TRITONBENCH: Benchmarking Large Language Model Capabilities for Generating Triton Operators

Anonymous ACL submission

Abstract

Triton, a high-level Python-like language designed for building efficient GPU kernels, is widely adopted in deep learning frameworks due to its portability, flexibility, and accessibility. However, programming and parallel optimization still require considerable trial and error from Triton developers. Despite advances in large language models (LLMs) for conventional code generation, these models struggle to generate accurate, performance-optimized Triton code, as they lack awareness of its specifications and the complexities of GPU programming. More critically, there is an urgent need for systematic evaluations tailored to Triton. In this work, we introduce TRITONBENCH, the first comprehensive benchmark for Triton operator generation. TRITONBENCH features two evaluation channels: a curated set of 184 real-world operators from GitHub and a collection of operators aligned with PyTorch interfaces. Unlike conventional code benchmarks prioritizing functional correctness, TRITONBENCH also profiles efficiency performance on widely deployed GPUs aligned with industry applications. Our study reveals that current state-of-the-art code LLMs struggle to generate efficient Triton operators, highlighting a significant gap in high-performance code generation.

1 Introduction

Triton (Tillet et al., 2019) language, a high-level Python-like programming language designed for implementing efficient GPU kernels, is playing an increasingly pivotal role in the ever-scaling deep learning ecosystems (Abadi et al., 2016; Paszke et al., 2019). Due to the superior portability, flexibility, lightweight design, and accessibility to less proficient programmers, Triton is prevalently adopted in modern Large Language Model (LLM) frameworks such as vLLM (Kwon et al., 2023), LightLLM (ModelTC, 2025), Liger-kernel (Hsu et al., 2024) and unsloth (Daniel Han and team, 2023). However, crafting high-performance operators remains challenging, especially for the intricate balance between memory hierarchy management, parallel thread coordination, and hardware-specific optimizations. Even though Triton abstracts away many complexities of low-level programming architectures like CUDA, it still requires developers to manually handle critical aspects such as pointer arithmetic and memory access patterns, making performance tuning a labor-intensive process that often involves extensive trial and error.

Current research in AI-assisted coding has reached a human-competitive level (Hui et al., 2024; Zhu et al., 2024), yet it is primarily restricted to general-purpose languages like Python. However, LLMs still face challenges in generating Domain Specific Language (DSL) code. Specifically for Triton, current models might be unfamiliar with Triton specification and the intricacies of GPU programming (Nichols et al., 2024). Most importantly, the ability of these models to produce high-quality Triton code remains unassessed. Therefore, a high-quality benchmark paired with performance-aware metrics is urgently required.

In this study, we present TRITONBENCH, a performance-aware benchmark framework for Triton generation, which contains two channels, namely TRITONBENCH-G and TRITONBENCH-T. Specifically, TRITONBENCH-G contains 184 carefully curated operators from existing GitHub repositories, reflecting the realistic demand for Triton operator development. As a complement, TRITONBENCH-T is composed of operator development tasks aligned with PyTorch interfaces, covering operators under-represented by public sources. Moreover, unlike the majority of code benchmarks merely prioritizing functional correctness (Chen et al., 2021; Austin et al., 2021a), TRITONBENCH emphasizes efficiency performance profiling against reference programs on NVIDIA GPUs, better aligning industrial demands.

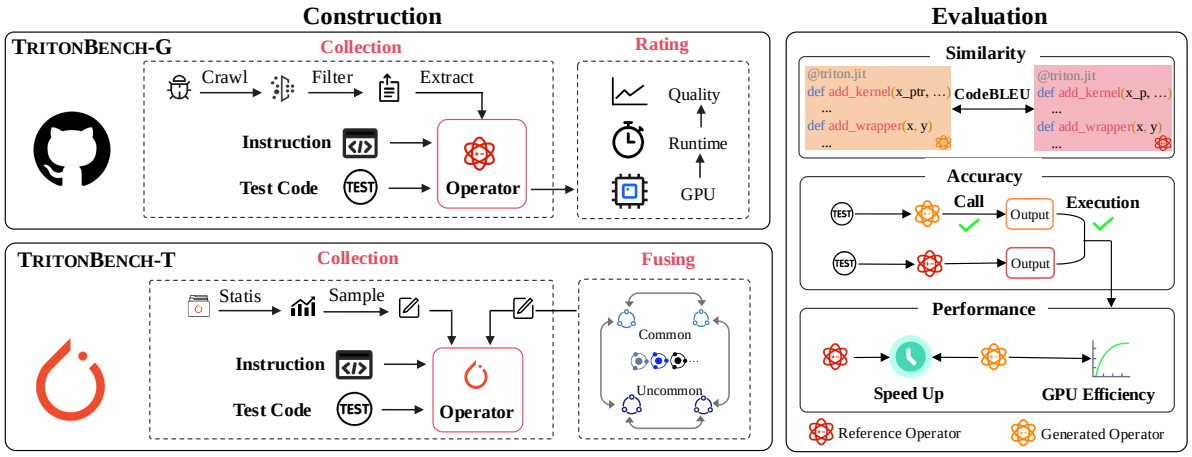


Figure 1: Illustration of the construction and evaluation of TritonBench.

As shown in Figure 1, for TritonBench-G, we follow three steps: 1) scrape and collect high-quality operators, 2) generate instructions via prompts, and 3) annotate test code with LLMs. Moreover, HPC experts evaluate GPU performance for all triton codes. For TritonBench-T, we provide operator generation tasks aligned with PyTorch. To construct these tasks, we first perform a frequency analysis to select torch operators, combine them into diverse sets, and provide paired instructions and test code. Our evaluation metrics include similarity, call and execution accuracy, speed up, and GPU efficiency.

We conduct extensive experiments across a broad range of LLMs. Overall, the difficulty of TritonBench-G is greater than that of TritonBench-T. The highest execution accuracy on TritonBench-G can reach 23.91%, while on TritonBench-T, it can reach 53.01%. For all correctly executed operators generated by the models, the best speed up on TritonBench-G is 1.56×, whereas, on TritonBench-T, it is 1.91×. Additionally, we perform in-depth analyses of LLMs' behavior on TritonBench and summarize the challenges in Triton generation. The results reveal that current LLMs are not yet fully capable of handling TritonBench, underscoring the challenge of enabling LLMs to generate Triton code effectively. We hope this work initiates evaluation in this under-explored area and fosters advancements in LLM-driven operator development.

2 Related Work

2.1 Triton Development

Triton (Mitkov et al., 2021) is an open-source, Python-like language and a compiler designed to simplify GPU programming in AI and HPC. It abstracts the complexities of CUDA by introducing a block-based programming model, automating low-level optimizations such as memory coalescing and tensor core utilization, and making it more accessible to researchers without HPC background. Nonetheless, Triton provides explicit control over memory access patterns and parallelism. This balance of productivity and flexibility makes it prevalently adopted in both academia and industry (Kwon et al., 2023; ModelTC, 2025; Hsu et al., 2024; Daniel Han and team, 2023). However, Triton developers must still laboriously tune critical parameters to exploit hardware capabilities. LLM code generation poses prospects for automating Triton development, which calls for a systematic evaluation of generated operators.

2.2 Code Benchmarks

The demand for proper measurement of coding capability arises as the program synthesis research advances. The primary practice of coding benchmarks is functional correctness testing, usually realized by test case construction and sandbox execution. For example, HumanEval (Belz et al., 2021) curate hand-written programs and test cases, and MBPP (Austin et al., 2021b) create programming problems by crowd-sourcing. The functionality test has recently extended to automated test generation for better coverage (Liu et al., 2023) and broader applications, including software engineering (Jimenez et al., 2024). Another vital aspect of coding benchmarking is performance profiling (Shypula et al.; Liu et al., 2023; Huang et al., 2024; Qiu et al., 2024). However, most existing frameworks focus on competition-style, single-process execution. While there are some

frameworks for evaluating parallel programming on CPUs (Nichols et al., 2024; Chaturvedi et al., 2024), benchmarks targeting GPU code remain scarce. As the deployment of deep learning models scales up, a comprehensive evaluation framework that considers both correctness and performance on GPU code becomes increasingly necessary.

2.3 LLMs for Code Generation

LLMs have recently demonstrated impressive capabilities in generating code from natural language instructions, as evidenced by models such as DeepSeek-Coder (Guo et al., 2024; Zhu et al., 2024) and Qwen-Coder (Hui et al., 2024), which have achieved strong performance on broad coding benchmarks. Despite their versatility, they often struggle with Domain-Specific Languages (DSLs) designed for higher levels of abstraction and improved efficiency in targeted contexts (Wąsowski and Berger, 2023). The main reason for this status is the limited availability of DSL datasets and benchmarks (Cassano et al., 2024; Pujar et al., 2023), coupled with their unique syntax and semantics (Pujar et al., 2023), posing significant challenges for LLMs (Buscemi, 2023). In this work, we focus on DSLs within the high-performance computing domain where the challenges we mentioned are more pronounced for involving the parallel programming model. We introduce the first comprehensive benchmark for Triton generation, providing a systematic evaluation framework that aims to guide future improvements in DSL-centric LLM code generation.

3 TritonBench-G

Triton (Tillet et al., 2019) is a DSL that abstracts away low-level complexities to simplify GPU programming for computation-intensive tasks, with flexibility for specialized applications like machine learning. Typically, a Triton operator includes at least a kernel and a wrapper. The kernel comprises code executed on the GPU, focusing on tensor element addressing and thread parallel coordination. Meanwhile, the wrapper offers a Python function that encapsulates the kernel call. Figure 2 shows an example of Triton operator.

We create TritonBench-G by curating high-quality human-authored Triton operators from Github, which reflects Triton's currently actual requirements. The following sections will explain data collection (§ 3.1), data statistics (§ 3.2), operator quality rating (§ 3.3), test code design (§ 3.4), and evaluation metrics (§ 3.5).

```
1  @triton.jit
2  def add_kernel(x_ptr, y_ptr, output_ptr, n_elements,
   BLOCK_SIZE):
3      pid = tl.program_id(axis=0)
4      block_start = pid * BLOCK_SIZE
5      offsets = block_start + tl.arange(0, BLOCK_SIZE)
6      mask = offsets < n_elements
7      x = tl.load(x_ptr + offsets, mask=mask)
8      y = tl.load(y_ptr + offsets, mask=mask)
9      output = x + y
10     tl.store(output_ptr + offsets, output, mask=mask)

11 def add(x: torch.Tensor, y: torch.Tensor):
12     output = torch.empty_like(x)
13     n_elements = output.numel()
14     grid = lambda meta: (triton.cdiv(n_elements, meta['
   BLOCK_SIZE']))
15     add_kernel[grid](x, y, output, n_elements, BLOCK_SI
   ZE=1024)
16     return output
```

Figure 2: Implementation of the Triton "add" operator. Lines 3-6 perform for tensor element addressing, followed by the calculation and storage in lines 7-10. The kernel is called in wrapper line 15.

3.1 Data Collection

Our process starts by gathering Triton-related GitHub repositories with more than 100 stars, which collectively encompass 95 repositories with 845 Python files. As Triton repositories with higher star counts are rare, 100 stars serve as an optimal threshold, striking a balance between quality and quantity. We then use prompt-based filtering (see prompt D in the Appendix) to process the candidate Python files and select 250 that specifically contain Triton code snippets.

Afterward, we perform a rigorous manual inspection of the Triton code to ensure its accuracy and clarity. This process involves filling in missing components, removing redundant sections, and debugging the operators. When a file contains multiple independent Triton operators, we split them into separate files. For operators that are solely kernels, we add the necessary wrappers to ensure they work as intended. Additionally, to ensure uniqueness, we leverage CodeBertScore (Zhou et al., 2023) to eliminate duplicates.

Finally, we generate the LLM instruction for each operator based on prompt D. The instructions provide essential details, including the operator's functionality, corresponding function names, and a comprehensive input/output demonstration. All instructions are carefully reviewed and manually verified to ensure they correctly reflect the intended behavior of each operator.

Difficulty	Instruction	Triton Operator			
	tok#	func#	params#	line#	tok#
d1 (1.6%)	296.67	2.00	1.33	26.00	369.0
d2 (14.7%)	363.26	2.41	2.70	45.56	678.1
d3 (35.3%)	353.80	3.80	3.34	102.42	1510.4
d4 (45.7%)	394.48	3.89	6.04	153.77	2689.1
d5 (2.7%)	469.60	6.60	6.00	249.80	4581.4

Table 1: Statistics of TRITONBENCH-G.

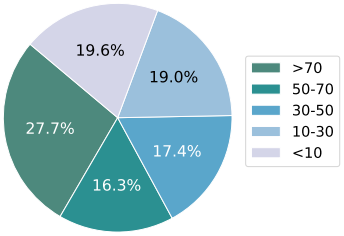


Figure 3: Distribution of GPU efficiency of the Triton operators in TRITONBENCH-G.

3.2 Data Statistics

Table 1 summarizes statistics of TRITONBENCH-G. In this benchmark, each operator is assigned a difficulty level, from **d1** (easiest) to **d5** (most challenging), by an LLM guided by prompt D, with subsequent manual verification by two domain experts. For each difficulty level, we report statistics including the average number of functions (**func#**), parameters (**params#**), lines (**lines#**), and tokens (**tok#**). Notably, the upward trend observed in these statistics as the difficulty level increases suggests the expert-driven grading scheme is largely reasonable.

Compared to existing code generation tasks (Chen et al., 2021; Austin et al., 2021a), the average instruction length in TRITONBENCH-G is substantially longer, which is a deliberate design decision. The extended instructions provide richer context, which can help the model understand nuanced requirements and generate high-quality operators. Additionally, this approach better reflects real-world operator development practices where detailed requirements are indispensable.

3.3 Operators Quality Rating

To systematically evaluate the quality of the Triton operators in TRITONBENCH-G, we compute the GPU efficiency for each operator. Detailed methodology for calculating GPU efficiency can be found in Appendix B. Our statistics indicate an average GPU efficiency of **43.0**%, which reflects the overall reliability of the operators in TRITONBENCH-G. The distribution of efficiency scores is shown in Figure 3. As shown in the figure, 19.6% of operators developed by professional Triton programmers have GPU performance below 10%, which underscores the challenges in developing and optimizing Triton operators.

3.4 Test Code Design

In contrast to traditional CPU-language benchmarks (Shypula et al.; Liu et al., 2023; Huang et al., 2024; Qiu et al., 2024) that predominantly rely on scalar test inputs,TRITONBENCH-G is built around tensor-based test inputs. We employ PyTorch to generate random tensors as replacements for conventional test cases. Specifically, we leverage a prompt D to generate the corresponding test code for each operator. In the case of the multi-branch operators, the generated test code is designed to invoke every branch within the operator. Moreover, we rigorously debug all branches to guarantee test reliability. On average, we generate 3.6 test branches per operator.

3.5 Evaluation Metrics

In contrast to traditional code evaluations, which mainly emphasize accuracy (Chen et al., 2021; Austin et al., 2021a), our TRITONBENCH-G introduces dedicated performance evaluations. Specifically, the systematic evaluation of Triton operators covers five key metrics:

`Similarity` assesses text-level resemblance using CODEBLEU (Ren et al., 2020). In our experiments, we assign equal weights of 0.25 to N-gram, weighted N-gram, syntax, and dataflow components to ensure a balanced evaluation.

`Call & Execution Accuracy` assess whether the code can run without error and whether its input-output behavior is correct, respectively.

`Speed Up` measures the relative execution time improvement for correctly executed operators. Specifically, if t_{gen} and t_{ref} represent the running times of the generated and reference operators, respectively, then $\texttt{SpeedUp}(gen) = \frac{t_{ref}}{t_{gen}}$.

`GPU Efficiency` evaluates how effectively the generated operator utilizes GPU resources, following the operator quality rating in § 3.3. For further details, please refer to Appendix B.

4 TRITONBENCH-T

The real-world Triton operators introduced in § 3 primarily focus on highly frequent operations. As a complement, we propose TRITONBENCH-T, which aligns the Triton wrapper with interfaces of the PyTorch library (Paszke et al., 2019). Together, TRITONBENCH-G and TRITONBENCH-T form a complementary evaluation framework. The following sections elaborate on the data construction (§ 4.1), data statistics (§ 4.2), test code and metrics (§ 4.3), and benchmark comparisons (§ 4.4).

4.1 Data Construction

We construct TRITONBENCH-T by selecting PyTorch operators based on their usage frequency in real-world coding and then fusing them (hereafter referred to simply as "operators"). First, we select operators that require GPU interactions, ensuring alignment with Triton's scope. Next, we sample 40 high-frequency operators and 40 low-frequency operators from the remaining pool. The frequency of each operator is determined by its usage probability in PyTorch-related code from The Stack V2 (Lozhkov et al., 2024) with those exceeding a predefined threshold 45% as common operators.

Subsequently, we fuse these operators in various configurations: combinations of common operators, combinations of common and uncommon operators, and combinations of uncommon operators. All combinations are valid, as the outputs of preceding operators serve as appropriate inputs for subsequent ones. The final set includes 166 operators, based on the latest (v2.6.0) version of the PyTorch library. Each operator is paired with its corresponding standard PyTorch call and document, while fused operators combine descriptions from all involved operators.

4.2 Data Statistic

The statistics of TRITONBENCH-T are presented in Table 2. Similar to TRITONBENCH-G, the operators are categorized into five difficulty levels (**d1** to **d5**) using an LLM guided by prompt D. These initial categorizations are then validated through manual review by two domain experts.

We report the following statistics: (1)**torch-op#** the average number of PyTorch operators, (2)**params#** the average number of parameters, (3)**math#**, the average token number of mathematical expressions, and (4)**description#**, the average token count of the descriptions. These statistics generally increase with the operator difficulty, similar trend that aligns with the observations in TRITONBENCH-G.

Difficulty	Torch-Align Operator			
	torch-op#	params#	math#	description#
d1 (13.3%)	1.36	2.82	23.50	50.41
d2 (22.3%)	1.97	3.78	40.73	61.19
d3 (32.5%)	2.70	4.91	74.64	67.89
d4 (29.5%)	2.16	5.24	47.31	71.02
d5 (2.4%)	2.75	2.75	30.50	88.50

Table 2: Statistics of TRITONBENCH-T.

4.3 Test Code and Metrics

The design of the test code in TRITONBENCH-T adheres to those of TRITONBENCH-G, employing randomly generated tensors for operator evaluation. For correctness and performance assessment, we utilize **`Call Accuracy`**, **`Execution Accuracy`**, and **`Speed Up`**, whose computation methods are consistent with those used in TRITONBENCH-G.

4.4 Benchmark Comparison

This section provides comparisons between TRITONBENCH-G and TRITONBENCH-T, which differ in key aspects and together provide a well-rounded evaluation.

Source & Distribution: TRITONBENCH-G is collected from **GitHub** and reflects real-world programming demands with a concentration of frequently used operators, e.g., `Attention` at 20.0%, `MatMul` at 10.9%, `LayerNorm` at 6.5%, `SoftMax` at 3.8%. In contrast, TRITONBENCH-T, sourced from **PyTorch**, presents a more diverse operator set including both common and uncommon operators.

Instruction Generation: TRITONBENCH-G combines **LLM generation** with **expert verification** while TRITONBENCH-T directly extracts instructions from **PyTorch documentation**. This difference underlines their complementary roles in probing different facets of the Triton generation.

Evaluation Metrics: Both benchmark channels assess **correctness** and **performance**. Additionally, TRITONBENCH-G incorporates a similarity-based assessment that offers direct comparisons with established implementations. In summary, the different designs of TRITONBENCH-G and TRITONBENCH-T enable a comprehensive and nuanced evaluation of Triton operator generation.

Model	Size	Similarity	Call Accuracy	Execution Accuracy	Speed Up	GPU Efficiency
Domain-Specific Models						
Qwen2.5-Coder	7B	9.19 / 14.54	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00
DeepSeek-Coder	6.7B	9.38 / 14.52	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00
Qwen2.5-Coder-sft	7B	**29.98** / 25.96	4.89 / 10.87	4.89 / 10.87	**1.56 / 1.22**	**51.71 / 46.70**
DeepSeek-Coder-sft	6.7B	25.52 / **30.34**	**9.78 / 11.96**	**9.87 / 11.96**	1.03 / 1.11	47.68 / 42.26
General-Purpose Models						
GPT-4o	-	9.87 / 20.67	10.87 / 17.93	10.33 / 16.84	0.97 / 1.19	48.80 / **53.33**
Claude-3.5-Sonnet	-	12.46 / 22.48	10.33 / 20.11	9.79 / 19.57	0.90 / **1.54**	**59.31** / 49.32
Qwen2.5-72B	72B	14.86 / 26.25	11.41 / 16.85	10.87 / 16.31	0.96 / 1.19	23.28 / 49.40
DeepSeek-R1	685B	**19.96** / 22.64	13.59 / 22.83	13.05 / 22.83	**1.11** / 1.22	44.83 / 46.70
GPT-o1	-	16.58 / **29.70**	**15.22 / 23.91**	**14.23 / 23.91**	0.92 / 1.14	54.25 / 46.37

Table 3: Main results of TRITONBENCH-G across baseline models, where the left side of "/" represents the zero-shot results and the right side represents the one-shot results.

5 Experiments

We conduct an extensive set of experiments on TRITONBENCH to rigorously evaluate the performance and capabilities of current LLMs.

5.1 Baselines and Setup

TRITONBENCH generally requires strong capabilities in code generation. Therefore, we select state-of-the-art LLMs that excel in programming tasks as baselines, including both specialized open-source models and general-purpose models. For specialized open-source models, we choose `Qwen2.5-Coder-7B-Instruct` (Hui et al., 2024) and `deepseek-coder-6.7b-instruct` (Guo et al., 2024). For general-purpose models, we include `Claude-3.5-Sonnet-0620`[1], `GPT-4o-0806`[2], `qwen2.5-72B-Instruct` (Yang et al., 2024), as well as the thought-driven models DeepSeek-R1 (Guo et al., 2025) and `GPT-o1-2024-12-17`[3].

In our experiments, all general-purpose models are deployed for direct inference. In contrast, domain-specific models undergo an additional supervised fine-tuning phase. Details of the training corpus can be found in § A. For evaluation, we consider both zero-shot and one-shot scenarios. In the one-shot setting, a BM25-based retrieval method (Robertson et al., 2009) is utilized to select the most relevant prompt from the training corpus.

5.2 Main results of TRITONBENCH-G

Table 3 illustrates the performances of baselines on TRITONBENCH-G. It is evident that domain-specific models generally underperform compared to general-purpose models. However, fine-tuning 7B domain-specific models with domain data significantly boosts accuracy. Qwen's accuracy rises from 0 to 4.89%, and DeepSeek's from 0 to 9.78% in zero-shot settings, with even more pronounced enhancements in one-shot settings due to the retrieval data from the same source as TRITONBENCH-G. The observed increase in `Speed Up` can be attributed to the relative simplicity of the correctly generated operators, which makes it easier for LLMs to produce efficient code. The high `GPU efficiency` shares the similar reasons.

General-purpose models, particularly DeepSeek-R1 and GPT-o1, excel across all metrics. Under one-shot conditions, DeepSeek-R1 achieves 22.83% in `Call and Execution Accuracy`, while GPT-o1 reaches 23.91%. The roughly 10% improvement from zero-shot to one-shot highlights the critical role of high-quality examples for Triton generation. Furthermore, the close alignment between `Call Accuracy` and `Execution Accuracy` indicates that only a few operators fail to produce correct results despite successfully invoked.

DeepSeek-R1 also leads in GPU execution times, with an improvement of 1.11× in zero-shot and 1.22× in one-shot settings. While GPU efficiency is strong across most models, Qwen2.5-72B exhibits lower efficiency in zero-shot settings, likely due to a higher proportion of less efficient operators. Finally, `Similarity` provides corroborative insights, as its variations mirror trends observed in other metrics.

5.3 Main Results of TRITONBENCH-T

From Table 4, we can observe that domain-specific models generally underperform general-purpose

[1] https://www.anthropic.com/news/claude-3-5-sonnet
[2] https://openai.com/index/hello-gpt-4o
[3] https://openai.com/o1

Model	Size	Call Accuracy	Execution Accuracy	Speed Up
Domain-Specific Models				
Qwen2.5-Coder	7B	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00
DeepSeek-Coder	6.7B	0.00 / 1.81	0.00 / 1.81	0.00 / **0.94**
Qwen2.5-Coder-sft	7B	17.47 / 16.27	17.47 / 15.67	**0.98** / 0.92
DeepSeek-Coder-sft	6.7B	**19.28 / 18.67**	**19.28 / 16.26**	0.91 / 0.85
General-Purpose Models				
GPT-4o	-	36.75 / 32.53	36.75 / 32.53	0.98 / 0.94
Claude-3.5-Sonnet	-	29.52 / 37.95	29.52 / 33.70	0.93 / 0.89
Qwen2.5-72B	72B	30.12 / 22.89	30.12 / 16.30	1.07 / 0.92
DeepSeek-R1	685B	**53.01 / 45.78**	**53.01 / 45.78**	1.03 / **1.91**
GPT-o1	-	32.53 / 43.37	32.53 / 43.37	**1.21** / 1.10

Table 4: Main results of TRITONBENCH-T across baseline models, where the left side of "/" represents the zero-shot results and the right side represents the one-shot results.

models. Nonetheless, fine-tuning with an $8k$ corpus considerably improves their performance. For instance, Qwen's zero-shot `Execution Accuracy` rises from 0 to 17.47%. In contrast, its one-shot improvement (15.67%) is slightly lower, likely due to the fact that the retrieved prompts and TRITONBENCH-T operators come from different sources (Github vs. Pytorch).

Among general-purpose models, DeepSeek-R1 demonstrates the strongest overall performance, achieving 53.01% `Call and Execution Accuracy` in the zero-shot setting. Although its accuracy drops by 7.23% in the one-shot setting, it still slightly surpasses GPT-o1. As for `Speed Up`, DeepSeek-R1 achieves the best performance of $1.91\times$ improvements. Most performance improvements in successfully executed operators stem from operator fusion. Triton's fused operators reduce redundant memory reads and writes compared to PyTorch, enhancing memory bandwidth utilization and boosting performance.

Overall, most models achieve better performance on TRITONBENCH-T than to TRITONBENCH-G, likely because TRITONBENCH-T features a more balanced distribution of operator difficulty, whereas TRITONBENCH-G is predominantly composed of higher-difficulty operators, namely, **d3** and **d4**.

6 Analysis

In this section, we examine the distribution of correct and incorrect operators across difficulty levels (**d1**–**d5**) for the top-performing models, DeepSeek-R1 and GPT-o1, as shown in Figure 4 and Figure 5. Additionally, we analyze the error patterns of incorrect operators and summarize the main challenges for each benchmark as detailed in Table 3 and Table 4. The zero-shot and one-shot settings are annotated as ^{0} and ^{1} respectively.

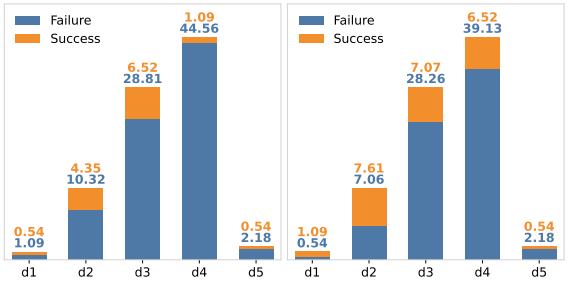
6.1 Challenges for TRITONBENCH-G

Figure 4 clearly shows that most operators are generated incorrectly. Both DeepSeek-R1 and GPT-o1 exhibit similar trends, with DeepSeek-R1 outperforming GPT-o1. Notably, when moving from the zero-shot to the one-shot setting, both models achieve significant improvements on **d4**. These improvements may stem from the prevalence of `Attention` and `Softmax` operators in **d4**, enabling models to leverage similar examples. In contrast, the simpler operators in **d2** and **d3** show only limited gains in the one-shot setting, likely due to the smaller, more idiosyncratic nature of these datasets that leads to lower retrieval similarity.

For the incorrectly written operators, we classify the 16 error types into 4 major categories, detailed in Appendix C which is presented in Table 5. Note that only compiler-reported errors were considered. The results show that, compared to the zero-shot setting, both DeepSeek-R1 and GPT-o1 in the one-shot setting demonstrate a significant increase in `Syntax` and `Name&Ref` errors but a reduction in `Attr&Type` and `Run&Logc` errors. This trend suggests that the training corpus may provide helpful guidance on logical structure and Triton specifications, thus enhancing overall accuracy. Furthermore, error sensitivity differs between models: DeepSeek-R1 is less susceptible to syntax errors, whereas GPT-o1 handles logical errors better.

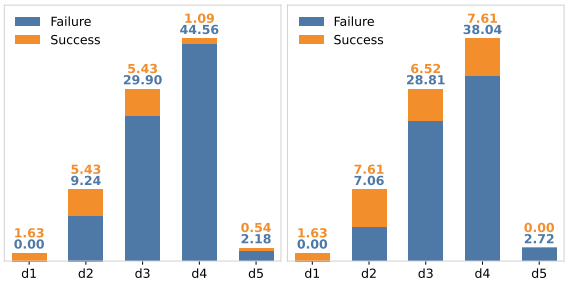
6.2 Challenges for TRITONBENCH-T

The execution results of TRITONBENCH-T (Figure 5) show the percentages of correctly generated operators. we can observe that DeepSeek-R1 generated more correct than incorrect operators,



(a) DeepSeek-R1^0 results.

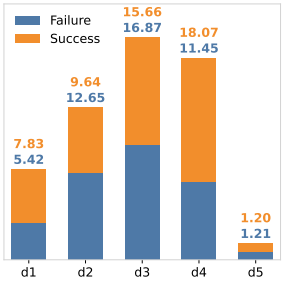
(b) DeepSeek-R1^1 results.



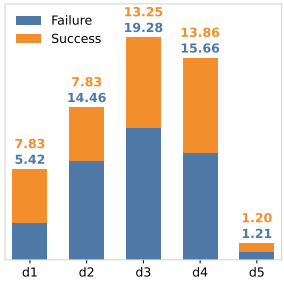
(c) GPT-o1^0 results.

(d) GPT-o1^1 results.

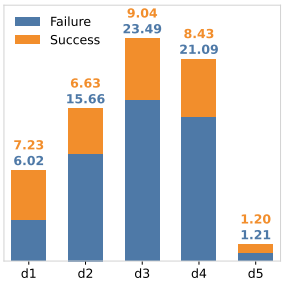
Figure 4: Execution results distribution across difficulty levels in TRITONBENCH-G.



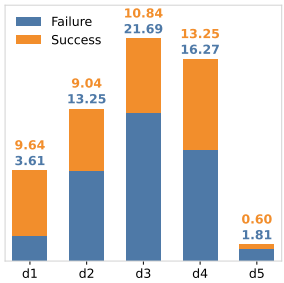
(a) DeepSeek-R1^0 results.



(b) DeepSeek-R1^1 results.



(c) GPT-o1^0 results.



(d) GPT-o1^1 results.

Figure 5: Execution results distribution across difficulty levels in TRITONBENCH-T.

Model	Syntax	Attr&Type	Name&Ref	Run&Logc
DeepSeek-R1^0	1.64	42.62	16.39	39.34
DeepSeek-R1^1	9.27	33.11	35.76	21.85
GPT-o1^0	10.3	38.18	28.48	23.03
GPT-o1^1	20.83	24.31	43.06	11.81

Table 5: Error statistics of execution failures in TRITONBENCH-G.

Model	Syntax	Attr&Type	Name&Ref	Run&Logc
DeepSeek-R1^0	0.00	31.96	14.43	53.61
DeepSeek-R1^1	0.00	36.79	20.75	42.45
GPT-o1^0	24.06	26.32	7.52	42.11
GPT-o1^1	25.25	25.25	22.22	27.27

Table 6: Error statistics of execution failures in TRITONBENCH-T.

which proves the point that the difficulty distributions in TRITONBENCH-T are smoother than TRITONBENCH-G.

However, while DeepSeek-R1's performance declines for difficulty **d2-d4** in the one-shot setting, GPT-o1 shows improved accuracy on these subsets. This finding indicates that GPT-o1 might be more adept at logical reasoning for Triton generation tasks, allowing it to efficiently use the provided sample. The differing trends also imply that sample operators affect models in diverse ways.

For execution error statistics in TRITONBENCH-T (Table 6), DeepSeek-R1 notably avoids `Syntax` errors entirely, while GPT-o1 maintains a high rate of such errors. Under the one-shot setting, DeepSeek-R1 shows a rise in `Attr&Type` and `Name&Ref` errors alongside a decline in `Run&Logc` Errors. Conversely, GPT-o1 experiences a significant increase in `Name&Ref` errors with a notable drop in `Run&Logc` errors. Comparing TRITONBENCH-G and TRITONBENCH-T, the one-shot setting consistently reduces `Run&Logc` errors. These variations in error patterns likely stem from the mixed influence of useful and irrelevant information in the provided samples.

7 Conclusion

In this work, we present TRITONBENCH, a dual-channel benchmark specifically designed for evaluating LLMs' generation for Triton operators. TRITONBENCH-G integrates real-world Triton operator samples from open repositories, while TRITONBENCH-T introduces complementary tasks that align with PyTorch interfaces. Our evaluation framework addresses both functional accuracy and the performance on NVIDIA GPUs. We also conduct extensive experiments and detailed analysis on our benchmark, and find that current LLMs struggle to generate high-quality Triton operators, underscoring the necessity for further advancement in generating accurate as well as performance-aware Triton code. We anticipate TRITONBENCH will serve as an essential framework for advancing automated operator generation for Triton.

Limitations

The primary limitation of this study is that the evaluations of TRITONBENCH were conducted exclusively on the NVIDIA A100 GPU, as it is widely adopted in industry and research applications. In future work, we plan to expand the evaluation to include a broader range of hardware architectures for more comprehensive performance insights.

Ethics Statement

This work adheres to ethical research practices and poses no potential risks. All code data used in TRITONBENCH are sourced exclusively from publicly available resources, including GitHub repositories and PyTorch documentations, ensuring no privacy concerns.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021a. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021b. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732.

Anya Belz, Shubham Agarwal, Yvette Graham, Ehud Reiter, and Anastasia Shimorina, editors. 2021. *Proceedings of the Workshop on Human Evaluation of NLP Systems (HumEval)*. Association for Computational Linguistics, Online.

Alessio Buscemi. 2023. A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *ArXiv preprint*, abs/2308.04477.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.

Aman Chaturvedi, Daniel Nichols, Siddharth Singh, and Abhinav Bhatele. 2024. Hpc-coder-v2: Studying code llms across low-resource parallel languages. *ArXiv preprint*, abs/2412.15178.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv preprint*, abs/2501.12948.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming-the rise of code intelligence. *CoRR*.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger kernel: Efficient triton kernels for llm training. *ArXiv preprint*, abs/2410.10989.

Dong Huang, Weiyi Shang, Yuhao Qing, Heming Cui, and Jie M Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. *ArXiv preprint*, abs/2402.02037.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *ArXiv preprint*, abs/2409.12186.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *ArXiv preprint*, abs/2402.19173.

Ruslan Mitkov, Vilelmini Sosoni, Julie Christine Giguère, Elena Murgolo, and Elizabeth Deysel, editors. 2021. *Proceedings of the Translation and Interpreting Technology Online Conference*. INCOMA Ltd., Held Online.

ModelTC. 2025. Lightllm: A python-based llm inference and serving framework. `https://github.com/ModelTC/lightllm`.

Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 281–294.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035.

Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matt Jones, Alessandro Morari, et al. 2023. Automated code generation for information technology tasks in yaml through large language models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE.

Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024. How efficient is llm-generated code? a rigorous & high-standard benchmark. *ArXiv preprint*, abs/2406.06647.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv preprint*, abs/2009.10297.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, et al. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.

Andrzej Wąsowski and Thorsten Berger. 2023. *Domain-Specific Languages*. Springer.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *ArXiv preprint*, abs/2412.15115.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore. Association for Computational Linguistics.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *ArXiv preprint*, abs/2406.11931.

A Training Corpus

The training corpus for supervised fine-tuning comprises two distinct components: real-world data sourced from GitHub and synthetically generated data produced through compiler operations.

The real-world data component incorporates Triton code extracted from GitHub repositories, which undergoes basic cleaning procedures as outlined in prompt D, undergoes a debugging process that is less rigorous than the methodology applied to TRITONBENCH-G. To prevent potential data leakage and ensure benchmark integrity, we systematically eliminate samples exhibiting high similarity to TRITONBENCH-G entries using the CODEBERTSCORE similarity metric (Zhou et al., 2023).

The synthetic data component is generated using Ninetoothed[4], a domain-specific language built upon Triton that offers enhanced abstraction capabilities. This framework facilitates the automated synthesis of valid Triton code through the processing of well-formed expressions. Each part of data containing $4K$ samples. This combined corpus

[4]https://github.com/InfiniTensor/ninetoothed

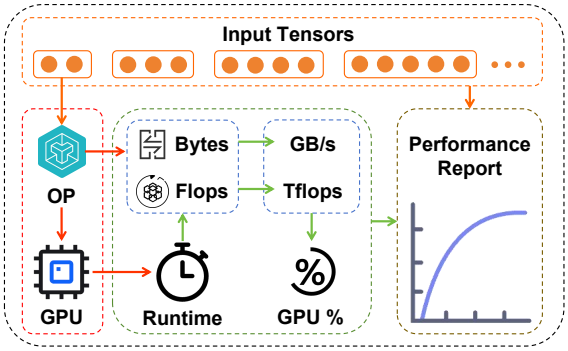


Figure 6: The workflow of operator performance evaluation

serves as the foundational training dataset for experimental models in one-shot learning settings. For all experiments, the fine-tuning process is carried out over 3 epochs with a learning rate of $5e-5$.

B Operator Performance Evaluation

For operator performance evaluation, we refer primarily to the official examples provided by Triton[5]. We provide evaluation scripts for each operator in TRITONBENCH-G. Figure 6 illustrates the workflow of our operator performance evaluation.

First, we define a set of tensors with increasing dimensions based on the characteristics of the operator. Next, each tensor is sequentially fed into the operator for execution. During each execution, we use the expert annotations for each operator to determine the total memory bandwidth (Bytes) and the total number of floating-point operations (Flops) based on the input tensors. More importantly, we use the `triton.testing.do_bench` method from the official Triton library[6] to measure the operator's execution time on the GPU. Specifically, we gradually increase the warm-up time and repetition time until the measured execution time stabilized, which means that most operators are run hundreds of thousands of times to ensure that the running time is measured accurately. After obtaining the execution time, we calculate the operator's performance metrics by dividing the total memory bandwidth and the total floating-point operations by the execution time to obtain throughput in GB/s and Tflops, respectively. We then calculate the GPU efficiency by calculating the ratio of the measured performance metrics (GB/s and Tflops) to the theoretical maximum performance of the NVIDIA A100 Tensor Core GPU. Repetition of the above process for tensors of increasing sizes obtains the performance metrics for each execution, which collectively form the operator performance report. We adopt the peak GPU efficiency from the performance report as the final measure of the operator's quality.

By following the evaluation workflow described above, we generate a detailed performance report for each operator in TRITONBENCH-G. Figure 7 illustrates the performance curves of several common operators. As the input dimensions increase, as can be seen from the figure, the GB/s or Tflops of the operators show an upward trend, eventually stabilizing. This suggests that the performance of the operator reaches a bottleneck beyond a certain scale, and further increases in input size result in diminishing returns in performance, aligning with the expected trend of operator performance.

C Error Categories

We provide the error type statistics of failure operators in TRITONBENCH. A total of 16 error types are identified in the integrated Call and Execution error results. For convenience in presentation, we categorize them into four main groups: Syntax Errors: including SyntaxError and IndentationError; Attrb&Type Errors: including AttributeError, TypeError, and NotImplementedError; Name&Ref Errors: including NameError, KeyError, IndexError, ModuleNotFoundError, and ImportError; Run&Logc Errors: including ValueError, ZeroDivisionError, RuntimeError, RecursionError, AssertionError, CompilationError, and ResultsError. ResultsError refers to the inconsistency between the execution results of the reference operator and the generated operator.

D Prompts

Here are the four prompts we use in our work: Filtering Prompt, Instruction Prompt, Difficulty Prompt, and Test Code Prompt. Specifically, the first is used to extract Triton-related code from crawled code files; the second instructs the large model to generate corresponding instructions based on Triton code; the third prompts the large model to score the difficulty of Triton operators according to the standards we proposed; and the last asks the large model to generate test code.

[5] https://triton-lang.org/main/getting-started/tutorials/
[6] https://triton-lang.org/main/pythonapi/generated/triton.testing.do_bench.html

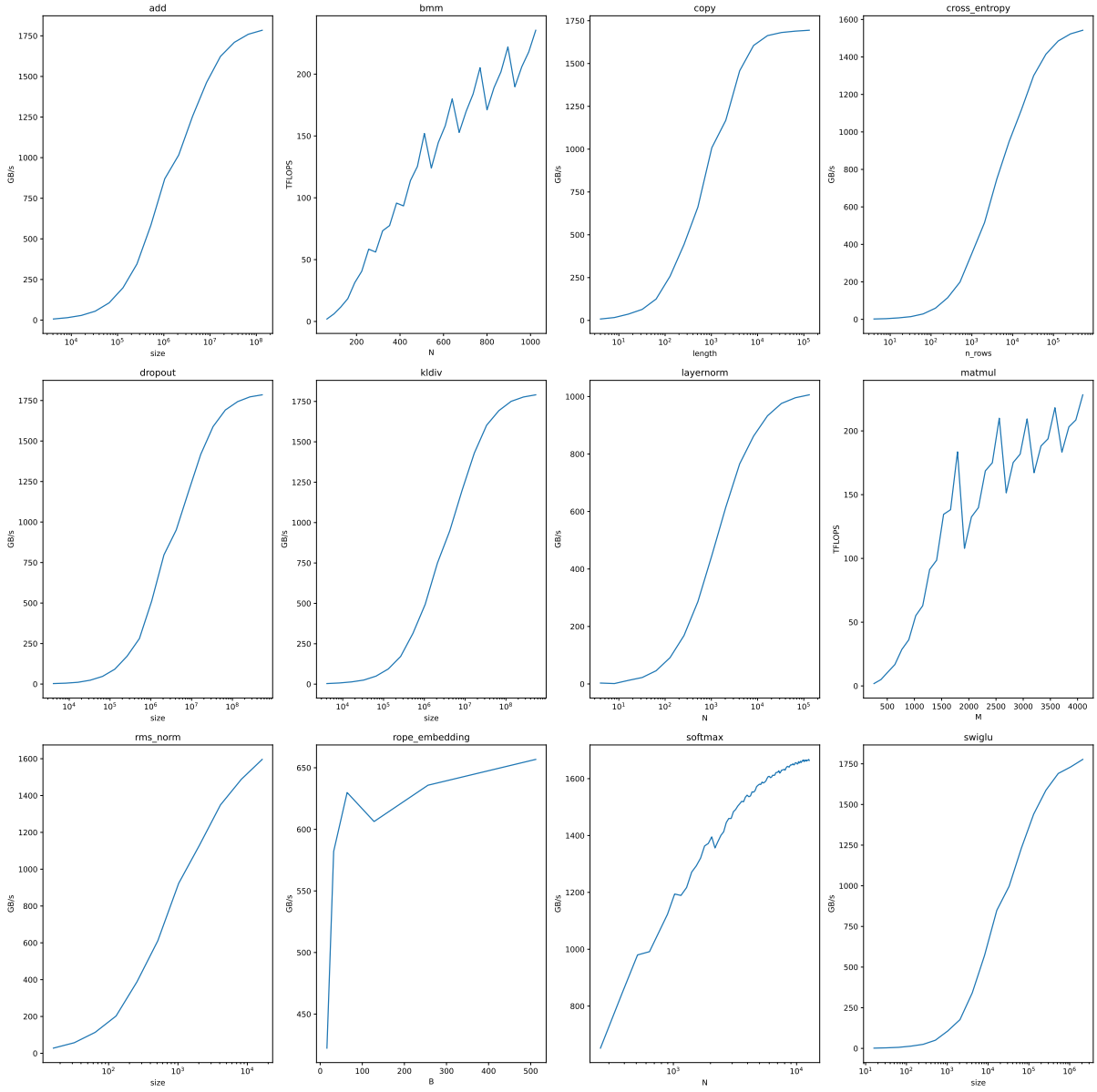


Figure 7: Performance Curves of Common Operators

Filtering Prompt

{code}
Please help me select **all** triton kernel functions decorated with `@triton.jit` and all code that calls these kernels, while only keeping the necessary imports (e.g., triton, torch) and the calling functions.

Note 1: Retain necessary comments related to the Triton code. Code can be optimized, but do not remove all kernel code and its corresponding calls just for brevity.

Note 2: If the triton kernel is decorated with a custom or third-party decorator other than triton, discard that kernel.

Note 3: If `@triton.jit` appears as a **string** in the code or is nested within a function body, then discard it.

Note 4: If there are multiple triton kernel functions decorated with `@triton.jit` and their calling wrapper functions, retain all of them, not just a subset.

1) Extract all triton operators (kernel functions decorated with `@triton.jit` and their calling functions) and output them in python code format. If no triton kernel function is found, discard it.

2) Provide a concise English description of each extracted operator (including both kernel and calling code) in the form of a python dictionary: `"description": "Use triton language to..."`

Instruction Prompt

{code}
Based on the above Triton operator code, generate a detailed description so that the large model can accurately reproduce the corresponding kernel and wrapper function.
Be clear about the logic and main functionality of the operator, specify the function name, inputs, and outputs, and describe any public variables clearly.
Try to describe the function's code implementation. Ensure that the large model can reproduce the corresponding function and parameter code based on these instructions.
Note that the output should maintain correct python syntax.

Test Code Prompt

{code}
Write a test code in Python for the above code. Ensure that all branch tests are in a single function starting with "`test_`", with no parameters.

Note 1: Particular attention should be paid to the fact that tensor parameters are of GPU type.

Note 2: Try to limit the number of branches to no more than 4.

Note 3: In branch tests, avoid modifying parameters that are later in the argument list with default values (especially if they have out parameters, do not assign them).

Note 4: Store the results of all branch calculations in a dictionary, where the dictionary key is `"test_case_n"`, with `n` representing the test case number.

Note 5: Ensure that the import paths match exactly as described in the operator documentation to maintain accuracy.

Note 6: The code should run directly, without `if __name__ == "__main__"`.

Difficulty Prompt

{code}
Please evaluate the complexity of the code in the following two aspects based on the requirements of the Triton operator and score it from simple to complex on a scale from 1 to 5:
1) Memory layout complexity: Analyze the memory access pattern, including memory tiling, array transposition, address alignment, cache utilization, and the number of global memory accesses.
2) Computation scheduling complexity: Examine instruction-level parallelism, computation-memory pipeline, thread block design, inter-thread communication, thread branch divergence, and hardware resource utilization.
The final score is the ceiling of the average score from both aspects, and only one complexity score is output in [ ].