num_blocks_x = (1 + block_size - 1) / block_size;
    const int num_blocks_y = (M + block_size - 1) / block_size;

    dim3 grid(num_blocks_x, num_blocks_y);
    dim3 block(block_size, 1);

    matmul_kernel<<<grid, block>>>(A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, K);

    return C;
}
"""

matmul_cpp_source = ("torch::Tensor matmul_cuda(torch::Tensor A, torch::Tensor B);")

# Compile the inline CUDA code for matrix-vector multiplication
matmul = load_inline(
    name="matmul",
    cpp_sources=matmul_cpp_source,
    cuda_sources=matmul_source,
    functions=["matmul_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)

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

    def forward(self, A: torch.Tensor, B: torch.Tensor) -> torch.Tensor:
        return self.matmul.matmul_cuda(A, B)
```

```
Output shape mismatch: Expected
torch.Size([256, 1]), got
torch.Size([256])
```

[H]

Figure 5: Visualization of CUDA kernel generated by Qwen2.5-Coder-7B-Instruct-bnb-4bit. **(left)** Generated CUDA Kernel for matrix multiplication between square matrices successfully compiles and passes the correctness test. **(right)** Generated CUDA Kernel for matrix-vector multiplication compiles but fails the correctness test. These contrasting results highlight the brittleness of current LLMs in CUDA kernel generation, where minor algorithmic variations can cause complete correctness failures despite successful compilation. The fundamental operations underlying both tasks—matrix indexing, accumulation loops, and memory coalescing—are nearly identical, yet the model struggles with the nuanced differences in tensor dimensionality and output formatting required for matrix-vector operations.