

Adapting Language Models for Low-Resource GPU Kernel Programming

Stanford CS224N Custom Project

Laasya Konidala
Department of Computer Science
Stanford University
laasya@stanford.edu

Natalia Pahlavan
Department of Computer Science
Stanford University
natalia9@stanford.edu

Annmaria Antony
Department of Computer Science
Stanford University
annmaria@stanford.edu

Abstract

Large language models perform well on high-resource programming languages, but they struggle to generate low-resource, compiler-verified languages such as AMD HIP, where open-source training data is scarce and performance constraints are strict. In this work, we investigate methods for improving HIP kernel generation with (1) using RAG over HIP/ROCm documentation and (2) SFT on a closed-loop synthetic data pipeline built on top of KernelBench. As part of this work, we constructed a synthetic dataset of 118 high-quality kernel examples containing the KernelBench task description, reference PyTorch architectures, generated HIP kernels, compilation and correctness results, speedup measurements, and full prompt histories across iterative attempts. We benchmark Qwen2.5-Coder-7B-Instruct on AMD MI350X GPUs using KernelBench on 3 metrics: compilation success, functional correctness, and execution speedup relative to PyTorch baselines. Our experiments show that retrieval from HIP documentation significantly improves compilation and correctness rates compared to a baseline model. Specifically, we found that the RAG pipeline improved the correctness rate of our baseline by 40%, 36%, and 22% across levels 1, 2, and 3, respectively. Further improvements are obtained through supervised fine-tuning on the synthetic kernel dataset. We discovered that the best kernels were produced when the RAG context included information about shared-memory usage and the SFT model was trained on iteratively refined examples. However, performance improvements remain limited at higher optimization thresholds.

1 Key Information to include

- Mentor: Tristan Thrush
- External Collaborators (if you have any): Simon Guo (Scaling Intelligence Lab)
- Sharing project: No.

2 Introduction

The performance, cost, and energy efficiency of AI workloads is consistently bottlenecked by the quality of their underlying kernels, low-level functions that leverage massive parallelism to implement tensor operations on GPUs [1]. Writing high-performance kernels requires deep knowledge of low-

resource languages like CUDA or HIP, hardware specifications, and various optimization techniques to achieve near-peak performance.

While Large Language Models (LLMs) have excelled at tasks such as software engineering [2], their ability to generate highly-performant GPU kernels remains limited as they struggle with languages such as HIP, CUDA, or Verilog where open-source training data is scarce despite the existence of the compiler as a perfect verifier [3]. This is a challenging problem for LLMs to solve as the task requires both correct and performant kernels. However, LLMs grapple with reasoning about performance issues due to the lack of information in their textual code input. In addition, there is a tradeoff between code correctness and performance, as hardware optimizations needed for efficiency commonly cause bugs to arise [4].

Recent work has explored using language models to generate GPU kernels and optimize them automatically. KernelBench [1] evaluates whether LLMs can generate efficient CUDA kernels for deep learning workloads and proposes a benchmark to measure correctness and performance. Other work has explored agentic approaches where models iteratively generate, compile, and refine kernels using execution feedback [5]. There has also been early work applying reinforcement learning to kernel generation tasks, where models improve through multiple rounds of execution feedback [6].

However, most prior work focuses on CUDA and NVIDIA GPUs. Much less work has examined kernel generation for AMD hardware or HIP [7]. In addition, many existing approaches rely heavily on large training corpora or iterative optimization loops but do not investigate techniques that are commonly used in low-resource settings, such as retrieval from language documentation or training on synthetic data. Our work focuses specifically on these settings.

Therefore, we look to investigate how LLMs can be improved on low-resource, compiler-verified programming languages, specifically focusing on AMD’s HIP as a case study. Specifically, we want to answer the following question: How can retrieval-augmented generation (RAG) on HIP/ROCm documentation and fine-tuning on synthetic data meaningfully improve an LLM’s ability to generate correct and performant HIP kernels?

We focus on GPU kernel generation for AMD hardware, which has not been previously explored as much as NVIDIA GPUs. We also generated a synthetic dataset of 118 HIP kernels and have used these samples to fine-tune with supervised fine-tuning (SFT). We found that injecting HIP documentation from our RAG pipeline significantly improved the correctness rate by 40%, 36%, and 22% across levels 1, 2, and 3, respectively. We found that SFT improved upon RAG gradually for levels 1 and 2, but decreased correctness for level 3 tasks. Our findings show that correctness and performance were significantly lower for level 3 tasks, suggesting that more complex training techniques are needed for these types of problems.

3 Related Work

3.1 KernelBench: Can LLMs Write Efficient GPU Kernels?

KernelBench [1] is an open-source benchmark tool to evaluate the ability of language models to generate optimized GPU kernels. Specifically, KernelBench contains 250 diverse tasks across three levels of deep learning workloads, ranging from individual operations like matrix multiplies to full machine learning architectures.

However, KernelBench leaves several important research directions unexplored. The work stops short of investigating various approaches used in low-resource settings such as retrieval augmented generation over language documentation, reinforcement learning from the execution outputs, or creating synthetic training data. Further, in their evaluation, they focus on CUDA kernels and NVIDIA GPUs, so there is room to explore how other hardware and programming libraries would perform, such as AMD GPUs and HIP.

3.2 Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization

Sakana AI’s AI CUDA Engineer framework [5] iteratively produces CUDA kernels using an LLM, compiles and executes them against PyTorch reference implementations, and evaluates both correctness and performance. A central component of this process is a kernel archive that stores previously generated implementations and their profiling results. For each optimization step, the agent retrieves

strong kernels from this archive and includes them as few-shot examples in the prompt which guides the model toward higher-quality implementations. The system forms a closed optimization loop, allowing the agent to explore the large design space of GPU kernels and progressively get to stronger solutions (as the dataset grows with more examples).

However, a key limitation of this study is that the synthetic data produced is not used to update the model. Our work addresses this limitation by treating the agentic optimization loop as a data generator such that the resulting corpus can support SFT for kernel generation.

3.3 Kevin: Multi-Turn RL for Generating CUDA Kernels

Recent work has explored training open-source models with RL for the GPU kernel generation task. Baronio et al. [6] introduces a multi-turn RL approach to improve CUDA kernel generation by allowing the model to iteratively refine its output using execution feedback to inform its policy. Their multi-turn training approach enables the model to gradually improve kernels by exploring different optimization strategies across iterations.

While Kevin demonstrates that RL can improve CUDA kernel generation, the work focuses exclusively on CUDA and relies on QwQ-32B, a large reasoning model with 32.5B parameters. We expand upon Kevin’s work by focusing on AMD hardware using Qwen2.5-Coder-7B-Instruct, a code-specific smaller model with 7.61B parameters. As the authors report, generating high-quality kernels often requires many iterations before reaching a correct and performant implementation, making RL training unstable and computationally expensive when using smaller models. We investigate whether supervised fine-tuning alone can use a smaller model to generate correct and performant HIP GPU kernels without requiring online exploration during training.

Our project addresses the gaps in these three works. We focus on HIP kernel generation, injected HIP documentation from a RAG pipeline at inference time, and created a synthetic dataset to support SFT.

4 Approach

The goals of our project are three-fold: (1) Applying a RAG pipeline to inject AMD HIP documentation into the model’s context. (2) Generating a large corpus of synthetic data using a closed-loop synthetic data generation process inspired by Sakana AI’s agentic framework. Lastly, (3) Using SFT to train the model to generate performant HIP kernels.

We used KernelBench as our baseline (Fig. 1) [1]. Given a reference implementation in PyTorch, the language model is prompted to output an optimized version of the implementation in HIP with any hardware-efficiency techniques. The LM-generated code is then compiled, evaluated on random inputs, and measured with wall-clock execution time. KernelBench contains 250 curated deep learning tasks, and their corresponding PyTorch implementations, spanning three levels of complexity. In this work, we use all 250.

We constructed a corpus from HIP documentation and example kernels. Given the target task, we use a RAG pipeline to retrieve relevant information from the corpus and provide it into the model’s context before it generates kernels. With this approach, we aim to reduce compilation failures from syntax errors.

To generate training data for SFT, we created a synthetic dataset of HIP kernels using a closed-loop generation process inspired by Sakana’s agentic kernel optimization systems [5]. Starting from PyTorch reference implementations in KernelBench, we prompted the language model to produce HIP kernels that implement the same functionality. Each generated kernel was compiled and evaluated using the KernelBench pipeline. Kernels that successfully compiled and passed functional correctness tests were stored as candidate training examples.

Through this process we generated a dataset of 118 HIP kernels across different difficulty levels. The resulting dataset includes both simple operations such as elementwise kernels and more complex kernels involving shared memory and thread synchronization. We then used these examples to perform supervised fine-tuning so that the model could learn the general structure and syntax of HIP kernels.

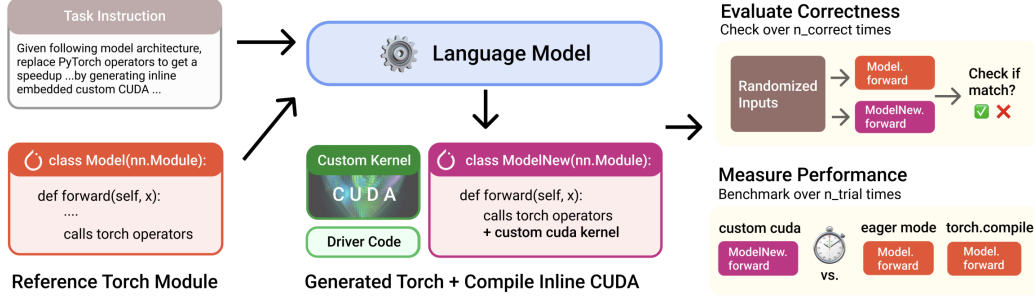


Figure 1: Overview of the KernelBench workflow. Image from KernelBench paper

5 Experiments

5.1 Data

For our RAG pipeline, we constructed a corpus from HIP documentation, API references, and example HIP kernels [8, 9, 10]. We cloned GitHub repositories from ROCm such as ROCm/HIP and ROCm/rocm-examples and used FireCrawl to scrape HIP documentation from ROCm web docs [11] into Markdown files. In total, the corpus contains 590 documents.

We ran our agentic generation pipeline on all three difficulty levels of KernelBench, producing 114 verified HIP kernels from 162 attempted problems. Refer to Section A.1 for the generated synthetic data. Table 1 summarizes the dataset composition.

Table 1: Synthetic HIP kernel dataset generated via agentic loop with Gemini 2.5 Flash. Each kernel passes compilation with `hipcc`, correctness checks against the PyTorch reference, and performance benchmarking on an AMD MI350X GPU.

Level	Description	Attempted	Solved	Rate	Avg. Speedup	Avg. Attempts
1	Single operators	44	22	50.0%	1.48×	3.3
2	Fused operators	100	76	76.0%	1.05×	1.7
3	Full architectures	18	16	88.9%	0.71×	1.9
Total		162	114	70.4%		

5.2 Evaluation method

We extended KernelBench to support AMD GPUs by building support to run on all three levels of tasks. We measured (1) compilation success rate on AMD GPUs, (2) functional correctness as determined by KernelBench’s reference outputs and (3) execution time relative to the PyTorch baseline. Specifically, KernelBench proposes fast_p , a metric ensuring that the code is both functionally correct and causes a performance speed-up of at least p -times (Eq. 1). Mathematically, as defined in KernelBench,

$$\text{fast}_p = \frac{1}{N} \sum_{i=1}^N \mathbb{1}(\text{correct}_i \wedge \{\text{speedup} > p\}) \quad (1)$$

where $p = 0$ indicates the correctness rate.

5.3 Experimental details

Our RAG pipeline used LangChain’s `MarkdownHeaderTextSplitter` and `RecursiveCharacterTextSplitter` to preserve context across headers and allow for chunking across specific separators such as code

blocks and newlines. Each chunk is embedded using BAAI’s bge-large-en-v1.5 [12] into 1024-dimensional embedding vectors. We used ChromaDB to store text chunks and their corresponding metadata for efficient retrieval. At inference time, we retrieve the top 10 relevant chunks by cosine similarity and prepend them to the prompt.

The synthetic data was generated using an automated generation loop. For each KernelBench task, we prompted the language model to produce a HIP implementation given the PyTorch reference code. The generated kernel was compiled using the HIP toolchain and executed against randomly generated test inputs. If the kernel successfully compiled and produced outputs matching the reference implementation within tolerance, the example was stored as a valid training sample.

This filtering step ensured that the training dataset only contained kernels that were syntactically valid and functionally correct. In practice, many generations failed to compile due to syntax or API errors, which further highlights the difficulty of the task.

We fine-tuned using SFT with gradient accumulation steps of 16, learning rate of $2e-4$, and trained for one epoch using the Qwen2.5-Coder-7B-Instruct [13] model.

5.4 Results

Compilation and correctness Figure 2A and Figure 2B show that incorporating RAG noticeably improves compilation and correctness across all thresholds, respectively. The improvements were stronger than expected for syntactic correctness as we did not expect the retrieved context, gathered from documentation and examples to be enough to teach HIP code correctness. Figure 2 demonstrates that RAG+SFT performed closely to just RAG on level 1 and level 3 tasks, but shows noticeable improvement on level 2 tasks. Level 2 tasks typically involve fused operations and require reasoning across multiple operators, so the improvement suggests that RAG+SFT helps the model learn common fusion patterns for performance gains.

Figure 2B shows that RAG+SFT had the highest correctness rate across the experiments for all three levels. This shows that SFT allowed the model to learn correctness patterns that RAG alone was not able to provide by training the model on kernels that had already passed compilation and correctness checks. Overall, the RAG and RAG+SFT experiments largely improved the compilation rate and correctness rate in comparison to the baseline.

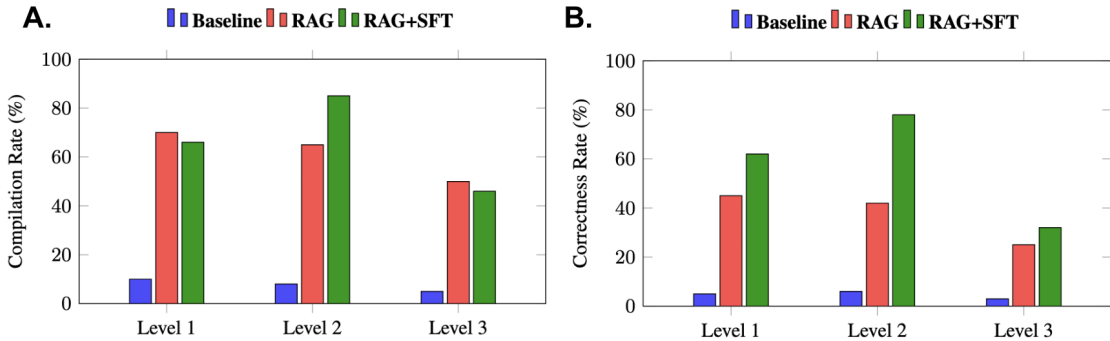


Figure 2: Metrics across difficulty levels for A. compilation rates and B. correctness rates.

Performance Figure 3 evaluates performance using the fast_p metric across increasing speedup thresholds. Across all three levels, Figure 4 shows that the fast_p score sharply declines as the speedup threshold p increases across all three levels, indicating that generating high-performance kernels remains challenging even when compilation and correctness improve. The level 3 fast_p performance shows that all three methods result in very low scores. Since level 3 kernels typically require several optimization strategies in combination like tiling and memory reuse to achieve high performance, performance optimization for these problems requires a deeper understanding of GPU hardware and memory hierarchies that solely documentation and fine-tuning have difficulty providing.

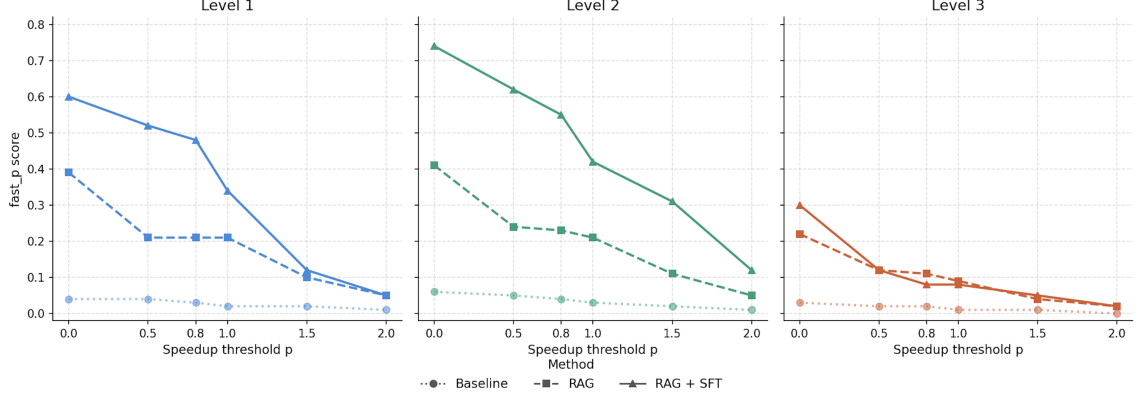


Figure 3: fast_p comparison across different difficulty levels.

6 Analysis

Across all levels, the baseline model frequently produced kernels that looked structurally correct but failed to compile or execute because of deeper misunderstandings about HIP kernels. The most common errors included incompatible function arguments, tensor shape mismatches, and tensors being used in device code while still residing in host memory. These failures suggest that the baseline model lacked a clear understanding of memory management and kernel engineering.

As expected, when we introduced retrieval over HIP/ROCm documentation, compilation rates improved significantly across all levels by providing the model with explicit examples of correct API usage, kernel launch patterns, and a few kernel examples. Interestingly, however, when the retrieved context contained high-quality HIP kernel examples, regardless of the problem level, the model often attempted very aggressive optimization strategies. In some cases, such as GEMM + Sigmoid + LogSumExp, this led to increased execution failures as the model attempted optimizations such as fusion and parallel data access patterns without fully understanding the constraints of the hardware or data layout.

Examining multi-attempt generation traces shows that the model can learn from execution feedback across iterations in synthetic data generation. Although the Gemini-generated synthetic data was not the focus of our study, it provided useful insight into how kernels evolved across attempts. In several kernels, the model gradually moved from separate operations to fused kernels, such as ConvTranspose2d + Add + Clamp + Scale and ConvTranspose3d + ResidualAdd + Multiply, which achieved speedups of roughly 1.5x by increasing data re-use.

The strongest SFT+RAG kernels emerged when the retrieved RAG context included HIP documentation on shared-memory usage or cache-friendly data access patterns, and the SFT model trained on iteratively refined examples. In these cases, the kernels utilized parallelization strategies like block-level reductions and indexing patterns that reuse nearby data. This suggests that combining retrieval-based knowledge with exposure to iterative execution feedback helps the model better apply classic optimization techniques.

However, even with these improvements, performance optimization remained difficult. Many generated kernels that compiled and produced correct outputs still failed to achieve meaningful speedups. Thus, while the model can learn syntactical patterns for writing valid kernels and performant kernels, achieving significant speedup over the PyTorch Eager baseline requires the model to have a deeper understanding of GPU architecture and parallel execution than is captured in the training data.

7 Conclusion

In this research, we investigated whether retrieval and SFT can improve an LLM’s ability to generate correct and performant AMD HIP kernels for KernelBench tasks. We found that our baseline, one-

shot approach frequently produced syntactically plausible kernels that failed to compile. Introducing retrieval over HIP/ROCm documentation significantly improved compilation rates, while SFT further improved the model’s ability to produce correct and optimize kernels. As part of this work, we constructed a synthetic dataset of 118 high-quality kernel examples containing the KernelBench task description, reference PyTorch architectures, generated HIP kernels, compilation and correctness results, speedup measurements, runtime metrics, and full prompt histories across iterative attempts. The main limitation of our work is that, due to computational constraints, we used a relatively small open-source model and trained it on a limited synthetic dataset consisting of 79 kernels in the train split. The synthetic dataset was also imbalanced: 53 of the 79 kernels were from level 2 problems, leading to an overrepresentation of fused kernels, limiting the SFT model’s ability to generalize across the full range of KernelBench tasks. Future work could explore RL approaches including PPO or multi-turn RL, train on a larger and more diverse synthetic dataset, or explore the kernel generation task on other hardware accelerators beyond GPUs.

Team contributions

Laasya: Building support for running HIP on KernelBench, baseline and SFT experiments, final report writing. Natalia: RAG and SFT experiments and paper writing. Annmaria: Synthetic data generation and paper writing.

References

- [1] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.
- [2] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- [3] Ali Tehrani, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *Advances in Neural Information Processing Systems*, 37:100965–100999, 2024.
- [4] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill. Hardware/software tradeoffs for increased performance. *ACM SIGPLAN Notices*, 17(4):2–11, 1982.
- [5] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.
- [6] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.
- [7] Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, et al. Towards automated kernel generation in the era of llms. *arXiv preprint arXiv:2601.15727*, 2026.
- [8] AMD. Hip: C++ heterogeneous-compute interface for portability. <https://github.com/ROCm/HIP>.
- [9] AMD. Amd rocm software. <https://github.com/ROCm/ROCm>.
- [10] AMD. Rocm examples. <https://github.com/ROCm/rocm-examples>.
- [11] AMD. Hip documentation. <https://rocm.docs.amd.com/projects/HIP/en/latest/>.
- [12] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pages 641–649, 2024.

- [13] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

A Appendix

A.1 A.1 Generated Synthetic Data

Our synthetic HIP kernels that were generated from PyTorch reference implementations are available at the following link: <https://github.com/AnnmariaKAntony/hip-kernel-bench/tree/main>.