DRTriton: Large-Scale Synthetic Data Driven Reinforcement Learning for Triton Kernel Generation

Siqi Guo
Texas A&M University
siqi@tamu.edu

Ming Lin
Oracle
linming04@gmail.com

Tianbao Yang*
Texas A&M University
tianbao-yang@tamu.edu

Abstract

Developing efficient CUDA kernels is a fundamental yet challenging task in the generative AI industry. Recent research leverages Large Language Models (LLMs) to automatically convert PyTorch reference implementations to CUDA kernels, significantly reducing engineering effort. State-of-the-art LLMs, such as GPT-5.2 and Claude-Sonnet-4.5, still struggle with this task. To address this challenge, we propose **DRTriton**, a scalable learning framework for training LLMs to convert PyTorch programs into highly optimized Triton kernels, which are then compiled to CUDA kernels at runtime. DRTriton consists of three key components: (i) a data synthetic algorithm CSP-DAG that guarantees full coverage and unbiased uniform sampling over the operator space with controlled difficulty; (ii) a curriculum RL framework with decoupled rewards that jointly optimizes conversion success rate and execution speed; and (iii) a test-time search algorithm that further improves the execution speed of the generated Triton kernels. With a warmup stage of SFT on limited PyTorch-Triton pairs curated using existing LLMs, DRTriton trained by RL on synthesized PyTorch programs generalizes effectively to real-world CUDA kernels that are challenging even for human experts. Experimental results show that DRTriton-7B achieves speedup over PyTorch on 92% of KernelBench Level 2 tasks, compared to 23% for GPT-5.2 and 19% for Claude-Sonnet-4.5.

1 Introduction

The rapid growth of generative AI has driven the industry to seek highly optimized CUDA kernels to reduce inference costs. However, developing efficient CUDA kernels remains a significant challenge even for human experts. For instance, the widely used FlashAttention Dao et al. [2022] library requires years of development effort. As new algorithms and models continue to emerge, it is increasingly impractical to manually create optimized CUDA kernels for each one. Automating efficient CUDA kernel development has become a critical need.

To simplify CUDA kernel development, practitioners often turn to more user-friendly domain-specific languages (DSLs) such as *Triton* [Tillet et al., 2019]. *Triton* offers an alternative API to CUDA with a PyTorch-like interface, achieving a good balance between development and runtime efficiency. Despite this advantage, writing efficient Triton kernels still requires substantial expertise in GPU programming and involves extensive trial and error. An automated tool capable of converting PyTorch reference implementations into optimized Triton kernels would therefore greatly alleviate this burden.

Thanks to the remarkable code generation capabilities of recent large language models (LLMs) [Chen et al., 2021, Roziere et al., 2023, Huang et al., 2025], it is promising to leverage LLMs for this conversion. While being powerful in various aspects, modern open-source and commercial LLMs still fall short in translating PyTorch implementations into efficient Triton kernels and/or CUDA kernels, even on relatively simple tasks [Ouyang et al., 2025, Li et al., 2025c].

*Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

In this work, we focus on converting PyTorch programs to Triton kernels using generative AI technologies. Specifically, we aim to train an LLM for this task. Several recent works [Li et al., 2025d, Woo et al., 2025] try to address this challenge by learning on PyTorch programs and their corresponding Triton kernels collected from public code repositories. While showing promising improvements over baseline models, the capacities of these approaches are inherently limited by the following factors:

Limited scale of training data. For example, Li et al. [2025d] use 14k samples and Woo et al. [2025] use 11k samples, which is insufficient for learning complex Triton or CUDA kernels.

Uncontrolled complexity and quality. When training with sparse rewards, it is crucial to carefully control the order and difficulty of samples presented to the model. If challenging samples are introduced too early, the model may consistently receive zero rewards, leading to poor gradient signals, wasted training tokens, and potentially misleading updates.

To address these challenges, we introduce a novel approach that leverages synthetic data. Instead of relying on limited real-world PyTorch code, we systematically generate PyTorch programs with varying difficulty and employ curriculum reinforcement learning, allowing the model to progress from simple to complex tasks. Our approach begins by collecting a representative subset of 61 widely used PyTorch operators, including those used in the KernelBench benchmark [Ouyang et al., 2025]. We then formulate PyTorch program generation as a constraint satisfaction problem over directed acyclic graphs (CSP-DAG), thereby enabling efficient and uniform sampling of valid programs composed of these operators. Based on this synthetic data generation strategy, we present **DRTriton**, a scalable framework that trains PyTorch-to-Triton LLMs based on large-scale synthetic PyTorch programs. DRTriton consists of three key components, which constitute our main contributions:

- **CSP-DAG sampling algorithm for PyTorch code generation.** We propose an efficient algorithm for synthesizing **valid** PyTorch programs. We formulate the PyTorch program generation as a constraint satisfaction problem (CSP) on directed acyclic graphs (DAGs), which can be solved using CP-SAT solvers [Perron and Didier]. This approach allows us to sample valid PyTorch programs in our code space.
- **Large-scale curriculum reinforcement learning with decoupled rewards.** We introduce a curriculum learning scheme and decouple rewards into the conventional reinforcement learning framework. It stabilizes our learning process under sparse reward and improves our learning efficiency, especially in the early stage.
- **Test-time search for compositional kernels.** We introduce an effective test-time search strategy that decomposes complex PyTorch programs into several smaller kernels and searches for optimal compositions of kernels. This test-time search strategy substantially improves both conversion success rate and execution speed when the PyTorch program cannot be translated to a single kernel.

Leveraging the above techniques, we learn a DRTriton-7B model initialized from Qwen-2.5-Coder-7B-Instruct, using 2,026 single-operator PyTorch-Triton pairs curated by DeepSeek-R1 and GPT-5.2, followed by curriculum RL on 100,000 synthetic PyTorch programs. The resulting model demonstrates strong performance: (i) On our synthetic benchmark with 20 operators, it achieves 99% accuracy, with 86% of generated Triton kernels outperforming the original PyTorch implementations. (ii) On KernelBench Level 3, the model attains 76% accuracy, with 54% of generated Triton kernels faster than the original PyTorch code and 34% faster than kernels produced by `torch.compile`.

2 Related Work

Kernel Compilers. Conventional kernel optimization relies on human-crafted optimization algorithms. TVM [Chen et al., 2018] and Ansor [Zheng et al., 2020] utilize machine learning to search for optimized schedules with manual specified patterns. TorchInductor [Ansel et al., 2024] has emerged as the default compiler (used by `torch.compile`) since PyTorch 2.0. It maps PyTorch models to Triton kernels through pattern matching and loop fusion. While effective, these compilers are bounded by predefined heuristics and templates. Our approach learns from uniformly sampled PyTorch code, avoiding reliance on pre-defined heuristics.

GPU Kernel Generation with LLMs. To overcome the rigidity of conventional compilers, recent works explored LLMs for direct kernel generation. Fisches et al. [2025] curated a dataset of 25k paired examples of PyTorch modules with their equivalent Triton kernel implementations, and then

trained a model named KernelLLM by SFT. However, their model suffers from hallucination and "fake" kernels. To improve KernelLLM, two recent works leveraged RL to fine-tune LLMs based on hand-curated data [Li et al., 2025d, Woo et al., 2025]. These approaches require curation of training samples from public datasets, which limits their scalability.

Multi-agent systems. Multi-agent systems have been proposed to coordinate planning, generation and verification [Hong et al., 2025, Wang et al., 2025a, Wei et al., 2025, Zhang et al., 2025, Liao et al., 2025, Li et al., 2025b]. Their capacity upper bound is restricted by the foundational large language models. Robust benchmarking [Lange et al., 2025] revealed that many such systems exploit testing loopholes such as eliminating redundant operators, hard-coding for specific input patterns. Our work focuses on training LLMs directly without introducing multi-turn agents, and is therefore complementary to these works.

Synthetic Code Generation. In the broader code generation landscape, methods like Magicoder [Wei et al., 2023] and WizardCoder [Luo et al., 2023] prompt LLMs to generate synthetic code for a given code snippet. This approach cannot guarantee the correctness and the coverage of the generated code due to the sampling nature of LLMs. Unlike prior work that relies on expensive LLM-based synthesis for generating kernel data [Paliskara and Saroufim, 2025, Liao et al., 2025], our framework uses the CSP-DAG algorithm to generate valid PyTorch programs with guaranteed correctness and uniform coverage, which aligns with classical program synthesis [Gulwani et al., 2017, Ellis et al., 2021].

Reinforcement Learning with Verifiable Rewards (RLVR). RLVR does not rely on human supervision, and has been shown to enable the emergence of new abilities or the discovery of knowledge beyond the original training data [Guo et al., 2025]. In our work, we observe similar phenomena: although our training pipeline does not explicitly instruct the model on how to write Triton kernels, the model learns to map PyTorch code to Triton kernel code on its own—a new skill that naturally emerges through RLVR.

One problem in RLVR for code generation is sparse reward in the early stage of the training. AutoTriton [Li et al., 2025d] and TritonRL [Woo et al., 2025] use compiler feedback plus correctness of the generated kernels as a single verifiable reward. This does not address the sparsity of the reward. To tackle this challenge, we introduce curriculum learning with decoupled rewards. The idea of decoupled rewards was first proposed in DRPO [Li et al., 2025a]. We show that the decoupled rewards greatly help us stabilize the training in the early stage, where the reward is sparse, accelerating the convergence speed of our training.

3 CSP-DAG for PyTorch Program Generation

Synthesizing PyTorch programs presents two challenges: (i) syntactic validity and (ii) satisfaction of tensor shape constraints across operators. For example, it is invalid to multiply two matrices if their dimensions do not match. To address these challenges, we propose an algorithm for synthesizing PyTorch programs with an arbitrary number of operators named CSP-DAG. The algorithm consists of two stages: (1) randomly generating N operators and linking them as a directed acyclic graph (DAG), and (2) finding the shapes of each operator's input tensors and output tensors via a CSP solver.

3.1 Generating DAGs

We represent a PyTorch program as a DAG where **edges** represent tensors with initially unknown shapes, and **nodes** represent operators that consume input edges and produce output edges. We assume each node may have multiple inputs but produces only one output.[2]

Operators fall into two categories: **OpCompute** operators take tensor inputs and produce tensor outputs (e.g., `torch.add`, `torch.matmul`, `torch.relu`), while **OpCreate** operators produce tensors without any tensor inputs (e.g., `torch.randn`, `torch.ones`). We list the full set of operators in Table 4 in Appendix A.

To generate a DAG, we maintain a list of candidate tensors, termed the candidate-tensor list, which is initialized to be an empty list. We iteratively add n nodes (operators) into the graph. At each iteration,

[2]An operator with two outputs, e.g., $a, b = f(x)$, can always be decomposed into two single-output operators: $a = f_1(x), b = f_2(x)$.

we first randomly select an operator type $\texttt{op} \in \text{OpCompute}$. Let m be the number of required inputs of $\texttt{op}$. Then, we randomly draw m tensors without replacement from the candidate-tensor list to create edges attached to the node $\texttt{op}$. If the list has fewer than m tensors, we randomly select $\texttt{op}' \in \text{OpCreate}$ and create a tensor with a line of code $\texttt{out} = \texttt{op'(shape)}$ and insert the resulting tensor into the candidate-tensor list, whose shape will be determined later. We repeat this step until the list has m tensors. After the node $\texttt{op}$ and its m edges $\text{in}_1, \text{in}_2 \ldots, \text{in}_m$ are created, we will create a line of code $\texttt{out} = \texttt{op}(\text{in}_1, \text{in}_2 \ldots, \text{in}_m)$ and insert $\texttt{out}$ into the candidate-tensor list.

We show an example of a DAG for a generated PyTorch program in Figure 1(left).

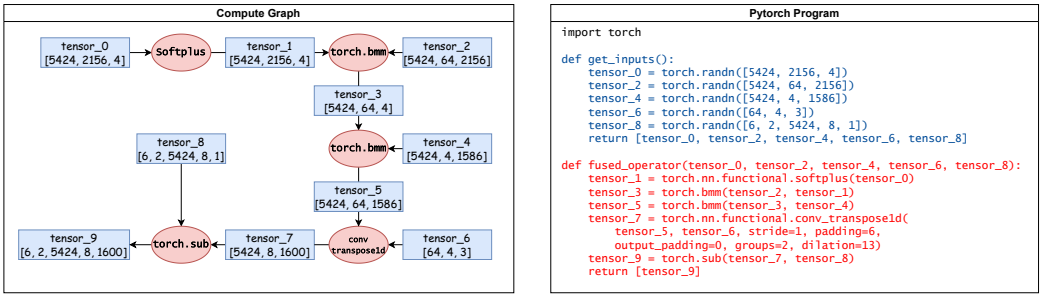


Figure 1: Left: A DAG with 5 nodes and 10 edges. Each node represents an operator, and each edge represents a tensor. Right: The synthesized PyTorch program. Blue part indicates the tensors and the red part indicates the operators.

3.2 Constraint Programming

The above process generates valid computation graphs, but does not guarantee that tensor shapes are compatible across operators. For instance, matrix multiplication requires that the inner dimensions match. To generate valid PyTorch code, we must ensure all tensors satisfy the shape constraints imposed by the operators. The shape compatibility problem can be formulated as a constraint satisfaction problem (CSP) and solved using constraint programming tools like OR-Tools.

Formally, for each tensor edge e, we denote the order of this tensor by n and the i-order has s_i dimensions. Hence, each tensor edge has n variables $s_1, \ldots, s_n$ to be solved. We restrict each dimension size to be an integer between 1 and 2^{15}. We let $e.s_i$ denote the variable s_i attached to the tensor e.

For each operator in the DAG, its order and dimension size are constrained by its neighbors in the DAG. Different operators impose different constraints, reflecting their underlying computational characteristics. We defer the full constraint specifications to Appendix B.

In addition to operator-specific constraints, we further constrain the minimum and maximum FLOPs of the generated PyTorch module, the minimum and maximum number of elements of all tensors in the DAG:

$$\text{FLOPS}_{\text{min}} \leq \sum_{\text{op} \in \text{nodes}} \text{flops}(\text{op}) \leq \text{FLOPS}_{\text{max}}, \quad \text{SIZE}_{\text{min}} \leq \sum_{e \in \text{edges}} \text{numel}(e) \leq \text{SIZE}_{\text{max}}, \quad (1)$$

where $\text{numel}(e) = \prod_{i=1}^{e.n} e.s_i$ is the total number of elements in the tensor e. The specific values of these bounds are provided in Appendix B.

The above inequalities form a typical constraint satisfaction problem (CSP). We use CP-SAT solver [Perron and Didier] to find a feasible solution. If multiple feasible solutions exist, the CP-SAT solver selects one at random. Figure 1(left) shows one solution of tensor shapes by the CP-SAT solver, whose corresponding PyTorch program is shown in Figure 1(right).

Our CSP-DAG algorithm guarantees coverage over all valid PyTorch programs composable from the given operators. Solving the constraints for a DAG of up to 20 operators takes **1–2 seconds** on a single CPU core. At scale, generating 100k programs on a 32-core machine requires roughly 1.5 hours in total.

Difficulty Level. In this work, we say that a PyTorch program is of difficulty level k if it has k OpCompute operators.

Remark. The format of our synthesized PyTorch programs closely resembles the intermediate representation (IR) used by PyTorch's compiler stack. In practice, most standard PyTorch models can be lowered into this format via `torch.export` [Ansel et al., 2024], which traces the computational graph and produces a flat, functional representation where each line corresponds to a single operator application.

4 Training Pipeline

In this section, we present our training pipeline based on the synthetic PyTorch programs. It consists of three training components — (i) SFT on a small set of PyTorch-Triton pairs on difficulty level 1; (ii) curriculum RL with decoupled rewards, and (iii) test-time search for optimal compositional kernels — supported by a Triton verifier described in Section 4.1.

4.1 Triton Verifier

We consider a Triton translation of a PyTorch program to be "correct" if (i) it is syntactically valid, (ii) it passes the faithfulness validation, and (iii) it produces the same output as the PyTorch implementation.

Syntactic validation. We apply a rule-based linter to confirm the presence of at least one `@triton.jit`-annotated kernel, then attempt compilation to catch remaining syntactic errors.

Faithfulness validation. Sometimes the LLM may simply copy the PyTorch implementation, which of course generates the same outputs. To prevent this, we employ monkey-patch testing by replacing each kernel with a no-op kernel to verify that the Triton kernels are genuinely utilized.

Correctness validation. We construct 5 random test cases with input-output pairs from the corresponding PyTorch program. We execute the Triton code on these inputs and compare its outputs with those produced by the PyTorch implementation using the same inputs. The test is passed only when all outputs of PyTorch against Triton are matched precisely.

4.2 Supervised Fine-Tuning

We employ SFT to provide a cold start for RL. Since the base model may not possess sufficient knowledge of the Triton language, directly applying RL would lead to extremely sparse and uninformative rewards, as most generated programs would fail to compile.

We construct our SFT dataset using the synthetic PyTorch programs as described in Section 3. Specifically, all SFT programs are **Level 1** programs (i.e., each containing exactly one OpCompute operator), which enables the model to first acquire basic PyTorch-to-Triton conversion skills before progressing to harder tasks in the RL stage. For each synthetic PyTorch program, we prompt DeepSeek-R1 or GPT-5.2 to generate the corresponding Triton kernel implementation. The Triton kernel is then verified, and we keep the Triton kernel codes that pass all verifications described in Section 4.1. Ultimately, this procedure yields a dataset of 2,026 PyTorch–Triton pairs covering 61 fundamental operators commonly used in PyTorch. Detailed information about the dataset construction process is provided in Appendix C.

We denote these PyTorch-Triton pairs as $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{2026}$. We fine-tune a pretrained model by minimizing the SFT loss given below:

$$\min_{\theta} -\frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \pi_\theta(\mathbf{y}|\mathbf{x}), \tag{2}$$

where π_θ denote the LLM with parameters θ.

4.3 RLVR with Decoupled Rewards

To further enhance the capability of the model, we conduct RL after SFT. While most existing studies adopt GRPO as the workhorse, we employ DRPO [Li et al., 2025a] which is designed to handle two decoupled reward signals: a correctness reward and a speed reward.

The key idea of DRPO is to decouple the accuracy reward from the speed reward in the learning process. Let q denote an input query, which is a PyTorch program to be translated, and let o denote an output Triton implementation. Let $s_\theta(o, q)$ denote the average log-likelihood score of the model: $s_\theta(o, q) = \frac{1}{|o|}\sum_{t=1}^{|o|} \log \pi_\theta(o_t|q, o_{<t})$, where o_t is the t-th token of o. The goal of our RL is to (i) improve the log-likelihood of correct translations while decreasing that of incorrect ones; (ii) favor faster Triton implementations than slower ones.

Let $\pi_{\text{old}}(\cdot)$ denote the old model to be improved. Let $\mathcal{Q} = \{q_1, \ldots, q_n\}$ denote a set of input queries. For each q_i, we let the model $\pi_{\text{old}}(\cdot)$ generate m answers $\{o_{i,1}, \ldots, o_{i,m}\}$. We split these outputs into a correct set $\mathcal{S}_+(q_i)$ and an incorrect set $\mathcal{S}_-(q_i)$. For each output o, we denote its speed reward by $r_s(o|q)$. With these notations, the policy optimization of DRPO is formulated as a sum of three terms:

$$\begin{aligned}\min_\theta -\frac{1}{n}\sum_{i=1}^{n}\sum_{o\in\mathcal{S}_+(q_i)} \omega(o|q)s_\theta(o,q) + \tau\log\Big(\frac{1}{|\mathcal{S}_-(q_i)|}\sum_{o'\in\mathcal{S}_-(q_i)}\exp\big(\frac{s_\theta(o',q)}{\tau}\big)\Big) \\ + \beta\left[\mathbb{D}_{\text{KL}}(\pi_{\text{old}}||\pi_\theta) - \delta\right]_+^2,\end{aligned} \tag{3}$$

where $\omega(o|q)$ is a speed-induced weight for a correct output defined by

$$\omega(o|q) = \frac{\exp(r_s(o|q)/\lambda)}{\sum_{o\in\mathcal{S}_+(q)}\exp(r_s(o|q)/\lambda)},$$

$\tau, \lambda, \delta, \beta > 0$ are hyper-parameters.

The first term is to increase the log-likelihood of correct outputs, with each correct output weighted using its speed reward. The second term is to decrease the log-likelihood of incorrect outputs. Using the log-sum-exp function also weights each incorrect output differently for the gradient calculation, i.e., a wrong output with a larger log-likelihood will get a higher weight for penalization. The last term is an adaptive regularization term, regularizing the change from the old model. It is motivated from the trust-region constraint $\mathbb{D}_{\text{KL}}(\pi_{\text{old}}||\pi_\theta) \le \delta$ using a squared-hinge penalty function $\beta[\cdot]_+^2 = \beta(0, \cdot)^2$.

Speed Reward. We define the speed reward as a function of the ratio between the execution time of the PyTorch program and that of the corresponding Triton program $r_s(o|q) = f\left(\frac{t_{\text{torch}}}{t_{\text{triton}}}\right)$, where t_{torch} and t_{triton} are the execution time of the PyTorch and Triton implementations, respectively. In the experiments, we have evaluated different functions f including the logarithmic function $f(s) = \log s$ and the power function $f(s) = s^\alpha$ with $\alpha > 0$. The best choice is the logarithmic function. We present the ablation studies of speed reward functions in Section 5.4.

4.4 Curriculum Reinforcement Learning

Curriculum learning has been widely adopted in reinforcement learning [Narvekar et al., 2020, Parashar et al., 2025, Wang et al., 2025b], as it improves learning efficiency and stability by gradually increasing task difficulty. We operationalize this by advancing to the next level once Pass@1 accuracy on held-out programs of the current level exceeds a threshold (50% in our experiments). We stop advancing once performance plateaus, which occurs at Level 5 in our experiments.

4.5 Test-time Search for Optimal Compositional Kernels

For PyTorch programs too complex to fit in a single Triton kernel, we propose a test-time search that identifies the optimal composition of partial Triton kernels. To achieve this, we employ a systematic search procedure over different fusion strategies. Given a PyTorch program with n sequential operators, our approach proceeds in three stages:

Fragment Extraction. We decompose the program into all possible contiguous subsequences of operators with length $l \in \{1, 2, \ldots, \min(5, n)\}$, where n is the number of operators of the program. Fragments are extracted in order of increasing length — first all single-operator fragments, then all two-operator fragments, and so on up to length 5. This generates $m = \sum_{l=1}^{\min(5,n)}(n - l + 1) = O(n)$ candidate fragments, making the extraction cost linear in the program length and thus affordable even for complex programs.

Kernel Generation and Verification. For each extracted fragment, we prompt our trained model to generate a corresponding Triton kernel implementation. We then verify the functional correctness of each generated kernel using the verification procedure described in Section 4.1. Let $m' \leq m$ denote the number of kernels that pass verification.

Code Reconstruction. For each of the m' verified kernels, we construct a candidate hybrid implementation by replacing the corresponding PyTorch fragment with the Triton kernel while keeping all other operators unchanged. This yields m' distinct hybrid implementations. We benchmark each candidate by measuring its execution time and select the fastest implementation as our final output. This ensures we obtain the optimal fusion strategy among all verified options while maintaining correctness guarantees.

5 Experiments

5.1 Experimental Setup

We employ Qwen-2.5-Coder-7B-Instruct as our base model. Our training pipeline proceeds in two stages. First, SFT operates on 2,026 single-operator PyTorch-Triton pairs covering 61 fundamental operators (detailed in Section 4.2), with ground-truth Triton implementations generated using DeepSeek-R1 or GPT-5.2. We train for 10 epochs with a learning rate 2×10^{-6} and a batch size of 64. Second, we apply DRPO in three sequential curriculum stages: 20k programs at Level 1, 60k at Level 2, and 20k at Level 5. We use learning rate 1×10^{-6}, DRPO hyper-parameters $(\beta_0, \tau, \lambda) = (100, 5, 0.1)$, KL constraint upper bound $\delta = 0.001$, and generate 8 rollouts per prompt. All experiments are conducted on 8×H100 GPUs. SFT takes approximately 2 hours; the full curriculum RL pipeline of 100k synthetic programs across three stages takes approximately 10 days.

Benchmarks. Our evaluation uses two complementary benchmarks. The synthetic benchmark contains 400 held-out PyTorch programs with no overlap with training data, spanning four difficulty levels: 100 single-operator programs (Level 1), 100 two-operator programs (Level 2), 100 five-operator programs (Level 5), and 100 twenty-operator programs (Level 20). This benchmark systematically evaluates the model's ability to scale from simple single-operator translation to complex multi-operator kernel fusion. The KernelBench benchmark comprises 250 PyTorch programs for neural network operations across three difficulty levels: Level 1 (100 tasks) contains single-kernel operators like convolution, matrix multiplications, and activations; Level 2 (100 tasks) includes fusion patterns such as convolution with bias and ReLU; and Level 3 (50 tasks) features complete ML architectures including MobileNet, VGG, and MiniGPT.

Evaluations. We employ the same verification protocol described in Section 4.1 to verify the generated Triton kernels. We report three metrics: (1) Acc as the percentage of kernels that pass verification, (2) Faster1 as the percentage of kernels that are both correct and achieve more than 1x speedup over the PyTorch execution, and (3) average speedup over PyTorch, computed using the geometric mean. We report average speedup in two ways: over the verified kernels only (Success), and over all tasks with each failed conversion counted as a 1× speedup, i.e., falling back to the original PyTorch implementation (Full).

Baselines. We compare against several baselines spanning commercial LLMs (GPT-5.2, Claude Sonnet 4.5), open-source models (DeepSeek-R1, Qwen-3-Coder-480B), and a specialized model (AutoTriton). Since test-time search is model-agnostic, we additionally apply it to AutoTriton and, on KernelBench, to Claude-Sonnet-4.5. The prompt template is given in the Appendix F.

5.2 Results on Synthetic Benchmarks

Table 1 and Figure 2 present our main results on the synthetic benchmark, demonstrating that DRTriton substantially outperforms all baselines across all difficulty levels in both accuracy and speed.

DRTriton outperforms all baselines on accuracy. DRTriton achieves 87% accuracy at Level 1 (vs. 68% for Claude) and 75% at Level 2 (vs. 49% for Claude). At Level 5, DRTriton achieves 15% accuracy while most baselines fail almost completely.

DRTriton generates substantially faster kernels. As shown in Figure 2, DRTriton's speedup curves lie clearly above all baselines across all difficulty levels. On Level 1, 32% of DRTriton's kernels

Table 1: Main results on synthetic benchmark. Acc (%): Pass@1 accuracy breakdown by difficulty level. Faster1 (%): percentage of correct kernels achieving $> 1\times$ speedup. Avg. speedup: geometric mean speedup over PyTorch, computed over correct kernels only (Success) and over all tasks with failed conversions counted as $1\times$ (Full).

Model	Level 1		Level 2		Level 5		Level 20		Avg. speedup	
	Acc	Faster1	Acc	Faster1	Acc	Faster1	Acc	Faster1	Success	Full
GPT-5.2	54	17	43	35	5	5	0	0	1.34	1.08
Claude-Sonnet-4.5	68	14	49	29	7	3	0	0	0.68	0.89
DeepSeek-R1	33	13	24	16	9	3	0	0	0.91	0.98
Qwen-3-Coder-480B	48	6	30	23	1	1	0	0	0.97	0.99
AutoTriton	36	7	15	8	2	1	0	0	1.08	1.01
+ test-time search	36	7	45	22	61	34	96	80	1.18	1.10
DRTriton (ours)	**87**	**32**	**75**	**54**	15	10	0	0	1.20	1.08
+ test-time search	**87**	**32**	**96**	**78**	**99**	**89**	**99**	**86**	**1.57**	**1.53**

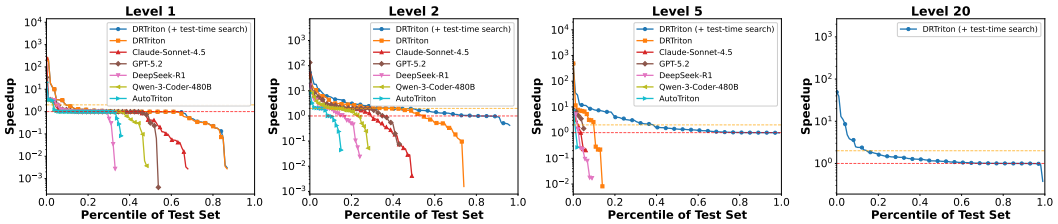


Figure 2: Speedup distribution across difficulty levels for DRTriton, DRTriton (with test-time search), and other baselines, on synthetic benchmarks. Each curve shows speedups of generated Triton kernels over PyTorch programs across the percentages of testing data. The red dashed flat line is the reference line for 1x speedup, and the orange dashed flat line is the reference line for 2x speedup.

exceed $1\times$ speedup, compared to 17% for GPT-5.2 and 14% for Claude; on Level 2 this rises to 54%, far exceeding all baselines. This demonstrates that DRPO's speed-based reward weighting $\omega(o|q)$ successfully guides the model toward generating optimized, not merely correct, kernels.

Effectiveness of curriculum RL. Figure 3a shows that DRPO training substantially improves over SFT. Peak performance at Level 1 and Level 2 is achieved at Stage 2, with a slight regression in Stage 3; Level 5 shows steady improvement throughout.

Effectiveness of test-time search. Test-time search dramatically improves performance on complex programs. Counterintuitively, longer and harder programs achieve higher accuracy and better speedup after test-time search. This occurs because longer programs contain more code segments with optimization potential, making it easier for DRTriton to identify and improve inefficient fragments. Test-time search improves average speedup from $1.20\times$ to $1.57\times$ over correct kernels, and from $1.08\times$ to $1.53\times$ over the full benchmark, demonstrating that fragment-based kernel composition preserves and enhances optimization quality. Notably, test-time search is **model-agnostic**: as shown in Table 1, applying it to AutoTriton also yields substantial improvements across all difficulty levels. We further disentangle the contributions of the trained model and the search in Section 5.4. The computational overhead of test-time search is reported in Appendix D.3.

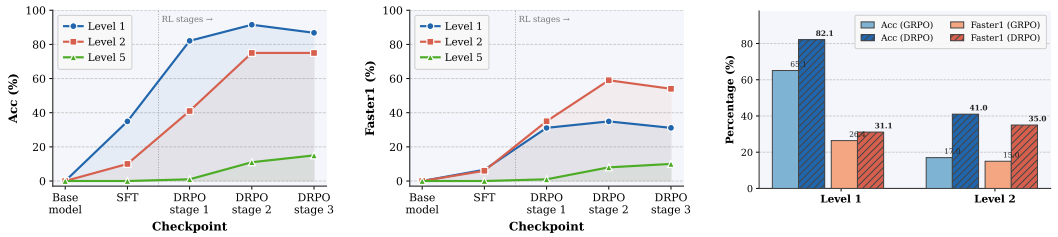
5.3 Results on KernelBench

While our synthesis framework generates programs in functional format, KernelBench provides object-oriented PyTorch code. To bridge this gap, we use `torch.export` to lower each KernelBench model into a flat functional IR, which has the same structure as our synthesized programs. Given the functional code, we then apply test-time search to identify the optimal kernel, and measure execution speed against both the original PyTorch implementation and the `torch.compile` baseline. An example of the rewritten code is provided in Appendix E.

Table 2 and Figure 4 (Appendix D.1) present the results. Test-time search substantially improves every model it is applied to, and DRTriton with test-time search achieves substantially higher accuracy and faster execution than all baselines evaluated without search. At Level 1 (single operators),

Table 2: Results on KernelBench real-world benchmark. We report accuracy (Acc) and Faster1 scores (percentage of correct kernels achieving $> 1\times$ speedup) against two baselines: Torch Eager (TE) and `torch.compile` (TC).

Model	Level 1			Level 2			Level 3		
	Acc	TE	TC	Acc	TE	TC	Acc	TE	TC
AutoTriton	44	12	6	45	14	12	32	14	6
GPT-5.2	40	23	23	32	23	11	30	18	10
Claude-Sonnet-4.5	46	8	8	36	19	12	36	12	12
DeepSeek-R1	25	8	7	22	15	8	8	2	2
Qwen-3-Coder-480B	21	5	1	18	14	1	24	0	0
AutoTriton (test-time search)	44	12	6	89	32	19	**86**	48	20
Claude-Sonnet-4.5 (test-time search)	67	**41**	**30**	**96**	68	42	**86**	**64**	30
DRTriton (test-time search)	**69**	17	12	**96**	**92**	**56**	76	54	**34**



(a) Curriculum RL progression.

(b) DRPO vs. GRPO ablation.

Figure 3: Training dynamics. (a) Performance progression across training stages on three difficulty levels, showing Acc and Faster1 metrics for the base model, after SFT, and across three DRPO stages. (b) Comparison of DRPO and GRPO trained on Stage 1 data (20k samples) from the same SFT checkpoint; DRPO consistently outperforms GRPO across all metrics.

DRTriton achieves the highest accuracy (69%) but modest speedups over Torch Eager (TE: 17%) and `torch.compile` (TC: 12%), as these operators are already well-optimized in PyTorch; here Claude-Sonnet-4.5 with test-time search reaches similar accuracy (67%) with a larger fraction of faster kernels (TE: 41%, TC: 30%). At Level 2, DRTriton's performance improves dramatically: 96% accuracy with 92% faster than TE and 56% faster than TC. Claude-Sonnet-4.5 with test-time search matches this accuracy, but DRTriton produces faster kernels on 24 and 14 more percentage points of tasks against TE and TC respectively, demonstrating DRTriton's strength in multi-operator fusion.

At Level 3 (complete architectures), DRTriton maintains 76% accuracy with 54% outperforming TE and 34% outperforming TC, the highest TC score among all methods. AutoTriton and Claude-Sonnet-4.5 with test-time search achieve higher accuracy at Level 3 (86%), and Claude-Sonnet-4.5 with search also produces more kernels faster than Torch Eager (64%), while DRTriton remains ahead against `torch.compile` (34% vs. 30% and 20%). We note that DRTriton attains this performance with a 7B open model. A detailed walkthrough of a Level 3 (LeNet-5) kernel generation — from object-oriented PyTorch to optimized Triton — is provided in Appendix G.

5.4 Ablation Studies

We conduct three ablation studies to validate key design choices and to separate the contributions of the trained model and test-time search.

Ablation on RL algorithm. Figure 3b compares DRPO with GRPO on identical Stage 1 training data from the same SFT checkpoint. For GRPO, we design the reward as $r(o) = 1 + f\left(\frac{t_{\text{torch}}}{t_{\text{triton}}}\right)$ if the output is valid, otherwise, $r(o) = 0$. The GRPO objective is detailed in Appendix H. DRPO consistently outperforms GRPO across all metrics.

Ablation on reward design. Table 5 in Appendix D.2 examines different reward function designs for the speed component $r_s(o)$ in DRPO. We compare the logarithmic form $r_s(o) = \log(t_{\text{torch}}/t_{\text{triton}})$ against power forms $r_s(o) = (t_{\text{torch}}/t_{\text{triton}})^{\alpha}$ for $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$. The logarithmic form consistently achieves the best Acc and Faster1, and is used in all other experiments.

Table 3: Decomposing the contributions of the model and test-time search on the synthetic benchmark. The same test-time search is applied to the base model (Qwen-2.5-Coder-7B-Instruct), the SFT-only checkpoint, and the checkpoint after each DRPO stage. Metrics are defined as in Table 1.

Checkpoint (+ search)	Level 1		Level 2		Level 5		Level 20		Avg. speedup	
	Acc	Faster1	Acc	Faster1	Acc	Faster1	Acc	Faster1	Success	Full
Base model	23	4	44	12	73	34	81	52	1.06	1.03
SFT-only	30	6	50	14	82	44	95	71	1.09	1.05
DRPO Stage 1	82	28	92	62	**100**	89	98	**87**	1.32	1.29
DRPO Stage 2	**92**	**36**	**98**	**83**	**100**	**91**	97	84	1.53	1.51
DRPO Stage 3 (DRTriton)	87	32	96	78	99	89	**99**	86	**1.57**	**1.53**

Contributions of the model versus search. To disentangle the gains due to the trained model from those due to test-time search, we apply the same test-time search to the base model, the SFT-only checkpoint, and the checkpoint after each DRPO stage (Table 3). Search alone is effective at raising accuracy on long programs: even the base model reaches 81% accuracy at Level 20, because it only needs to succeed on some fragments. However, search contributes little to speed on its own: the base model and SFT-only checkpoint reach full-benchmark average speedups of only 1.03× and 1.05×, with Level 1 and Level 2 Faster1 at or below 14%. The largest gain comes from the first DRPO stage, which raises the full-benchmark average speedup from 1.05× to 1.29× and Level 2 Faster1 from 14% to 62%. Subsequent stages further increase speed (1.51× and 1.53×). Consistent with Figure 3a, Stage 2 gives the best Level 1 and Level 2 results, while Stage 3 gives the highest average speedup. These results indicate that test-time search mainly improves coverage, while the speed of the final kernels is driven primarily by the RL-trained model.

6 Conclusion

We have presented DRTriton, a scalable framework for training PyTorch-to-Triton LLMs via large-scale synthetic data and curriculum reinforcement learning. DRTriton-7B outperforms state-of-the-art commercial LLMs and specialized models on the synthetic benchmark, and on KernelBench it achieves the highest accuracy at Levels 1–2 and the most kernels faster than `torch.compile` at Levels 2–3, remaining competitive with a frontier commercial model equipped with the same test-time search despite having only 7B parameters. These results demonstrate that a small SFT warmup on synthetic data combined with RL on large-scale synthetic programs generalizes effectively to complex, human-authored GPU kernels. Our current operator set is limited to 61 PyTorch operators, and extending coverage to sparse operations, custom CUDA extensions, and native CUDA generation is an important future direction. We believe this work establishes synthetic data-driven curriculum RL as a scalable and promising paradigm for automated GPU kernel generation.

References

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pages 835–850, 2021.

Zacharias V Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, et al. Kernelllm: Making kernel development more accessible, 6 2025. *Corresponding authors: Aram Markosyan, Mark Saroufim*, 2025.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

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

Charles Hong, Sahil Bhatia, Alvin Cheung, and Sophia Shao. Autocomp: Llm-driven code optimization for tensor accelerators. In *Machine Learning for Computer Architecture and Systems 2025*, 2025.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193, 2025.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

Gang Li, Yan Chen, Ming Lin, and Tianbao Yang. Drpo: Efficient reasoning via decoupled reward policy optimization. *arXiv preprint arXiv:2510.04474*, 2025a.

Haonan Li, Keyu Man, Partha Kanuparthy, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. Tritonforge: Profiling-guided framework for automated triton kernel optimization. *arXiv preprint arXiv:2512.09196*, 2025b.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066, 2025c.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025d.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: a framework and survey. *J. Mach. Learn. Res.*, 21(1), January 2020. ISSN 1532-4435.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Sahan Paliskara and Mark Saroufim. Kernelbook, 5 2025. URL https://huggingface.co/datasets/GPUMODE/KernelBook.

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, and Shuiwang Ji. Curriculum reinforcement learning from easy to hard tasks improves llm reasoning, 2025. URL https://arxiv.org/abs/2506.06632.

Laurent Perron and Frédéric Didier. Cp-sat. URL https://developers.google.com/optimization/cp/cp_solver/.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 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, 2019.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194*, 2025a.

Zhenting Wang, Guofeng Cui, Yu-Jhe Li, Kun Wan, and Wentian Zhao. Dump: Automated distribution-level curriculum learning for rl-based llm post-training, 2025b. URL https://arxiv.org/abs/2504.09710.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*, 2023.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. Tritonrl: Training llms to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

A Full set of operators

Table 4: Complete List of Operators

Category	Operator Class	PyTorch Function	#In
OpCreate	`Randn`	`torch.randn`	0
	`Ones`	`torch.ones`	0
	`Zeros`	`torch.zeros`	0
Elementwise Binary	`Add`	`torch.add`	2

Continued on next page

Table 4 – *Continued from previous page*

Category	Operator Class	PyTorch Function	#In
	Mul	torch.mul	2
	Sub	torch.sub	2
	Div	torch.div	2
	Maximum	torch.maximum	2
	Minimum	torch.minimum	2
Elementwise Ternary	Lerp	torch.lerp	3
Reduction	Max	torch.max	1
	Min	torch.min	1
	Sum	torch.sum	1
	Mean	torch.mean	1
	ArgMax	torch.argmax	1
	ArgMin	torch.argmin	1
	Var	torch.var	1
	Norm	torch.norm	1
Matrix Operators	Matmul	torch.matmul	2
	Bmm	torch.bmm	2
	Transpose	torch.transpose	1
	Triu	torch.triu	1
	Tril	torch.tril	1
Unary Activation	ReLU	torch.relu	1
	LeakyReLU	torch.nn.functional.leaky_relu	1
	Sigmoid	torch.sigmoid	1
	Tanh	torch.tanh	1
	Swish	torch.nn.functional.silu	1
	GELU	torch.nn.functional.gelu	1
	SELU	torch.selu	1
	ELU	torch.nn.functional.elu	1
	Hardsigmoid	torch.nn.functional.hardsigmoid	1
	HardTanh	torch.nn.functional.hardtanh	1
	Softplus	torch.nn.functional.softplus	1
	Softsign	torch.nn.functional.softsign	1
	LogSigmoid	torch.nn.functional.logsigmoid	1
	Clamp	torch.clamp	1
Unary with Dimension	Softmax	torch.softmax	1
	LogSoftmax	torch.log_softmax	1
Unary Mathematical	Cos	torch.cos	1
	Sin	torch.sin	1
	Exp2	torch.exp2	1
	Abs	torch.abs	1
Cumulative Operators	CumMax	torch.cummax	1
	CumMin	torch.cummin	1
	CumSum	torch.cumsum	1
Normalization	BatchNorm	torch.nn.functional.batch_norm	1
	LayerNorm	torch.nn.functional.layer_norm	1
	GroupNorm	torch.nn.functional.group_norm	1
	InstanceNorm	torch.nn.functional.instance_norm	1
Pooling (1D)	AvgPool1d	torch.nn.functional.avg_pool1d	1
	MaxPool1d	torch.nn.functional.max_pool1d	1
Pooling (2D)	AvgPool2d	torch.nn.functional.avg_pool2d	1
	MaxPool2d	torch.nn.functional.max_pool2d	1
Pooling (3D)	AvgPool3d	torch.nn.functional.avg_pool3d	1
	MaxPool3d	torch.nn.functional.max_pool3d	1
Convolution	Conv1d	torch.nn.functional.conv1d	2
	Conv2d	torch.nn.functional.conv2d	2

Continued on next page

Table 4 – *Continued from previous page*

Category	Operator Class	PyTorch Function	#In
	`Conv3d`	`torch.nn.functional.conv3d`	2
Transposed Convolution	`ConvTranspose1d`	`torch.nn.functional.conv_transpose1d`	2
	`ConvTranspose2d`	`torch.nn.functional.conv_transpose2d`	2
	`ConvTranspose3d`	`torch.nn.functional.conv_transpose3d`	2
Tensor Manipulation	`Cat`	`torch.cat`	$\geq$2
	`Stack`	`torch.stack`	$\geq$2

B Constraint Details

Before describing the constraints for each operator category, we first clarify the broadcasting mechanism and our notation convention. **Broadcasting** is a mechanism that allows operators to work with inputs of different shapes by automatically expanding dimensions. Specifically, two dimensions are compatible for broadcasting if they are equal, or if one of them is 1. When broadcasting occurs, the dimension of size 1 is virtually replicated to match the larger dimension. For example, consider adding two tensors: tensor a with shape $(3, 1, 5)$ and tensor b with shape $(3, 4, 5)$. These are compatible for broadcasting because in each dimension position (comparing from right to left), we have: dimension 3 has $5 = 5$ (compatible), dimension 2 has 1 and 4 (compatible, 1 broadcasts to 4), and dimension 1 has $3 = 3$ (compatible). The output has shape $(3, 4, 5)$. However, tensor a with shape $(3, 2, 5)$ and tensor b with shape $(3, 4, 5)$ would fail to broadcast because dimension 2 has sizes 2 and 4, where neither is equal nor is either 1. Broadcasting also works across tensors with different orders (number of dimensions). For instance, tensor a with shape $(5,)$ and tensor b with shape $(3, 4, 5)$ can be broadcast together. Aligning from the right, a is treated as having shape $(1, 1, 5)$, which is compatible with $(3, 4, 5)$, yielding an output of shape $(3, 4, 5)$.

Since constraint checking always occurs on the trailing (rightmost) dimensions of input tensors, and operators can have multiple inputs with different orders, we introduce a global variable $N = \max(\text{in}_1.n, \text{in}_2.n, \ldots, \text{in}_m.n)$ representing the maximum order among all m inputs. We then define right-aligned versions $\text{in}'_1, \text{in}'_2, \ldots, \text{in}'_m$, each of order N, where $\text{in}'_x.s_i = \text{in}_x.s_{i-(N-\text{in}_x.n)}$ for all $i \in \{N - \text{in}_x.n + 1, \ldots, N\}$, and $\text{in}'_x.s_i = 1$ for all $i \in \{1, \ldots, N - \text{in}_x.n\}$. This right-alignment pads leading dimensions with size 1, enabling uniform constraint specification across inputs of varying orders. Unless explicitly stated otherwise, all constraints below are defined on these right-aligned variables $\text{in}'_1, \text{in}'_2, \ldots, \text{in}'_m$ rather than on the original inputs.

1. **ElementwiseOp.** This category of operators takes two inputs and produces one output $c = \text{op}(a, b)$. The shapes of inputs can be broadcast. The constraints are:

$$
\begin{aligned}
&c.n = \max(a.n, b.n) \\
&c.s_i = \max(a.s_i, b.s_i), \quad \forall i \in \{1, \ldots, N\} \\
&(a.s_i = b.s_i) \vee (a.s_i = 1) \vee (b.s_i = 1), \\
&\quad \forall i \in \{1, \ldots, N\}
\end{aligned}
$$

2. **ReduceOp.** This category of operators takes one input and arguments dim and keepdim. Let a denote the input, c denote the output, dim' denote the actual dimension index after

right-alignment: $\text{dim}' = N - a.n + \text{dim} + 1$. The constraints are:

$$
\begin{aligned}
&\text{dim} < a.n \\
&\text{if keepdim} = \text{True}: \\
&\quad c.n = a.n \\
&\quad c.s_i = \begin{cases} a.s_i & \text{if } i \neq \text{dim}' \\ 1 & \text{if } i = \text{dim}' \end{cases} \\
&\text{if keepdim} = \text{False}: \\
&\quad c.n = a.n - 1 \\
&\quad c.s_i = \begin{cases} a.s_{i+1} & \text{if } i < \text{dim}' \\ a.s_i & \text{if } i > \text{dim}' \end{cases} \\
&\forall i \in \{1, \ldots, N\}
\end{aligned}
$$

3. **Matmul.** Matrix multiplication with broadcasting on batch dimensions. Let a and b denote the inputs and c denote the output:

$$
\begin{aligned}
&a.n \geq 2, \quad b.n \geq 2 \\
&a.s_N = b.s_{N-1} \\
&c.s_{N-1} = a.s_{N-1} \\
&c.s_N = b.s_N \\
&c.n = \max(a.n, b.n) \\
&c.s_i = \max(a.s_i, b.s_i), \quad \forall i \in \{1, \ldots, N-2\} \\
&(a.s_i = b.s_i) \vee (a.s_i = 1) \vee (b.s_i = 1), \\
&\quad \forall i \in \{1, \ldots, N-2\}
\end{aligned}
$$

4. **Transpose.** Let a denote the input, c denote the output. Transposing the last two dimensions has the constraints:

$$
\begin{aligned}
&a.n \geq 2 \\
&c.n = a.n \\
&c.s_i = a.s_i, \quad \forall i \in \{1, \ldots, N-2\} \\
&c.s_{N-1} = a.s_N \\
&c.s_N = a.s_{N-1}
\end{aligned}
$$

5. **ConvNd.** Convolution with parameters spatial dims $m \in \{1, 2, 3\}$, output channels C_{out}, kernel sizes $k_1, \ldots, k_m$ for each spatial dimension, stride s, padding p, dilation d, and groups g:

$$
\begin{aligned}
&a.n = c.n = m + 2 \\
&2p \leq \max_j k_j \\
&c.s_{N-m-1} = a.s_{N-m-1} \\
&a.s_{N-m} \equiv 0 \pmod{g} \\
&c.s_{N-m} = C_{\text{out}} \equiv 0 \pmod{g} \\
&\text{For spatial dimension } i \in \{N-m+1, \ldots, N\}: \\
&\quad a.s_i \geq k_{i-(N-m)} \\
&\quad c.s_i = \left\lfloor \frac{a.s_i + 2p - d(k_{i-(N-m)} - 1) - 1}{s} \right\rfloor + 1
\end{aligned}
$$

Other operators like PoolNd, ConvTransposeNd are handled similarly to ConvNd.

6. **BatchNorm/LayerNorm/GroupNorm/InstanceNorm.** Normalization operators preserve the input shape:

$$
\begin{aligned}
&c.n = a.n \\
&c.s_i = a.s_i, \quad \forall i \in \{1, \ldots, N\}
\end{aligned}
$$

For specific normalization types, additional constraints may apply (e.g., GroupNorm requires the number of channels to be divisible by the number of groups).

FLOPs and Size Constraints. In our experiments, we set $\text{FLOPS}_{\text{min}} = 2^{34}$, $\text{FLOPS}_{\text{max}} = 2^{35}$, $\text{SIZE}_{\text{min}} = 32$, and $\text{SIZE}_{\text{max}} = 2^{32}$.

C SFT Dataset Construction Details

Our dataset construction follows a multi-stage process designed to ensure both coverage and quality across diverse PyTorch operators.

Initial Data Generation. We begin by sampling 200 synthetic PyTorch programs for each of the 61 fundamental operators, yielding 12,200 programs in total. For each program, we prompt DeepSeek-R1 to generate the corresponding Triton kernel implementation. We then validate each generated kernel through execution testing to verify functional correctness. Programs for which DeepSeek-R1 fails to produce correct Triton kernels are discarded. This initial filtering process results in 1,464 valid PyTorch–Triton pairs.

Iterative Refinement. To identify operators requiring additional training data, we train an initial model using the 1,464-sample dataset through supervised fine-tuning (SFT) followed by one stage of reinforcement learning (RL). We then evaluate the trained model's operator-level success rate by generating and validating Triton kernels for 100 test samples per operator. This analysis reveals that certain operators exhibit extremely low success rates, indicating insufficient training data coverage. For operators with poor performance in the evaluation phase, we generate additional training data using GPT-5.2. Specifically, we prompt GPT-5.2 to produce 100 additional PyTorch–Triton pairs for each under-performing operator. After validation and filtering, this augmentation yields 562 additional correct pairs.

Final Dataset. We merge the initial 1,464 samples with the 562 augmented samples to form our final training dataset of 2,026 PyTorch–Triton pairs. This dataset maintains broad coverage across all 61 operators while providing enhanced representation for operators that proved challenging for the base model. The distribution of operators in the final dataset reflects both the initial uniform sampling strategy and the targeted augmentation based on empirical performance needs.

D Additional Experiment Results

D.1 Speedup Curves on KernelBench

Figure 4 shows the speedup distribution across all three KernelBench difficulty levels for DRTriton (with test-time search) and all baseline models.

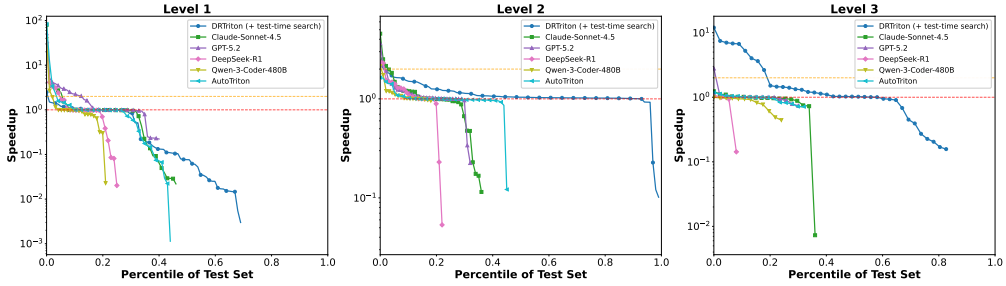


Figure 4: Speedup distribution across difficulty levels for DRTriton (with test-time search), and other baselines, on KernelBench. Each curve shows speedups of generated Triton kernels over PyTorch programs across the percentages of testing data. The red dashed flat line is the reference line for 1x speedup, and the orange dashed flat line is the reference line for 2x speedup.

D.2 Ablation on Speed Reward Function

Table 5 compares different functional forms for the speed reward component $r_s(o)$ in DRPO, using Qwen-2.5-Coder-1.5B as the base model.

Table 5: Ablation study on speed reward function design. All models are trained with DRPO on Stage 1 data using Qwen-2.5-Coder-1.5B as the base model, starting from the SFT checkpoint.

Reward Function	Level 1	
	Acc	Faster1
Power ($\alpha = 0.25$)	38.1	14.4
Power ($\alpha = 0.50$)	30.9	5.2
Power ($\alpha = 0.75$)	30.9	2.1
Power ($\alpha = 1.0$)	32.0	3.1
Logarithmic	**42.3**	**18.6**

D.3 Test-Time Search Overhead

Table 6 reports the wall-clock cost of test-time search on both benchmarks. Despite the larger number of fragments evaluated, the total overhead remains modest in absolute terms and is negligible relative to the overall training cost.

Table 6: Test-time search overhead on synthetic and KernelBench benchmarks. Generation and validation times are reported in hours.

Benchmark	Setting	Fragments evaluated	Generation (h)	Validation (h)	Total (h)
Synthetic	w/o test-time search	400	0.01	0.11	0.12
	w/ test-time search	10,612	0.13	2.72	2.85
KernelBench	w/o test-time search	250	0.01	0.07	0.08
	w/ test-time search	16,859	0.23	3.30	3.53

E PyTorch Code Rewriting Example

The following shows a typical multi-layer perceptron (MLP) implementation from KernelBench:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self, input_size, layer_sizes, output_size):
        """
        :param input_size: The number of input features
        :param layer_sizes: A list of ints containing the sizes
                            of each hidden layer
        :param output_size: The number of output features
        """
        super(Model, self).__init__()

        layers = []
        current_input_size = input_size

        for layer_size in layer_sizes:
            layers.append(nn.Linear(current_input_size, layer_size))
            layers.append(nn.ReLU())
            current_input_size = layer_size

        layers.append(nn.Linear(current_input_size, output_size))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        """
        :param x: The input tensor, shape (batch_size, input_size)
```

```
        :return: The output tensor, shape (batch_size, output_size)
        """
        return self.network(x)

# Test configuration
batch_size = 128
input_size = 16384
layer_sizes = [16384, 16384]
output_size = 8192

def get_inputs():
    return [torch.rand(batch_size, input_size)]

def get_init_inputs():
    return [input_size, layer_sizes, output_size]
```

Using `torch.export` to trace the computational graph, we obtain the following functional representation. The rewritten code explicitly materializes all model parameters (weights and biases) and expresses the forward pass as a sequence of functional operators in `fused_operator`. This functional representation aligns with our training data distribution and enables DRTriton to identify optimization opportunities.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

def get_inputs():
    # Initialize weights and biases for first linear layer
    tensor_0 = torch.empty((16384, 16384), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_0, a=2.23606797749979)
    tensor_1 = torch.empty(16384, device=None, dtype=None)
    nn.init.uniform_(tensor_1, -0.0078125, 0.0078125)

    # Initialize weights and biases for second linear layer
    tensor_2 = torch.empty((16384, 16384), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_2, a=2.23606797749979)
    tensor_3 = torch.empty(16384, device=None, dtype=None)
    nn.init.uniform_(tensor_3, -0.0078125, 0.0078125)

    # Initialize weights and biases for output layer
    tensor_4 = torch.empty((8192, 16384), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_4, a=2.23606797749979)
    tensor_5 = torch.empty(8192, device=None, dtype=None)
    nn.init.uniform_(tensor_5, -0.0078125, 0.0078125)

    # Input tensor
    tensor_6 = torch.rand(128, 16384)

    return [tensor_6, tensor_0, tensor_1, tensor_2, tensor_3,
            tensor_4, tensor_5]


def fused_operator(tensor_6, tensor_0, tensor_1, tensor_2,
                   tensor_3, tensor_4, tensor_5):
    # First linear layer + ReLU
    tensor_20 = F.linear(tensor_6, tensor_0, tensor_1)
    tensor_21 = F.relu(tensor_20, inplace=False)

    # Second linear layer + ReLU
    tensor_22 = F.linear(tensor_21, tensor_2, tensor_3)
    tensor_23 = F.relu(tensor_22, inplace=False)

    # Output layer
    tensor_24 = F.linear(tensor_23, tensor_4, tensor_5)

    return [tensor_24]
```

F Evaluation Prompts

For KernelBench evaluation, we use the Triton backend and employ the default prompt provided by the KernelBench pipeline.

For evaluation on our synthetic benchmarks, we use the prompt template shown in the listing below. The prompt specifies detailed requirements for kernel structure, memory efficiency, and performance optimization, and includes a concrete example to guide code generation.

```
## Task: Convert PyTorch to Efficient Triton Kernel

Convert the PyTorch `fused_operator()` function into an **efficient**
Triton GPU kernel that produces identical numerical results while
maximizing performance.

## Requirements

1. **Kernel Structure**: Use `@triton.jit` on **exactly one** kernel function; create
     entrypoint `triton_fused_operator` to launch the kernel. **Only one kernel is
     allowed** - all operations must be
   fused into a single kernel.
2. **Memory Efficiency**:
   - Use `tl.load()`/`tl.store()` with masks for bounds checking
   - Ensure coalesced memory access patterns (contiguous loads/stores)
   - Handle tensor strides correctly to minimize memory transactions
   - Minimize redundant loads/stores
3. **Constants**: Use `tl.constexpr` for block sizes and compile-time
   constants
4. **Grid/Block Optimization**:
   - Calculate grid dimensions from tensor shapes using `triton.cdiv()`
   - Choose block sizes (typically 128-1024) that maximize GPU utilization
5. **Computation Efficiency**:
   - **All operations must be fused into a single kernel** - only one `@triton.jit`
     decorated function is allowed
   - Use vectorized operations when applicable
   - Minimize register pressure by reusing variables
6. **Entrypoint**: Only handle tensor prep, kernel launch, and return results; NO
     computation logic
7. **Output**: Wrap code in `<triton_code>` tags (no markdown blocks); include imports

**Important**: Handle the tensor shapes from `get_inputs()` - use masks to prevent out
     -of-bounds access. **Prioritize efficiency** - the kernel should be optimized for
     GPU execution.

Available triton.language methods: abs, arange, argmax, argmin, atomic_add, cast, cdiv
     , clamp, cos, dot, exp, load, log, max, min, sigmoid, sin, softmax, sqrt, store,
     sum, where, zeros, ...
[full list omitted for brevity]

## Example

**Input (PyTorch):**
def get_inputs():
    tensor_0 = torch.randn([128], dtype=torch.float32)
    tensor_1 = torch.randn([128], dtype=torch.float32)
    return [tensor_0, tensor_1]

def fused_operator(tensor_0, tensor_1):
    tensor_2 = torch.add(tensor_0, tensor_1)
    return [tensor_2]

**Output (Triton):**
<triton_code>
import torch
import triton
import triton.language as tl

@triton.jit
def add_kernel(x_ptr, y_ptr, output_ptr, n_elements,
               BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(axis=0)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
    y = tl.load(y_ptr + offsets, mask=mask, other=0.0)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)

def triton_fused_operator(tensor_0, tensor_1):
    n_elements = tensor_0.numel()
    output = torch.empty_like(tensor_0)
    BLOCK_SIZE = 1024
    grid = (triton.cdiv(n_elements, BLOCK_SIZE),)
    add_kernel[grid](tensor_0, tensor_1, output, n_elements,
                     BLOCK_SIZE=BLOCK_SIZE)
```

```
    return [output]
</triton_code>

## Convert This Code

<torch_code>
{pytorch_impl}
</torch_code>

Generate an **efficient** Triton kernel that:
- Uses **exactly one kernel function** (only one `@triton.jit`
  decorated function)
- Minimizes memory transactions (coalesced access)
- Uses optimal block sizes for the tensor shapes
- Fuses all operations into the single kernel
- Maximizes GPU utilization

Return only the Triton code wrapped in <triton_code> tags.
```

G Case Study on KernelBench

This section demonstrates DRTriton's complete pipeline on a Level 3 KernelBench task: the LeNet-5 architecture. We show three stages of transformation: (1) the original object-oriented PyTorch implementation, (2) the functional IR obtained by tracing the computational graph with `torch.export`, and (3) the optimized Triton kernel produced by DRTriton with test-time search.

G.1 Original PyTorch Code

The following shows the original LeNet-5 implementation from KernelBench, written in standard object-oriented PyTorch style with `nn.Module` and class-based layer definitions.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self, num_classes):
        """
        LeNet-5 architecture implementation in PyTorch.

        :param num_classes: The number of output classes.
        """
        super(Model, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride
    =1)

        # Fully connected layers
        self.fc1 = nn.Linear(in_features=16*5*5, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, out_features=num_classes)

    def forward(self, x):
        """
        Forward pass of the LeNet-5 model.

        :param x: The input tensor, shape (batch_size, 1, 32, 32)
        :return: The output tensor, shape (batch_size, num_classes)
        """
        # First convolutional layer with ReLU activation and max pooling
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        # Second convolutional layer with ReLU activation and max pooling
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        # Flatten the output for the fully connected layers
        x = x.view(-1, 16*5*5)

        # First fully connected layer with ReLU activation
        x = F.relu(self.fc1(x))
```

```
        # Second fully connected layer with ReLU activation
        x = F.relu(self.fc2(x))

        # Final fully connected layer
        x = self.fc3(x)

        return x

# Test code for the LeNet-5 model (larger batch & image)
batch_size = 4096
num_classes = 20

def get_inputs():
    return [torch.rand(batch_size, 1, 32, 32)]

def get_init_inputs():
    return [num_classes]
```

G.2 Functionally Rewritten PyTorch Code

Similar to the example in Appendix E, we transform the original object-oriented code into the following functional format.

```
import torch
import torch.nn
import torch.nn.functional
import math

def get_inputs():
    'Initialize parameters and return tensors needed for forward pass'
    tensor_0 = torch.empty((6, 1, 5, 5), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_0, a=2.23606797749979)
    tensor_1 = torch.empty(6, device=None, dtype=None)
    nn.init.uniform_(tensor_1, -0.2, 0.2)
    tensor_2 = torch.empty((16, 6, 5, 5), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_2, a=2.23606797749979)
    tensor_3 = torch.empty(16, device=None, dtype=None)
    nn.init.uniform_(tensor_3, -0.08164965809277261, 0.08164965809277261)
    tensor_4 = torch.empty((120, 400), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_4, a=2.23606797749979)
    tensor_5 = torch.empty(120, device=None, dtype=None)
    nn.init.uniform_(tensor_5, -0.05, 0.05)
    tensor_6 = torch.empty((84, 120), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_6, a=2.23606797749979)
    tensor_7 = torch.empty(84, device=None, dtype=None)
    nn.init.uniform_(tensor_7, -0.09128709291752768, 0.09128709291752768)
    tensor_8 = torch.empty((20, 84), device=None, dtype=None)
    nn.init.kaiming_uniform_(tensor_8, a=2.23606797749979)
    tensor_9 = torch.empty(20, device=None, dtype=None)
    nn.init.uniform_(tensor_9, -0.1091089451179962, 0.1091089451179962)
    tensor_11 = tensor_0.cuda(None)
    tensor_13 = tensor_1.cuda(None)
    tensor_15 = tensor_2.cuda(None)
    tensor_17 = tensor_3.cuda(None)
    tensor_19 = tensor_4.cuda(None)
    tensor_21 = tensor_5.cuda(None)
    tensor_23 = tensor_6.cuda(None)
    tensor_25 = tensor_7.cuda(None)
    tensor_27 = tensor_8.cuda(None)
    tensor_29 = tensor_9.cuda(None)
    tensor_30 = torch.rand(4096, 1, 32, 32)
    tensor_31 = tensor_30.cuda()
    return [tensor_31, tensor_15, tensor_19, tensor_13, tensor_21, tensor_29,
     tensor_11, tensor_23, tensor_27, tensor_25, tensor_17]

def fused_operator(tensor_31, tensor_15, tensor_19, tensor_13, tensor_21, tensor_29,
     tensor_11, tensor_23, tensor_27, tensor_25, tensor_17):
    'Forward pass operations'
    tensor_32 = F.conv2d(tensor_31, tensor_11, tensor_13, (1, 1), (0, 0), (1, 1), 1)
    tensor_33 = F.relu(tensor_32)
    tensor_34 = F.max_pool2d(tensor_33, kernel_size=2, stride=2)
    tensor_35 = F.conv2d(tensor_34, tensor_15, tensor_17, (1, 1), (0, 0), (1, 1), 1)
    tensor_36 = F.relu(tensor_35)
    tensor_37 = F.max_pool2d(tensor_36, kernel_size=2, stride=2)
    tensor_38 = tensor_37.view(-1, 400)
    tensor_39 = F.linear(tensor_38, tensor_19, tensor_21)
```

```
    tensor_40 = F.relu(tensor_39)
    tensor_41 = F.linear(tensor_40, tensor_23, tensor_25)
    tensor_42 = F.relu(tensor_41)
    tensor_43 = F.linear(tensor_42, tensor_27, tensor_29)
    return [tensor_43]
```

G.3 Optimized Triton Kernel with Test-Time Search

Given the functionally rewritten code, we then apply test-time search to identify the optimal compositional kernel. In this example, the optimal solution fuses the first `F.conv2d` and `F.relu` operations into a single Triton kernel (`_triton_fused_operator`), while keeping the remaining operations in PyTorch.

```
import torch
import triton
import triton.language as tl

@triton.jit
def triton_fused_operator_kernel(
    # Input tensors
    x_ptr,  # Pointer to tensor_31 of shape [B, C_in, H_in, W_in] -> [4096, 1, 32, 32]
    w_ptr,  # Pointer to tensor_11 of shape [C_out, C_in, kH, kW] -> [6, 1, 5, 5]
    t_ptr,  # Pointer to tensor_13 of shape [C_out] -> [6]
    # Output tensor
    output_ptr,  # Pointer to output tensor_33 of shape [B, C_out, H_out, W_out] ->
     [4096, 6, 28, 28]
    B,  # Batch size from output [4096]
    C_in,  # Number of input channels from tensor_31 [1]
    H_in,  # Height of input from tensor_31 [32]
    W_in,  # Width of input from tensor_31 [32]
    C_out,  # Number of output channels from tensor_11 [6]
    kH,  # Height kernel size from tensor_11 [5]
    kW,  # Width kernel size from tensor_11 [5]
    H_out,  # Output height from convolution [28]
    W_out,  # Output width from convolution [28]
    total_output_elements,  # Total output elements (B * C_out * H_out * W_out)
    BLOCK_SIZE: tl.constexpr,  # Processing block size
):
    # Calculate indices for current block
    block_start = tl.program_id(0) * BLOCK_SIZE
    indices = block_start + tl.arange(0, BLOCK_SIZE)
    # Create mask for valid output positions
    mask = indices < total_output_elements
    # Compute output indices
    # Shape of output: [B, C_out, H_out, W_out]
    # Output indices
    b = indices // (C_out * H_out * W_out)  # Batch index
    c_out = (indices // (H_out * W_out) % C_out)  # Channel index
    h_out = (indices // (W_out) % H_out)  # Height index
    w_out = indices % W_out  # Width index

    # Initialize accumulator
    total = tl.full((BLOCK_SIZE,), 0.0, dtype=tl.float32)

    # Loop over kernel
    for c_in in tl.range(0, C_in):
        for kh in tl.range(0, kH):
            for kw in tl.range(0, kW):
                # Compute positions
                # Input indices
                # tensor_31 indices: [B, C_in, H_in, W_in] -> [b, c_in, h_in, w_in]
                # Calculate input positions
                # H_in and W_in indices
                h_in = h_out + (kh - 0)  # (0 padding)
                w_in = w_out + (kw - 0)  # (0 padding)

                # Condition mask for valid input positions
                mask_in = (mask) & (h_in >= 0) & (h_in < H_in) & (w_in >= 0) & (w_in <
     W_in)

                # Step 1: Load input values
                # tensor_31 indices: [B, C_in, H_in, W_in] -> [b, c_in, h_in, w_in]
                # Position
                # tensor_31 index position
                # [4096, 1, 32, 32] -> [b, c_in, h_in, w_in]
```

```
                # tensor_31 index = (b * 1 * 32 * 32) + (c_in * 32 * 32) + (h_in * 32)
     + (w_in)
                x_index = (b * C_in * H_in * W_in) + (c_in * H_in * W_in) + (h_in *
    W_in) + (w_in)
                x = tl.load(x_ptr + x_index, mask=mask_in, other=0.0)

                # Step 2: Load weight values
                # tensor_11 indices: [C_out, C_in, kH, kW] -> [c_out, c_in, kh, kw]
                # Position
                # tensor_11 index = (c_out * C_in * kH * kW) + (c_in * kH * kW) + (kh
    * kW) + (kw)
                w_index = (c_out * C_in * kH * kW) + (c_in * kH * kW) + (kh * kW) + (
    kw)
                w = tl.load(w_ptr + w_index)

                # Update accumulator
                # Update accumulator
                total = total + (tl.where(mask_in, x * w, 0.0))

    # Step 3: Load bias term
    # tensor_13 indices: [C_out] -> [c_out]
    # Position
    # tensor_13 index = (c_out)
    t_index = (c_out)
    t = tl.load(t_ptr + t_index)

    # Update accumulator with bias
    total = total + t

    # Apply ReLU activation
    # ReLU activation: max(0, x)
    relu1 = tl.where(total > 0, total, 0.0)
    relu2 = tl.where(relu1 > 0, relu1, 0.0)

    # Write output
    # Output position
    # tensor_33 indices: [B, C_out, H_out, W_out] -> [b, c_out, h_out, w_out]
    # Output position = (b * C_out * H_out * W_out) + (c_out * H_out * W_out) + (h_out
     * W_out) + (w_out)
    output_index = (b * C_out * H_out * W_out) + (c_out * H_out * W_out) + (h_out *
    W_out) + (w_out)
    # Write output
    tl.store(output_ptr + output_index, relu2, mask=mask)

def _triton_fused_operator(tensor_11: torch.Tensor, tensor_13: torch.Tensor, tensor_31
    : torch.Tensor) -> torch.Tensor:
    # Ensure input tensors are in correct shapes
    assert tensor_11.is_cuda, "tensor_11 must be a CUDA tensor"
    assert tensor_13.is_cuda, "tensor_13 must be a CUDA tensor"
    assert tensor_31.is_cuda, "tensor_31 must be a CUDA tensor"

    # Extract shapes
    B = tensor_31.shape[0]  # Batch size from tensor_31 [4096]
    C_in = tensor_31.shape[1]  # Number of input channels from tensor_31 [1]
    H_in = tensor_31.shape[2]  # Height of input from tensor_31 [32]
    W_in = tensor_31.shape[3]  # Width of input from tensor_31 [32]

    C_out = tensor_11.shape[0]  # Number of output channels from tensor_11 [6]
    kH = tensor_11.shape[2]  # Height kernel size from tensor_11 [5]
    kW = tensor_11.shape[3]  # Width kernel size from tensor_11 [5]

    H_out = (H_in + 2 * 0 - (kH - 1) - 1) + 1  # Output height [28]
    W_out = (W_in + 2 * 0 - (kW - 1) - 1) + 1  # Output width [28]

    # Initialize output tensor
    tensor_33 = torch.empty((B, C_out, H_out, W_out), dtype=torch.float32, device=
    tensor_31.device)

    # Total output elements
    total_output_elements = B * C_out * H_out * W_out

    # Set block size and grid size
    BLOCK_SIZE = 1024  # Optimal for modern GPUs
    grid_size = (triton.cdiv(total_output_elements, BLOCK_SIZE),)

    # Launch kernel
    triton_fused_operator_kernel[grid_size](
        tensor_31,  # tensor_31
        tensor_11,  # tensor_11
        tensor_13,  # tensor_13
        tensor_33,  # tensor_33
```

```
        B=B,
        C_in=C_in,
        H_in=H_in,
        W_in=W_in,
        C_out=C_out,
        kH=kH,
        kW=kW,
        H_out=H_out,
        W_out=W_out,
        total_output_elements=total_output_elements,
        BLOCK_SIZE=BLOCK_SIZE
    )

    return [tensor_33]

def triton_fused_operator(tensor_31, tensor_15, tensor_19, tensor_13, tensor_21,
    tensor_29, tensor_11, tensor_23, tensor_27, tensor_25, tensor_17):
    'Forward pass operations'
    tensor_33, = _triton_fused_operator(tensor_11, tensor_13, tensor_31)
    tensor_34 = F.max_pool2d(tensor_33, kernel_size=2, stride=2)
    tensor_35 = F.conv2d(tensor_34, tensor_15, tensor_17, (1, 1), (0, 0), (1, 1), 1)
    tensor_36 = F.relu(tensor_35)
    tensor_37 = F.max_pool2d(tensor_36, kernel_size=2, stride=2)
    tensor_38 = tensor_37.view(-1, 400)
    tensor_39 = F.linear(tensor_38, tensor_19, tensor_21)
    tensor_40 = F.relu(tensor_39)
    tensor_41 = F.linear(tensor_40, tensor_23, tensor_25)
    tensor_42 = F.relu(tensor_41)
    tensor_43 = F.linear(tensor_42, tensor_27, tensor_29)
    return [tensor_43]
```

H GRPO Objective

Group Relative Policy Optimization (GRPO) [Shao et al., 2024] is a policy optimization method that generates multiple outputs for each input and computes a group-relative advantage function. For a model π_θ and input q, GRPO maximizes:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}_q \mathbb{E}_{\{o_i\}_{i=1}^G \sim \pi_{\text{old}}(\cdot|q)} \left[\frac{1}{G} \sum_{i=1}^{G} \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \min \left(r_{i,t} A(o_i|q), \text{clip}(r_{i,t}, 1-\epsilon, 1+\epsilon) A(o_i|q) \right) \right] - \beta \mathbb{D}_{\text{KL}}(\pi_\theta || \pi_{\text{ref}}) \tag{4}$$

where $r_{i,t} = \frac{\pi_\theta(o_{i,t}|q, o_{i,<t})}{\pi_{\text{old}}(o_{i,t}|q, o_{i,<t})}$ represents the importance sampling ratio, π_{ref} denotes a fixed reference policy, and $\mathbb{D}_{\text{KL}}(\cdot, \cdot)$ is the Kullback-Leibler divergence. The advantage function $A(o_i|q)$ measures the normalized deviation of output o_i's reward from the group mean:

$$A(o_i|q) = \frac{r(o_i|q) - \text{mean}(r(o_1|q), \cdots, r(o_G|q))}{\text{std}(r(o_1|q), \cdots, r(o_G|q))} \tag{5}$$

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state the three main contributions (CSP-DAG, curriculum RL with decoupled rewards, and test-time search), and the experimental results in Section 5 directly support these claims with quantitative results on both synthetic and KernelBench benchmarks.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in the Conclusion: the operator set is currently limited to 61 PyTorch operators, and the framework targets Triton as the compilation backend. Extending to sparse operations, custom CUDA extensions, and native CUDA generation are identified as important future directions.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not include theoretical proofs. The contributions are algorithmic and empirical in nature.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The experimental setup is fully described in Section 5.1, including the base model (Qwen-2.5-Coder-7B-Instruct), training hyperparameters (learning rate, batch size, DRPO parameters), curriculum stages, and evaluation benchmarks. The prompt template used for evaluation is provided in Appendix F.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: We will release the data and code after the acceptance of the paper.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: All training and evaluation details are provided in Section 5.1, including model architecture, optimizer settings, learning rates, batch sizes, DRPO hyperparameters $(\beta_0, \tau, \lambda) = (100, 5, 0.1)$, KL constraint bound $\delta = 0.001$, number of rollouts, and curriculum stage configurations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Error bars and confidence intervals are not reported, as the primary metrics (Pass@1 accuracy, Faster1 percentage, and average speedup) are computed over fixed held-out benchmark sets. Running multiple independent training runs would be computationally prohibitive given the scale of the RL training pipeline.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Training was conducted on 8×H100 GPUs. SFT took approximately 2 hours, and the full curriculum RL pipeline of 100k synthetic programs across three stages takes approximately 10 days in total. CSP-DAG data generation for 100k programs on a 32-core machine requires roughly 1.5 hours. Test-time search overhead is reported in Appendix D.3: approximately 2.85 hours on the synthetic benchmark and 3.53 hours on KernelBench. All training details are provided in Section 5.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research focuses on automated GPU kernel generation for performance optimization and does not involve human subjects, sensitive data, or applications with foreseeable negative societal impact. The work conforms with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The positive societal impact is reducing the cost and expertise barrier for high-performance GPU kernel development, which could democratize access to efficient AI computing. The primary negative risk is potential misuse of automated kernel generation to accelerate development of computationally intensive applications; however, this risk is indirect and not specific to this work.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper presents a model training framework for GPU kernel generation. The trained model does not pose high risk for misuse beyond standard concerns applicable to any code generation system, and no special safeguards beyond standard responsible release practices are required.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All existing assets used in the paper are properly cited. These include the Qwen-2.5-Coder-7B-Instruct base model, the KernelBench benchmark [Ouyang et al., 2025], the DRPO algorithm [Li et al., 2025a], and baseline models (DeepSeek-R1, GPT-5.2, AutoTriton). The CP-SAT solver from OR-Tools is also cited [Perron and Didier].

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The paper introduces a new synthetic PyTorch-Triton dataset of 2,026 pairs and the DRTriton-7B model. Details of the dataset construction process are provided in Section 4.2 and Appendix C.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects. All data is synthetically generated.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects, so IRB approval is not required.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a core component of this research. DeepSeek-R1 and GPT-5.2 are used to generate the SFT training data (PyTorch-Triton pairs), and the trained DRTriton model itself is an LLM fine-tuned for PyTorch-to-Triton kernel conversion. These usages are described in detail in Sections 4.2 and 5.1.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.