
BEYOND TRITONRL: REINFORCEMENT LEARNING FOR TRITON KERNEL OPTIMIZATION WITH MODULAR EXTENSIONS

Anonymous Authors¹

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 end-to-end implementation and evaluation of all components. However, our implementation significantly underperformed the baseline, achieving 15% on Level 1 tasks and 3% on Level 2 tasks (vs. baseline 63% and 7%). We provide honest analysis of why theoretically motivated extensions failed to improve performance and lessons learned about limited-budget 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.

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?

¹Anonymous 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.

- 055 3. Can adaptive curriculum learning enable better performance on complex fusion tasks?
 056
 057
 058
 059
 060
 061 4. How much does timing noise affect RL training, and can calibrated measurements help?

062 **Problem Definition:** Given a natural language description
 063 and reference PyTorch implementation, generate a functionally correct and performant Triton kernel. The challenge
 064 is particularly acute for Level 2 tasks involving kernel fusion (e.g., fused GEMM+bias, Conv+BatchNorm+ReLU),
 065 where multiple operations must be correctly combined while
 066 maintaining numerical accuracy.
 067

068 **Background:** TritonRL uses supervised fine-tuning (SFT)
 069 followed by reinforcement learning with best-of-N sampling.
 070 The model generates multiple candidate kernels and selects
 071 those with highest rewards based on correctness verification
 072 and performance measurement. However, the baseline
 073 verification is fragile (single test case, no early filtering),
 074 and the training progression is fixed rather than adaptive to
 075 model capabilities.
 076
 077

3 RELATED WORK

079 **Kernel Generation:** Prior work includes TVM’s AutoScheduler and Ansor for automated kernel optimization,
 080 but these focus on search-based methods rather than learned
 081 approaches.
 082
 083

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

089 **Curriculum Learning:** Progressive task difficulty has
 090 proven effective in RL. Our adaptive curriculum applies
 091 this principle by monitoring Level 1 (basic operations) per-
 092 formance before increasing Level 2 (fusion) task exposure.
 093

094 **Improvements Over Prior Work:** We improve upon TritonRL through: (1) hardened verification with multi-input
 095 testing vs. single test case, (2) staged evaluation for com-
 096 putational efficiency, (3) dynamic curriculum vs. fixed task
 097 distribution, and (4) calibrated timing with statistical rigor.
 098 These extensions are complementary and can be indepen-
 099 dently enabled/disabled.
 100

4 OVERVIEW

101 The system architecture consists of four independently tog-
 102 gleable extension modules integrated into the TritonRL
 103 pipeline:
 104
 105

106 **Training Pipeline:** SFT Phase (Qwen2.5-7B + LoRA) →
 107 RL Phase (Best-of-N Sampling) → Extension Stack (Multi-
 108
 109

Input, Staged Eval, Curriculum, Timing) → Reward Func-
 055 tion [0-1]

056 **Data Flow:** KernelBook (18K samples) → 1K training
 057 subset → SFT → RL → Evaluation
 058

059 **Key Design Choice:** Modular architecture allows ablation
 060 studies and independent validation of each extension’s con-
 061 tribution.
 062
 063

5 METHOD

5.1 Implementation Details

064 **Base Model:** Qwen/Qwen2.5-Coder-7B-Instruct, a state-of-
 065 the-art code generation model.
 066

Training Configuration:

- **Hardware:** 8 × 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)

067 **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

068 **Motivation:** Single test case verification allows models to
 069 overfit to specific input patterns.
 070

071 **Implementation:** Generate 5 diverse test inputs per kernel:
 072 (1) original input, (2) scaled values ($\times 2.0$), (3) different
 073 random seed, (4) larger dimensions, (5) different precision.
 074 Test generation logic is custom-implemented using PyTorch
 075 tensor operations.
 076

5.3 Extension 2: Staged Evaluation

077 **Motivation:** Evaluating all generated kernels wastes com-
 078 pute on invalid candidates.
 079

080 **Implementation:** Five-stage funnel with increasing cost:
 081 (1) AST Check (ms), (2) Compilation (seconds), (3) Tiny
 082 Run on 4×4 tensor (seconds), (4) Full Run with multi-input
 083 (seconds), (5) Timing measurement (minutes). Candidates
 084 failing any stage receive partial credit (0.0, 0.3, 0.5, 0.7
 085 rewards).
 086

087 **Benefit:** Unit testing showed ~35-40% throughput improve-
 088 ment by filtering invalid kernels early.
 089

110 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.

119 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 ~15% to ~5%.

129 5.6 What We Implemented vs. Adopted
130 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

144 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

156 6 EVALUATION
157 6.1 Experimental Settings

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

161 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 completed but results significantly underperformed the baseline. We successfully completed both SFT and RL training phases, but our implementation achieved lower correctness rates than the TritonRL baseline on both Level 1 and Level 2 tasks.

Actual Results:

Task Level	Baseline	Ours (Best)	Diff
Level 1 (Basic)	63%	15%	-48%
Level 2 (Fusion)	7%	3%	-4%
SFT Loss	N/A	0.757→0.199	N/A
RL Tasks	N/A	10/10 completed	N/A

Table 1. Comparison with TritonRL baseline - our implementation underperformed

Why Our Implementation Underperformed:

1. **Evaluation Verifier Issues:** Initial evaluation failures (0% valid rate) were caused by code extraction bugs. After fixing the verifier, evaluation revealed our model generated 15% correct on Level 1 and 3% on Level 2—significantly below baseline.
2. **Simplified Training Pipeline:** Due to time/budget constraints, we used only 1,000 samples (vs. full 18K dataset), 1 SFT epoch, and limited RL iterations. The baseline likely trained on more data with more iterations.
3. **Extension Integration:** While individual extensions showed promise in unit tests (e.g., 35–40% throughput gain for staged evaluation), their combined effect in the full training pipeline may have introduced unexpected interactions or the training time was insufficient to benefit from them.
4. **Model Selection:** We used Qwen2.5-Coder-7B-Instruct, which may not be optimal for Triton kernel generation compared to the model used in TritonRL baseline.

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 Can and Cannot Claim:

What we achieved:

- Complete end-to-end training pipeline execution
- Honest evaluation showing 15% Level 1 and 3% Level 2 correctness
- SFT loss reduction from 0.757 to 0.199
- RL training completed 10/10 tasks with slight improvement over SFT

What we cannot claim:

- **Our extensions did not improve over baseline** - we underperformed by 48% on Level 1 and 4% on Level 2
- No ablation studies to isolate individual extension contributions
- Cannot determine if poor performance was due to extensions, limited training data, or model choice

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:

- **Results significantly underperformed baseline.** Our best model achieved 15% (Level 1) and 3% (Level 2) vs. baseline 63% and 7%.
- **Limited training scale.** Used 1,000 samples vs. full 18K dataset, 1 SFT epoch, and minimal RL iterations due to budget constraints.
- **No ablation studies.** Cannot isolate whether poor performance was due to extensions, data scarcity, model choice, or training duration.
- **Negative result interpretation unclear.** Uncertain if extensions hurt performance or if insufficient training prevented benefits.

What This Report Provides:

- Complete end-to-end implementation and evaluation
- Honest negative results: 15%/3% vs. baseline 63%/7%
- Analysis of why our approach underperformed
- Lessons for future RL-based code generation research
- Demonstration that good ideas don’t always translate to improvements

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 end-to-end implementation of four modular extensions, (2) honest negative results: 15% Level 1 and 3% Level 2 vs. baseline 63% and 7%, (3) analysis of why theoretically sound extensions failed to improve performance, (4) lessons for future RL-based code generation research.

Results Summary: Training completed successfully (SFT loss: 0.757→0.199, RL: 10/10 tasks), but our implementation significantly underperformed the baseline. Likely causes include limited training data (1K vs. 18K samples), insufficient training iterations (1 SFT epoch), possible negative interactions between extensions, or suboptimal model choice.

220 **Key Takeaways:** (1) Theoretically motivated extensions
221 don't guarantee improvements—our "better" verification,
222 evaluation, curriculum, and timing did not translate to bet-
223 ter results. (2) Limited training budgets make it difficult
224 to distinguish between "extensions hurt performance" vs.
225 "insufficient training to see benefits." (3) Negative results are
226 valuable: future work should validate extensions on small
227 scale before scaling up, and ablation studies are critical. (4)
228 Honest reporting of failures is as important as celebrating
229 successes.

230 **Code Repository:** [https://github.com/
231 vanshajagrawal/beyondtritonrl](https://github.com/vanshajagrawal/beyondtritonrl)

232 **Project Presentation:** [https://
233 drive.google.com/drive/folders/
234 1B3xkmlPtHR3myixEjc5Lv-1dxUp9M15E](https://drive.google.com/drive/folders/1B3xkmlPtHR3myixEjc5Lv-1dxUp9M15E)

237 REFERENCES

- 238 [1] TritonRL: Reinforcement Learning for Triton Kernel
239 Generation.
- 240 [2] KernelBook Dataset, ScalingIntelligence/KernelBook,
241 HuggingFace Hub.
- 242 [3] P. Tillet, H. T. Kung, and D. Cox, "Triton: An Interme-
243 diate Language and Compiler for Tiled Neural Network
244 Computations," *MAPL 2019*.
- 245 [4] Qwen Team, "Qwen2.5-Coder Technical Report," 2024.
- 246 [5] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large
247 Language Models," *ICLR 2022*.
- 248 [6] R. Nakano et al., "WebGPT: Browser-assisted question-
249 answering with human feedback," *NeurIPS 2021*.

250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274