Beyond TritonRL: Reinforcement Learning for Triton Kernel Optimization with Modular Extensions

Anonymous Authors[1]

Abstract

The performance of machine learning systems critically depends on efficient low-level kernel implementations. While TritonRL demonstrates that reinforcement learning can improve kernel generation, it achieves only 7% correctness on challenging kernel fusion tasks. This project implements four modular extensions to address specific weaknesses: multi-input testing to prevent overfitting, staged evaluation for computational efficiency, adaptive curriculum learning for progressive difficulty, and calibrated timing to reduce measurement noise. We provide complete implementations and unit-level verification of all components. Training runs encountered implementation bugs that prevented evaluation, limiting this work to implementation contributions and lessons learned about the challenges of end-to-end RL training for code generation.

1 Introduction

Motivation: The performance of machine learning systems critically depends on efficient low-level kernel implementations. Triton, a Python-based domain-specific language, enables developers to write GPU kernels more easily than CUDA, but generating optimal Triton code remains challenging. Recent work (TritonRL) has shown that reinforcement learning can improve kernel generation, but performance on complex tasks—particularly kernel fusion (Level 2 tasks)—remains limited, with only 7% correctness on the most challenging benchmarks.

Problem Importance: As ML models scale, the computational efficiency of kernels becomes a bottleneck. Kernel fusion, which combines multiple operations into a single kernel to reduce memory transfers, is crucial for performance but particularly difficult to generate correctly. While traditional compiler frameworks like TVM, Ansor, and Mirage provide automated optimization for standard operations, they require manual extension to support novel operators that emerge from ML research (e.g., new attention variants, custom activations, specialized normalization schemes). LLM-based kernel generation offers a compelling alternative: given a natural language description and PyTorch reference, it can generate acceptable Triton kernels for novel operations without requiring domain experts to hand-engineer compiler passes.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Machine Learning and Systems (MLSys) Conference. Do not distribute.

Contributions: This project implements four modular extensions to the TritonRL baseline:

1. **Multi-input testing** to prevent overfitting to single test cases
2. **Staged evaluation** to efficiently filter invalid kernels early
3. **Adaptive curriculum learning** to progressively increase task difficulty
4. **Calibrated timing** to reduce noise in performance measurements

While full training runs were initiated, implementation bugs in the evaluation pipeline prevented completion. We provide complete implementations, unit-level verification of all components, and an honest analysis of challenges encountered in implementing end-to-end RL training for code generation.

2 Problem

Research Questions:

1. Can multi-input testing improve generalization of RL-generated Triton kernels?
2. Does staged evaluation improve training efficiency by early rejection of invalid candidates?
3. Can adaptive curriculum learning enable better performance on complex fusion tasks?

4. How much does timing noise affect RL training, and can calibrated measurements help?

Problem Definition: Given a natural language description and reference PyTorch implementation, generate a functionally correct and performant Triton kernel. The challenge is particularly acute for Level 2 tasks involving kernel fusion (e.g., fused GEMM+bias, Conv+BatchNorm+ReLU), where multiple operations must be correctly combined while maintaining numerical accuracy.

Background: TritonRL uses supervised fine-tuning (SFT) followed by reinforcement learning with best-of-N sampling. The model generates multiple candidate kernels and selects those with highest rewards based on correctness verification and performance measurement. However, the baseline verification is fragile (single test case, no early filtering), and the training progression is fixed rather than adaptive to model capabilities.

3 Related Work

Kernel Generation: Prior work includes TVM's AutoScheduler and Ansor for automated kernel optimization, but these focus on search-based methods rather than learned approaches.

RL for Code Generation: Recent work like AlphaCode and CodeRL demonstrates RL's potential for program synthesis. TritonRL adapts these ideas to kernel generation but achieves only 7% accuracy on fusion tasks, motivating our improvements.

Curriculum Learning: Progressive task difficulty has proven effective in RL. Our adaptive curriculum applies this principle by monitoring Level 1 (basic operations) performance before increasing Level 2 (fusion) task exposure.

Improvements Over Prior Work: We improve upon TritonRL through: (1) hardened verification with multi-input testing vs. single test case, (2) staged evaluation for computational efficiency, (3) dynamic curriculum vs. fixed task distribution, and (4) calibrated timing with statistical rigor. These extensions are complementary and can be independently enabled/disabled.

4 Overview

The system architecture consists of four independently toggleable extension modules integrated into the TritonRL pipeline:

Training Pipeline: SFT Phase (Qwen2.5-7B + LoRA) $\rightarrow$ RL Phase (Best-of-N Sampling) $\rightarrow$ Extension Stack (Multi-Input, Staged Eval, Curriculum, Timing) $\rightarrow$ Reward Function [0-1]

Data Flow: KernelBook (18K samples) $\rightarrow$ 1K training subset $\rightarrow$ SFT $\rightarrow$ RL $\rightarrow$ Evaluation

Key Design Choice: Modular architecture allows ablation studies and independent validation of each extension's contribution.

5 Method

5.1 Implementation Details

Base Model: Qwen/Qwen2.5-Coder-7B-Instruct, a state-of-the-art code generation model.

Training Configuration:

- **Hardware:** $8\times$ NVIDIA A100 40GB GPUs (p4d.24xlarge)
- **Memory Optimization:** 8-bit quantization via bitsandbytes
- **LoRA Parameters:** rank=16, alpha=32, dropout=0.05
- **Batch Size:** 2 per device (16 total across 8 GPUs)

Language & Libraries: Python 3.10+, PyTorch 2.1+, Triton 2.1+, Transformers 4.36+, PEFT 0.7+, Accelerate 0.24+

5.2 Extension 1: Multi-Input Testing

Motivation: Single test case verification allows models to overfit to specific input patterns.

Implementation: Generate 5 diverse test inputs per kernel: (1) original input, (2) scaled values ($\times 2.0$), (3) different random seed, (4) larger dimensions, (5) different precision. Test generation logic is custom-implemented using PyTorch tensor operations.

5.3 Extension 2: Staged Evaluation

Motivation: Evaluating all generated kernels wastes compute on invalid candidates.

Implementation: Five-stage funnel with increasing cost: (1) AST Check (ms), (2) Compilation (seconds), (3) Tiny Run on 4×4 tensor (seconds), (4) Full Run with multi-input (seconds), (5) Timing measurement (minutes). Candidates failing any stage receive partial credit (0.0, 0.3, 0.5, 0.7 rewards).

Benefit: Unit testing showed $\sim$35-40% throughput improvement by filtering invalid kernels early.

5.4 Extension 3: Adaptive Curriculum

Motivation: Fixed task distributions force models to train on difficult fusion tasks prematurely.

Implementation: Dynamic sampling schedule starting with 10% Level 2 (fusion) tasks, linearly increasing to 50% when Level 1 accuracy exceeds 40%. This ensures solid foundations before tackling complexity.

5.5 Extension 4: Calibrated Timing

Motivation: Single timing measurements contain significant noise from GPU scheduling and thermal effects.

Implementation: Statistical measurement protocol with 10 warmup runs, 50 measurement trials using CUDA events, and trimmed mean aggregation (remove 10% outliers). Unit testing showed coefficient of variation decreased from $\sim$15% to $\sim$5%.

5.6 What We Implemented vs. Adopted

Implemented:

- All four extension modules (`extensions/` directory)
- Multi-input test generation logic
- Staged evaluation pipeline with partial rewards
- Adaptive curriculum scheduler
- Calibrated timing with CUDA events
- Integration layer for configuration

Adopted:

- Base model weights (Qwen2.5-Coder-7B-Instruct)
- KernelBook dataset (18K PyTorch→Triton pairs)
- LoRA training framework (PEFT library)
- PyTorch, Triton, Transformers libraries
- RL best-of-N sampling strategy from TritonRL

6 EVALUATION

6.1 Experimental Settings

Hardware: Initially 8× A100 40GB (p4d.24xlarge), later g5.2xlarge (1× A10G) due to budget

Dataset:

- **Training:** 1,000 samples from KernelBook
- **Evaluation:** 20 Level 2 fusion tasks from KernelBench

Metrics: Valid Rate (% passing AST), Compiled Rate (% compiling), Correct Rate (% with correct outputs - PRIMARY), Speedup (vs. PyTorch)

Hyperparameters: SFT: 1 epoch, lr=2e-4; RL: N=10 candidates, K=3; Multi-input: 5 variations; Curriculum: p=0.1→0.5

6.2 Results

Training was initiated but encountered implementation bugs. We successfully completed SFT training but evaluation failed due to code extraction issues. The following presents actual measurements and identified problems:

Actual Training Metrics:

Metric	Result	Notes
SFT Loss	0.757 → 0.199	Completed
SFT Valid Rate	0%	Verifier bug
SFT Compiled	0%	Verifier bug
SFT Correct	0%	Verifier bug
RL Tasks	10/10	Completed
RL Checkpoint	Empty	Path bug

Table 1. Actual training results from g5.2xlarge run

Known Issues Encountered:

1. **Code Extraction Bug:** Evaluation verifier failed to extract generated Triton code from model output, causing all kernels to be marked as invalid (0% valid rate). The model generates text in chat format with markdown code blocks, but the extraction regex was too strict.
2. **Checkpoint Path Mismatch:** RL phase saved to `rl_best_of_n/` instead of configured `rl_final/`, causing evaluation to fail when loading the checkpoint.
3. **Instance Termination:** All training instances were terminated before bugs could be debugged and fixed, preventing rerun.

Unit-Level Verification:

Individual extension components were tested:

- **Multi-Input:** Verified on 10 sample kernels, successfully caught edge cases
- **Staged Eval:** Tested on 50 candidates, measured 35-40% throughput gain
- **Curriculum:** Scheduling logic verified in isolation
- **Timing:** Validated on 5 reference kernels, reduced variance from 15% to 5%

What We Cannot Claim:

- No valid experimental results on KernelBench Level 2 tasks
- Cannot report improvements over TritonRL baseline
- Extensions' impact on model accuracy unvalidated
- No ablation studies conducted

6.3 Lessons Learned

End-to-End Integration Challenges:

1. **Output Format Brittleness:** LLM code generation requires robust parsing that handles various markdown formats, code fence variations, and chat templates. Simple regex extraction is insufficient.
2. **Configuration Management:** Path mismatches between training and evaluation scripts cause silent failures. Centralized configuration with validation is critical.
3. **Iterative Debugging Needs:** RL training for code generation requires multiple iterations to debug verifier/reward pipeline. Cloud instance costs make this expensive without incremental testing.
4. **Verification Before Training:** Running end-to-end ”dry runs” with dummy models before expensive GPU training would catch integration bugs early.

6.4 Limitations

Primary Limitations:

- **No successful end-to-end evaluation.** Implementation bugs prevented measuring extension effectiveness.
- **Component verification only.** Extensions tested in isolation (10-50 samples), not integrated system.
- **No baseline comparison.** Cannot compare against TritonRL due to incomplete runs.

What This Report Provides:

- Complete implementation of four modular extensions
- Architecture and design decisions
- Unit-level verification of components
- Honest analysis of implementation challenges
- Lessons for future work on RL-based code generation

7 Conclusion

This project implements a modular extension framework for improving RL-based Triton kernel generation, targeting kernel fusion tasks where baseline TritonRL achieves 7% correctness. Through four complementary extensions—multi-input testing, staged evaluation, adaptive curriculum, and calibrated timing—we address specific weaknesses in the baseline approach.

Contributions: (1) Complete implementations of four modular extensions, (2) architectural designs enabling ablation studies, (3) unit-level validation of each component, (4) honest analysis of end-to-end integration challenges in RL-based code generation systems.

Limitations: Implementation bugs in the evaluation pipeline prevented successful end-to-end training. SFT training completed (loss: 0.757→0.199) but code extraction failures caused 0% valid rate. This work represents implementation contributions and lessons learned rather than experimental validation of improvements.

Key Takeaways: (1) Robust verification, efficient evaluation, adaptive training progression, and precise measurements are critical for RL-based program synthesis, but (2) end-to-end integration of LLM code generation, verification, and RL reward computation is surprisingly fragile and requires extensive testing infrastructure. (3) Output format parsing, path management, and incremental validation are as important as algorithmic contributions. Future work must prioritize dry-run validation before expensive GPU training.

Code Repository: https://github.com/vanshajagrawal/tritonrl

Project Presentation: https://drive.google.com/drive/folders/1B3xkmlPtHR3myixEjc5Lv-ldxUp9M15E

References

[1] TritonRL: Reinforcement Learning for Triton Kernel Generation.

[2] KernelBook Dataset, ScalingIntelligence/KernelBook, HuggingFace Hub.

[3] P. Tillet, H. T. Kung, and D. Cox, “Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations,” *MAPL 2019*.

[4] Qwen Team, “Qwen2.5-Coder Technical Report,” 2024.

[5] E. J. Hu et al., “LoRA: Low-Rank Adaptation of Large Language Models,” *ICLR 2022*.

[6] R. Nakano et al., “WebGPT: Browser-assisted question-answering with human feedback,” *NeurIPS 2021*.