MULTI-TURN RL WITH STRUCTURAL AND PERFORMANCE AWARE REWARDS FOR CUDA KERNEL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement Learning with Verifiable Rewards (RLVR) has emerged as a powerful technique to enhance the reasoning capacity of LLMs for optimized code generation. However, existing RLVR approaches primarily rely on outcome-based signals such as correctness and speedup, overlooking performance-critical structural properties of programs that are essential for generating optimized code. In this work, we propose CUDAPERF, a reflective RL framework that incorporates both verifiable execution rewards and structural code-aware rewards derived from parallelization features (e.g., memory coalescing, occupancy, Arithmatic Intensity, and synchronization patterns). CUDAPERF operates in two stages: (1) an offline pairwise ranking module that learns to distinguish strong and weak program candidates via contrastive comparisons, and (2) an online RL training phase that jointly optimizes for correctness, performance, and structural efficiency through a unified reward signal. To further enhance learning, CUDAPERF utilizes iterative refinement using execution feedback enabling progressive improvement of generated candidates. We also introduce a dataset comprising 2.9k C $\rightarrow$ CUDA and 1k PyTorch $\rightarrow$ CUDA programs, each paired with diverse input configurations and multiple CUDA implementations encompassing diverse optimization strategies. CUDAPERF is evaluated across multiple benchmarks comprising both C $\rightarrow$ CUDA and PyTorch $\rightarrow$ CUDA transformations. Empirical findings suggest that CUDAPERF significantly outperforms strong baselines, including Qwen-3-32B (for C $\rightarrow$ CUDA) and CUDA Agent (for PyTorch $\rightarrow$ CUDA) by achieving up to 5X & 3.32X improvements in speedup, and 17% & 7% improvements in correctness, respectively.

1 INTRODUCTION

Automatic generation of high-performance parallel programs remains a fundamental challenge in high-performance computing (HPC). While modern large language models (LLMs) have demonstrated strong capabilities in code synthesis, they often struggle to produce correct and performant GPU programs, particularly for low-level parallel programming models such as CUDA. The difficulty arises from the need to jointly reason about functional correctness and complex parallel execution patterns, which are typically guided by expert knowledge and iterative profiling.

Recent advances in reinforcement learning with verifiable rewards (RLVR) have shown promise in improving reasoning tasks, including code generation and optimization. Recently, a large volume of works use RLVR and Agentic AI workflows to generate efficient GPU kernels (Baronio et al., 2025; Li et al., 2025; Du et al., 2025; Liao et al., 2025; Lange et al., 2025). However, existing approaches largely rely on binary or sparse signals such as compilation success or correctness, which are insufficient for guiding models toward performance-optimal parallel implementations. In contrast, expert GPU programmers rely not only on execution feedback but also on structural properties of code, such as memory access patterns, thread mapping strategies, and synchronization behavior, which strongly influence performance (NVIDIA, 2023).

In this work, we propose a novel reinforcement learning framework CUDAPERF for generating high-performance CUDA programs from sequential code. Our approach integrates execution harness–based

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

verifiable rewards with structure-aware signals derived from fine-grained program features, enabling the model to learn both what works (via execution feedback) and why it works (via structural patterns). To further improve learning efficiency, we introduce an offline pairwise ranking mechanism that learns to distinguish strong and weak candidates based on performance and leverages this signal to guide policy optimization.

We evaluate CUDAPERF on program translation tasks spanning C $\rightarrow$ CUDA and PyTorch $\rightarrow$ CUDA, demonstrating consistent improvements in both correctness and execution efficiency over strong baselines. Our results highlight the importance of combining verifiable execution feedback with structure-aware inductive biases for scalable and effective program optimization, bridging the gap between neural code generation and expert-level performance engineering. In summary, the paper makes the following contributions:

- We propose **CUDAPERF, a reinforcement learning framework** for generating high-performance CUDA programs that integrates execution-based verifiable rewards with structure-aware code features. CUDAPERF naturally extends to a multi-task setting, supporting both C $\rightarrow$ CUDA and PyTorch $\rightarrow$ CUDA translation. Empirical results demonstrate that CUDAPERF achieves up to **5**$\times$ and **3.32**$\times$ speedups over strong baselines such as Qwen-3-32B and CUDA-Agent on these tasks, respectively.
- We propose a unified learning strategy that combines an **offline pairwise ranking mechanism** with **online RL optimization** to construct a composite reward signal. The offline stage learns to distinguish strong and weak program candidates via contrastive comparisons, while the online phase jointly optimizes correctness, performance, and structural efficiency by integrating execution feedback with structure-aware rewards. Empirical results show that combining structural and verifiable rewards yields more optimized CUDA programs.
- We curate a dataset comprising **2.9k C $\rightarrow$ CUDA** and **1k PyTorch $\rightarrow$ CUDA** programs, each paired with diverse input configurations and multiple CUDA implementations encompassing diverse optimization strategies to enable robust evaluation of both correctness and performance.

2 METHODOLOGY

CUDAPERF works in two stages. Firstly, an offline pairwise ranking module is trained that learns to distinguish between strong and weak CUDA candidates via a scoring mechanism that favors optimized CUDA kernels. Then this ranking module is combined with execution based signals in online RL training phase to guide policy optimization. Figure 1 shows the workflow of CUDAPERF.

2.1 OFFLINE TRAINING

Structural Reward

One of the key contribution of this work is the integration of performance-relevant structural features in the reward. Drawing on the Nvidia CUDA guide (NVIDIA, 2023) and prior literature (Williams et al., 2009; Volkov, 2010; Hong & Kim, 2009; Grewe, 2014; Fung et al., 2007; Cui et al., 2012; Pai et al., 2013) on performance modeling and optimization on GPU a set of core parallel code features that significantly influence program performance are identified that includes memory coalescing, arithmetic intensity & operation count, occupancy (threads per block), control divergence, synchronization, atomic operations and data transfer overhead. While prior works in systems and architecture studied these factors, existing LLM-based code generators and RL-driven approaches largely ignore such structural signals and rely primarily on execution-based feedback (Liao et al., 2025; Du et al., 2025; Lange et al., 2025; Baronio et al., 2025; Dai et al., 2026; Li et al., 2025).

To incorporate the mentioned CUDA structural properties into the reward function in RL, an offline lightweight ranker is trained to map extracted code features to a scalar performance score. More details of the feature extraction techniques is described in Appendix A.2. The feature vector looks like following,

$$\phi(y) = [\texttt{coal}, \texttt{ai}, \texttt{occ}, \texttt{div}, \texttt{xfer}, \texttt{atomics}, \texttt{syncthreads}, \texttt{kernels}, \texttt{tpb}, \texttt{gmem_access}, \texttt{ops}]$$

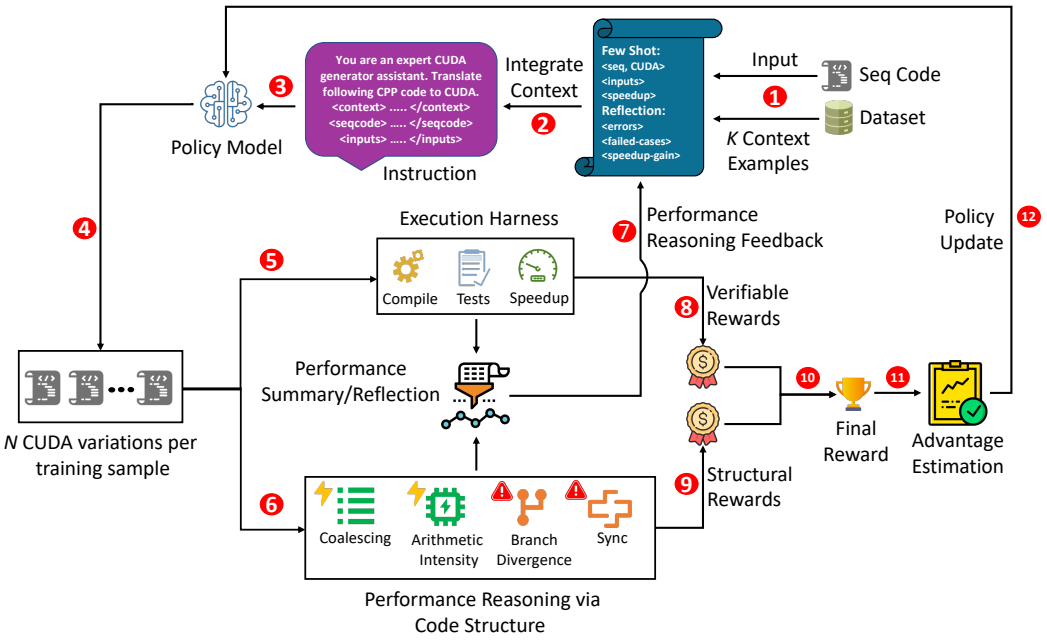


Figure 1: **CUDAPERF workflow.** The framework generates multiple CUDA candidates from sequential programs, evaluates them using execution-based verifiable rewards and structure-aware performance features, and iteratively refines the policy using reflection-driven feedback and reinforcement learning updates.

Then feature normalization is applied $\hat{\phi}_j(y) = \frac{\phi_j(y)-\mu_j}{\sigma_j}$; where μ_j and σ_j denote the mean and standard deviation of feature j over the training split. Each training task contain multiple CUDA candidates. We sort the CUDA candidate programs for each task by measured speedup. The speedup is measured as the ratio of sequential and CUDA runtime. The bottom quartile is treated as the weak candidate set, while the top quartile is treated as the strong candidate set. We then sample pairwise preferences (y^+, y^-) where y^+ is sampled from the top-performing quartile and y^- is sampled from the bottom-performing quartile, subject to $\text{speedup}(y^+) > \text{speedup}(y^-)$.

We train an MLP ranker $s_\psi(\hat{\phi}(y))$ that maps the normalized feature vector of a candidate program to a scalar structural score. The model consists of fully connected layers with ReLU activations and dropout, followed by a final linear layer producing a single score. The ranker is trained to assign higher scores to faster candidates within each pair.

Given a pair (y^+, y^-), the ranker is optimized using the logistic pairwise ranking loss:

$$\mathcal{L}_{\text{rank}}(\psi) = \mathbb{E}_{(y^+,y^-)}\left[\log\left(1+\exp\left(-(s_\psi(\hat{\phi}(y^+)) - s_\psi(\hat{\phi}(y^-)))\right)\right)\right].$$

This objective encourages the score difference $s_\psi(\hat{\phi}(y^+)) - s_\psi(\hat{\phi}(y^-))$ to be positive whenever y^+ has higher measured speedup than y^-. In practice, this corresponds to applying the softplus loss to the negative score difference. The model is trained with AdamW, mini-batch optimization, weight decay, dropout regularization, and gradient clipping. After training, the ranker provides a learned structural reward:

$$R_{\text{str}}(y) = s_\psi(\hat{\phi}(y)) \tag{1}$$

This score in Eq. 1 is considered as **Structural Reward** during online reinforcement learning.

2.2 ONLINE RL TRAINING

2.2.1 PROBLEM SETUP

We formalize the task as learning a conditional policy $\pi_\theta(y \mid x)$, where x represents a sequential input program and y represents a candidate CUDA implementation where the policy is parametrized by θ.

For each input x the policy generates g candidates $Y(x) = y_1, y_2, ..., y_g$. This enables group-relative evaluation. The objective is to learn an optimal policy π_θ^* (Eq. 2), that maximizes the expected reward over generated programs, where the reward integrates both verifiable execution feedback $R_{\text{ver}}(y, x)$ and structure-aware performance signals $R_{\text{str}}(y, x)$.

$$\pi_\theta^* = \arg\max_{\pi_\theta} \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \left[R_{\text{ver}}(y, x) + R_{\text{str}}(y, x) \right] \tag{2}$$

Here R_{ver} captures correctness and runtime efficiency while R_{str} provides a learned prior over performance-relevant CUDA code structure. These code structural features introduce structured inductive bias that complements execution-based feedback with code structure-based signals. This unified formulation of composite reward enables the model to reason not only from observed performance outcomes (from execution harness) but also from interpretable code-level properties, thereby facilitating a transition from purely black-box optimization to more principled, white-box reasoning over program performance.

Verifiable Reward

The verifiable reward is measured by passing each generated CUDA response through an execution harness that evaluates using compilation correctness, test case pass ratio and speedup gain. The verifiable reward consists of following components:

Compilation Penalty. A fixed hard penalty is applied if the candidate CUDA fails to compile.

$$R_{\text{comp}}(y) = \begin{cases} -\alpha_{\text{comp}}, & \text{if compilation fails} \\ 0, & \text{otherwise} \end{cases}$$

Correctness Penalty. Incorrect outputs are penalized proportionally to the pass rate where pass rate is computed as $p(y) = \frac{\#\ passed\ tests}{\#\ total\ test\ cases}$.

$$R_{\text{corr}}(y) = -\alpha_{\text{wrong}} \cdot (1 - p(y))$$

Performance Reward. For correct programs meaning programs that compile and also pass all test cases, we define the speedup as $S = \frac{T_{\text{CPU}}(y)}{T_{\text{GPU}}(y)}$. Here $T_{\text{CPU}}(y)$ and $T_{\text{CPU}}(y)$ are the sequential program and CUDA program median runtime across 5 executions. Then the performance reward is computed as:

$$R_{\text{speed}}(y) = \alpha_{\text{speed}} \cdot \text{clip}\big(\log_2(S),\ s_{\min},\ s_{\max} \big)$$

The log-scaling and clipping is performed to transform the speedup reward to a bounded range which stabilizes training and prevent extreme performance outliers from becoming dominating entities in the reward signal.

Stability Penalty. To discourage unstable kernels, high runtime variance is penalized using the coefficient of variation $\text{cv}(y)$:

$$R_{\text{cv}}(y) = -\alpha_{\text{cv}} \cdot \text{cv}(y),$$

The final **Verifiable Reward** is given by:

$$R_{\text{ver}}(y, x) = \begin{cases} R_{\text{comp}}(y), & \text{if compilation fails} \\ R_{\text{corr}}(y), & \text{if } p(y) < 1 \\ R_{\text{corr}}(y) + R_{\text{speed}}(y) + R_{\text{cv}}(y), & \text{if } p(y) = 1 \end{cases}$$

We empirically set the values $\alpha_{\text{comp}} = 3.0$, $\alpha_{\text{wrong}} = 2.0$, $\alpha_{\text{speed}} = 1.0$ and $\alpha_{\text{cv}} = 0.5$. Also, to bound the speedup reward to moderate slowdowns and upto 16X speedup we set the $s_{\min} = -0.5$ and $s_{\max} = 4.0$.

3 Experimental Results

Base Models. For the RL training the policy model is based on Qwen-3-32B (Qwen3-Team, 2025). We finetune using QLoRA (4 bit) (Dettmers et al., 2023). We train on two A100 40 GB GPUs.

The CUDA executions are pinned to a single GPU to avoid interference between training and benchmarking. We focus on two problems: i) C to CUDA and ii) PyTorch to CUDA transformation. Our training recipe remains almost identical for both of the problems.

Execution Harness. The execution harness consists of three components: compilation check module, test case check module and speedup calculation module. The compilation is checked to ensure that the generated CUDA candidates do not have any syntactic errors. For each sample in our dataset we have at least five test cases and the test case check module runs each candidate CUDA program through these test cases and checks how many of these test cases are passed. Finally, the speedup calculation module runs each candidate CUDA five times, takes the average runtime, calculates the speedup compared to the sequential program.

Training offline ranker. Offline training is done only once to teach the ranker model to identify preferable CUDA kernels. We use existing dataset like BabelTower (Wen et al., 2022), CUDA Agent (Dai et al., 2026) to some extent which has sequential and their corresponding CUDA implementations along with test cases. But for training our offline ranker, we need to have a dataset where for each sequential program, there are multiple valid CUDA kernels such that the ranker can learn to identify which kernels are preferable. So, to increase diversity and meet our criterial we also used the CodeNet (Puri et al., 2021) dataset that contains programs from AIZU (Aizu online judge, 2026) and AtCoder (Atcoder, 2026) online judges, which includes thousands of programs along with test cases. Then, we employ LLM (Qwen-3-32B (Qwen3-Team, 2025)) to generate at least 10 different CUDA variations for each sequential program. We also employed LLM to generate randomized test inputs following the approach of (Baronio et al., 2025) where necessary. We collected around 50k samples however we use a rigorous filtering where we only considered those samples where we have at least 5 CUDA candidates that successfully compile, pass all the test cases, and at least one variant has $\geq$1.5X speedup compared to the sequential program. This results in around 2903 C and 1013 PyTorch programs, where for each sequential (C/PyTorch) program, there are at least 5 valid CUDA candidates.

For training the offline ranker we only need the CUDA variants for each sequential program as the goal is to increase ranking score for the CUDA kernels that are most performant. We employ a lightweight multi-layer perceptron (MLP) ranker to learn a scalar structural score for each CUDA program candidate before starting RL training. The network consists of fully connected layers with ReLU activations and dropout regularization. We use hidden dimensions of 64 and 32 with a dropout probability of 0.1. The model is optimized using AdamW with learning rate 3×10^{-4}, weight decay 10^{-4}, batch size 64, and gradient clipping threshold 1.0. Training is performed for 50 epochs as we observe loss saturates after that. More details are provided in the Appendix A.14.

Multi-turn RL Training. The RL training is done separately for C→CUDA and PyTorch→CUDA. For the RL training we start from the base Qwen-3-32B (Qwen3-Team, 2025) as the policy model. For each training sample which is a sequential program we generate 16 candidate CUDA programs. Each candidate is passed through the execution harness and we collect the compilation status, test case pass ratio and also, the observed speedup compared to the sequential program. This provides us with the verifiable reward R_{ver}. Then for each candidate CUDA program the code structural properties are also extracted and we generate a structural reward score R_{str} using the offline trained ranker. Both these rewards are combined to obtain the composite reward. Using these composite reward the policy model learns to identify the optimal CUDA candidates for each sequential program. We compute the GRPO loss according to Eq. 3 directly adapted from (Guo et al., 2025) which updates the policy by maximizing the GRPO objective.

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)}$$

$$\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min\left(\frac{\pi_\theta(o_{i,t}\mid q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}\mid q,o_{i,<t})}\hat{A}_{i,t},\ \text{clip}\left(\frac{\pi_\theta(o_{i,t}\mid q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}\mid q,o_{i,<t})},1-\epsilon,\,1+\epsilon\right)\hat{A}_{i,t}\right)\right]$$
$$-\beta D_{\text{KL}}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big) \tag{3}$$

Here $\hat{A}_{i,t} = \frac{r_i - \text{mean}(r)}{\text{std}(r)}$ and r_i represents the composite reward. Hence, $r_i = R_{\text{ver}}(i) + R_{\text{str}}(i)$; which means the combination of verifiable rewards and structural rewards for i^{th} CUDA candidate. Here G represents each group and o_i represents each candidate CUDA program. $\pi_{\theta_{\text{old}}}$ and π_θ represents the

old and the new policy model, respectively. We set the clipping parameter $\epsilon = 0.2$ which checks the policy updates and helps to prevent instability during training. Also, we set $\beta D_{\text{KL}} = 0$ following Baronio et al. (2025) so that the policy model can deviate freely from the base model. For each training sample, we normalize the reward across 16 generated candidates and then the GRPO loss is computed for the entire batch following the approach of Baronio et al. (2025).

Structured Feedback and Iterative Refinement. Also, for each candidate we take upto 5-refinement turns as the policy model may not be able to generate correct CUDA programs in the first attempt. In each refinement turn, we provide feedback from both execution harness and code structural properties. These feedbacks are incorporated in the instructions for the policy model. If a program fails to compile the compiler error messages are extracted and truncated to provide actionable debugging signals. If the program compiles but fails certain test cases then the failed cases are included in the feedback such that the policy model can readjust based on the test cases. If a program passes all test cases then we include the observed speedup in the feedback. Finally, we also include the code structural feature values in the feedback. These feedbacks are formatted as structured text and appended to the prompt. Iterative refinement is performed by conditioning the model on both previous generation and its associated feedback. To prevent context explosion for multiple refinement turns we discard context from previous turns and instead use another LLM (Qwen-3-32B (Qwen3-Team, 2025)) to summarize the changes that are applied to the previous turn and include the summary in the prompt.

Prompt Construction. The prompts are constructed by combining structured task content with perforamnce-aware few-shot examplers. For each task the prompt contains the sequential code and input test cases and these are provided using structured delimiters: [SEQ_CODE], [INPUTS]. For constructing the few-shot pool we randomly sample D_k samples from the dataset. Then these samples are sorted according to the speedup. From these sorted dataset only top N candidates are retained. Finally we randomly sample k examples as few-shot context. The few-shot context is given within a [EXAMPLE] block in the prompt. Each selected sample is formatted using explicit delimiters: [SEQ_CODE], [CUDA], [INPUTS], [SPEEDUP]. The details of the prompts used for the instructions are described in the Appendix. We empirically set the values to $k = 5$, $N = 20$ and $D_k = 50$.

We train for 60 epochs as we found rewards usually reach a plateau after that. Also, we sample 16 candidates from policy model at each turn using nucleus sampling with $p = 0.9$ and temperatures [0.2, 0.7] to enable balance between conservative and diverse generations. The maximum generation length is set to 1536 tokens which is sufficient for capturing complete CUDA programs. For tackling out-of-memory issues during training we compute masked log-probabilities over a truncated window of 1024 tokens.

Metrics. As the prime focus of this work is to optimize CUDA kernels we evaluate our approach using correctness also the performance of the generated CUDA programs. The correctness is determined through testing the generate CUDA candidates using the test cases. If a CUDA candidate passes all the test cases, only then do we consider it a correct candidate. The second metric performance is calculated by taking the ratio of sequential and CUDA runtimes. We report performance using the geometric mean of speedups across tasks in the evaluation set.

3.1 C TO CUDA TRANSLATION

Dataset. For RL Training on this task we use the curated dataset of 2903 C-to-CUDA programs. We keep around 10% of the dataset (290 samples) for evaluation. In addition to that, we also evaluate on the BabelTower (Wen et al., 2022) evaluation set of 233 pairs of C and CUDA programs.

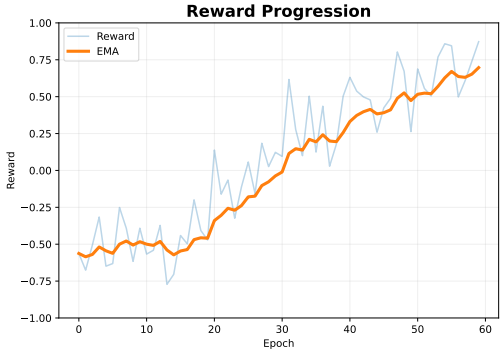


Figure 2: Reward Progression over epochs for C-to-CUDA translation. (CUDAPERF Dataset)

Baselines. As baselines we compare with state-of-the-art open-source model Qwen-3-32B (Qwen3-Team, 2025), Qwen2.5-Coder-32B (Hui et al., 2024), CodeLlama-34b-Instruct-hf (Roziere et al., 2023) and closed source model OpenAI O4-mini (OpenAI, 2026). In addition to that, we also compare with task specific models state-of-the-art QiMeng-MuPa (Ke et al., 2025) and CodeRosetta (TehraniJamsaz et al., 2024) which are specifically designed for C to CUDA translation.

Table 1: Comparison of correctness (all test cases passed) and performance (geometric mean of speedup) across multiple benchmarks. CUDAPERF consistently outperforms strong baselines in both functional correctness and execution efficiency. All results are reported for pass@5.

Dataset	Task	Model	Correctness (All test cases passed)	Performance (Geo. Mean of Speedup)
CUDAPERF Dataset (Ours)	C → CUDA	OpenAI O4-mini (OpenAI, 2026)	41%	1.3X
		CodeRosetta (TehraniJamsaz et al., 2024)	45%	1.5X
		CodeLlama-34b-Instruct-hf (Roziere et al., 2023)	53%	3.5X
		Qwen2.5-Coder-32B (Hui et al., 2024)	61%	5.3X
		QiMeng-MuPa (Ke et al., 2025)	63%	5.9X
		Qwen-3-32B (Qwen3-Team, 2025)	72%	6.2X
		CUDAPERF (Ours)	**89%**	**11.02X**
BabelTower	C → CUDA	OpenAI O4-mini (OpenAI, 2026)	52%	1.9X
		CodeLlama-34b-Instruct-hf (Roziere et al., 2023)	68%	3.1X
		CodeRosetta (TehraniJamsaz et al., 2024)	71%	3.2X
		Qwen2.5-Coder-32B (Hui et al., 2024)	79%	4.4X
		Qwen-3-32B (Qwen3-Team, 2025)	86%	4.92X
		QiMeng-MuPa (Ke et al., 2025)	**94%**	6.01X
		CUDAPERF (Ours)	**94%**	**9.02X**
KernelBench	PyTorch → CUDA	OpenAI O4-mini (OpenAI, 2026)	29%	0.62X
		CodeLlama-34b-Instruct-hf (Roziere et al., 2023)	41%	0.25X
		Qwen2.5-Coder-32B (Hui et al., 2024)	47%	0.44X
		Qwen-3-32B (Qwen3-Team, 2025)	51%	0.49X
		Cuda Agent (Dai et al., 2026)	86%	3.09X
		CUDAPERF (Ours)	**93%**	**6.41X**

Results. From Table 1 it can be observed that CUDAPERF achieves state-of-the-art results on both our and also the BabelTower dataset. On both datasets, CUDAPERF outperforms closed source OpenAI O4-mini (OpenAI, 2026) by achieving 48% & 42% better results in terms of correctness. Also, on both datasets CUDAPERF achieves 9.72X & 5.23X better speedup compared to OpenAI O4-mini (OpenAI, 2026). Also, on both datasets CUDAPERF achieves 17% & 8% better results in terms of correctness and 4.82X & 4.1X better speedup than the Qwen-3-32B (Qwen3-Team, 2025) model. Also, on our dataset CUDAPERF outperforms current state-of-the-art model QiMeng-MuPa (Ke et al., 2025) by achieving 26% better correctness and 5.12X better speedup. In BabelTower dataset CUDAPERF achieves comparable performance with Qimeng-MUPA (Ke et al., 2025) in terms of correctness. However, CUDAPERF achieves 3.01X better speedup compared to QiMeng-MuPa (Ke et al., 2025) which indicates that our RL based training along with verifiable and structural rewards helps to generate more efficient CUDA kernels. We provide more detailed results in Appendix A.18. Also, we provide the ablation results using single turn reward progression for this task in Appendix A.17.

3.2 PYTORCH TO CUDA TRANSLATION

Dataset. We use the curated dataset which contains 1013 PyTorch-CUDA programs for RL training on this task. For evaluation we use the KernelBench Ouyang et al. (2025) dataset which contains 250 PyTorch programs consisting of mostly ML workloads across three different levels.

Baselines. We choose Qwen2.5-Coder-32B (Hui et al., 2024), Qwen-3-32B (Qwen3-Team, 2025), CodeLlama-34b-Instruct-hf (Roziere et al., 2023) as open-source and OpenAI O4-mini (OpenAI, 2026) as closed source baselines. For task specific model we choose Cuda Agent (Dai et al., 2026), which is state-of-the-art models specifically designed for PyTorch to CUDA translation.

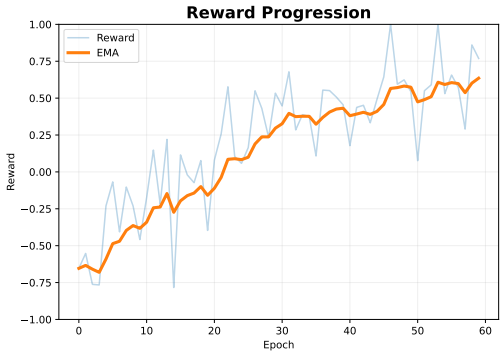


Figure 3: Reward Progression over epochs for PyTorch-to-CUDA translation. (KernelBench Dataset)

Results. Results from Table 1 show that CUDAPERF performs better than specilized models like CUDA Agent (Dai et al., 2026). CUDAPERF achieves 7% better results in terms of correctness than the state-of-the-art CUDA Agent (Dai et al., 2026). Also, it achieves 3.32X better speedup compared to the best performing model CUDAAgent. CUDAPERF outperforms OpenAI O4-mini (OpenAI, 2026) by achieving 64% better correctness and a significant 5.79X speedup gain. CUDAPERF also achieves 42% better correctness and 5.92X better speedup than the Qwen-3-32B (Qwen3-Team,

Table 2: Comparing performance of CUDAPERF for different sizes of policy model. We choose 3 different Qwen-3 model (Qwen3-Team, 2025) sizes (8B, 14B and 32B). The base policy model that we use for RL training is given in square brackets. All results are reported for pass@5.

Dataset	Task	Model	Correctness (All test cases passed)	Performance (Geo. Mean of Speedup)
CUDAPERF Dataset (Ours)	C → CUDA	CUDAPERF (Qwen-3-8B)	69%	6.9X
		CUDAPERF (Qwen-3-14B)	77%	9.1X
		CUDAPERF (Qwen-3-32B)	**89%**	**11.02X**
BabelTower	C → CUDA	CUDAPERF (Qwen-3-8B)	76%	4.3X
		CUDAPERF (Qwen-3-14B)	81%	5.9X
		CUDAPERF (Qwen-3-32B)	**94%**	**9.02X**
KernelBench	PyTorch → CUDA	CUDAPERF (Qwen-3-8B)	71%	3.4X
		CUDAPERF (Qwen-3-14B)	83%	4.5X
		CUDAPERF (Qwen-3-32B)	**93%**	**6.41X**

2025) model. We provide more detailed results in Appendix A.18. The ablation results using single turn reward progression for this task is provided in Appendix A.17.

3.3 ABLATION ON MODEL SIZE

We experimented with different model sizes as starting point for our CUDAPERF training. The results are reported in Table 2. Each time we use a different size for Qwen-3-32B (Qwen3-Team, 2025) model as our base policy model in RL. It can be observed from Table 2 that the increasing the model size also results in better correctness and performance results for CUDAPERF. This is also consistent with the findings of (Baronio et al., 2025) paper that reported using a stronger prior helps to obtain better results. Hence, we use Qwen-3-32B (Qwen3-Team, 2025) as our base policy model for RL training.

3.4 ABLATION ON VERIFIABLE AND STRUCTURAL REWARDS

We also perform experiments by only using one of the reward signals and masking off the other one. Table 3 shows the results. It can be observed that both components contribute significantly in achieving better correctness and speedup results. Without structural rewards, the correctness drops 8%, 7% & 9%, and performance drops 1.72X, 2.92X & 2.11X across the CUDAPERF, BabelTower and KernelBench dataset respectively. And, without verifiable rewards, the correctness drops 14%, 13% & 17%, and performance drops 4.12X, 4.42X & 2.71X across the CUDAPERF, BabelTower and KernelBench dataset respectively.

Table 3: Analyzing effects on Verifiable and Structural rewards in CUDAPERF. We use the Qwen-3-32B (Qwen3-Team, 2025) model as the RL base policy model. All results are reported for pass@5.

Dataset	Task	Model	Correctness (All test cases passed)	Performance (Geo. Mean of Speedup)
CUDAPERF Dataset (Ours)	C → CUDA	CUDAPERF (with structural reward only)	75%	6.9X
		CUDAPERF (with verifiable reward only)	81%	9.3X
		CUDAPERF	**89%**	**11.02X**
BabelTower	C → CUDA	CUDAPERF (with structural reward only)	81%	4.6X
		CUDAPERF (with verifiable reward only)	87%	6.1X
		CUDAPERF	**94%**	**9.02X**
KernelBench	PyTorch → CUDA	CUDAPERF (with structural reward only)	76%	3.7X
		CUDAPERF (with verifiable reward only)	84%	4.3X
		CUDAPERF	**93%**	**6.41X**

4 RELATED WORKS

BabelTower (Wen et al., 2022) uses back-translation to improve C to CUDA translation using an unsupervised setting. CodeRosseta (TehraniJamsaz et al., 2024) tackles the problem of C to CUDA translation by integrating AST Entity Recognition in the pre-training phase. QiMeng-MuPa (Ke et al., 2025) targets the same task of translating C to CUDA programs. They introduce a Mutual-Supervised Learning framework that works in an iterative manner, where a translation and a tester module

generate data for each other to improve collectively. These works are not RL-based and do not include any reward signals. Also, these works mostly focus on C to CUDA kernel generation. But these approaches do not consider the problem of CUDA kernel optimization.

OptiML (Bhattacharjee et al., 2026) proposes an inference-only framework to guide CUDA kernel optimization through the Monte Carlo Tree Search (MCTS) method; however, their approach is evaluated only on 5 tasks. Prior works have also demonstrated the effectiveness of using correctness and execution-based signals to guide learning in code generation and related domains (Chen et al., 2021; Nijkamp et al., 2022; Cobbe et al., 2021). CodeV-R1 (Zhu et al., 2025) extends RLVR to structured domains by incorporating testbench-based verification and data synthesis pipelines to ensure correctness and scalability. However, these approaches primarily rely on outcome-based rewards like correctness or functional equivalence. Reinforcement learning techniques like GRPO have been extensively applied to different domains like math (Wang et al., 2025a) and coding (Wei et al., 2025; Waghjale et al., 2024; Shypula et al., 2023). Also, the benefits of using multi-turn over single-turn RL is explored in various studies (Goldie et al., 2025; Wang et al., 2025b; Zhou et al., 2024; Gehring et al., 2024).

Some recent works have explored RLVR for CUDA kernel generation and optimization. CUDA-L1 (Li et al., 2025) introduces a contrastive RL framework that improves kernel performance using speedup-based rewards, demonstrating strong gains on KernelBench (Ouyang et al., 2025) dataset. CUDA Agent (Dai et al., 2026) proposes a large-scale agentic RL system that leverages automated profiling and verification environments to learn high-performance CUDA kernels. In parallel, multi-turn RL approaches such as (Baronio et al., 2025) explicitly model iterative refinement by learning from sequences of execution feedback, highlighting the importance of trajectory-level optimization in CUDA kernels. However their training data overlaps substantially with the original KernelBench (Ouyang et al., 2025) tasks, making a full-set comparison inappropriate which is also mentioned in CUDA Agent (Dai et al., 2026). KernelEvolve (Liao et al., 2025) is an automated, agent-driven system that generates and optimizes GPU kernels. It uses a search-based optimization process with execution feedback for generating optimized GPU programs. AKG Kernel Agent (Du et al., 2025) proposes a multi-agent collaborative framework that incorporates outcome-based signals like correctness and speedup to automate kernel generation and tuning. Lange et al. (2025) proposed agentic workflows along wth LLM-based verification and efficient filtering for translating PyTorch programs to optimized CUDA kernels. Most prior work optimizes CUDA kernels using outcome-based signals such as correctness and speedup. However, these approaches rely purely on execution feedback and do not explicitly incorporate structural properties of parallel programs into the reward. In contrast, CUDAPERF introduces a composite reward that combines verifiable execution signals with structure-aware features derived from parallelization patterns.

5 LIMITATIONS

One of the limitations of CUDAPERF is its training reliance on execution-based feedback, which introduces significant computational overhead. In particular, the need to repeatedly compile and benchmark candidate programs can be time-consuming, especially for large-scale kernels or diverse input configurations. As a result, this execution-driven reward formulation substantially increases the overall training time of the RL process. Another point worth mentioning is that CUDAPERF includes iterative refinement; it remains limited by the model and reward design, which can restrict its ability to discover optimization strategies beyond observed patterns.

6 CONCLUSION

In this work, we introduced CUDAPERF, which leverages execution-based verifiable rewards with structure-aware signals derived from parallelization features to guide policy to generate optimized CUDA kernels. Extensive experiments across multiple benchmarks demonstrate that CUDAPERF consistently outperforms strong baselines in both correctness and performance. We believe CUDAPERF represents a step toward more principled and scalable learning frameworks for automated program optimization. Unlike purely black-box approaches, our method leverages fine-grained structural properties of programs, enabling the incorporation of interpretable, white-box reasoning signals derived from high-level performance characteristics signals derived from high-level over program performance.

AI USE STATEMENT

In this work, we used generative AI tools for language polishing, including grammar, spelling, and improving clarity of presentation.

REFERENCES

Aizu online judge. Aizu online judge. `https://onlinejudge.u-aizu.ac.jp.`, 2026. Accessed: 2026-05-07.

Atcoder. Atcoder. `https://atcoder.jp/.`, 2026. Accessed: 2026-05-07.

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

Arijit Bhattacharjee, Heng Ping, Son Vu Le, Paul Bogdan, Nesreen K Ahmed, and Ali Jannesari. Optiml: An end-to-end framework for program synthesis and cuda kernel optimization. *arXiv preprint arXiv:2602.12305*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. An accurate gpu performance model for effective control flow divergence optimization. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 83–94. IEEE, 2012.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

Jinye Du, Quan Yuan, Zuyao Zhang, Yanzhi Yi, Jiahui Hu, Wangyi Chen, Yiyang Zhu, Qishui Zheng, Wenxiang Zou, Xiangyu Chang, et al. Akg kernel agent: A multi-agent framework for cross-platform kernel synthesis. *arXiv preprint arXiv:2512.23424*, 2025.

Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 407–420. IEEE, 2007.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Anna Goldie, Azalia Mirhoseini, Hao Zhou, Irene Cai, and Christopher D Manning. Synthetic data generation & multi-step rl for reasoning & tool use. *arXiv preprint arXiv:2504.04736*, 2025.

Dominik Grewe. Mapping parallel programs to heterogeneous multi-core systems. 2014.

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

Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 152–163, 2009.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Changxin Ke, Rui Zhang, Shuo Wang, Li Ding, Guangli Li, Yuanbo Wen, Shuoming Zhang, Ruiyuan Xu, Jin Qin, Jiaming Guo, et al. Qimeng-mupa: Mutual-supervised learning for sequential-to-parallel code translation. *arXiv preprint arXiv:2506.11153*, 2025.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

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

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

NVIDIA. *NVIDIA CUDA C Programming Guide*, 2023.

OpenAI. Openai api. `https://developers.openai.com/api/docs/models/o4-mini`, 2026. Accessed: 2026-05-07.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving gpgpu concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News*, 41(1):407–418, 2013.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

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

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

Ali TehraniJamsaz, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *Advances in Neural Information Processing Systems*, 37:100965–100999, 2024.

Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, pp. 16. San Jose, CA, 2010.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15362–15376, 2024.

Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Liyuan Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, et al. Reinforcement learning for reasoning in large language models with one training example. *arXiv preprint arXiv:2504.20571*, 2025a.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025b.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pp. 23685–23700. PMLR, 2022.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. Archer: Training language model agents via hierarchical multi-turn rl, 2024. *URL https://arxiv. org/abs/2402.19446*, 2024.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, et al. Qimeng-codev-r1: Reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.24183*, 2025.

A Appendix

A.1 Source Code and Data

We provide our RL based model CUDAPerf and also our curated dataset in the anynomous github repo: https://anonymous.4open.science/r/Project-4-585B/README.md

A.2 Structural Feature Extraction for Performance-Aware RL

To enable performance-aware reasoning without expensive execution, we design a structural feature extractor that maps a candidate CUDA program y into a compact feature representation capturing GPU-relevant properties. These features are integrated into the reinforcement learning reward to guide the model toward efficient parallel implementations.

A.3 Overview

Given a CUDA program y, we extract a feature vector:

$$\phi(y) = \big[\text{coal}, \text{ai}, \text{occ}, \text{div}, \text{xfer}, \text{atomics}, \text{sync}, \text{kernels}, \text{tpb}, \text{gmem}, \text{ops}\big] \in \mathbb{R}^d$$

Each component serves as a proxy for performance-critical aspects such as memory efficiency, compute intensity, and parallel scalability.

A.4 Kernel Extraction

We first identify CUDA kernels using pattern matching:

$$\mathcal{K}(y) = \{(k_i, B_i)\}_{i=1}^{K}$$

where k_i and B_i denote the kernel name and body, respectively. The number of kernels is:

$$\text{kernels}(y) = |\mathcal{K}(y)|$$

A.5 Memory Coalescing

For each global memory access $A[j]$, we analyze the index expression $j = f(\texttt{threadIdx.x}, \cdots)$ and approximate the stride:

$$s = \frac{\partial j}{\partial \texttt{threadIdx.x}}$$

We define a coalescing score:

$$\text{coal}(s) = \begin{cases} 1.0 & s = 1 \\ 0.4 & s = 2 \\ 0.1 & s = 4 \\ -0.5 & s > 4 \\ 0.2 & s = 0 \\ 0.0 & \text{otherwise} \end{cases}$$

Kernel-level and program-level scores:

$$\text{coal}_k = \frac{1}{N_k} \sum_{i=1}^{N_k} \text{coal}(s_i), \quad \text{coal}(y) = \frac{1}{K} \sum_{k=1}^{K} \text{coal}_k$$

A.6 Arithmetic Intensity

We approximate arithmetic intensity as:

$$\text{AI}_k = \frac{\text{ops}_k}{\text{gmem}_k + \epsilon}$$

To stabilize the scale:

$$\text{ai}_k = \tanh(\lambda \cdot \text{AI}_k)$$

where $\lambda = 0.25$. The final value is averaged across kernels.

A.7 Operation and Memory Counts

We compute:

$$\text{ops}_k = n_+ + n_- + n_\times + n_\div + 4 \cdot n_{\text{popc}}$$

Global memory accesses are approximated as:

$$\text{gmem}_k = \#\{\text{array accesses in } B_k\}$$

A.8 Occupancy Proxy

We approximate occupancy using threads-per-block (TPB):

$$\text{occ}(y) = \begin{cases} 1.0 & 128 \leq \text{TPB} \leq 256 \\ 0.4 & 64 \leq \text{TPB} < 128 \text{ or } 256 < \text{TPB} \leq 512 \\ -0.3 & \text{otherwise} \end{cases}$$

A.9 Control Divergence

We penalize non-uniform branching:

$$\text{div}_k = -\alpha \cdot N_{\text{non-uniform ifs}}$$

where conditionals dependent on thread or block indices are excluded.

A.10 Synchronization and Atomics

We directly count:

$$\text{atomics}(y) = \sum_k n^{(k)}_{\text{atomic}}, \quad \text{sync}(y) = \sum_k n^{(k)}_{\text{sync}}$$

A.11 Data Transfer Overhead

We penalize excessive host-device transfers:

$$\text{xfer}(y) = -\lambda_1 \max(0, N_{\text{memcpy}} - 2) - \lambda_2 N_{\text{loop-memcpy}} + \lambda_3 \mathbb{I}_{\text{efficient}}$$

Algorithm 1 Structural Feature Extraction

```
Input: CUDA program y
Extract kernels K(y)
for each kernel k ∈ K(y) do
    Count operations ops_k
    Count memory accesses gmem_k
    Compute arithmetic intensity ai_k
    Analyze memory stride → coal_k
    Compute divergence div_k
    Count atomics and synchronization
end for
Estimate occupancy from TPB
Compute transfer cost xfer(y)
Aggregate features into φ(y)
return φ(y)
```

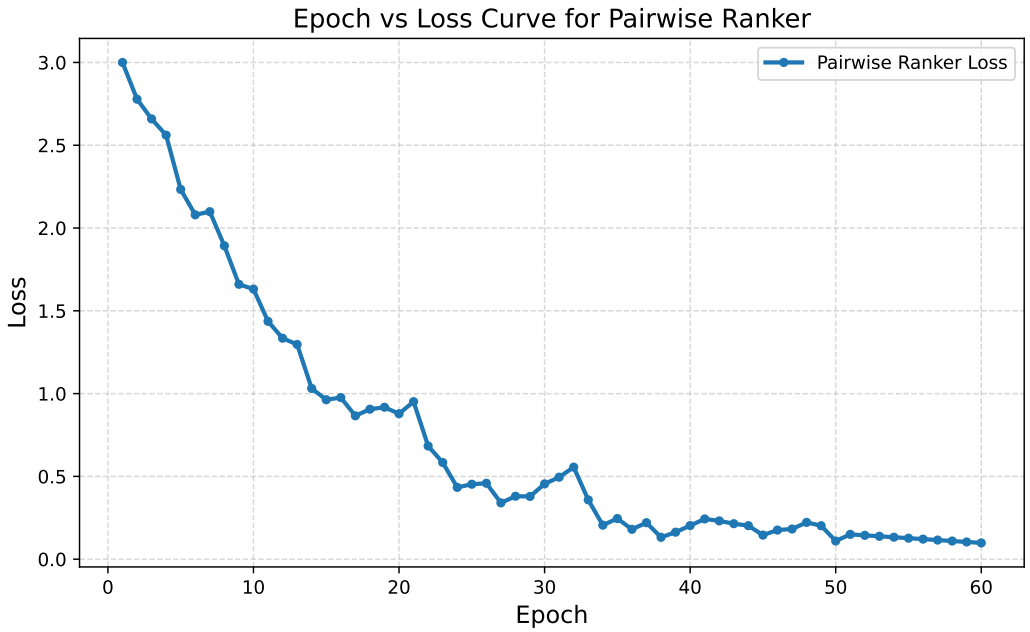


Figure 4: Epoch vs loss curve for Pairwise Ranking MLP model

A.12 Structural Reward

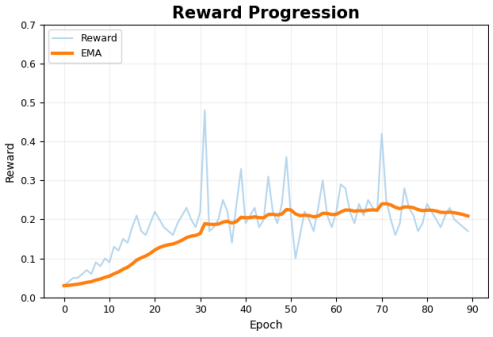
We normalize features and compute a learned structural score:

$$R_{\text{str}}(y) = w^\top \cdot \frac{\phi(y) - \mu}{\sigma}$$

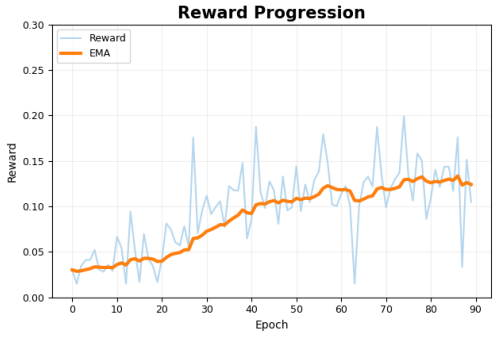
This provides a dense, execution-free signal that correlates with performance.

A.13 Feature Extraction Algorithm

Algorithm 1 shows the overall feature Extraction Algorithm.



(a) Single turn epoch vs reward for C to CUDA translation



(b) Single turn epoch vs reward for PyTorch to CUDA translation

Figure 5: Comparison single vs. multi-turn training reward progression

A.14 MLP Ranker Architecture

We employ a lightweight multi-layer perceptron (MLP) ranker to learn a scalar quality score for each CUDA program candidate. The network consists of fully connected layers with ReLU activations and dropout regularization. Given a normalized feature vector $\phi(y)$, the model predicts a ranking score:

$$s(y) = f_\theta(\phi(y)),$$

where f_θ denotes the parameterized MLP ranker.

The default architecture uses hidden dimensions of 64 and 32 with a dropout probability of 0.1. The model is optimized using AdamW with learning rate 3×10^{-4}, weight decay 10^{-4}, batch size 512, and gradient clipping threshold 1.0. It can be observed from Figure 4 that training loss saturates after 50 epochs.

A.15 Training Times

The MLP based offline ranker needs 2 hours 16 minutes to train. C-to-CUDA RL training requires 3 days 12 hours and 12 minutes. PyTorch-to-CUDA training requires 3 days 8 hours and 44 minutes. All models are trained on two A100 GPUs with 40 GB GPU memory.

A.16 Baseline Sources

For CodeLlama-34b-Instruct-hf (https://huggingface.co/meta-llama/CodeLlama-34b-Instruct-hf), Qwen2.5-Coder (https://huggingface.co/Qwen/Qwen2.5-Coder-32B), Qwen-3-32B (https://huggingface.co/Qwen/Qwen3-32B) models are accessed using the HuggingFace API. We use the $agent_workdir$ for CUDA Agent execution from their open-source GitHub repository (https://github.com/BytedTsinghua-SIA/CUDA-Agent/tree/main). For QiMeng-MuPa we use the translator-Llama-3-8B model (https://huggingface.co/kcxain/translator-Llama-3-8B) which is a finetuned version of the Meta-Llama-3-8B model. We choose this model as this version produces the best results on the BabelTower dataset.

A.17 Single vs. Multi-turn Reward Progression Ablation

Figure 5a and Figure 5b shows the rewards progression for single turn RL setup. It can be observed that for both C-CUDA and PyTorch-CUDA the reward saturates at a much lower value than multi-turn RL setup as described in the main section of the paper.

A.18 Results

.

Table 4: Comparing performance of CUDAPERF across 3 different datasets using Pass@1, 3, 5 metrics.

Dataset	Model	Correctness (All test cases passed)			Performance (Speedup)		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
CUDA-Perf-Dataset	OpenAI O4-mini (OpenAI, 2026)	31%	36%	41%	0.39X	0.95X	1.3X
	CodeRosetta (TehraniJamsaz et al., 2024)	29%	31%	45%	0.48X	0.96X	1.5X
	CodeLlama-34b-Instruct-hf (Roziere et al., 2023)	29%	37%	53%	0.58X	1.5X	3.5X
	Qwen2.5-Coder-32B (Hui et al., 2024)	23%	42%	61%	0.81X	1.6X	5.3X
	QiMeng-MuPa (Ke et al., 2025)	20%	49%	63%	0.91X	2.1X	5.9X
	Qwen-3-32B (Qwen3-Team, 2025)	31%	44%	72%	1.2X	2.3X	6.2X
	CUDAPERF	**45%**	**64%**	**89%**	**3.1X**	**5.7X**	**11.02X**
BabelTower	OpenAI O4-mini (OpenAI, 2026)	29%	31%	52%	0.88X	0.91X	1.9X
	CodeLlama-34b-Instruct-hf (Roziere et al., 2023)	28%	44%	68%	0.77X	1.2X	3.1X
	CodeRosetta (TehraniJamsaz et al., 2024)	40%	57%	71%	0.79X	1.9X	3.2X
	Qwen2.5-Coder-32B (Hui et al., 2024)	56%	67%	79%	0.97X	1.3X	4.4X
	Qwen-3-32B (Qwen3-Team, 2025)	66%	72%	86%	1.1X	1.9X	4.92X
	QiMeng-MuPa (Ke et al., 2025)	**82%**	89%	**94%**	1.2X	2.4X	6.01X
	CUDAPERF	76%	**91%**	**94%**	**2.5X**	**4.9X**	**9.02X**
KernelBench	OpenAI O4-mini (OpenAI, 2026)	23%	27%	29%	0.23X	0.39X	0.62X
	CodeLlama-34b-Instruct-hf (Roziere et al., 2023)	22%	27%	41%	0.17X	0.22X	0.25X
	Qwen2.5-Coder-32B (Hui et al., 2024)	29%	34%	47%	0.19X	0.21X	0.44X
	Qwen-3-32B (Qwen3-Team, 2025)	27%	38%	51%	0.20X	0.22X	0.49X
	Cuda Agent (Dai et al., 2026)	34%	69%	86%	1.2X	2.3X	3.09X
	CUDAPERF	**55%**	**82%**	**93%**	**1.9X**	**2.8X**	**6.41X**

Here we provide more detailed results for the datasets and the baseline models using Pass@1, 3, 5 metrics. It can be observed from Table **??** that CUDAPERF achieves better results in most cases outperforming strong baselines like Qwen-3-32B (Qwen3-Team, 2025) model, Cuda Agent (Dai et al., 2026) and QiMeng-MuPa (Ke et al., 2025) across different dataset.

A.19 PROMPTS FOR POLICY MODEL

Below is the skeleton for the prompts that are used in the RL training phase for policy models to generate CUDA programs.

```
[FEWSHOT]
[EXAMPLE]
[SEQ_CODE]
<Sequential C/PyTorch program>
[/SEQ_CODE]

[CUDA]
<Optimized CUDA implementation>
[/CUDA]

[INPUTS]
<Test inputs>
[/INPUTS]
[SPEEDUP]
<reported speedup>
[/SPEEDUP]
[/EXAMPLE]

[EXAMPLE]
[SEQ_CODE]
<Sequential C/PyTorch program>
[/SEQ_CODE]

[CUDA]
<Optimized CUDA implementation>
[/CUDA]

[INPUTS]
<Test inputs>
[/INPUTS]
[SPEEDUP]
<reported speedup>
[/SPEEDUP]
[/EXAMPLE]
...
[/FEWSHOT]

[FEEDBACK_HISTORY]
<Summarized feedback of previous turns>
[/FEEDBACK_HISTORY]

[FEEDBACK]
```

```
<Most recent feedback>
[COMPILE_ERROR]
[/COMPILE_ERROR]
[FAILED_CASE]
[/FAILED_CASE]
[/FEEDBACK]
Convert the following sequential program into an efficient
CUDA C++ program that produces the same output on
the given hardware.

When generating the program try to use
different optimization strategies including:
memory coalescing,
arithmetic intensity & operation count,
occupancy (threads per block),
control divergence, synchronization,
atomic operations and data transfer overhead.

Use the codes within <EXAMPLE></EXAMPLE> block as some references.

Use the information in [FEEDBACK][/FEEDBACK]
block to improve the CUDA program.

Only output CUDA code compilable by nvcc as a
single translation unit.

Output CUDA code within a <OUTPUT_CUDA></OUTPUT_CUDA> Block

[SEQ_CODE]
<Target sequential program>
[/SEQ_CODE]

[INPUTS]
<Test inputs>
[/INPUTS]
```