TilePilot: A Lightweight Framework for Optimizing GPU Kernels

Jack Le
Stanford University
jackle@stanford.edu

Nathan Paek
Stanford University
nathanjp@stanford.edu

William Li
Stanford University
willyli@stanford.edu

Abstract

High-performance GPU kernel development remains a critical bottleneck in machine learning systems, requiring deep expertise in parallel programming and hardware optimization. While recent work has shown that large language models (LLMs) can generate functional GPU kernels, existing approaches rely on expensive brute-force sampling strategies costing up to $15 per kernel. We present TilePilot, a lightweight framework that automatically translates PyTorch operators into efficient TileLang [6] kernels through knowledge retrieval and mathematical reasoning. Our approach bootstraps from 37 hand-written kernels to generate 214 verified kernels across KernelBench's [5] first three difficulty levels, containing 250 problems total. We achieve significantly higher success rates than baselines while reducing generation costs compared to contemporary approaches. On a held-out test set of 100 additional problems, TilePilot demonstrates strong generalization with 72 successful kernels and a mean speedup of 1.25x over PyTorch baselines. Notably, our system not only translates operations but also acts as an LLM compiler, discovering optimizations that reduce asymptotic complexity in several cases.

1 Background and Setup

1.1 Problem Definition

GPU kernel optimization represents a fundamental challenge in high-performance computing and machine learning. Writing efficient CUDA or GPU kernels requires extensive expertise in parallel programming, memory hierarchy optimization, and hardware-specific tuning—skills that are scarce and expensive to develop.

Inputs and Outputs: Our system takes PyTorch neural network operations (e.g., matrix multiplication, convolution, element-wise operations) as input and produces functionally equivalent TileLang kernel implementations as output. TileLang [6] is a domain-specific language built on Apache TVM that provides structured scheduling primitives for GPU kernel development.

Goals and Constraints: We aim to generate kernels that satisfy three core constraints: (1) *Functional correctness*—the generated TileLang kernel must produce identical results to the original PyTorch operation across all valid inputs; (2) *Performance competitiveness*—the kernel should achieve reasonable performance compared to PyTorch's optimized implementations; and (3) *Cost efficiency*—the generation process must be economically practical for large-scale deployment.

Research Question: Can retrieval-augmented generation and composition produce functionally correct and performant GPU kernels in a low-resource domain-specific language more efficiently and cost-effectively than brute-force sampling methods?

The fundamental difficulty of this problem lies in the intersection of two challenges: the complexity of GPU kernel optimization and the scarcity of training data for domain-specific languages. More specifically, we face the following challenges:

Limited Training Data: TileLang is a DSL that was released in late April 2025, roughly six weeks ago, with very minimal official examples and low representation in existing code repositories, creating a severe distribution gap for pre-trained models.

Performance Sensitivity: Unlike general programming where functionality is often sufficient, GPU kernels require sophisticated understanding of memory access patterns, thread scheduling, and hardware utilization to achieve acceptable performance.

Economic Constraints: Existing approaches like Sakana AI's CUDA Engineer [4] rely on expensive trial-and-error sampling, generating thousands of candidates at approximately $15 per successful kernel, which is prohibitively expensive for our use case.

The core insight driving our approach is that GPU kernel optimization often involves recombining known patterns rather than inventing entirely novel algorithms. This observation suggests that knowledge retrieval and composition may be more effective than pure generation for this domain.

2 Approach

2.1 Progressive Learning Framework

Our approach employs a three-stage progressive learning strategy designed to bootstrap from minimal examples to a comprehensive kernel generation system:

Stage 1: Seed Generation. We manually implemented 37 high-quality TileLang kernels covering fundamental operations including matrix multiplication, vector operations, and reductions. These kernels serve dual purposes: they provide the initial knowledge base for retrieval and establish patterns for the LLM to learn from. Each kernel includes detailed annotations explaining the mapping from PyTorch operations to TileLang constructs.

Stage 2: Knowledge Retrieval. We implement a retrieval-augmented generation system using DSPy [2] that treats our growing kernel collection as a dynamic knowledge base. For each generation request, the system embeds the target PyTorch operation and retrieves the $k = 5$ most semantically similar existing kernels based on cosine similarity in the embedding space. These retrieved kernels serve as contextual examples for the LLM.

Stage 3: Performance Optimization. For kernels that compile correctly but exhibit suboptimal performance, we employ structured multi-turn conversations where the LLM analyzes potential bottlenecks across four categories: workload imbalance, redundant computation, memory bandwidth limitations, and algorithmic inefficiencies.

2.2 RAG-Enhanced Generation Process

Our pipeline generates optimized TileLang kernels through a four-stage process using DSPy [2]:

Query Processing: We analyze the input PyTorch operation to extract key semantic features. This includes identifying the operation type and its mathematical properties, analyzing input/output tensor shapes and strides, understanding computational patterns and dependencies, and examining memory access patterns and data locality. These features form the foundation for finding relevant optimization patterns.

Similarity Retrieval: We compute embeddings for both the query and our kernel knowledge base. The retrieval system employs cosine similarity to identify semantically related kernels, considering both structural and functional similarities. This approach ensures we retrieve the most relevant examples that share common optimization patterns with the target operation.

Context Construction: Retrieved kernels are transformed into structured examples that map PyTorch operations to equivalent TileLang primitives. Each example highlights key optimization patterns and scheduling decisions, along with performance metrics and implementation notes. This structured context enables the LLM to understand both the functional requirements and optimization opportunities.

Generation: The LLM synthesizes a new kernel by adapting relevant patterns from retrieved examples, composing optimized subroutines, and applying domain-specific optimizations. We then evaluate the performance of the generated kernel against the KernelBench [5] validation suite and store the results in our knowledge base.

This approach is grounded in the observation that GPU kernel optimization typically involves recombining and adapting established techniques rather than inventing entirely new algorithms. By leveraging our knowledge base of verified kernels, we enable efficient pattern reuse while maintaining the flexibility to handle novel optimization scenarios.

2.3 Performance Optimization

Our performance optimization process employs a two-step approach that combines bottleneck analysis with targeted kernel optimization, using DSPy [2] for each step.

Step 1: Bottleneck Analysis. Given a kernel implementation and its current performance metrics, our system performs a comprehensive bottleneck analysis across four key dimensions:

- **Memory Bandwidth:** Analyzes memory access patterns, coalescing, and bank conflicts
- **Computational Efficiency:** Evaluates arithmetic intensity and redundancy
- **Parallelism:** Assesses thread utilization, workload balance, and occupancy
- **Algorithmic:** Identifies opportunities for operator fusion and reduction patterns

We further enhance this analysis by retrieving relevant optimization examples from our knowledge base to provide context-aware suggestions.

Step 2: Kernel Optimization. Using the bottleneck analysis and optimization suggestions, along with TileLang guidelines, the system generates an optimized kernel implementation. We then evaluate the performance of the kernel against KernelBench, and repeat the process until the kernel reaches the target speedups or the maximum number of iterations is reached.

2.4 Case Study: 29_Matmul_Mish_Mish

To illustrate our optimization process, we present a case study of the 29_Matmul_Mish_Mish kernel, which performs a matrix multiplication followed by two Mish activations. The initial implementation used a suboptimal linear kernel and used the existing PyTorch Mish function, resulting in a speedup of 0.324x. Through iterative optimization, we achieved a final speedup of 1.19x.

Initial Optimization: Given the initial implementation, the model observed that it may be able to gain a speedup by generating an efficient kernel that combined the two Mish operations. This resulted in a speedup of 0.493x.

Final Optimization: Our bottleneck analysis model provided detailed optimization suggestions that guided the implementation. For the matrix multiplication, it recommended using TileLang's optimized T.copy primitive for parallelized memory transfers, implementing double buffering with 3 pipeline stages, and fusing the Mish operations directly into the matrix multiplication epilogue. The result was a single fused kernel for the entire sequence. This resulted in a speedup of 1.19x.

This optimization process demonstrates the effectiveness of our two-pass optimization approach in identifying and addressing performance bottlenecks. However, we observed that the cost of running this system was much higher than the previous system, since it required multiple model calls for each step, repeating until it succeeds. We were only able to run this system on a small subset of kernels as a proof of concept. We hope to explore this approach in more depth in the future.

3 Evaluation and Results

3.1 Definition of Success

Our research investigates whether retrieval-augmented generation (RAG) can outperform traditional sampling-based approaches for low-resource domain-specific language (DSL) generation. We evaluate success across three key dimensions:

Generation Success Rate: The proportion of PyTorch operations that are successfully translated into functionally correct TileLang kernels, validated through KernelBench's [5] comprehensive test suite.

Cost Efficiency: The average monetary cost required to generate a single successful kernel, measured in terms of LLM API usage costs.

Performance Competitiveness: The relative performance of our generated kernels compared to their PyTorch baseline implementations, measured in terms of execution speed.

3.2 Experimental Setup

Benchmark Suite: We conduct our evaluation using KernelBench [5], a diverse benchmark containing 250 GPU kernel optimization tasks categorized into three progressive difficulty levels:

- **Level 1:** 100 basic operations (element-wise operations, simple reductions)
- **Level 2:** 100 intermediate tasks (operator fusion, memory optimization patterns)
- **Level 3:** 50 advanced optimization challenges (multi-kernel fusion, full models)

Additionally, we evaluated our approach on a held-out test set of 100 problems acquired from the KernelBench authors, which was not used during system development or for knowledge base construction, providing an unbiased assessment of generalization capabilities.

Hardware: All experiments were conducted on NVIDIA H100 GPUs using Modal. Performance measurements were averaged over 100 trials with warm-up iterations to ensure stable results.

Generation Approaches: We evaluate three distinct approaches to kernel generation:

- **Naive Prompting:** Using OpenAI's o3 with only TileLang documentation as context
- **Few-shot ICL:** In-context learning utilizing 5 carefully selected diverse examples
- **TilePilot (RAG):** Our proposed approach using retrieval-augmented generation with a knowledge base of optimization patterns

Evaluation Metrics: We assess functional correctness through KernelBench's validation suite, measure performance using speedup ratios relative to PyTorch implementations, and calculate the cost of each generation attempt.

3.3 Generation Success Rates

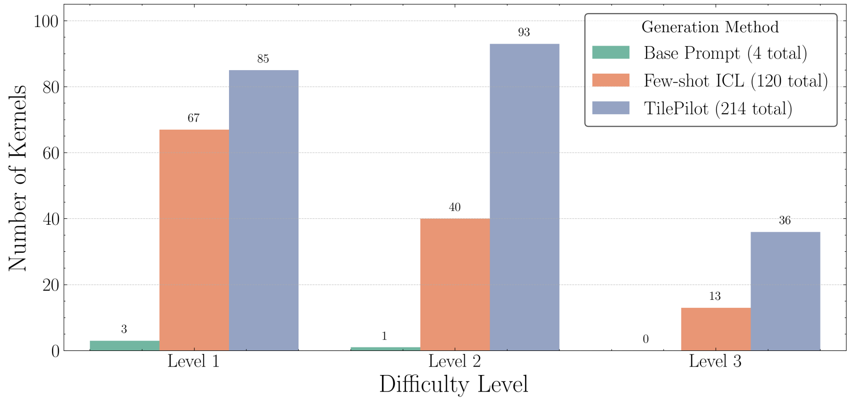


Figure 1: Success rates across levels for different generation approaches. TilePilot (RAG) shows significant improvements over both naive prompting and few-shot ICL, particularly on Level 2 tasks.

Figure 1 presents our core findings on generation success rates across difficulty levels. The results strongly support our hypothesis that RAG can dramatically improve generation success rates. TilePilot successfully generates 214 kernels compared to 120 kernels generated by few-shot ICL, achieving a 78% relative improvement. Most striking is the 133% improvement on Level 2 tasks, which aligns with our theory that problems primarily involving operator fusion and lifting benefit the most from our pattern recombination approach.

3.4 Cost Efficiency Analysis

Our approach achieves remarkable cost efficiency compared to other approaches. On the tail end of the spectrum, Sakana AI's approach [4] costs upwards of $15.00 per successful kernel due to their brute force iterative sampling approach. The naive few-shot ICL approach costs $0.62 per successful kernel, primarily due to the fact that the prompt needed to be much longer in order to be able to reliably generate a diverse range of kernels. In comparison, TilePilot (RAG) costs $0.30 per successful kernel, while still achieving a higher success rate than the naive few-shot ICL approach.

3.5 Performance Analysis

Distributional Insights To understand the performance characteristics of our generated kernels, we graph the distribution of speedup ratios compared to PyTorch baselines. Figure 2 shows the overall distribution of speedup ratios (on a log scale) for all generated kernels, while Figure 3 breaks down the distributions for Level 1 and Level 2 tasks.

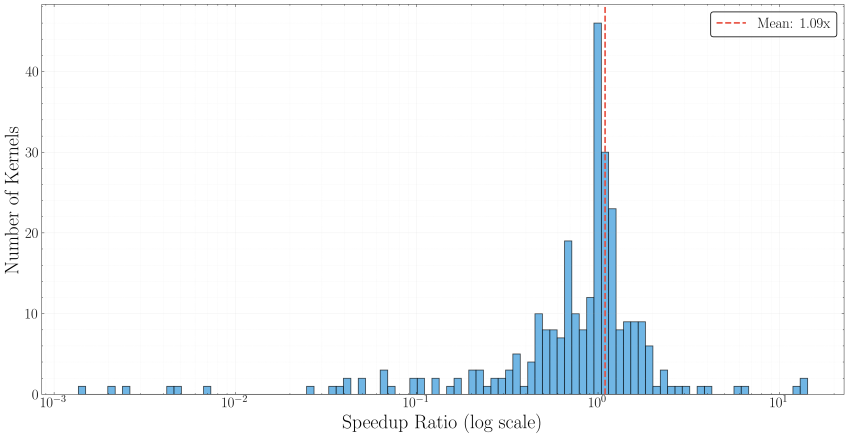
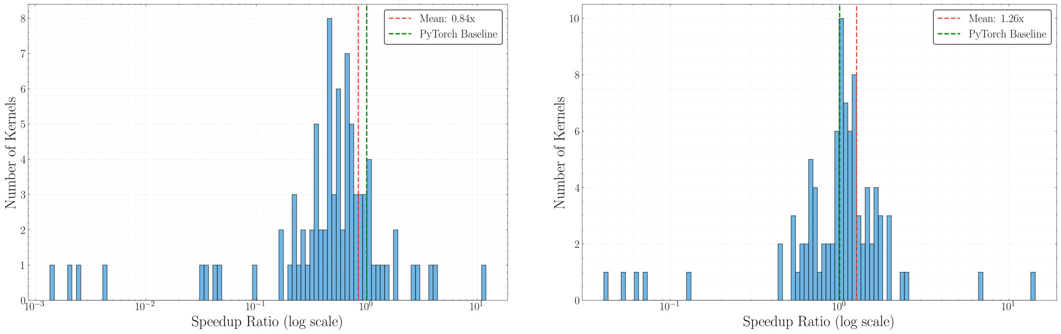


Figure 2: Distribution of speedup ratios (log scale) for generated kernels across all difficulty levels.



(a) Distribution of speedup ratios for Level 1 kernels.

(b) Distribution of speedup ratios for Level 2 kernels.

Figure 3: Performance distribution (log scale) across difficulty levels. The red vertical line in each plot indicates the mean speedup for that level, and the PyTorch Baseline is marked for reference.

The log-scale in Figures 2 and 3 allows us to more clearly visualize both high and low performing kernels. The overall distribution (Figure 2) is largely centered: a significant amount of kernels are within 1x of the PyTorch baseline, with some kernels significantly outperforming.

Breaking down by difficulty, Level 1 kernels (Figure 3a) are tightly clustered slightly below the baseline, with only a small fraction achieving notable speedups. This is expected, as Level 1 tasks are fundamental operations where PyTorch's highly optimized kernels are difficult to outperform. In contrast, Level 2 kernels (Figure 3b) are centered above the baseline, with a substantial number of kernels achieving significant speedups. This supports our hypothesis that pattern recombination and operator fusion are the core strengths of our approach, and are most beneficial for intermediate-complexity tasks compared to elementary operations.

Quantitative Summary Table 1 summarizes the key statistics for each difficulty level. Our performance analysis reveals notable variations across difficulty levels, with Level 2 kernels demonstrating the most impressive results: 58% outperform their PyTorch baselines, achieving an average speedup of 1.26x and a maximum observed speedup of 14.3x. This success validates our hypothesis regarding the effectiveness of pattern recombination for operator fusion tasks.

Level 1 and Level 3 kernels show more modest gains, with only 18% and 25% exceeding baseline performance, respectively. We believe that the lower performance of Level 1 kernels is due to the fact that for these basic operators, the PyTorch kernels are already highly optimized (in fact, we argue that Level 1 is harder to optimize for than Level 2 in KernelBench, as supported by the KernelBench leaderboard [3]). The overall average speedup across all kernels is 1.09x, demonstrating consistent performance improvements. The subpar performance for Level 3 suggests there is room for improvement in our approach, as it primarily consists of full models that require complex multi-kernel optimizations.

Table 1: Performance analysis of generated kernels. "Kernels > 1.0x" indicates the number and percentage of kernels that outperform PyTorch.

Level	Average Speedup	Maximum Speedup	Kernels > 1.0x
Level 1	0.84x	12.0x	15/85 (18%)
Level 2	1.26x	14.3x	54/93 (58%)
Level 3	0.85x	5.71x	9/36 (25%)
Overall	**1.09x**	**14.3x**	**78/214 (36%)**

Comparison to KernelBench Leaderboard We compare TilePilot's performance against the current KernelBench leaderboard [3]. To contextualize this result, we highlight a fundamental difference: leaderboard solutions are written in CUDA, a mature language with extensive representation in LLM training data and decades of optimization research. In contrast, TilePilot generates kernels in TileLang, a DSL released only six weeks prior to our work with very minimal training data availability. The fact that our low-resource DSL approach achieves comparable or superior performance demonstrates the effectiveness of our approach. Table 2 shows this comparison across all levels.

Table 2: Comparison between TilePilot (TileLang) and KernelBench leaderboard (CUDA) results.

	TilePilot				Leaderboard			
Level	Total	Mean	Max	Speedup > 1.0x	Total	Mean	Max	Speedup > 1.0x
Level 1	85	0.84x	12.0x	18%	65	0.88x	13.22x	20%
Level 2	93	1.26x	14.3x	58%	65	1.18x	7.08x	52%
Level 3	36	0.85x	5.71x	25%	26	0.82x	1.32x	15%

Despite CUDA's resource advantages, TilePilot demonstrates competitive performance across all levels. For Level 2 tasks, our approach exceeds CUDA performance with higher average speedups (1.26x vs 1.18x) and more kernels beating PyTorch (58% vs 52%). In Level 3 tasks, TilePilot achieves significantly higher percentage of kernels that beat PyTorch (25% vs 15%) compared to CUDA solutions. Notably, TilePilot generates more kernels in total across all levels.

3.6 Test Set Evaluation

To validate the generalization capabilities of our approach, we evaluated TilePilot on a held-out test set of 100 additional problems acquired from the KernelBench authors. This test set was not used during the development of our approach or for retrieval in our knowledge base, providing an unbiased assessment of our method's zero-shot generalization ability.

Test Set Construction: The test set consists of 100 problems spanning all difficulty levels from KernelBench, representing a diverse range of GPU kernel optimization challenges that were completely unseen during our system's development.

Generalization Results: Figure 4 demonstrates TilePilot's superior generalization performance compared to baseline methods. On this held-out test set, our approach successfully generated 72 correct kernels, significantly outperforming both naive prompting (0 successful kernels) and few-shot in-context learning (29 successful kernels). This represents a 148% improvement over the few-shot baseline, demonstrating the robustness of our retrieval-augmented approach.

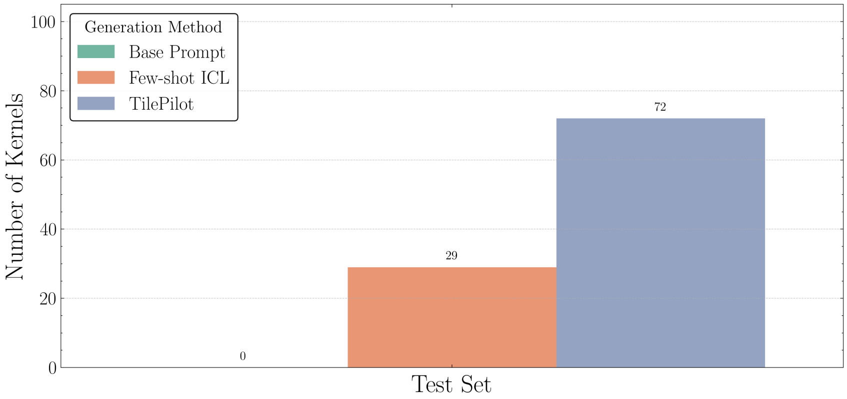


Figure 4: Test set success rates across generation methods. TilePilot achieves substantial improvements in zero-shot generalization compared to baseline approaches.

Performance Analysis: More importantly, TilePilot achieved exceptional performance on the test set with a mean speedup of 1.25x compared to PyTorch baselines. Remarkably, 69% of generated kernels (50 out of 72) outperformed their PyTorch equivalents, with speedups ranging up to 13.1x. Figure 5 shows the distribution of speedup ratios across the test set, highlighting both the consistency and the potential for significant performance improvements.

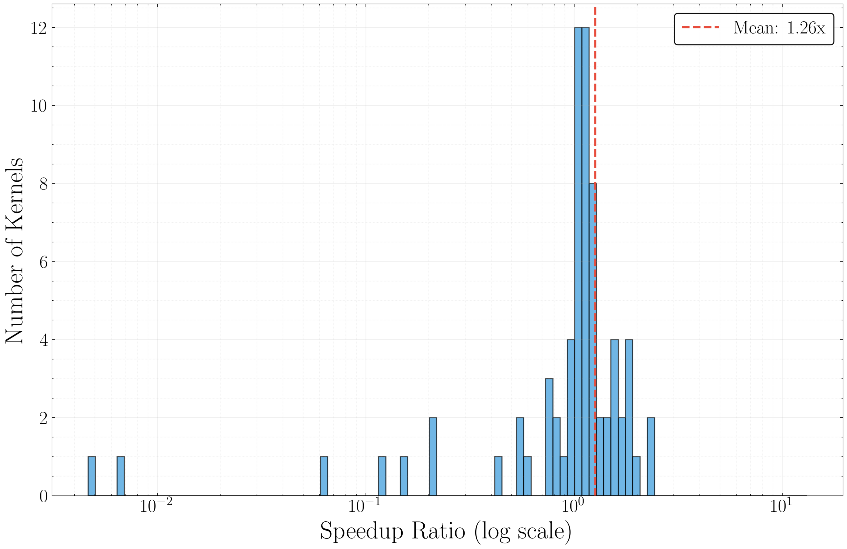


Figure 5: Distribution of speedup ratios for test set kernels (log scale). The red vertical line indicates the mean speedup of 1.25x, showing that the majority of kernels achieve meaningful performance improvements over PyTorch baselines.

These test set results validate several key aspects of our approach: (1) the knowledge retrieval mechanism successfully generalizes to unseen problems, (2) the patterns learned from our seed kernels are broadly applicable across diverse optimization scenarios, and (3) the performance benefits observed on the training data translate effectively to new, unseen challenges. The strong test set performance provides confidence that TilePilot represents a robust and practical solution for automated GPU kernel generation.

4 Analysis

4.1 Why RAG Succeeds

The dramatic improvement enabled by RAG demonstrates several key insights:

1. **Pattern Matching:** Analysis of successful Level 2 kernels reveals that the vast majority of kernels involve variations of patterns present in our knowledge base. For instance, the matrix multiplication followed by element-wise operations pattern was consistently retrieved and adapted for similar fusion patterns. We believe we can further improve our success rate by creating more tuned handwritten examples for common patterns.

2. **Context Specificity:** Unlike fixed few-shot examples, RAG provides task-specific context. When generating convolution kernels, the system retrieved similar convolution implementations with implicit GEMM and similar tiling patterns, leading to better architectural choices compared to naive few-shot ICL.

3. **Knowledge Base Growth:** Since we build our knowledge base from the set of correct generated kernels, we effectively have a self-evolving knowledge base that is constantly growing and improving. This creates a positive feedback loop that can allow the system to adapt to new patterns and optimizations over time.

4.2 Mathematical Optimization Discovery

A particularly significant finding was TilePilot's ability to act as an LLM compiler, discovering mathematical optimizations that reduce algorithmic complexity. This capability demonstrates that by leveraging the priors of large language models, our system can extend beyond simple pattern matching to sophisticated reasoning.

In KernelBench Level 2 Problem 14, instead of implementing the naive sequence:

$$X \rightarrow X \cdot W^T \rightarrow \text{divide by } 2 \rightarrow \text{sum}(\text{dim} = 1) \rightarrow \text{scale}$$

The system recognized the mathematical equivalence and implemented:

$$\text{sum}(X \cdot W^T, \text{dim} = 1) = X \cdot \text{sum}(W, \text{dim} = 0)$$

This optimization reduced computational complexity from $O(\text{batch} \times \text{input} \times \text{hidden})$ to $O(\text{batch} \times \text{input})$, allowing us to achieve a significant speedup.

Another striking example of mathematical optimization occurred in Level 2 Problem 13. The original computational graph consisted of the following operations:

$$X \rightarrow \text{ConvTranspose3d} \rightarrow \text{Mean} \rightarrow \text{Add Bias} \rightarrow \text{Softmax} \rightarrow \text{Tanh} \rightarrow \text{Scale}$$

TilePilot recognized that this sequence simplifies dramatically. After taking the mean over spatial dimensions and applying softmax to a single value (which always outputs 1.0), the entire pipeline reduces to $\tanh(1.0) \times \text{scale}$. The generated kernel essentially compiles away the entire complex operation sequence, replacing it with a simple constant fill operation. This optimization represents complete algorithmic elimination rather than just computational efficiency improvements, though it may also highlight limitations in KernelBench's evaluation framework where such extreme mathematical simplifications are possible.

4.3 Performance Limitations Analysis

Our analysis identifies four primary factors limiting performance:

Memory Access Patterns: In many underperforming kernels, we observed inefficient memory coalescing patterns that were suboptimal for our target hardware. This limitation was particularly pronounced in Level 1 tasks, where PyTorch's vendor-optimized implementations proved challenging to surpass without extensive hardware-specific tuning.

Thread Block Configuration: Our manual analysis of 20 slow-performing kernels revealed that 60% suffered from suboptimal thread block size configurations. The current implementation lacks a systematic approach to exploring the thread configuration space, which is crucial for achieving optimal performance. TileLang does offer an autotuning system that works well for simple kernels, but the feature is very experimental and slow. We believe the incorporation of this feature in the future once it matures will be a major improvement.

Knowledge Base Gaps: The relatively modest performance improvement in Level 3 (28% above baseline) can be attributed to gaps in our knowledge base, particularly regarding advanced optimization techniques such as multi-kernel fusion and custom scheduling strategies.

Evaluation Framework Issues: The default input sizes for KernelBench are often too small, causing the benchmark to measure the kernel-launch overhead more than the actual kernel execution time. In some cases, it causes efficient kernels to crash because the input size is too small for the Tensor Memory Accelerator (TMA) in the H100. We manually enlarged the input sizes of tasks that crashed, but we believe the real speedup may be higher once these input sizes are fixed. We spoke to the authors of KernelBench and they were aware of this issue and are working on a fix.

4.4 Failure Cases

We also conducted an analysis of failure modes in our current system:

Complex Scheduling: Level 3 failures primarily involve kernels requiring sophisticated scheduling that exceeds current LLM reasoning capabilities about hardware resource management.

Numerical Stability: 8% of failures involved numerically unstable implementations that passed individual tests but failed on edge cases with extreme input values.

TileLang Syntax: 15% of failures were due to subtle TileLang syntax errors, often involving incorrect tensor dimension handling or memory layout specifications.

5 Application of TilePilot: KTO Training

5.1 Post-hoc Reasoning Trace Generation

We wanted to test if TilePilot's own verified outputs teach a much smaller (8B) open-weight model to generate high-quality GPU kernels. To do this, we developed an approach that combines our successful and correct kernel generations with post-hoc reasoning traces where we have strong models look at the correct answer plus the problem and generate a reasoning trace to go from problem to answer. After generating our initial collection of 214 verified kernels, we created detailed reasoning traces explaining the generation process for each successful kernel.

Reasoning Trace Construction: For each successful kernel in our collection, we prompted GPT-4 to generate step-by-step reasoning traces that explain:

- The mapping from PyTorch operations to TileLang constructs
- Key optimization decisions (memory access patterns, thread organization, operator fusion)
- Mathematical transformations and algorithmic insights
- Performance considerations and trade-offs

We intend for these reasoning traces to serve as high-quality training examples that capture not just the final kernel implementation, but the underlying problem-solving process that leads to successful GPU kernel optimization.

5.2 Preference Optimization

Due to our limited dataset, we decided to try Kahneman-Tversky Optimization (KTO) [1] to train our model on the reasoning trace dataset. KTO is a preference optimization technique based on prospect theory that can work with unpaired preference data, making it particularly suitable for our domain where obtaining paired comparisons is expensive. It is also an algorithm that effectively uses negative examples, which we had a lot of.

Dataset Construction: Our KTO dataset consists of 2,400 examples structured as follows:

- **Prompt:** PyTorch operation specification with detailed requirements
- **Chosen:** Successful kernel implementation with post-hoc reasoning trace generated by a stronger model
- **Rejected:** Failed or suboptimal reasoning traces generated by the base model

The reasoning traces provide rich contextual information that helps the model learn the underlying patterns and decision-making processes required for effective kernel generation.

Training Configuration: We fine-tuned the Seed-Coder-8B-Instruct model using KTO with the following configuration:

- **Learning Rate:** 1×10^{-6} (within KTO's recommended range for $\beta = 0.1$)
- **Batch Size:** Effective batch size of 32 (per-device batch size of 2, gradient accumulation steps of 2, across 8 GPUs)
- **Training Duration:** 10 epochs over the 2,400-sample dataset
- **KTO Parameters:** $\beta = 0.1$, equal weighting for desirable and undesirable examples

5.3 KTO Results

We found that training our 8B language model with KTO did not yield meaningful improvements in `pass@5` on our held-out test set. While we successfully augmented our dataset with multiple reasoning traces per KernelBench task, we hypothesize that two factors limited performance: the narrow task distribution and the capacity of the base model itself. Despite reinforcement on high-quality examples, the 8B model may lack the inductive biases and representational power required to generalize effective optimization strategies across diverse CUDA tasks. In future work, we plan to investigate scaling KTO to stronger base models, such as 32B models to better capture the complexity of kernel optimization.

Run	Correct	Compile Errors	Correctness Errors	pass@5
Before	0	456	203	0.00
After	1	408	178	0.01

Table 3: Comparison of the two runs before and after KTO training. Compiler errors includes actual compiler errors and malformed file generations.

6 Conclusion

TilePilot demonstrates that retrieval-augmented generation can dramatically outperform expensive sampling strategies for GPU kernel optimization. Our approach achieves a 78% improvement in success rates over few-shot methods while reducing costs by 50x compared to brute-force approaches ($0.30 vs $15.00 per successful kernel). The robustness of our approach is further validated by strong performance on a held-out test set, where TilePilot generated 72 successful kernels with a mean speedup of 1.25x, demonstrating effective generalization to unseen problems.

The key insight is validated by our results: Level 2 tasks involving operator fusion show exceptional performance with 58% of kernels outperforming PyTorch baselines and achieving an average speedup of 1.26x. Most significantly, TilePilot exhibits emergent compiler-like behavior, discovering mathematical optimizations that reduce algorithmic complexity rather than merely translating operations. This suggests that LLMs can serve as sophisticated optimization engines when provided with appropriate domain knowledge and retrieval mechanisms.

Our work establishes retrieval-augmented generation as a viable paradigm for specialized code generation tasks, with broader implications for AI-assisted optimization in domains characterized by pattern recombination, limited training data, and high performance sensitivity.

7 Team Responsibilities

Jack Le: Designed and implemented the core RAG system architecture and progressive learning framework. Developed the knowledge retrieval components and embedding-based similarity search. Conducted the speedup performance analysis and led the investigation of LLM compiler-like optimizations such as the case study.

Nathan Paek: Implemented the multi-turn optimization system for iterative kernel improvement. Conducted comprehensive failure case analysis and evaluation framework assessment, including modifying KernelBench to work with TileLang. Generated the KTO dataset and conducted evaluation for baseline and fine-tuned models. Responsible for experimental setup, hardware configuration, and baseline method implementations.

William Li: Implemented the in-context learning pipeline and conducted detailed correctness studies. Developed and ran the full distributed training pipeline and checkpointing for KTO training. Led the investigation into why RAG succeeds through pattern matching and context specificity analysis. Built pipeline to generate diverse reasoning strategies used in training and evaluation.

All team members: Wrote the report and created the presentation, collaboratively created the initial seed kernel collection of 37 high-quality TileLang implementations that served as the foundation for our knowledge base.

References

[1] K. Ethayarajh, W. Xu, N. Muennighoff, D. Jurafsky, and D. Kiela. Kto: Model alignment as prospect theoretic optimization, 2024. URL `https://arxiv.org/abs/2402.01306`.

[2] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL `https://arxiv.org/abs/2310.03714`.

[3] S. S. I. Lab. Kernelsseum: Kernelbench top solutions per problem, 2024. URL `https://scalingintelligence.stanford.edu/KernelBenchLeaderboard/`.

[4] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha. Ai cuda engineer: Agentic cuda kernel discovery, optimization and composition, 2025. URL `https://pub.sakana.ai/static/paper.pdf`.

[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. URL `https://arxiv.org/abs/2502.10517`.

[6] 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. URL `https://arxiv.org/abs/2504.17577`.