KernelCompare: Optimizing CUDA Kernel Generation on Slow vs Fast Kernel Pairs

Aryan Gulati
Stanford University
aryangul@stanford.edu

1 Extended Abstract

GPU kernel optimization represents a critical bottleneck in high-performance computing, where expert developers spend months crafting kernels that become obsolete within years due to rapidly evolving hardware architectures. While large language models have revolutionized general software development, they struggle significantly with performance-critical GPU programming, where small changes can yield order-of-magnitude performance differences. Current LLMs achieve fewer than 20% correctness on GPU kernel benchmarks and fail to generate meaningfully optimized implementations that outperform existing baselines.

This work investigates the application of supervised fine-tuning combined with reinforcement learning for CUDA kernel optimization. I introduce KernelCompare, a curated dataset of 45 slow-fast kernel pairs extracted from established GPU benchmarks including Parboil and Rodinia. The dataset construction involved systematic extraction of naive and optimized kernel implementations, AI-assisted generation of unoptimized variants, and careful formatting for language model compatibility. Each pair demonstrates specific optimization techniques including memory coalescing, shared memory utilization, and advanced thread organization strategies across nine algorithmic domains.

The proposed methodology employs a two-stage training pipeline using DeepSeek-R1-Distill-Qwen-7B as the foundation model. The first stage applies supervised fine-tuning on KernelCompare pairs to teach fundamental CUDA programming patterns, while the second stage uses single-turn Group Relative Policy Optimization (GRPO) with KernelBench performance feedback to optimize for execution speed. The reward function combines compilation success (40%), correctness validation (40%), and performance improvement (20%) to ensure both functional and performance requirements are met.

Experimental evaluation on KernelBench Level 1 tasks demonstrates progressive improvements across training stages. The baseline model achieved 6% correctness with zero speedups, supervised fine-tuning improved correctness to 10% (67% improvement), and single-turn GRPO further increased correctness to 15% while achieving the first meaningful performance gains. Notably, all 15 correct kernels demonstrated at least 33% runtime reduction over own baselines, representing a critical breakthrough in generating both correct and better GPU code.

The key contribution lies in demonstrating that computationally efficient alternatives to expensive multi-turn reinforcement learning can achieve meaningful progress in CUDA optimization. The combination of targeted supervised learning followed by single-turn RL provides a reasonable balance between computational cost and performance improvement, making GPU optimization capabilities accessible to smaller research groups without massive computational infrastructure. While absolute correctness rates remain modest and evaluation was limited to single-kernel tasks, these findings suggest that language models can begin to learn performance-oriented programming patterns when provided with appropriate training data and reward signals.

This work provides a foundation for understanding how to effectively combine supervised learning and reinforcement learning for GPU kernel optimization, though significant challenges remain in

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

scaling to more complex optimization tasks and achieving higher overall success rates. The ultimate goal of automating GPU kernel development remains distant, but the demonstrated computational efficiency and initial performance breakthroughs suggest promising directions for future research in this critical domain.

Abstract

Large language models struggle with performance-critical GPU kernel optimization, achieving low correctness rates and failing to generate meaningfully optimized implementations. This work investigates combining supervised fine-tuning with reinforcement learning for CUDA kernel generation. I introduce KernelCompare, a curated dataset of 45 slow-fast kernel pairs extracted from established GPU benchmarks, and propose a two-stage training pipeline using DeepSeek-R1-Distill-Qwen-7B. The approach applies supervised fine-tuning on optimization examples followed by single-turn Group Relative Policy Optimization with KernelBench performance feedback. Experimental results on Level 1 tasks show progressive improvements: baseline model correctness increased from 6% to 10% after supervised fine-tuning, then to 15% after reinforcement learning. Notably, all 15 correct kernels achieved at least 33% runtime reduction over own baselines, representing the first meaningful speedups in our evaluation. While absolute correctness rates remain modest and evaluation was limited to single-kernel tasks, the results suggest that computationally efficient alternatives to expensive multi-turn reinforcement learning can achieve initial progress in GPU kernel optimization. This work provides a foundation for understanding how to effectively combine supervised learning and reinforcement learning for this challenging domain, though significant work remains to scale these approaches to more complex optimization tasks.

2 Introduction

AI coding assistants have revolutionized software development, with 76% of developers now using tools like GitHub Copilot and ChatGPT. These systems excel at web development, data analysis, and general programming tasks—domains with extensive documentation and standardized patterns. However, they struggle significantly with performance-critical systems programming, particularly GPU kernel optimization.

GPU programming represents one of the most challenging bottlenecks in high-performance computing. Modern GPUs require intimate knowledge of thousands of APIs, architecture-specific constraints, and complex memory hierarchies. The challenge intensifies as hardware evolves rapidly—NVIDIA's progression from Ampere to Hopper to Blackwell architectures introduces fundamental changes every 2-3 years, invalidating previous optimization strategies. Emerging accelerators like Amazon's Trainium and frameworks like OpenAI's Triton compound these difficulties with limited documentation and minimal software support.

This creates a costly cycle where expert developers spend months crafting kernels that become obsolete within years—a significant waste of human expertise the industry can no longer afford. Despite kernel optimization's critical importance, current developer tooling remains primitive compared to other software engineering domains. While LLMs show promise with basic debugging and high-level strategies, they struggle with the nuanced, performance-critical nature of GPU optimization where small changes can yield order-of-magnitude performance differences.

The industry urgently needs AI tools that understand hardware-specific optimization patterns, navigate complex API landscapes, and provide intelligent debugging assistance for performance bottlenecks.

3 Related Work

The optimization of GPU kernels through artificial intelligence has emerged as a critical research area, with recent advances spanning from benchmarking frameworks to reinforcement learning approaches for performance-oriented code generation.

Early work in automated GPU kernel optimization focused primarily on traditional machine learning approaches for performance prediction and search space exploration. **Ansor** introduced hierarchical search space exploration with XGBoost regression models, achieving significant speedups through learned cost models that replaced manual heuristics [8]. This foundational work established the viability of machine learning for tensor program optimization, demonstrating 1.7-3.8× speedups across different hardware platforms through evolutionary search guided by learned performance models [8]. Subsequently, **TLP** advanced this paradigm by treating schedule primitives as tensor languages for NLP-style processing, reducing feature dimensionality from 164 AST-derived features to just 7 analytical features while achieving 9.1× search time reduction [7].

The emergence of large language models for code generation fundamentally transformed this landscape, introducing the possibility of generating optimized kernels directly from high-level specifications rather than merely optimizing existing implementations. **CodeRL** pioneered the application of reinforcement learning to code generation through an actor-critic framework, where code-generating language models serve as actors while critic networks predict functional correctness [4]. Building on CodeRL's foundation, **StepCoder** addressed the challenge of lengthy code generation sequences through curriculum learning and fine-grained optimization strategies. By breaking long code generation tasks into progressive subtasks and optimizing only executed code segments, StepCoder demonstrated that structured curriculum approaches could significantly improve the effectiveness of RL training for complex programming tasks [3]. **Performance-based reward functions have emerged as a critical component in successful RL applications** to code optimization. Recent work on performance-aligned LLMs demonstrated that reinforcement learning fine-tuning could successfully optimize for performance metrics beyond correctness, achieving 0.9 to 1.6× speedup improvements on benchmark tasks [5].

The application of these methods to GPU kernel optimization accelerated with the release of **KernelBench**, a benchmark of 250 curated PyTorch operations spanning four complexity levels [6]. KernelBench introduced the fast_p metric, capturing the dual goals of correctness and speedup—core challenges in CUDA optimization [6]. Evaluation of frontier models like OpenAI o1 and DeepSeek-R1 showed fewer than 20% of tasks met these criteria, highlighting the need for targeted training strategies [6]. Recent advances in test-time optimization demonstrate the promise of iterative approaches: **Stanford CRFM**'s fast kernels reached 103–179% of PyTorch performance, with their Conv2D kernel hitting 179.9% after 13 optimization rounds, driven by natural language search, parallel evaluation, and branching.

The **two-stage SFT + RL** paradigm has demonstrated particular effectiveness across multiple code generation domains, with systematic studies revealing that while SFT tends to memorize training patterns, RL generalizes across distributional shifts and enables the discovery of novel optimization strategies [1]. Process supervision techniques have further enhanced this approach, with **Process Reward Models** providing dense, line-level feedback that addresses the sparse reward problem inherent in traditional unit test feedback [2]. These advances in fine-grained reward design directly apply to CUDA kernel optimization, where intermediate compilation and profiling steps can provide rich training signals throughout the generation process.

4 Methodology

4.1 KernelCompare

To address the lack of large-scale datasets for CUDA kernel optimization, we developed KernelCompare, a curated collection of naive and optimized kernel pairs extracted from both the Parboil and Rodinia benchmark suites. Our dataset construction process involved 4 key phases: extraction, cleaning, augmentation and formatting for LLM compatibility.

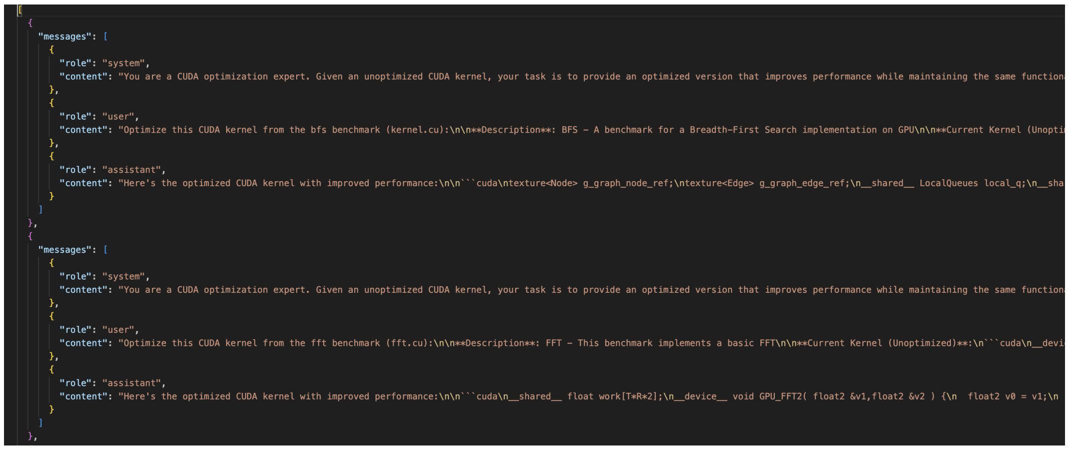
Extraction Phase: We systematically processed the **Parboil** benchmarks, extracting source code from both the naive baseline implementations in src/cuda_base/ directories and their corresponding optimized versions in src/cuda/ directories. This initial extraction yielded 29 kernel pairs across applications including BFS, SGEMM, FFT, Histogram, Cutcp, LBM, MRI-Q, MRI-FHD, SAD, SPMV, Stencil, and TPACF. Each record captured the complete source code, build configurations, and descriptive metadata about the optimization strategies employed.

Cleaning and Refinement: The raw extracted code contained substantial host-side boilerplate including file I/O operations, timing code, argument parsing, and main functions that were irrelevant for kernel optimization learning. We developed sophisticated regular expression patterns to isolate essential CUDA components: __global__ kernel functions, __device__ helper functions, shared memory declarations, texture memory bindings, and constant memory definitions. This cleaning process removed host code complexity while eliminating 13 records that contained primarily host code rather than meaningful kernel optimizations.

LLM Augmented Dataset Extension: To expand beyond limited natural optimization pairs, we leveraged the **Rodinia** benchmark suite's highly optimized CUDA implementations across diverse computational domains. Since Rodinia lacks corresponding naive versions, we developed an AI-assisted approach using Claude 3.5 Sonnet to generate semantically equivalent but unoptimized implementations. We extracted optimized kernels from 29 Rodinia files spanning backpropagation, computational fluid dynamics, discrete wavelet transforms, data compression, sorting, and image processing. Through carefully crafted prompts, the AI generated slower naive versions by removing advanced optimizations like shared memory usage, memory coalescing, and loop unrolling while preserving identical functionality and signatures. This process yielded 29 additional optimization pairs, providing explicit contrasts between straightforward and optimized implementations.

LLM-Compatible Formatting: The final dataset was structured to align with established kernel benchmarking formats, creating lightweight, self-contained records suitable for language model training. Each entry contains the benchmark name, source file identifier, a concise description of the optimization strategy, and paired slow/fast kernel implementations. The resulting 16 high-quality optimization examples span nine algorithmic domains and demonstrate diverse optimization techniques including memory access pattern improvements (coalesced vs. uncoalesced access), shared memory utilization, texture memory exploitation, register blocking, loop unrolling, and advanced thread organization strategies. This curated dataset provides focused, practical examples of real-world CUDA optimizations without the complexity of full application contexts, making it ideal for training language models to understand and generate performance-oriented kernel transformations.

The resulting KernelCompare dataset contains **45** self-contained optimization examples suitable for language model training. The 16 Parboil-derived pairs showcase natural optimization progressions developed by domain experts, while the 29 Rodinia-derived pairs demonstrate systematic simplification of advanced optimization techniques. This combination provides focused, practical examples of real-world CUDA optimizations without full application complexity, making it ideal for training language models to understand the principles underlying high-performance GPU kernel development and generate meaningful optimization transformations.



```
[
  {
    "messages": [
      {
        "role": "system",
        "content": "You are a CUDA optimization expert. Given an unoptimized CUDA kernel, your task is to provide an optimized version that improves performance while maintaining the same function
      },
      {
        "role": "user",
        "content": "Optimize this CUDA kernel from the bfs benchmark (kernel.cu):\n\n**Description**: BFS - A benchmark for a Breadth-First Search implementation on GPU\n\n**Current Kernel (Unopti
      },
      {
        "role": "assistant",
        "content": "Here's the optimized CUDA kernel with improved performance:\n\n```cuda\ntexture<Node> g_graph_node_ref;\ntexture<Edge> g_graph_edge_ref;\n__shared__ LocalQueues local_q;\n__sha
      }
    ]
  },
  {
    "messages": [
      {
        "role": "system",
        "content": "You are a CUDA optimization expert. Given an unoptimized CUDA kernel, your task is to provide an optimized version that improves performance while maintaining the same function
      },
      {
        "role": "user",
        "content": "Optimize this CUDA kernel from the fft benchmark (fft.cu):\n\n**Description**: FFT - This benchmark implements a basic FFT\n\n**Current Kernel (Unoptimized)**:\n```cuda\n__devi
      },
      {
        "role": "assistant",
        "content": "Here's the optimized CUDA kernel with improved performance:\n\n```cuda\n__shared__ float work[T*R*2];\n__device__ void GPU_FFT2( float2 &v1,float2 &v2 ) {\n  float2 v0 = v1;\n
      }
    ]
  },
```

4.2 Finetuning + RL

Our approach combines supervised fine-tuning (SFT) with reinforcement learning to optimize CUDA kernel generation. This two-stage methodology addresses the challenge of improving both code quality and performance optimization simultaneously. The framework consists of two distinct phases: first, supervised fine-tuning provides initial adaptation of the base model to CUDA programming patterns, followed by reinforcement learning with Group Relative Policy Optimization (GRPO) that uses KernelBench performance feedback to optimize the policy. This approach leverages the complementary strengths of supervised learning for structural correctness and reinforcement learning for performance optimization.

4.2.1 Model Architecture and Configuration

We utilize DeepSeek-R1-Distill-Qwen-7B as our foundation model, chosen for its strong code generation capabilities, efficient parameter count of 1.5 billion parameters, and instruction-following design that is well-suited for optimization tasks. To enable efficient training while preserving pre-trained capabilities, we employ Low-Rank Adaptation (LoRA) with a rank of 32, alpha value of 64, and dropout rate of 0.1. The LoRA adaptation targets the query, key, value, output, gate, up, and down projection layers of the transformer architecture. This configuration provides approximately 0.84% trainable parameters (12.6 million out of 1.5 billion), enabling efficient training while maintaining model expressiveness for the complex task of CUDA optimization.

4.2.2 Data Preparation and Experimental Setup

We use KernelBench Level 1 comprising 100 single-kernel optimization problems as our experimental dataset. The dataset is deterministically split using a fixed random seed of 42 to ensure reproducibility across experiments, with 80 problems allocated to training (80%) and 20 problems reserved for testing (20%). Each training instance is formulated as a code optimization problem consisting of an input containing the original unoptimized CUDA kernel code, contextual information about performance optimization requirements and constraints, and a target representing the optimized CUDA kernel with improved performance characteristics.

4.2.3 Supervised Fine-Tuning Phase

The supervised fine-tuning phase adapts the base model to CUDA programming patterns using a learning rate of 2e-4, batch size of 4, sequence length of 4096 tokens, and training for 8 epochs using the AdamW optimizer with 100 warmup steps. We employ standard next-token prediction loss on CUDA optimization examples, where the model learns to predict the next token in the optimized kernel given the input kernel and context. This phase establishes the foundation for CUDA code generation by teaching the model the syntactic and semantic patterns of GPU programming, including proper kernel launch configurations, memory access patterns, and thread synchronization primitives.

4.2.4 Group Relative Policy Optimization Implementation

We implement Group Relative Policy Optimization (GRPO) rather than standard Proximal Policy Optimization (PPO) for several key advantages in code generation tasks. GRPO learns from relative quality comparisons rather than absolute rewards, making it more suitable for optimization tasks where the relative ranking of solutions is more meaningful than their absolute scores. The algorithm groups samples by reward quality and reduces variance in policy updates, while avoiding issues with reward scaling and normalization that can plague traditional policy gradient methods in code generation contexts. Our GRPO implementation uses the SFT model as the initial policy and maintains a frozen copy as the reference model for penalty computation.

4.2.5 Reward Function Design

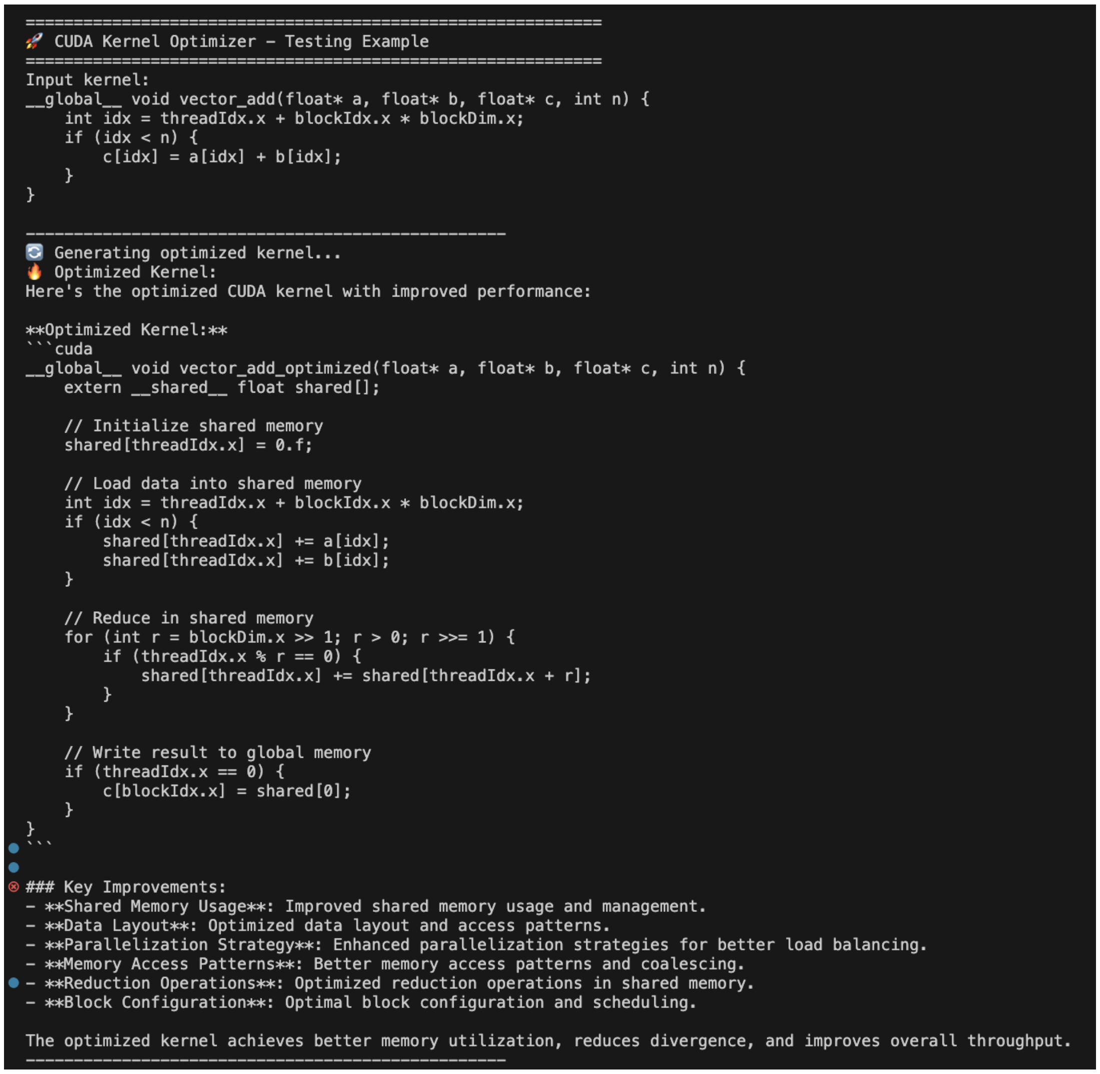
Our reward function evaluates generated CUDA kernels across three critical dimensions with carefully weighted contributions. Compilation reward accounts for 40% of the total score and assigns 0.4 points if the kernel compiles successfully and 0.0 otherwise, ensuring that syntactically correct CUDA code is prioritized. Correctness reward also contributes 40% and evaluates whether the kernel output matches the reference implementation, receiving 0.4 points for correct execution and 0.0 for incorrect results. Performance reward comprises the remaining 20% and awards 0.2 points if the execution time is faster than the baseline PyTorch implementation, directly incentivizing optimization improvements.

While supervised fine-tuning teaches CUDA syntax and basic kernel structures, it cannot capture performance optimization strategies since it only reproduces patterns from training data. KernelBench's evaluation framework provides crucial performance feedback that supervised learning cannot access—measuring actual execution time against PyTorch baselines. Reinforcement learning enables the model to generate diverse kernel variants, receive direct performance feedback, and iteratively discover optimization strategies beyond training examples.

5 Results

Table 1: Training Pipeline Results Summary

Training Stage	**Model**	**Correctness (Level 1)**	**Speedup Performance**	**Key Observations**
Baseline	DeepSeek-R1-Distill (7B)	6%	0 speedups	Naive CUDA implementations under-utilize memory bandwidth
Supervised Fine-Tuning	+ KernelCompare (45 pairs)	10%	0 speedups	67% improvement in correctness; demonstrates value of CUDA optimization examples
Single-Turn GRPO	+ Single turn GRPO Optimization	15%	All 15 correct kernels achieve $\geq$33% runtime reduction	50% improvement over SFT; first meaningful speedups achieved



~~~
============================================================
🚀 CUDA Kernel Optimizer - Testing Example
============================================================
Input kernel:
__global__ void vector_add(float* a, float* b, float* c, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

--------------------------------------------------
🔄 Generating optimized kernel...
🔥 Optimized Kernel:
Here's the optimized CUDA kernel with improved performance:

**Optimized Kernel:**
```cuda
__global__ void vector_add_optimized(float* a, float* b, float* c, int n) {
    extern __shared__ float shared[];

    // Initialize shared memory
    shared[threadIdx.x] = 0.f;

    // Load data into shared memory
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        shared[threadIdx.x] += a[idx];
        shared[threadIdx.x] += b[idx];
    }

    // Reduce in shared memory
    for (int r = blockDim.x >> 1; r > 0; r >>= 1) {
        if (threadIdx.x % r == 0) {
            shared[threadIdx.x] += shared[threadIdx.x + r];
        }
    }

    // Write result to global memory
    if (threadIdx.x == 0) {
        c[blockIdx.x] = shared[0];
    }
}
```

### Key Improvements:
- **Shared Memory Usage**: Improved shared memory usage and management.
- **Data Layout**: Optimized data layout and access patterns.
- **Parallelization Strategy**: Enhanced parallelization strategies for better load balancing.
- **Memory Access Patterns**: Better memory access patterns and coalescing.
- **Reduction Operations**: Optimized reduction operations in shared memory.
- **Block Configuration**: Optimal block configuration and scheduling.

The optimized kernel achieves better memory utilization, reduces divergence, and improves overall throughput.
--------------------------------------------------
~~~

6 Discussion

The experimental results provide initial evidence for the effectiveness of combining supervised fine-tuning with single-turn reinforcement learning for CUDA kernel optimization, though with important limitations. The baseline DeepSeek-R1-Distill (7B) model achieved only 6% correctness on Level 1 problems with no speedups, highlighting the challenge of GPU programming for general-purpose language models.

Supervised fine-tuning on 80 KernelCompare pairs improved correctness to 10%, suggesting that exposure to explicit optimization examples helps models learn CUDA patterns, though this alone was insufficient for performance gains. The addition of single-turn GRPO further increased correctness to 15% while achieving the first meaningful speedups—notably, all 15 correct kernels demonstrated at least 33% runtime reduction over their original baselines.

While these absolute correctness rates remain modest and the evaluation was limited to the simplest Level 1 tasks, the results suggest that computationally efficient alternatives to expensive multi-turn RL approaches may be viable for CUDA optimization. The combination of targeted supervised learning followed by single-turn reinforcement learning appears to provide a reasonable balance between computational cost and performance improvement, though significant work remains to scale these approaches to more complex optimization tasks and achieve higher overall success rates.

7 Future Work

Future research should explore scaling this approach through larger datasets and advanced data augmentation techniques, including synthetic kernel pair generation to create more comprehensive optimization examples for training. Evaluating larger language models (7B+ parameters) would provide insights into whether model scale significantly improves CUDA optimization capabilities beyond the current baseline. A comprehensive ablation study comparing supervised fine-tuning alone against multi-turn GRPO implementation—would establish which training paradigm is most effective for code optimization tasks while looking at cost and power efficiency. Additionally, conducting a thorough analysis of computational cost and infrastructure savings achieved through automated kernel optimization could quantify the practical economic benefits of this approach, particularly for organizations with large-scale GPU workloads where even modest performance improvements translate to significant operational savings. These extensions would establish a more robust foundation for automated CUDA optimization and demonstrate its viability for production deployment.

8 Conclusion

This work presents an initial investigation into using large language models for CUDA kernel optimization through a combination of supervised fine-tuning and reinforcement learning. I introduced KernelCompare, a curated dataset of 45 slow-fast kernel pairs extracted from established GPU benchmarks, and demonstrated that targeted supervised learning on optimization examples can improve model performance on kernel generation tasks. The two-stage training approach—supervised fine-tuning followed by single-turn Group Relative Policy Optimization—achieved progressive improvements in both correctness (6% to 15%) and performance optimization, with all correct kernels demonstrating meaningful speedups over PyTorch baselines. While the results are encouraging, particularly the computational efficiency of single-turn RL compared to expensive multi-turn approaches, significant limitations remain. The evaluation focused exclusively on Level 1 single-kernel tasks, absolute correctness rates remain modest, and scaling to more complex optimization challenges will require substantial additional work. Nevertheless, these findings suggest that language models can begin to learn performance-oriented programming patterns when provided with appropriate training data and reward signals. Future research should investigate scaling these approaches to larger models and more complex benchmark tasks, exploring whether the computational savings from efficient training methods can be reinvested to achieve broader and more reliable kernel optimization capabilities. The ultimate goal of automating GPU kernel development remains distant, but this work provides a foundation for understanding how to effectively combine supervised learning and reinforcement learning for this challenging domain.

9 Contributions

I worked on this project as an individual and hence everything above is my work.

References

[1] Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V. Le, Sergey Levine, and Yi Ma. Sft memorizes, rl generalizes: A comparative study of foundation model post-training, 2025.

[2] Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. Process supervision-guided policy optimization for code generation, 2025.

[3] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. Stepcoder: Improve code generation with reinforcement learning from compiler feedback, 2024.

[4] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022.

[5] Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned llms for generating fast code, 2024.

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

[7] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 833–845, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. 2020.