Accurate and Efficient CUDA Generation through Reinforcement Learning

Ao Li Kexin Li Weizhe Chen
Department of Electrical and Computer Engineering, University of Toronto
{dino.li, cassiekx.li, wz.chen}@mail.utoronto.ca

Abstract

The project explores reinforcement learning for generating accurate, efficient GPU kernels that address the limitations of existing one-shot code generation. By emphasizing correctness validation and speedup, the system mirrors a compiler-inspired multi-stage architecture that improves training stability and modularity. Using GRPO for policy optimization, the framework rewards compilability, functional correctness, and runtime speed, enabling iterative refinement of Triton kernels.

1 Introduction

The rapid advancement of Large Language Models (LLMs) has revolutionized modern computing and transformed many aspects of daily life. However, the high cost of LLM training and inference demands significant computational resources and financial investment, making efficient GPU execution critical for practical deployment at scale. Although GPUs provide massive parallelism, fully harnessing GPU performance requires writing highly optimized CUDA code that accounts for both software optimization and hardware architecture. The evolving NVIDIA GPU architectures have made this co-optimization increasingly complex, posing significant challenges for developers and compilers alike. To simplify this process, domain-specific languages (DSLs) such as TVM[1], Hidet[2], TileLang[3] and Triton[4] raise the level of abstraction by automatically compiling high-level operator code into efficient CUDA kernels. In parallel, LLMs have emerged as automated coding agents that can synthesize, debug, and optimize GPU kernels directly from high-level specifications. However, benchmark studies such as KernelBench [5] and TritonBench [6] indicate that correctness rates remain around 20–30% and performance often fails to match optimized library baselines, highlighting that jointly guaranteeing correctness and efficiency remains an open challenge. Motivated by these limitations, this project lies at the intersection of program synthesis, GPU performance optimization, and reinforcement learning (RL). This work investigates an RL-driven system for automatically generating functionally correct and hardware-efficient GPU kernels. Instead of attempting a one-shot generation process, the system separates correctness validation from performance optimization to enable the reliable iterative improvement at the intermediate representation (IR) level.

2 Background

Efficient GPU programming and reinforcement learning represent two core areas underlying automated kernel optimization. This section presents background on GPU programming and RL concepts, and then discusses related work in automated kernel generation and optimization.

2.1 CUDA Kernel Programming

A *CUDA kernel* is a function executed in parallel by many GPU threads. Efficient CUDA programming requires thread and block scheduling optimization, memory efficiency, and occupancy

maximization. Unlike high-level Python or PyTorch code, CUDA exposes low-level parallelism and hardware-specific constraints, making it hard for developers to implement high-performance and correct implementations.

Triton [4] is a Python-embedded DSL and compiler framework designed for high-performance GPU kernels with a simplified programming model. It enables developers and automated systems to write efficient parallel kernels at a higher abstraction level, while still achieving competitive performance compared to hand-optimized CUDA implementations. Listing 1 in Appendix B provides a simplified Triton example for vector addition. Thus, instead of directly emitting CUDA C++ source code, this work produces Triton implementations, which are later lowered to CUDA/PTX during compilation.

2.2 RL Problem Formulation

We formulate kernel generation and optimization as an *RL* problem. We can define an agent that writes kernel code, receives feedback from compilation, runs correctness tests, and runtime profiling, and updates its policy to improve over time. The environment comprises a Triton compilation toolchain, PyTorch test execution harness, and a runtime profiler. The reward is a composite function of correctness, determinism, and execution speed. Similar to prior work on program synthesis, this setup treats code generation as an iterative decision process rather than a single-shot prediction.

Formally, we consider the task of learning a policy $\pi_\theta(c \mid x)$ that maps a high-level operator specification x (e.g., tensor operation description) to a kernel implementation c in Triton. After compilation, execution, and verification, the agent receives a multi-component reward, which we aim to maximize $J(\theta) = \mathbb{E}_{c\sim\pi_\theta(\cdot|x)}[R(c;x)]$, where the reward $R(c;x)$ integrates compilability, where the kernel can be parsed and lowered without errors, correctness, which is functional verifications to PyTorch reference (multi-seed validation), and performance, which is the latency, inversely proportional to kernel runtime. The challenge lies in balancing these objectives while maintaining safety and reproducibility in evaluation. To this end, we incorporate sandboxed compilation, runtime checks (e.g., `cuda-memcheck`), and multi-run variance analysis, as advocated by recent benchmarks [5, 7].

2.3 Related Work

There exist several benchmarking and evaluation frameworks. KernelBench [5] and TritonBench [6] provide systematic evaluations, showing that models often produce syntactically valid and partially correct kernels but still struggle with runtime efficiency; they also highlight the importance of multi-seed evaluation due to sensitivity to input and seed variation. Recent work applies RL and contrastive learning to improve kernel quality. AutoTriton [8] treats Triton generation as an RL task using correctness and runtime feedback. CUDA-L1 [9] adopts contrastive RL to favor faster, more stable CUDA implementations. Kevin [10] uses multi-turn RL with iterative rollouts and performance-based selection, improving generalization to unseen operators. Astra [11] and TritonRL [12] employ multi-agent systems for generation, verification, and optimization, while Agentic-CUDA [7] offers an integrated agentic benchmarking framework with compilation, execution, and deterministic validation. Beyond GPU kernels, EquiBench [13] investigates equivalence checking as a means to evaluate LLM reasoning in code synthesis, providing valuable insights into formal correctness verification for generated programs. Such frameworks could be adapted to verify semantic equivalence between generated and reference kernels. While CUDA kernel generation is an active research area, recent studies show that achieving both functional correctness and performance remains a challenge.

3 Design

Our framework design draws inspiration from the traditional compiler stack, with the goal of achieving both functional correctness and performance optimization in a modular, decoupled manner.
Existing works on automatic CUDA generation often attempt to generate correct and optimized code in a single step. While this "one-shot" approach simplifies the pipeline, it couples two distinct objectives, correctness and optimization, hindering model interpretability and training stability.
To address this issue, our design, shown in Figure 1, explicitly decouples correctness verification from optimization. This approach mirrors the structure of a three-stage compiler:

1. Frontend: Responsible for validating code correctness—ensuring that programs are syntactically and semantically sound before further processing.

2. Mid-end: Performs hardware-agnostic optimizations (e.g., loop transformations, memory access improvements) on an intermediate representation (IR).
3. Backend: Applies target-specific optimizations and translates IR into executable target code.

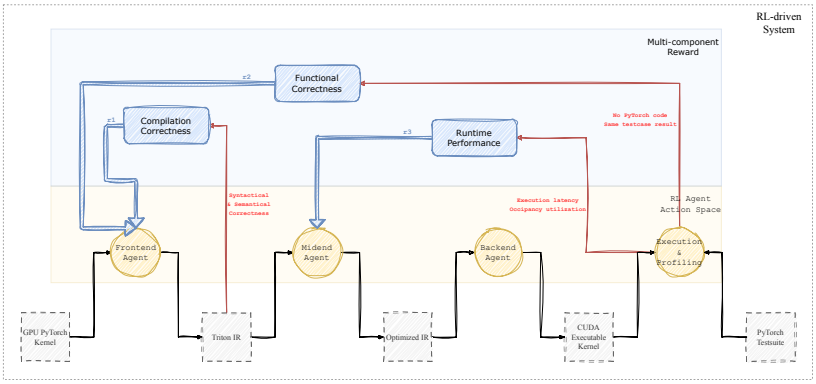

Figure 1: RL System Design Pipeline

Our framework mimics this architecture by introducing separate RL agents for the frontend and mid-end stages. Due to time and hardware constraints, we will not implement a backend optimizer in this project, though the design leaves room for future extension. The *frontend agent* focuses on ensuring functional correctness. It takes as input a GPU kernel implementation in PyTorch and converts it into the corresponding Triton IR. An important advantage is that PyTorch natively supports Triton as its backend representation, allowing us to use existing PyTorch–Triton pairs as contrastive samples for supervised pretraining and RL fine-tuning. Once correctness is verified, the *mid-end agent* performs IR-level optimizations. It learns to rewrite the Triton IR to improve execution efficiency, targeting better performance metrics such as latency reduction, memory coalescing, and occupancy utilization with the use of profiling and specific test cases. This multi-agent, decoupled design not only improves modularity and training efficiency but also provides clearer interpretability and potential for reuse across hardware backends.

4 Methodology

Our training pipeline implements a two-stage reinforcement learning approach that mirrors a compiler stack architecture. This separation ensures that the system first establishes a valid functional kernel code before attempting performance optimizations.

4.1 Model Architecture and Dataset

Both agents start from the pre-trained checkpoint `JinnP/Qwen3-8B-Kernelbook-SFT-HF` and are fine-tuned with LoRA (rank $r = 16$, alpha $\alpha = 32$, dropout 0.05) targeting attention and MLP projection layers. The dataset is `ScalingIntelligence/KernelBench` level_1, with prompts containing PyTorch model code, examples of Triton kernel implementations and randomized input tensors. Generated code is dynamically loaded, executed, and verified against PyTorch references for reward computation.

4.2 Group Relative Preference Optimization (GRPO) Algorithm

We employ GRPO as our learning algorithm which is a critic-free policy optimization algorithm that uses relative comparisons within groups instead of value function estimation used by Proximal Policy Optimization (PPO). For each prompt x, GRPO generates $K = 4$ completions $\{c_1, \ldots, c_K\}$ and computes advantages as:

$$A_i = r_i - \bar{r}, \quad \text{where } \bar{r} = \frac{1}{K} \sum_{j=1}^{K} r_j \tag{1}$$

This advantage is then used to update the policy, encouraging the model to generate outputs that perform better than the group average. GRPO is especially advantageous for language generation tasks where training a separate value network is unstable or computationally expensive. Our GRPO training configuration is shown in Appendix C.

4.3 Reward Functions

The reward computation implementation is shown in Listing 3 in Appendix D. Details are as follows.

Frontend Correctness Agent. It translates PyTorch code into valid Triton IR which the reward is based on: compilability and functional correctness. The reward function is:

$$R_{\text{frontend}}(c; x) = \begin{cases} -10.0 & \text{if compilation or correctness fails} \\ 1.0 & \text{if correctness passes} \end{cases} \tag{2}$$

Correctness is verified by comparing Triton output against PyTorch reference using `torch.allclose()` with tolerances rtol $= 10^{-5}$, atol $= 10^{-8}$. Negative rewards penalize invalid generation and prevent reward hacking. This stage ensures the agent learns strict Triton syntax and semantics.

Mid-end Performance Agent. It optimizes validated IR from Frontend Agent, applying code transformations (e.g., block tiling, memory coalescing) to minimize execution latency. By conditioning on known-correct code, we decouple logic generation from hardware optimization. The reward function adds performance metrics:

$$R_{\text{midend}}(c; x) = \begin{cases} -10.0 & \text{if compilation or correctness fails} \\ 1.0 + \frac{t_{\text{torch}}}{t_{\text{triton}}} & \text{if correctness passes} \end{cases} \tag{3}$$

where t_{torch} and t_{triton} are execution times measured over 100 runs using CUDA events (with 10 warmup runs).

5 Numerical Experiments

We evaluate our approach using KernelBench [5], a benchmark suite specifically designed to assess the ability of large language models (LLMs) to generate high-performance machine-learning kernels. KernelBench measures both functional correctness and runtime efficiency, providing a comprehensive assessment of model performance.

Figure 2 summarizes the evaluation outcomes. Approximately 40% of the generated kernels conform to the expected format, and the geometric mean of the performance metric across all valid kernels is approximately 3.25. Although this demonstrates promising baseline capability, a substantial portion of failures stems from issues unrelated to kernel logic.

One predominant source of failure is the `Invalid import` category. Further inspection revealed that many kernels fail to compile under the latest version of PyTorch due to a recent, non–backward-compatible API change introduced earlier this year, which affects a substantial portion of the benchmark samples (e.g., PyTorch no longer supports `from torch._inductor.runtime.triton_heuristics import grid` as a valid import, which is common in kernel code). We expect that resolving this compatibility issue will lead to a substantial improvement in both functional correctness and performance, enabling the model to more fully demonstrate its kernel-generation capabilities. Figure 3 illustrates an example of a generated Triton kernel that performs element-wise vector addition, demonstrating the model's ability to produce correct low-level kernel implementations when compatibility issues are not encountered.

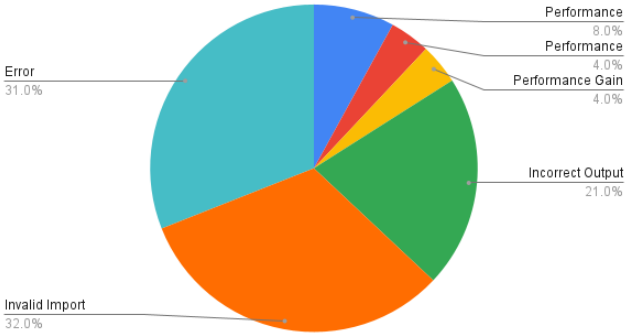

Figure 2: Output distribution

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

@triton.jit
def add_kernel(x_ptr, y_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    block_start = tl.program_id(0) * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
    y = tl.load(y_ptr + offsets, mask=mask, other=0.0)

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

def triton_add(x: torch.Tensor, y: torch.Tensor):
    assert x.is_cuda and y.is_cuda

    x = x.contiguous()
    y = y.contiguous()
    out = torch.empty_like(x)
    n_elements = x.numel()

    BLOCK_SIZE = 1024
    grid = lambda meta: (
        (n_elements + meta["BLOCK_SIZE"] - 1) // meta["BLOCK_SIZE"],
    )

    add_kernel[grid](x, y, out, n_elements, BLOCK_SIZE=BLOCK_SIZE)
    return out

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

    def forward(self, x, y):
        return triton_add(x, y)
```

Figure 3: Vector Addition Triton Code Generation Sample

6 Future Work

Hypereparameter tuning for the reward function, LoRA configuration, and systematic LLM prompt engineering could be further improved. Another promising direction is exploring more hardware-agnostic performance metrics, such as FLOPs, to improve generalization across GPU architectures and speed up model training.

7 Conclusion

This project demonstrates that a two-stage reinforcement learning approach effectively addresses the challenge of generating both correct and efficient GPU kernels by decoupling correctness validation from performance optimization. GRPO proved well-suited for code generation tasks, eliminating the need for a value network while maintaining training stability. Key insights include the importance of strong negative rewards to prevent reward hacking and the value of separating correctness and performance objectives. Future work should extend the pipeline with a backend agent for hardware-specific optimizations, adopt GFLOPs as a hardware-independent performance metric, streamline training flow through kernel caching and pre-computed timings, refine reward functions through ablation studies, and optimize prompt engineering for better guidance.

References

[1] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: an automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594.

[2] Y. Ding, C. H. Yu, B. Zheng, Y. Liu, Y. Wang, and G. Pekhimenko, "Hidet: Task-mapping programming paradigm for deep learning tensor programs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 370–384. [Online]. Available: https://doi.org/10.1145/3575693.3575702

[3] L. Wang, Y. Cheng, Y. Shi, Z. Tang, Z. Mo, W. Xie, L. Ma, Y. Xia, J. Xue, F. Yang, and Z. Yang, "Tilelang: A composable tiled programming model for ai systems," 2025. [Online]. Available: https://arxiv.org/abs/2504.17577

[4] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

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

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

[7] R. T. Lange, Q. Sun, A. Prasad, M. Faldor, Y. Tang, and D. Ha, "Towards robust agentic cuda kernel benchmarking, verification, and optimization," 2025. [Online]. Available: https://arxiv.org/abs/2509.14279

[8] S. Li, Z. Wang, Y. He, Y. Li, Q. Shi, J. Li, Y. Hu, W. Che, X. Han, Z. Liu, and M. Sun, "Autotriton: Automatic triton programming with reinforcement learning in llms," 2025. [Online]. Available: https://arxiv.org/abs/2507.05687

[9] X. Li, X. Sun, A. Wang, J. Li, and C. Shum, "Cuda-l1: Improving cuda optimization via contrastive reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2507.14111

[10] C. Baronio, P. Marsella, B. Pan, S. Guo, and S. Alberti, "Kevin: Multi-turn rl for generating cuda kernels," 2025. [Online]. Available: https://arxiv.org/abs/2507.11948

[11] A. Wei, T. Sun, Y. Seenichamy, H. Song, A. Ouyang, A. Mirhoseini, K. Wang, and A. Aiken, "Astra: A multi-agent system for gpu kernel performance optimization," 2025. [Online]. Available: https://arxiv.org/abs/2509.07506

[12] J. Woo, S. Zhu, A. Nie, Z. Jia, Y. Wang, and Y. Park, "Tritonrl: Training llms to think and code triton without cheating," 2025. [Online]. Available: https://arxiv.org/abs/2510.17891

[13] A. Wei, J. Cao, R. Li, H. Chen, Y. Zhang, Z. Wang, Y. Sun, Y. Liu, T. S. F. X. Teixeira, D. Yang, K. Wang, and A. Aiken, "Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking," *arXiv preprint arXiv:2502.12466*, 2025.

Appendix

A Assentation of Teamwork

All the members contribute in a fair manner in this project.

B Triton Example

Listing 1 provides a simplified Triton example for vector addition.

Listing 1: Triton kernel example

```
@triton.jit
def vector_add(a_ptr, b_ptr, c_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(axis=0)
    offset = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offset < n_elements
    a = tl.load(a_ptr + offset, mask=mask)
    b = tl.load(b_ptr + offset, mask=mask)
    tl.store(c_ptr + offset, a + b, mask=mask)
```

C GRPO Training Configuration

Our GRPO training configuration is shown in Listing 2.

Listing 2: GRPO training configuration

```
config = GRPOConfig(
    learning_rate=1e-6,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=1,
    max_prompt_length=512,
    max_completion_length=512,
    num_generations=4,
    beta=0.1,
)
```

D Reward Function Implementation

The reward function implementation is shown in Listing 3.

Listing 3: Reward Function Implementation

```
def compute_reward(torch_model, triton_model, *inputs):
    if not verify_compilance(triton_model.__name__, triton_model.__file__):
        return -10.0
    if not verify_correctness(torch_model, triton_model, *inputs):
        return -10.0
    correctness_reward = 1.0
    torch_time = measure_time(torch_model, *inputs)
    triton_time = measure_time(triton_model, *inputs)
    performance_reward = (torch_time / triton_time)
    return correctness_reward + performance_reward
```

Contest for Information Sharing

As part of this course, we may share selected project materials (e.g., reports, presentation slides, and presentation recordings) on the course webpage as learning resources for future students. Additionally, we may use anonymized project information for internal statistical analysis of course outcomes. Please indicate your preferences below. *Your choices will not affect your grade in any way.*

Consent for Sharing Project Materials

Please keep the item you wish and remove the other one.

- All group members consent to allow the project materials (report, slides, and presentation recording) to be shared on the course page for future students.

Consent for Use of Project Information in Statistical Analysis

Please keep the item you wish and remove the other one.

- All group members consent to allow anonymized information about the project (e.g., topic, methods, outcomes, grades) to be used by the instructor for statistical analysis and course improvement.

Optional Comments

Please kindly keep our source code private if this project is ever selected to be shared.

Group Identification

- Group number: G5
- Names of group members: Ao Li, Kexin Li, Weizhe Chen
- Signature of group members: Ao Li, Kexin Li, Weizhe Chen
- Date: 2025/12/07