

---

# GEAK: INTRODUCING TRITON KERNEL AI AGENT & EVALUATION BENCHMARKS

---

Jianghui Wang\*   Vinay Joshi\*   Saptarshi Majumder   Xu Chao   Bin Ding   Ziqiong Liu  
 Pratik Prabhanjan Brahma   Dong Li   Zicheng Liu   Emad Barsoum

*Advanced Micro Devices, Inc. (AMD)*

## ABSTRACT

The demand for AI-generated GPU kernels is rapidly growing, influenced by the need for scalable, hardware-optimized solutions in both industry and academia. As deep learning workloads grow in complexity and diversity, it is imperative to automate low-level kernel development to meet performance and productivity demands. Major cloud providers, semiconductor companies, and research institutions are now investing heavily in AI-driven code generation for GPUs, aiming to reduce manual optimization efforts while achieving near-expert performance on hardware like AMD Instinct™ MI300X. The Triton language, a Python-based DSL for GPU programming, has emerged as a popular target for such AI-generated kernels due to its balance of performance and ease-of-coding. In this work, we present an evaluation suite for Triton-based GPU kernels and GEAK (Generating Efficient AI-centric GPU Kernels)—a framework that leverages cutting-edge LLMs to generate performant Triton code specifically for AMD GPUs, including the AMD Instinct™ MI300X and MI250. GEAK leverages inference-time compute scaling to produce Triton-based GPU kernels using a reasoning loop adapted from Reflexion-style feedback mechanisms. On two evaluation benchmarks, GEAK significantly outperformed the baselines of directly prompting frontier LLMs as well as Reflexion-based generation pipelines by achieving correctness up to 63% and execution speed up of up to 2.59X. These results highlight the promise of GEAK-like agentic code generation for accelerating the adoption of diverse hardware platforms and democratizing access to expert-level kernel performance.

**Keywords** Triton language · GPU kernel generation · Automated Code Generation · Inference time scaling

## 1 Introduction

As AI workloads scale in both complexity and hardware diversity, there is a growing demand for intelligent systems that can generate high-performance GPU kernels without manual tuning. This demand is especially critical in environments where tight coupling between software and hardware efficiency determines practical feasibility—ranging from hyperscaler data centers used for serving frontier LLMs to academic HPC clusters. Recent advances in large language models (LLMs) have shown promising capabilities in code generation tasks, but achieving correct and efficient GPU code remains an open challenge. AI-assisted GPU kernel development, particularly for emerging hardware like AMD Instinct™ MI300X GPUs, can substantially improve productivity and deployment speed.

To this end, we introduce GEAK (Generating Efficient AI-centric GPU Kernels), AMD’s new agentic framework for automatic Triton kernel generation targeting AMD Instinct™ GPUs. GEAK is built to push the frontier of AI-assisted code generation by combining state-of-the-art LLMs with a structured reasoning and feedback loop. Alongside GEAK,

---

\*Primary authors

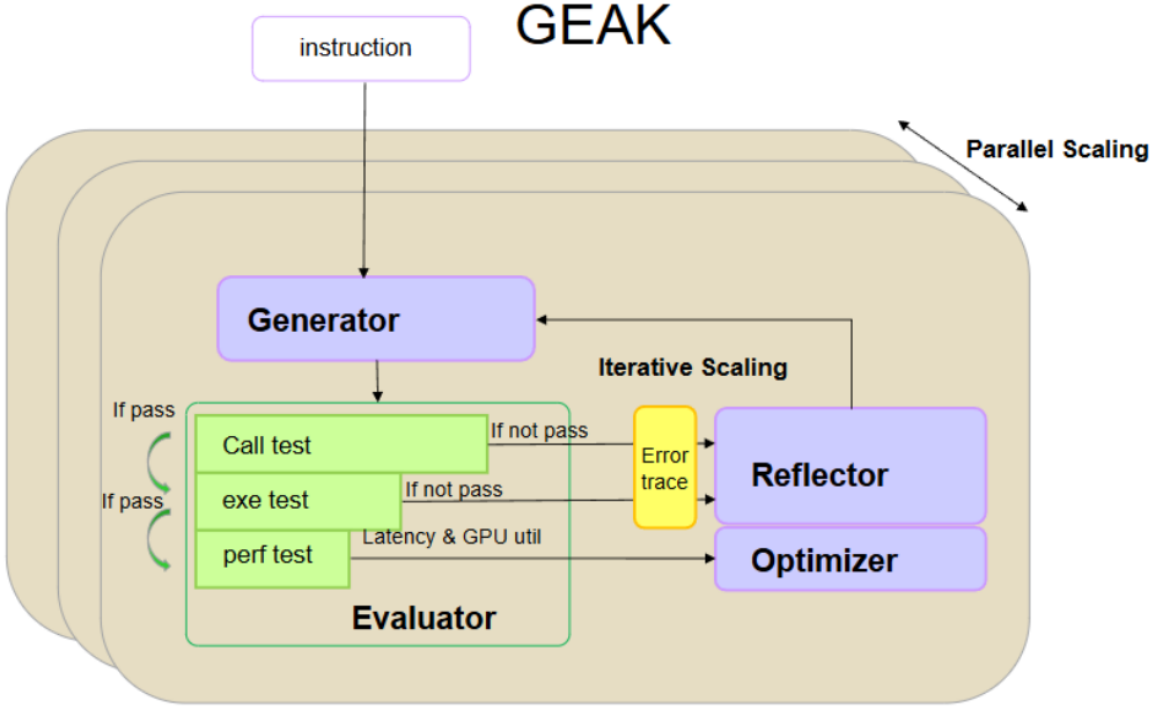


Figure 1: Illustration of GEAK pipeline.

we release two benchmark suites for evaluating AI-generated Triton kernels, enabling measurement of both execution correctness and runtime performance of generated kernels.

#### Key Contributions:

1. We introduce **GEAK framework**, a modular, agent-based system that uses inference-time compute scaling with frontier LLMs to generate Triton GPU kernels from minimal task descriptions. GEAK incorporates multiple agents—generation, evaluation, reflection, and optimization—to iteratively refine code quality and execution performance.
2. We establish two benchmark suites for evaluating Triton kernel generation. The **TritonBench-revised** is a revised subset of 184 kernels adapted from TritonBench-G[1] with stricter testing harnesses. In addition, we release the **ROCm Triton Benchmark**, a new suite of 30 real-world kernels sampled from open-source AMD ROCm repositories.
3. We demonstrate that GEAK significantly outperforms direct prompting of state-of-the-art LLMs, achieving correct kernel generation rate up to 54.89% on TritonBench-revised and 63.33% on the ROCm Triton benchmark—compared to less than 15% when directly prompting strong LLMs without agentic feedback.
4. We also present a detailed study on one of the performant kernels from ROCm Triton Benchmark (please see Appendix A).
5. We open source our GEAK agent implementation ([github link](#)) and GEAK evaluation framework ([github link](#)).

Furthermore, GEAK-generated kernels demonstrate an average speedup of up to 2.59× over their reference counterparts on the TritonBench-revised set.

## 2 Related Work

**Benchmarks** As program synthesis research advances, the need for rigorous and multidimensional benchmarks has become increasingly important. Most existing coding benchmarks concentrate on functional correctness, often assessed through manually crafted test cases and isolated execution environments. For instance, HumanEval provides expert-written programming tasks paired with curated test suites, while MBPP[2] collects problems through crowdsourcing.

To improve coverage and scalability, more recent approaches have turned to automated test generation[3], enabling broader application in areas such as software engineering and developer assistance[4].

In addition to correctness, performance profiling is gaining traction as a crucial axis in benchmarking ([5];[3];[6];[7]). However, most existing frameworks emphasize competition-style tasks or sequential execution and provide limited support for parallel or hardware-specific evaluation. While a few benchmarks address parallel CPU programming ([8];[9]), GPU-targeted benchmarks remain scarce, despite the growing importance of hardware efficiency in deploying large-scale deep learning models.

To address this gap, benchmarks like KernelBench [10] and TritonBench [1] have emerged, offering domain-specific assessments of LLM performance on GPU kernel generation. KernelBench [10] focuses on correctness and runtime efficiency across diverse workloads using metrics like "fast\_p", while TritonBench [1] highlights the challenges LLMs face in generating performant code for the Triton DSL. These benchmarks reveal a substantial gap between general-purpose code generation and the demands of high-performance, hardware-aware synthesis, underscoring the need for specialized, compute-oriented evaluation frameworks.

**LLMs for Code Generation** More recent work has explored the use of large language models for code generation, but often in general-purpose contexts with limited success on performance-critical low level code, such as DeepSeek-Coder[11] and Qwen-Coder[12], which have achieved strong performance on broad coding benchmarks.

There are also LLMs specifically trained for the kernel generation task, such as KernelLLM. However, its capability is limited to translating PyTorch code into Triton kernels, restricting its general applicability. In contrast, our agent can generate Triton code based on both natural language instructions and reference code, allowing for more flexible and context-aware code synthesis. Furthermore, specialized models like KernelLLM often lack the broader reasoning and problem-solving capabilities exhibited by larger state-of-the-art LLMs such as GPT-4.1 and Claude 4, which are critical for tackling complex optimization tasks involving multiple constraints and trade-offs.

**Inference-Time Compute Scaling** Inference-time compute scaling refers to the strategy of allocating increased computational resources—such as longer context windows, multiple reasoning passes, or parallel generations—during the inference phase of machine learning models to boost performance on complex tasks. Notably, this approach enables significant improvements in large language model (LLM) performance without requiring additional training. For example, the Chain-of-Thought [13] prompting technique enhances reasoning by encouraging models to answer questions in a step-by-step manner. Similarly, the Reflexion [13] framework improves agent performance by integrating verbalized feedback into an episodic memory buffer, which guides more informed decision-making in future attempts. Our proposed GEAK agent also leverages inference-time compute scaling to improve code generation and optimization quality. In addition, we incorporate a Reflexion-style module for debugging, enabling the agent to iteratively refine its outputs based on past execution feedback.

### 3 Benchmarks

We experiment with Triton kernel evaluation benchmarks, consisting of a modified set of 184 kernels originally sourced from TritonBench-G and an additional set of 30 kernels from various open-sourced ROCm repositories.

#### 3.1 TritonBench-revised Benchmark

The Triton kernels and corresponding evaluation codebase were borrowed and adapted from TritonBench but the following enhancements were made:

37 out of the 184 kernels in TritonBench-G [1] had errors while running on AMD GPUs (such as Shared memory errors, invalid HIP arguments, ModuleNotFound). We fixed these kernel to be AMD GPU compliant.

For several kernels, the original TritonBench repository had missing call functions for which execution accuracy script was effectively comparing empty strings as output. This was corrected by introducing the following:

- Adding call to the test functions in the kernels missing it.
- Developing tolerance based tensor comparison of output tensors produced by test functions instead of STDOUT string comparisons.
- Ensuring consistent seed for random tensor generations used in producing inputs in unit tests.

For some kernels, unit tests were written inconsistently that produced unexpected outputs. We have fixed these kernels to have consistent unit tests.

Table 1: List of kernels and their sources in the RoCM repositories Triton benchmark

| ROCm Kernel  | Repository link        |
|--|------------------------|
| 1. test_tma_store_gemm<br>2. moe-gemm<br>3. layernorm<br>4. triton_multreduce_matmul_kernel.py<br>5. multreduce_matmul_dot_kernel.py<br>6. gemm<br>7. test_block_pointer_matmul<br>8. test_block_copy<br>9. test_add_kernel<br>10. softmax | ROCm/triton            |
| 1. rmsnorm_fwd<br>2. rmsnorm_bwd<br>3. test_chained_dot_fp8<br>4. test_matmul_MXFP<br>5. test_gemm_fusion  | ROCm/aiter             |
| 1. test_triton_swizzle2d<br>2. test_triton_sort<br>3. test_triton_flip<br>4. test_reverse_range  | ROCm/aotriton          |
| 1. test_load_reduce.py<br>2. test_chained_matmul   | ROCm/vllm              |
| 1. test_random_int<br>2. test_randn<br>3. test_kernel_sub<br>4. test_kernel_dot<br>5. test_batched_vecmat  | ROCm/pytorch           |
| 1.test_flashattention_fwd<br>2.test_iv_dependent_matmul  | ROCm/xformers          |
| 1.test_gemm_no_scf   | ROCm/bitsandbytes      |
| 1.test_cast_matmul   | ROCm/TransformerEngine |

### 3.2 ROCm Triton Benchmark

The following kernels have been written by Triton expert engineers and released publicly to help enable the ecosystem around running AI workloads efficiently on AMD GPUs. We also took assistance from frontier LLMs to refactor these, as well as their corresponding unit tests, in the same format as the TritonBench-revised benchmark. This enables us to compare the accuracy and efficiency of our agent in a consistent manner.

### 3.3 Evaluation Metrics

We use the same metrics as reported in TritonBench [1]:

- **Call Accuracy:** Fraction of AI-generated kernels that can compile and run without any errors.
- **Execution Accuracy:** Percentage of AI-generated kernels that satisfy all unit tests.  
Execution accuracy is computed for kernels that successfully compile.
- **Speedup:** Relative execution time improvement of AI-generated kernels over reference ground truth kernels. We define it as: expectation of the ratio of median reference kernel latency and generated kernel latency over all unit tests.  
Speedup is only computed for kernels that satisfy all unit tests.

## 4 GEAK

### 4.1 Pipeline

As shown in Figure 1, the agentic AI system comprises four core modules: 1) Generator, 2) Reflector, 3) Evaluator, and 4) Optimizer. The Generator produces code based on user query and contextual information. The Evaluator follows a cascaded design: it first performs a functionality test to verify correctness. If the code fails this test, the corresponding error trace is fed back to the Reflector for further analysis. If the code passes, the Evaluator proceeds to assess its performance, including latency and memory efficiency. If the generated code fails to execute correctly, the Reflector analyzes both the code and the resulting error trace to identify potential issues. Finally, the Optimizer takes as input the functionally correct code and formulates strategies to enhance its performance with respect to latency and computational efficiency. To improve the performance of our agent, we employ some composable and configurable techniques as mentioned in below.

### 4.2 Modules

**1-shot prompting** To enable 1-shot prompting, the most similar Triton code from existing datasets is used in the prompt. The datasets used for 1-shot prompt do not overlap with the benchmark. We observed that 1-shot sample retrieval from datasets is most effective with code similarity rather than instruction similarity.

**Knowledge Injection** We enhance the prompt with domain-specific knowledge on writing efficient Triton kernels, including detailed hardware specifications. This incorporation of low-level optimization principles significantly improves the accuracy and quality of LLM-generated code.

**Reflexion** To enhance the system’s self-correcting capabilities, we leverage a Reflexion[13] module that supports introspective debugging and iterative refinement. When an LLM-generated kernel does not pass the functionality test, the resulting error trace is provided as feedback to the reflector for further analysis and correction. The agent is tasked with analyzing the cause of failure and proposing a corresponding solution. We scale the number of iterations to improve kernel generation.

**LLM selection for kernel agent** We have explored the use of multiple LLMs like GPT-4.1, O1 and Gemini 2.5 Pro. We observed significant variation in the outputs produced by different models, indicating that the capability of the underlying LLM can substantially influence the results.

**LLM as Optimizer** The Optimizer LLM is tasked with identifying potential optimization directions based on previous code generations and their corresponding performance metrics. These historical records, which include the generated code and associated performance results, are sorted in ascending order of performance. This structured presentation helps guide the LLM toward proposing more effective optimization strategies. [14]

**Debugging trap** When LLM’s generated code has bugs, the error trace is provided to the Reflector for correction. However, we’ve observed that sometimes code can undergo several reflection cycles while still being plagued by the same bug, which is what we refer to as the debugging trap. To prevent the agent from getting stuck in a debugging trap, we impose a limit on the number of debugging attempts per code snippet using a parameter `max_perf_debug_num`. If the code continues to fail after reaching this threshold, the agent is required to discard the current approach and generate a new strategy along with fresh code.

**Parallel Scaling** To ensure reliable kernel generation, we run multiple instances of GEAK multiple times in parallel and independently. To introduce diversity in the generated code, the LLM output is sampled with the parameter temperature set to 1. Our experiments show that such diversity in generation can yield correct and faster kernels otherwise underexplored. Moreover, further experiments indicate that combining sequential and parallel scaling yields additional performance improvements. To obtain unbiased estimates of accuracy, we estimate it using `pass@k` metric.

## 5 Experiments

### 5.1 Main Results

We first establish our baseline by directly prompting frontier LLMs, such as GPT-4.1, Gemini 2.5 Pro, and Claude 3.7 Sonnet, to produce the kernels. As shown in Table 2, across all models, direct prompting yields low execution

Table 2: Baseline Results

| Model                                | Call Accuracy (%) | Exec Accuracy % | Speedup     |
|--------------------------------------|-------------------|-----------------|-------------|
| TritonBench-modified benchmark       |                   |                 |             |
| GPT4.1                               | 14.67 / 19.02     | 8.70 / 14.13    | 0.52 / 0.53 |
| GPT4o                                | 10.87 / 14.13     | 7.07 / 9.24     | 0.51 / 0.53 |
| Gemini2.5pro                         | 20.65 / 21.74     | 14.13 / 16.85   | 1.33 / 0.96 |
| Claude 3.7                           | 11.41 / 20.11     | 7.07 / 15.22    | 0.61 / 0.96 |
| ROCm benchmark (0-shot results only) |                   |                 |             |
| GPT 4.1                              | 0                 | 0               | 0           |
| Gemini-2.5 Pro                       | 40                | 16.66           | 0.91        |

Note: numbers to the left of ‘/’ indicate 0-shot results and to the right are 1-shot prompting results.

Table 3: GEAK on TritonBench-Modified Benchmark

| Difficulty level | Correctly generated kernels | Total kernels in dataset | Exec Accuracy (%) | Average Speedup |
|------------------|-----------------------------|--------------------------|-------------------|-----------------|
| 1                | 2 / 3                       | 3                        | 66.67 / 100.0     | 1.24 / 1.16     |
| 2                | 23 / 22                     | 27                       | 85.19 / 81.48     | 4.78 / 1.69     |
| 3                | 39 / 41                     | 65                       | 60.00 / 63.08     | 1.57 / 3.02     |
| 4                | 32 / 34                     | 84                       | 38.10 / 40.48     | 2.24 / 2.86     |
| 5                | 1 / 1                       | 5                        | 20.00 / 20.00     | 0.14 / 0.61     |
| overall          | 97 / 101                    | 184                      | 52.72 / 54.89     | 2.42 / 2.59     |

Note: numbers to the left of ‘/’ indicate results on MI250 and to the right are results on MI300.

accuracy and suboptimal performance, indicating that even state-of-the-art LLMs struggle to produce correct and efficient low-level code without further guidance.

Introducing one-shot prompting—where a single example retrieved from TritonBench trainset is provided—consistently improves both call accuracy and execution accuracy compared to zero-shot prompting. In most cases, one-shot prompting also leads to higher speedup, suggesting that the additional context helps LLMs generate more efficient kernels. However, a notable exception is observed with Gemini 2.5 Pro, where one-shot prompting results in lower speedup than direct prompting, despite improvements in accuracy.

Table2 shows the direct prompting results on the ROCm benchmark. Direct prompting with GPT-4.1 fails to generate any valid kernels, underscoring the challenge of targeting non-CUDA platforms. In contrast, Gemini 2.5 Pro achieves better results than GPT-4.1 on both the TritonBench-revised and ROCm benchmarks, though the overall performance remains limited.

As shown in Table3 and Table4, compared to direct LLM prompting, our system GEAK delivers significantly stronger results. On MI300, GEAK achieves 54.89% execution accuracy and a 2.59× speedup on TritonBench-revised Benchmark, and 63.33% execution accuracy and 0.92x speedup on ROCm Benchmark, demonstrating the effectiveness of our agent framework. We also present a case study in Appendix A on *triton\_test\_flip.py* kernel analyzing potential strategies for performance gain.

Table 4: GEAK on ROCm Benchmark

| Difficulty level | Correctly generated kernels | Total kernels in dataset | Exec Accuracy (%) | Average Speedup |
|------------------|-----------------------------|--------------------------|-------------------|-----------------|
| overall          | 19                          | 30                       | 63.33             | 0.92            |

Table 5: Sequential Scaling of GEAK on TritonBench-Modified Benchmark on MI250

| #iterations | call acc(%)  | exec acc(%)  | speedup     |
|-------------|--------------|--------------|-------------|
| iter0       | 21.2         | 13.04        | 1.02        |
| iter1       | 33.15        | 26.63        | <b>1.89</b> |
| iter2       | 41.85        | 30.98        | 1.69        |
| iter3       | 47.28        | 34.24        | 1.39        |
| iter4       | 50.54        | 37.5         | 1.39        |
| iter5       | 57.61        | 37.5         | 1.63        |
| iter6       | 50.04        | 37.5         | 1.31        |
| iter7       | 54.39        | 39.13        | 1.48        |
| iter8       | 62.5         | 40.22        | 1.67        |
| iter9       | 56.52        | 40.76        | 1.45        |
| iter10      | 62.5         | 40.22        | 1.75        |
| iter11      | 55.43        | 40.76        | 1.45        |
| iter12      | 57.07        | 42.39        | 1.16        |
| iter13      | 59.78        | 41.85        | 1.75        |
| iter14      | 61.41        | 40.76        | 1.76        |
| iter15      | 57.07        | 43.48        | 1.65        |
| iter16      | 60.33        | 42.93        | 1.57        |
| iter17      | 60.87        | 43.48        | 1.85        |
| iter18      | 61.96        | 43.48        | 1.74        |
| iter19      | <b>63.04</b> | <b>44.02</b> | 1.55        |

## 5.2 Ablation Experiments

### 5.2.1 Sequential Scaling

We investigate the impact of increasing inference-time compute on accuracy and performance, focusing on the sequential scaling dimension.

The experimental results in Table 5 indicate that increasing the number of iterations leads to a steady improvement in both the accuracy and performance of GEAK. As the number of iterations increases, we observe a monotonic improvement in both call and execution accuracy, culminating at iter19 with 63.04% call accuracy and 44.02% execution accuracy—more than 3× higher than the baseline.

In terms of performance, speedup improves significantly in the early iterations, with iter1 already reaching 1.89×, the highest across all settings. This sharp gain is likely due to early corrections of suboptimal code generations. Although the speedup fluctuates slightly in later iterations, it consistently stays above 1.4× from iter7 onwards, suggesting that continued refinement contributes to runtime efficiency as well.

Notably, iteration count beyond 10 still yields meaningful gains: call accuracy increases from 56.52% (iter9) to 63.04% (iter19), and execution accuracy from 40.76% to 44.02%. These results validate the effectiveness of iterative refinement in improving both correctness and efficiency.

Overall, the analysis confirms that sequential compute budget is a powerful scaling axis for GEAK, enabling it to progressively refine outputs with improved correctness and performance.

### 5.2.2 Parallel Scaling

We demonstrate how increasing inference-time compute through parallel scaling improves both accuracy and performance. Specifically, we fix the number of sequential iterations to 10 and explore how varying the number of parallel runs during inference affects the results. Experiments are conducted on both MI250 and MI300 GPUs to ensure the findings generalize across hardware generations.

Figure 2 and Figure 3 show inference time scaling laws on ROCm and TritonBench-revised benchmarks, respectively. Both call and execution accuracy scale almost log-linearly with respect to the number of parallel runs,

These findings underscore the complementary role of parallel scaling alongside sequential refinement as two orthogonal axes to boost inference quality. Together, they provide a flexible mechanism for tuning the accuracy-performance tradeoff under a given compute budget.

Table 6: TritonBench-Modified Benchmark: GEAK Sequential@10 + Parallel@10 on MI250 - GPT4.1

| Pass@K | call acc(%) | exec acc(%) |
|--------|-------------|-------------|
| 1      | 56.52       | 40.76       |
| 2      | 66.30       | 45.65       |
| 3      | 72.83       | 48.91       |
| 4      | 78.80       | 50.0        |
| 5      | 81.52       | 50.0        |
| 6      | 82.61       | 49.46       |
| 7      | 85.87       | 52.17       |
| 8      | 86.41       | 53.26       |
| 9      | 86.41       | 53.26       |

Table 7: TritonBench-Modified Benchmark: GEAK Sequential@10 + Parallel@10 on MI300 - GPT4.1

| Pass@K | call acc(%) | exec acc(%) |
|--------|-------------|-------------|
| 1      | 54.78       | 35.38       |
| 2      | 71.31       | 43.95       |
| 3      | 78.85       | 47.53       |
| 4      | 83.13       | 49.55       |
| 5      | 85.92       | 50.92       |
| 6      | 87.87       | 51.96       |
| 7      | 89.30       | 52.81       |
| 8      | 90.37       | 53.55       |
| 9      | 91.20       | 54.24       |
| 10     | 91.85       | 54.89       |

The parallel scaling analysis reveals several critical insights for the GEAK framework’s design and implementation strategy. First, parallel scaling provides complementary benefits to sequential refinement, enabling the framework to achieve higher accuracy without proportionally increasing latency. This orthogonal improvement axis allows GEAK to optimize both accuracy and efficiency simultaneously.

The consistent scaling patterns across different hardware platforms demonstrate that GEAK’s parallel scaling mechanisms are robust and portable. The framework can leverage available computational resources effectively regardless of the underlying GPU architecture, making it suitable for diverse deployment environments.

The substantial improvements in Call Accuracy relative to Execution Accuracy indicate that parallel scaling particularly benefits high-level reasoning and decision-making processes. This suggests that GEAK’s parallel mechanisms are especially valuable for complex inference tasks that require sophisticated logical reasoning rather than simple execution steps.

Finally, the scaling characteristics support GEAK’s flexible architecture design, where parallel compute resources can be dynamically allocated based on task complexity and accuracy requirements. This adaptability ensures that the framework can optimize performance across different use cases while maintaining computational efficiency.

### 5.2.3 Effect of different Modules

To assess the contribution of individual components within our agent architecture, we conducted ablation experiments by evaluating performance across various combinations of modules. Results in Table9 show that each module—knowledge injection, one-shot, and Optimizer—contributes positively to both execution accuracy and speedup, highlighting the complementary roles these components play in enhancing the agent’s overall performance.

Specifically, knowledge injection alone provides a substantial lift over the baseline, improving call accuracy from 14.67% to 52.72% and execution accuracy from 8.70% to 20.11%. This underscores the critical role of prior knowledge in enabling correct kernel generation.

Adding one-shot prompting further improves accuracy, pushing call accuracy to 54.35% and execution accuracy to 27.17%. However, the speedup remains under 1.0 (i.e., 0.99), suggesting that while one-shot helps with correctness, it has limited impact on runtime efficiency on its own.



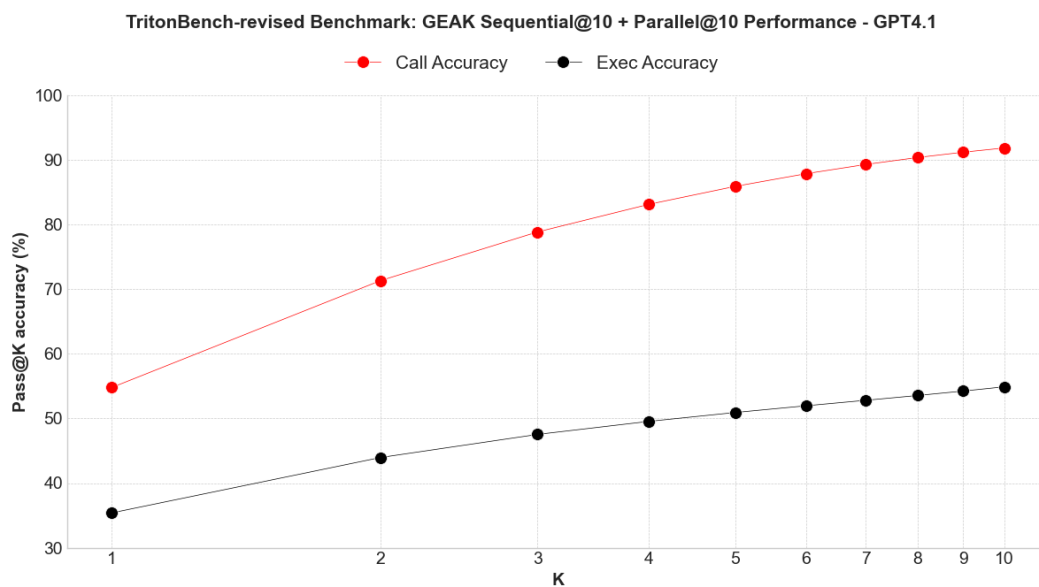


Figure 2: Inference time compute scaling study on TritonBench-revised benchmark.

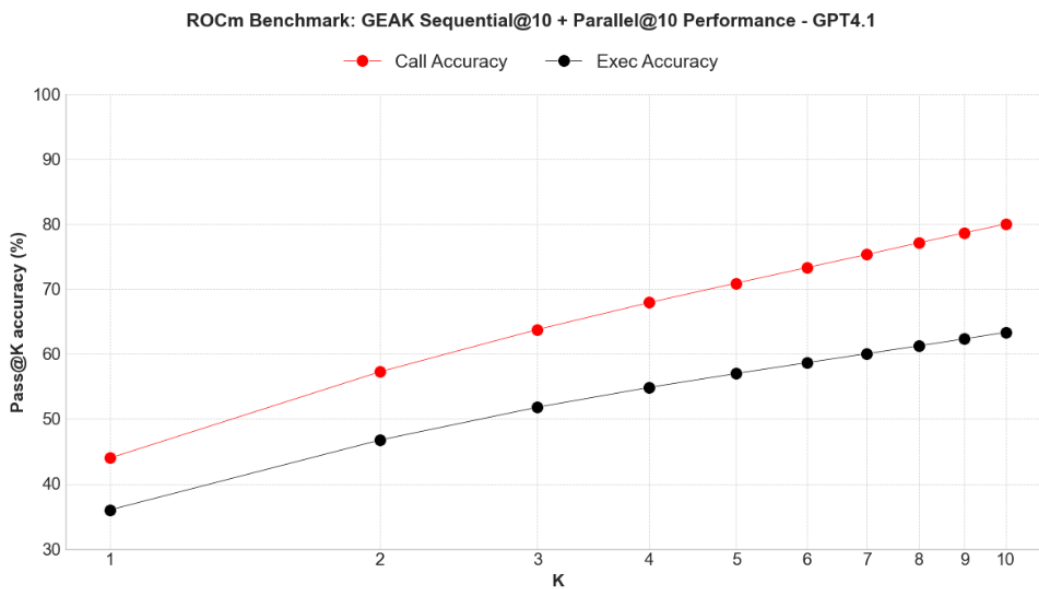


Figure 3: Inference time compute scaling study on ROCm benchmark.

Table 8: ROCm Benchmark: GEAK Sequential@10 + Parallel@10 on MI300 - GPT4.1

| Pass@K | call acc(%) | exec acc(%) |
|--------|-------------|-------------|
| 1      | 43.99       | 36.00       |
| 2      | 57.25       | 46.74       |
| 3      | 63.77       | 51.80       |
| 4      | 67.92       | 54.85       |
| 5      | 70.91       | 56.99       |
| 6      | 73.30       | 58.66       |
| 7      | 75.33       | 60.05       |
| 8      | 77.11       | 61.25       |
| 9      | 78.66       | 62.33       |
| 10     | 80.00       | 63.33       |

Table 9: Modules

| Knowledge Injection | 1-shot | Optimizer | call acc(%) | exec acc(%) | speedup |
|---------------------|--------|-----------|-------------|-------------|---------|
|                     |        |           | 14.67       | 8.70        | 0.52    |
| ✓                   |        |           | 52.72       | 20.11       | 0.86    |
| ✓                   | ✓      |           | 54.35       | 27.17       | 0.99    |
| ✓                   | ✓      | ✓         | 56.52       | 40.76       | 1.45    |

The introduction of the Optimizer module shows the most significant impact on speedup. When combined with the other two modules, it boosts execution accuracy to 40.76% and yields a notable speedup of 1.45x, the highest among all configurations. This indicates that the Optimizer not only improves the likelihood of generating correct kernels but also helps produce more efficient code in terms of latency and performance.

## 6 Conclusion

In this work, we presented GEAK, a modular and agent-based framework that leverages inference-time compute scaling with frontier LLMs to automatically generate efficient Triton GPU kernels from text task descriptions. By coordinating multiple specialized agents, GEAK iteratively improves both the correctness and performance of generated code without requiring additional training.

To rigorously assess kernel generation capabilities, we introduced two benchmarks: TritonBench-revised benchmark, a curated subset of TritonBench-G with improved test harnesses, and the newly constructed ROCm benchmark, which consists of real-world kernels from open-source AMD repositories. These benchmarks provide a diverse and challenging testbed for evaluating both functional correctness and runtime efficiency.

By fully open-sourcing the benchmarks and code for running the agent, we expect to engage the open-source community to accelerate the development of GPU kernels. We invite developers, researchers, and AI enthusiasts to explore the agent and benchmarks, and we hope this will foster innovation and collaboration within the AI community to come up with even better methods and final output kernels that can significantly improve the efficiency of training and inference for large-scale AI models.

## Acknowledgments

We would like to acknowledge the following folks for constructive discussions and feedback during the project - Alan Lee, Peng Sun, Vinayak Gokhale, Jason Furmanek, Sharunas Kalade, Graham Schelle, Sampsa Rikonen, Doug Lehr, Zhaoyi Li, Yonatan Dukler, Vikram Appia, Arseny Moskvichev, Stephen Youn and Steve Reinhart.

## References

- [1] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025.

- [2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.
- [3] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.
- [4] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.
- [5] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2024.
- [6] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. Effibench: Benchmarking the efficiency of automatically generated code, 2025.
- [7] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark, 2025.
- [8] Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 281–294. ACM, June 2024.
- [9] Aman Chaturvedi, Daniel Nichols, Siddharth Singh, and Abhinav Bhatele. Hpc-coder-v2: Studying code llms across low-resource parallel languages, 2024.
- [10] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.
- [11] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.
- [12] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shangaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024.
- [13] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.
- [14] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024.

## A Case study: Flip kernel

We studied kernel `test_triton_flip.py`, from ROCm Triton Benchmark, which achieved 2.26x speedup over Triton expert written kernel. The following was the triton kernel generated by GEAK:

```

1 @triton.jit
2 def flip_kernel(X, Z, N: tl.constexpr, M: tl.constexpr):
3     """
4     Processes 2D blocks of data, flipping each block horizontally.
5     Loads a (N, M) block from X, flips each row horizontally, and stores into Z.
6     Parameters:
7     X: pointer to flattened input tensor
8     Z: pointer to flattened output tensor
9     N: number of rows in the block
10    M: number of columns in the block
11    """
12    offs_row = tl.arange(0, N)
13    offs_col = tl.arange(0, M)
14    rows = offs_row
15    cols = offs_col
16    mask_row = rows < N
17    mask_col = cols < M
18    mask = mask_row[:, None] & mask_col[None, :]
19    flipped_cols = M - 1 - cols

```

```

20 x_ptrs = X + rows[:, None] * M + flipped_cols[None, :]
21 z_ptrs = Z + rows[:, None] * M + cols[None, :]
22 vals = tl.load(x_ptrs, mask=mask, other=0)
23 tl.store(z_ptrs, vals, mask=mask)

```

Corresponding Triton expert written kernel is below:

```

1 @triton.jit
2 def flip_kernel(X, Z, N: tl.constexpr, M: tl.constexpr):
3     offx = tl.arange(0, M)
4     offy = tl.arange(0, N) * M
5     off2d = offx[None, :] + offy[:, None]
6     x = tl.load(X + off2d)
7     x = tl.flip(x)
8     tl.store(Z + off2d, x)

```

## A.1 Performance Analysis

### Expert written code limitations:

#### 1. Double Memory Access Pattern:

```

1 x = tl.load(X + off2d) # Load entire block
2 x = tl.flip(x)         # Flip in registers
3 tl.store(Z + off2d, x) # Store entire block

```

- Loads the entire (N×M) block into registers
- Performs flip operation on register data
- Stores the entire block back to memory
- This creates unnecessary memory bandwidth usage

#### 2. Register Pressure:

- Must hold the entire block in registers simultaneously
- For large blocks, this can exceed register capacity
- May cause register spilling to local memory

#### 3. Limited Flexibility:

- ‘tl.flip()’ behavior may not be optimally tuned for all tensor shapes
- Less control over the exact memory access pattern

GEAK generated code advantages:

#### 1. Optimized Memory Access Pattern:

```

1 flipped_cols = M - 1 - cols
2 x_ptrs = X + rows[:, None] * M + flipped_cols[None, :] # Read from
3     flipped positions
4 z_ptrs = Z + rows[:, None] * M + cols[None, :]         # Write to normal
5     positions
6 vals = tl.load(x_ptrs, mask=mask, other=0)
7 tl.store(z_ptrs, vals, mask=mask)

```

- Single pass operation: Reads from flipped source positions and writes directly to destination
- No intermediate storage of the entire block
- More cache-friendly access pattern

#### 2. Better Memory Efficiency:

- Lower register usage since it doesn’t need to hold entire blocks
- Reduced memory bandwidth requirements
- Better cache utilization due to direct addressing

### 3. Explicit Masking:

```
1 mask = mask_row[:, None] & mask_col[None, :]
```

- Handles boundary conditions explicitly
- Prevents out-of-bounds memory accesses
- More robust for arbitrary tensor sizes

### 4. Coalesced Memory Access:

- The addressing pattern ‘rows[:, None] \* M + cols[None, :]’ maintains good memory coalescing
- Sequential threads access nearby memory locations

## Performance Implications:

- Memory Bandwidth: GEAK generated kernel uses  $\sim 50\%$  less memory bandwidth (single read+write vs double read+write)
- Register Usage: GEAK generated kernel has lower register pressure, allowing for larger block sizes
- Cache Efficiency: GEAK generated kernel’s direct addressing pattern is more cache-friendly
- Scalability: GEAK generated kernel scales better with larger tensor dimensions

## B Disclaimer

We clarify the following evaluation practices to help others accurately assess the correctness scope of AI-generated GPU kernel code:

### The Critical Role of Robust Unit Testing

- Strong test suites are non-negotiable. Evaluating AI-generated code must be paired with broad and well-crafted unit tests. Without sufficient coverage, code that is partially correct or even blatantly flawed may appear to pass—leading to misleading conclusions.
- Why coverage matters. Test coverage quantifies the percentage of code exercised by the tests. High coverage (statement, branch, or decision coverage) increases confidence that the code has been exercised across varied execution paths, thereby reducing the risk of undetected failures.
- LLMs struggle without strong test guidance. Recent studies like EvalPlus have shown that augmenting existing test suites with automatically generated test cases (e.g. 80 $\times$  more tests for HumanEval) can reduce flawed code passing rates by up to 28.9 percentage points. Similarly, TestGenEval finds that even top models like GPT-4o struggle to generate test suites with average coverage above 35.2%, limiting their ability to catch corner-case bugs.

### Our Benchmark Test Coverage Comparison

- ROCm Triton Benchmark: Includes wide and thorough unit test suites covering multiple input conditions, edge cases, and reference outputs. This provides confidence that passing kernels are truly correct—both functionally and numerically.
- TritonBench-G (original): Contains at most six unit tests per kernel, offering limited coverage and risk of false positives—i.e. code that passes tests yet fails under untested conditions.
- TritonBench-revised (our variant): Retains the same test harness from TritonBench-G unchanged. While functionally consistent, this limited coverage may falsely inflate apparent correctness if code is overfitted only to the narrow test cases.

### Recommendations & Evaluation Scope

- We strongly recommend using benchmarks with high coverage (like ROCm Triton benchmarks) when evaluating AI-generated kernels, especially in high-performance computing contexts where subtle numerical errors matter.

- When using narrower test suites (e.g. TritonBench-Revised), one must report expected test-coverage limitations clearly. Without caveats, reported correctness numbers (like  $\geq 50\%$  pass rate) could mislead readers into overestimating general correctness.
- For future work, integrating automatically generated test cases—via LLMs guided to expand coverage—or using mutation-based or static-analysis-assisted techniques (as in EvalPlus, ASTER, or TestGen-LLM frameworks) would help raise coverage and reveal hidden bugs.

By combining wide coverage benchmarks with clear reporting of limitations, one can ensure an accurate and trustworthy evaluation of AI-generated GPU kernels.