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

MS CSE Project Report

Vanshaj Agrawal[1]

[1]Pennsylvania State University, MS CSE
Partially done while collaborating with colleagues at Carnegie Mellon University

December 15, 2025

Abstract

Machine learning system performance fundamentally depends on efficient GPU kernels. TritonRL uses reinforcement learning to generate Triton kernels from natural language descriptions, achieving 63% correctness on basic operations (Level 1) but only 7% on kernel fusion tasks (Level 2). This work 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. Due to computational constraints, training used 200 samples (1.1% of the full dataset) with 2 RL fusion tasks over `total_time`. Training completed successfully with SFT loss decreasing from 0.757 to 0.199 and RL final loss of 0.17. Final evaluation on held-out test set achieved `level1_acc`% correctness on Level 1 tasks and `level2_acc`% on Level 2 fusion tasks. The limited training scale restricts comparison to the baseline system. This work provides methodological insights on sample efficiency versus verification rigor tradeoffs and the challenges of evaluating multi-component RL systems at limited scale.

1 Introduction

Machine learning system performance depends critically on efficient GPU kernel implementations. The gap between theoretical peak performance and achieved throughput often stems from suboptimal kernel design. Triton, a Python-based domain-specific language for GPU programming, offers more accessibility than CUDA while maintaining performance, but writing efficient Triton kernels still requires expert knowledge.

TritonRL [1] recently demonstrated that reinforcement learning can train language models to generate Triton kernels from natural language descriptions. Their approach combines supervised fine-tuning (SFT) with reinforcement learning using best-of-N sampling, achieving 63% correctness on basic operations (KernelBench Level 1) but only 7% on more complex kernel fusion tasks (Level 2).

1.1 Problem Statement

This project investigates whether modular extensions targeting specific weaknesses can improve upon the TritonRL baseline. The work focuses on four research questions:

1. Can multi-input testing improve generalization beyond single test cases?

2. Does staged evaluation with early filtering improve training efficiency?
3. Can adaptive curriculum learning help with complex fusion tasks?
4. Does calibrated timing measurement reduce noise in the reward signal?

The core task is: given a natural language description and PyTorch reference implementation, generate a functionally correct and performant Triton kernel. The challenge is particularly acute for Level 2 fusion tasks (e.g., fused GEMM+bias+ReLU, Conv2d+BatchNorm+ReLU) where multiple operations must be efficiently combined.

2 Related Work

2.1 Kernel Optimization

Traditional approaches to kernel optimization rely on search-based methods. TVM [5] and Ansor [6] use auto-tuning to explore large search spaces of possible implementations. These approaches achieve strong performance but require substantial computational resources and domain-specific optimizations.

2.2 RL for Code Generation

Recent work demonstrates that reinforcement learning can effectively guide code generation. AlphaCode [7] applies RL to competitive programming problems. CodeRL [8] uses program execution feedback as rewards. These approaches show that execution-based feedback provides stronger learning signals than supervised learning alone.

2.3 TritonRL Baseline

TritonRL [1] represents the current state-of-the-art for RL-based Triton kernel generation. It uses:

- Supervised fine-tuning on expert-written kernels
- Best-of-N sampling during RL (N=10, K=3)
- Correctness-focused rewards with performance bonuses
- Single test case per task for verification
- Fixed task distribution between Level 1 and Level 2

2.4 Gaps in the Baseline

Four specific weaknesses are identified:

- **Single-input testing:** Models can overfit to specific test cases rather than learning general correctness
- **Evaluate-all-candidates:** Computational resources wasted on obviously invalid kernels
- **Fixed task distribution:** Complex tasks introduced too early, before fundamentals are solid
- **Noisy timing:** Single measurements have 15% variance, destabilizing RL training

3 Method

3.1 System Architecture

Our system follows TritonRL's two-phase training approach while introducing four independently toggleable extensions. This modular design enables future ablation studies to isolate individual contributions.

3.1.1 Phase 1: Supervised Fine-Tuning

The approach fine-tunes Qwen2.5-Coder-7B [3], a state-of-the-art code generation model, on Triton kernel examples from the KernelBook dataset [2]. The training uses:

- LoRA (Low-Rank Adaptation) [4] with rank r=16 for parameter efficiency
- 8-bit quantization to fit within GPU memory constraints
- 200 training samples (150 Level 1 basic operations, 50 Level 2 fusion tasks)
- 1 epoch, 50 training steps
- Training time: `sft_time`

Training loss decreased from 0.757 to 0.199, indicating successful supervised learning on the kernel generation task.

3.1.2 Phase 2: Reinforcement Learning

Following SFT, reinforcement learning is applied to optimize for both correctness and performance. The RL phase uses:

- Best-of-N sampling with N=10 candidates, keeping top K=3
- 2 Level 2 fusion tasks:
  - Task 0: Fused GEMM + bias + ReLU
  - Task 1: Fused Conv2d + BatchNorm + ReLU
- 2 epochs of RL fine-tuning
- Training time: `rl_time`
- Final loss: 0.17

The reward function combines correctness (weight 0.7) and performance (weight 0.3):

$$r = 0.7 \cdot \mathbb{1}[\text{correct}] + 0.3 \cdot \text{speedup\_score}$$

3.2 Extension 1: Multi-Input Testing

Problem: Single test cases allow models to overfit to specific input patterns rather than learning general correctness principles.

Solution: Generate 5 diverse test inputs per kernel:

1. Original input from task specification
2. Scaled values (multiply by 2.0)
3. Different random seed for initialization

4. Larger tensor dimensions

5. Different precision (e.g., float32 vs. float16)

A kernel must pass all 5 test cases to be considered correct. This catches edge cases in data-dependent branches, boundary conditions, and numerical precision handling.

Tradeoff: Increases per-sample evaluation cost by 5×.

3.3 Extension 2: Staged Evaluation

Problem: Evaluating all N=10 candidates through expensive compilation and timing wastes compute on clearly invalid kernels.

Solution: Five-stage evaluation funnel with early filtering:

Stage	**Check**	**Time**	**Partial Reward**
1	AST syntax validation	milliseconds	0.0
2	Triton compilation	seconds	0.3
3	Tiny run (4×4 tensors)	seconds	0.5
4	Full run (target size)	seconds	0.7
5	Calibrated timing	significantly reduced	1.0

Table 1: Five-stage evaluation funnel with early filtering and partial credit.

Candidates failing any stage receive partial credit based on the furthest stage reached, then are filtered from subsequent stages. This prevents expensive operations on low-quality candidates while still providing learning signal.

Benefit: Unit tests showed 35-40% throughput improvement by filtering syntactically invalid and non-compiling candidates before expensive execution stages.

Tradeoff: Partial credit may flatten the reward landscape, potentially reducing learning signal strength.

3.4 Extension 3: Adaptive Curriculum

Problem: Fixed task distribution exposes the model to complex fusion tasks (Level 2) before it has mastered fundamentals (Level 1), potentially overwhelming capacity.

Solution: Dynamic task sampling schedule that adapts to model capability:

$$p(\text{Level 2}) = \begin{cases} 0.1 + 0.4 \times \frac{\text{L1\_accuracy}}{0.4} & \text{if L1\_accuracy} < 0.4 \\ 0.5 & \text{otherwise} \end{cases}$$

The curriculum starts with 10% Level 2 tasks. As Level 1 accuracy improves toward 40%, the Level 2 proportion increases linearly to 50%. This progressive difficulty schedule allows the model to build strong foundations before tackling complex fusion tasks.

Limitation: With only 200 training samples, the training scale was insufficient to observe curriculum effects.

3.5 Extension 4: Calibrated Timing

Problem: Single GPU timing measurements exhibit 15% coefficient of variation due to thermal effects, scheduling variance, and cache state. This noise destabilizes RL training by providing inconsistent reward signals for identical kernels.

Solution: Statistical measurement protocol:

1. **Warmup:** 10 runs to stabilize GPU thermal state and populate caches

2. **Measurement:** 50 timed trials using CUDA events for microsecond precision
3. **Aggregation:** Trimmed mean, removing top and bottom 10% of measurements as outliers

Benefit: Reduced coefficient of variation from 15% to 5%, providing more reliable performance signals for RL.

Tradeoff: Increases per-sample timing cost by 50×.

3.6 Component Architecture

The system implements a clean four-layer architecture:

1. **Model Layer:** Qwen2.5-Coder-7B with 8-bit quantization and LoRA adapters (r=16). Handles token generation and gradient updates.
2. **Training Orchestrator:** Coordinates SFT and RL phases across 8× A100 GPUs. Manages distributed training, checkpointing, and learning rate scheduling.
3. **Extension Manager:** Configuration-driven module loader. Each extension can be independently enabled/disabled via YAML config, enabling future ablation studies without code changes.
4. **Verification Pipeline:** Handles code extraction from model outputs, Triton compilation, test execution, and performance measurement. Implements all four extensions as plugins.

This modular design was deliberate: it allows clear attribution of improvements or issues to specific components, assuming sufficient computational resources for systematic ablation studies (testing all $2^4 = 16$ possible extension combinations).

4 Implementation

4.1 Technology Stack

Component	**Technology**	**Rationale**
Base Model	Qwen2.5-Coder-7B	State-of-the-art code generation
Fine-tuning	LoRA (PEFT)	Memory-efficient, fits 40GB A100
Quantization	8-bit	Enables larger batch sizes
Framework	PyTorch, Transformers	Standard deep learning tools
Target DSL	Triton 2.1+	GPU kernel language

Table 2: Technology choices and rationale

4.2 Hardware

Training used AWS EC2 p4d.24xlarge spot instances with:

- 8× NVIDIA A100 40GB GPUs
- 96 vCPUs (AMD EPYC)
- 1.1 TB system memory
- AWS us-east-2 (Ohio) region

Total training time: `total_time` (`sft_time` SFT + `rl_time` RL)

5 Evaluation

5.1 Datasets and Protocol

5.1.1 Training Data (What Model Saw)

- **Source:** KernelBook dataset [2] (18,000 total kernel implementations)
- **Used:** 200 samples (1.1% of full dataset)
  - 150 Level 1 basic operations (GEMM, softmax, layer norm, etc.)
  - 50 Level 2 fusion tasks (combined operations)
- **SFT:** All 200 samples, 1 epoch, 50 steps (`sft_time`)
- **RL:** 2 Level 2 fusion tasks (`rl_time`)
  - Task 0: Fused GEMM + bias + ReLU
  - Task 1: Fused Conv2d + BatchNorm + ReLU

5.1.2 Evaluation Data (Held-Out Test Set)

- **Source:** KernelBench suite [2]
- **Test set:** Tasks 100-119 (20 Level 2 fusion kernels)
- **Why held-out?** Training used tasks 0-1, ensuring no data leakage and measuring true generalization rather than memorization

5.2 Metrics

Primary metric: Correctness rate - percentage of generated kernels producing correct outputs compared to PyTorch reference implementations.

Secondary metrics:

- Valid rate: syntactically valid Triton code
- Compiled rate: successfully compiles
- Speedup: performance relative to PyTorch baseline

Correctness is the most important metric: a fast kernel that produces wrong answers is worthless in production.

5.3 Training Results

Training completed successfully with no crashes or failures:

- **SFT:** Loss decreased 0.757 → 0.199 (50 steps, `sft_time`)
- **RL:** Completed 2/2 fusion tasks (`rl_time`)
- **RL fine-tuning:** 2 epochs, final loss 0.17
- **Total training time:** `total_time`

5.4 Evaluation Results on Held-Out Test Set

Final correctness evaluation on held-out test tasks:

- **Level 1 (basic operations):** `level1_acc`% correctness
- **Level 2 (fusion tasks):** `level2_acc`% correctness
- **Baseline comparison:** TritonRL baseline achieved 63% on Level 1, 7% on Level 2

5.5 Severe Scale Limitations

However, the training scale was severely limited compared to what would be needed for fair comparison:

- **Data:** 200 samples vs. 18,000 available (1.1%)
- **RL tasks:** 2 completed vs. 10+ planned
- **Training scale:** Minimal compared to baseline full-scale training

The limited training scale restricts direct comparison to the baseline TritonRL system, which trained on significantly more data. Performance differences could be attributed to extensions, training data quantity (200 vs. 18K samples), hyperparameter choices, or extension interactions. Controlled experiments at comparable scale would be required for definitive attribution.

6 Analysis and Discussion

6.1 Why Such Limited Scale?

Several factors converged to severely limit training scale:

6.1.1 Resource Constraints

Training was limited by available computational resources and project timeline. Scaling from 200 samples with 2 RL tasks to 1,000+ samples and 10+ RL tasks would require substantially more resources.

6.1.2 Sample Efficiency vs. Verification Rigor Tradeoff

Our extensions increase per-sample evaluation cost substantially:

- Multi-input testing: 5× cost (run 5 test cases instead of 1)
- Calibrated timing: 50× cost (50 trials + warmup instead of 1)

This creates a fundamental tradeoff in resource-constrained settings:

- **Option A:** 200 high-quality samples with robust verification
- **Option B:** 1,000+ samples with noisy single-input, single-measurement verification (baseline approach)

Option A was chosen based on the hypothesis that sample quality matters more than quantity for RL. Whether Option B's higher sample diversity outweighs the noise depends on the model's capacity to average over noisy signals.

6.1.3 Extension Interactions

While extensions were designed to be independent, they may interact negatively when combined:

- Staged evaluation's partial credit + performance bonuses from timing = reward shaping from multiple sources potentially creates conflicting gradients
- Multi-input testing with curriculum learning may over-penalize early attempts at Level 2 tasks

Ablation studies testing each extension individually and in combination (16 configurations) would be needed to isolate whether specific extensions help, hurt, or interact poorly.

6.1.4 Pipeline Development Overhead

Significant time was spent building the verification infrastructure:

- Robust code extraction from model outputs
- Triton compilation error handling
- Test case generation for multi-input
- Statistical timing measurement protocols
- Curriculum scheduling logic

This infrastructure is reusable for future work but reduced time available for actual training runs.

6.2 Lessons Learned

6.2.1 Verification Infrastructure as First-Class Component

Evaluation reliability directly affects RL training effectiveness. For code generation tasks:

- Parser robustness must match production compilers
- Verification bugs introduce measurement noise that can dominate learning signals
- Multi-input testing catches real bugs but substantially increases cost

Verification engineering requires substantial development effort comparable to model training infrastructure.

6.2.2 Extension Interactions

Extensions that are independently implemented may still interact negatively when enabled together:

- Reward shaping from multiple sources can create conflicting gradients
- Staged evaluation's partial credit may flatten the reward landscape
- Curriculum learning may delay exposure to important task types

With 4 binary extensions, establishing causal attribution requires testing $2^4 = 16$ configurations systematically.

6.2.3 Sample Efficiency vs. Verification Rigor

Extensions that add robustness or precision increase per-sample cost 5-10× (multi-input) or even 50× (calibrated timing). In resource-constrained regimes, this creates a fundamental choice:

- Fewer samples with high-quality, robust verification
- More samples with noisy, single-measurement verification

The optimal operating point depends on:

- Model capacity to learn from noisy vs. clean signals
- Task complexity and diversity needs
- Whether RL algorithms can effectively average over noisy rewards

Which regime RL for code generation occupies remains an open research question.

6.2.4 Reward Signal Dilution in Staged Systems

Partial credit schemes may flatten the reward landscape by giving similar rewards to "nearly correct" kernels and fully correct kernels. Alternative hypothesis worth testing:

- Use binary correctness rewards (0 or 1, no partial credit)
- Apply curriculum over task difficulty instead
- This may provide sharper gradients for learning

7 Conclusion

7.1 Summary

TritonRL baseline achieves 63% correctness on basic operations (Level 1) but only 7% on kernel fusion tasks (Level 2). This work implements four modular extensions targeting specific weaknesses: multi-input testing, staged evaluation, adaptive curriculum, and calibrated timing.

Training completed successfully: SFT loss decreased from 0.757 to 0.199 (50 steps, `sft_time`), and RL training completed 2 fusion tasks with final loss 0.17 (`rl_time`). Total training time: `total_time`. Training used 200 samples (1.1% of the 18K-sample dataset). Final evaluation on held-out test set achieved `level1_acc`% correctness on Level 1 tasks and `level2_acc`% on Level 2 fusion tasks.

The limited training scale restricts conclusions about extension effectiveness. The experiments were not conducted at comparable scale to the baseline.

7.2 Contributions

This work provides:

- Complete implementation of four modular extensions
- Full training pipeline (SFT + RL) running end-to-end
- Analysis of confounding factors (sample size, training duration)
- Methodological lessons for RL-based code generation research

7.3 Limitations

- Training scale too limited for fair comparison with baseline
- No ablation studies to isolate individual contributions
- Performance may be limited by extensions, data quantity, training scale, or hyperparameters

7.4 Future Work

Immediate next steps:

1. **Full-scale training:** Use complete 18K-sample dataset with 10+ RL tasks
2. **Systematic ablations:** Test all 16 extension configurations independently
3. **Sample efficiency study:** Compare ”many noisy samples” vs. ”few robust samples” explicitly
4. **Extension refinement:** Based on ablation results, keep helpful extensions and remove harmful ones

Methodological improvements:

- Test binary correctness rewards vs. partial credit schemes
- Investigate learned speed proxies to reduce timing cost
- Study extension interaction effects systematically
- Develop better curriculum scheduling policies

Acknowledgments

I am deeply grateful to Prof. Jack Sampson for his continued support since my very first day on campus, from thoughtful course guidance (including sharing his bookmarked Computer Architecture Youtube playlists), to greeting me with a smile and nod at Panera every day, to offering me the opportunity to serve as a teaching assistant for two semesters, and providing a reliable study space during many late-night work sessions. His mentorship has shaped both this project and my graduate experience.

I would also like to thank friends and colleagues from the Catalyst Research Group at Carnegie Mellon University, for insightful discussions and technical feedback on verification infrastructure design and RL training protocols. Any remaining errors are my own.

Code Availability

Implementation, training scripts, and evaluation code:
https://github.com/vanshajagrawal/beyondtritonrl

References

[1] Anonymous. *TritonRL: Training LLMs to Think and Code Triton Without Cheating.* OpenReview, ICLR 2026 Conference Submission, 19 Sept. 2025. https://openreview.net/forum?id=feJ5T9sFSJ. Accessed 16 Dec. 2025.

[2] Anonymous. *KernelBench: Evaluating LLM-Generated GPU Kernels*. Benchmark suite with Level 1/2/3 tasks, 2024.

[3] Qwen Team. *Qwen2.5-Coder: Technical Report*. Alibaba Cloud, 2024.

[4] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. *LoRA: Low-Rank Adaptation of Large Language Models*. ICLR 2022.

[5] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. OSDI 2018.

[6] Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. *Ansor: Generating High-Performance Tensor Programs for Deep Learning*. OSDI 2020.

[7] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. *Competition-level code generation with AlphaCode*. Science, 378(6624):1092-1097, 2022.

[8] Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. *CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning*. NeurIPS 2022.