DICE: DIFFUSION LARGE LANGUAGE MODELS EXCEL AT GENERATING CUDA KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion large language models (dLLMs) have emerged as a compelling alternative to autoregressive (AR) LLMs, owing to their capacity for parallel token generation. Their capacity for holistic structural planning and non-sequential refinement motivates exploring dLLMs for code generation. Despite this potential, tailoring dLLMs for CUDA kernel generation remains challenging, obstructed not only by the high specialization but also by the severe lack of high-quality training data. To address these challenges, we construct ***CuKe***, an augmented supervised fine-tuning (SFT) dataset optimized for high-performance CUDA kernels. On top of it, we propose a bi-phase curated reinforcement learning (***BiC-RL***) framework consisting of a CUDA kernel infilling stage and an end-to-end CUDA kernel generation stage. With data scheduling during training, the model learns how to write optimized CUDA kernels progressively. Leveraging this training framework, we introduce ***DICE***, a series of diffusion large language models designed for CUDA kernel generation, spanning three parameter scales: 1.7B, 4B, and 8B. Extensive experiments on KernelBench and MultiKernelBench demonstrate that DICE significantly outperforms both autoregressive and diffusion LLMs of comparable scale, establishing a new state-of-the-art for CUDA kernel generation.

1 INTRODUCTION

Large language models (LLMs) have shown powerful ability in a range of language tasks, and the dominant paradigm for LLMs is based on autoregressive modeling (Achiam et al., 2023; Grattafiori et al., 2024; Liu et al., 2024; Guo et al., 2025; Yang et al., 2025; Comanici et al., 2025) for a long time. Recently, diffusion large language models (dLLMs) (Nie et al., 2025; Zhu et al., 2025; Ye et al., 2025; Cheng et al., 2025; Wang et al., 2026; Liu et al., 2025) have emerged as a compelling alternative to autoregressive (AR) LLMs. Different from the token-by-token generation mechanism introduced by AR LLMs, dLLMs mask the generated sequence and then refine the sequence through iterative denoising steps, which can account for global information. This difference is particularly appealing for code generation (Singh et al., 2023; Gong et al., 2026; Xie et al., 2025b; Peng et al., 2025; Labs et al., 2025; Fan et al., 2026), where global structural planning and back-and-forth refinement (Xie et al., 2025a) are often as important as local token prediction (Li et al., 2025d; Ni et al., 2025b).

CUDA kernel generation is a particularly demanding task in code generation. A successfully generated kernel must not only be syntactically and functionally correct but also provide speedup benefits. Recent studies have explored kernel generation primarily with AR LLMs (Ouyang et al., 2025; Baronio et al., 2026; Fisches et al., 2025; Woo et al., 2025; Li et al., 2025c; 2026), but the potential of dLLMs for this task remains largely unexplored. We argue that dLLMs are a promising fit for CUDA kernel generation for the following reasons. ① ***faster inference speed:*** CUDA kernel generation often requires producing long code sequences, making decoding efficiency especially important, while dLLMs provide a clear advantage through parallel generation (DeepMind, 2025; Labs et al., 2025; Song et al., 2025), as illustrated in the left panel of Figure 1. ② ***intrinsic generation characteristics:*** CUDA kernel generation involves strong non-local dependencies across distant code regions, including shared memory usage, synchronization, etc. Such dependencies are not well aligned with strict left-to-right decoding, but fit naturally with the bidirectional attention and iterative refinement of dLLMs. Figure 2 demonstrates the non-sequential generation nature of dLLMs. ③ ***data efficiency:*** high-quality kernel data is scarce (Kong et al., 2025), making data efficiency a central

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

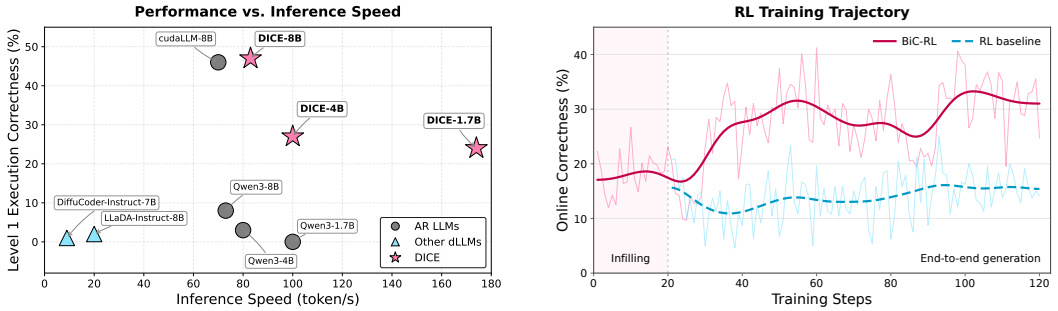


Figure 1: **Left Panel:** Execution correctness on KernelBench Level 1 versus generation throughput for DICE and baseline models. Throughput is measured on a single NVIDIA RTX A6000 with a batch size of 1. Qwen3 and cudaLLM[1] use vLLM (Kwon et al., 2023); LLaDA-Instruct and DiffuCoder-Instruct use dInfer (Ma et al., 2025), and DICE uses JetEngine[2]; **Right Panel:** RL training trajectory comparison of the BiC-RL framework and baseline RL on SDAR-Chat-8B.

concern. Prior studies report advantages of diffusion models when limited pre-training data are reused over sufficiently many epochs (Ni et al., 2025a; Prabhudesai et al., 2025). In our task-specific experiments, SDAR (Cheng et al., 2025) achieves larger average SFT gains than Qwen3 (Yang et al., 2025) on the same CuKe dataset (Appendix A.1), supporting further investigation of diffusion large language models for data-limited CUDA kernel generation.

Motivated by these observations, we introduce ***DICE***, a family of dLLMs tailored for CUDA kernel generation, spanning parameter sizes of 1.7B, 4B, and 8B. Rather than treating diffusion as a drop-in replacement for autoregression, DICE is designed to exploit the specific strengths of dLLMs for this task through task-specific SFT and RL. Our key contributions are summarized as follows: ① We introduce ***CuKe***, a specifically curated supervised fine-tuning dataset that contains a large number of high-performance CUDA kernels; ② We propose ***BiC-RL***, a new reinforcement learning paradigm for the CUDA kernel generation task that features a hierarchical progression from easy to hard across both the CUDA kernel data and task complexity; ③ Leveraging BiC-RL, we present ***DICE***, which is the first specialized dLLM designed for CUDA kernel generation, to our best knowledge; ④ Experiments on KernelBench and MultiKernelBench demonstrate competitive performance against evaluated autoregressive and diffusion LLMs at comparable parameter scales.

2 Preliminary

2.1 Autoregressive Language Models

For a sequence $\mathbf{x} = [x^1, x^2, \ldots, x^L]$, autoregressive LLMs factorize its probability as:

$$p(\mathbf{x}) = \prod_{\ell=1}^{L} p(x^\ell \mid x^{<\ell}; \theta), \tag{1}$$

where θ denotes the model parameters and $x^{<\ell}$ represents the prefix of preceding tokens. During inference, the model predicts x^ℓ conditioned on the previous context in a strictly left-to-right manner. The inherent sequential dependency introduces significant inference latency, particularly for long sequences. Furthermore, this rigid generation process diverges from iterative non-linear human programming workflows (Li et al., 2025a), which necessitate the exploration of alternative generation paradigms that align more effectively with the generation requirements of kernel programming.

2.2 Block Diffusion Language Models

The block diffusion mechanism, also named semi-autoregressive diffusion decoding, has been widely adopted by several works (Arriola et al., 2025; Fathi et al., 2025; Han et al., 2023), as it

[1] https://huggingface.co/ByteDance-Seed/cudaLLM-8B
[2] https://github.com/Labman42/JetEngine

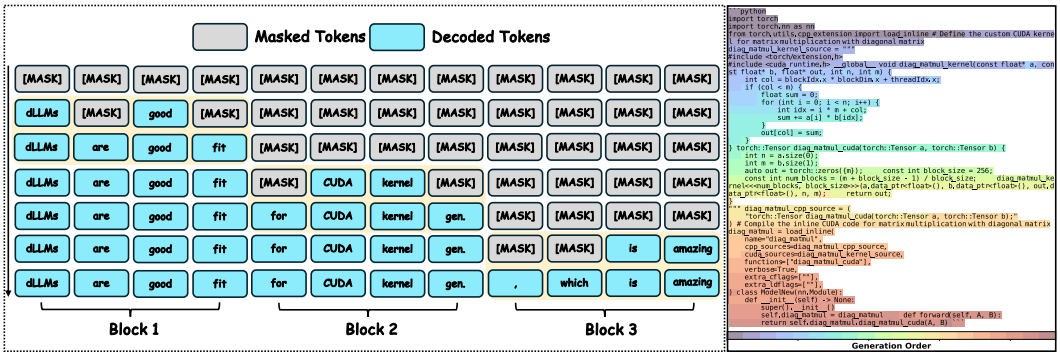


Figure 2: The inference paradigm of dLLMs. **Left Panel**: The sequence is divided into several blocks, where the block length equals four in this figure. The block diffusion mechanism enables models to generate autoregressively between blocks, while performing parallel discrete decoding within blocks. **Right Panel**: An actual generation trajectory for a CUDA kernel. While the overall trend remains autoregressive, we can clearly observe non-autoregressive behavior during the generation process.

enables variable-length generation and the KV cache mechanism. This hybrid approach decomposes a sequence $\mathbf{x}$ into B contiguous, non-overlapping blocks, denoted as $[x^1, x^2, \ldots, x^B]$, where each block contains L' tokens. The model uses autoregressive decoding between blocks and non-autoregressive decoding within blocks (Cheng et al., 2025). Consequently, the training objective is formulated as a weighted cross-entropy loss over these blocks:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim p_{\text{data}}, b\sim U[1,B], t}\left[\frac{1}{t}\sum_{\ell=1}^{L'} \mathbf{1}[x_t^{b,\ell} = \text{[MASK]}] \log p_\theta(x_0^{b,\ell} \mid x_t^b, x^{<b})\right], \tag{2}$$

where L' is the block size, x_t^b represents the corrupted sequence for block b, and $x_0^{b,\ell}$ signifies the ℓ-th original token within that block. Thus, the model is trained to reconstruct the original block x^b from its noised version x_t^b, conditioned on the preceding uncorrupted blocks $x^{<b}$.

As shown in Figure 2, the left part illustrates blockwise decoding and the right part displays a CUDA kernel generation trajectory, with colors indicating token generation order.

3 CuKe Dataset Construction

To address the scarcity of high-performance data in the field of CUDA kernel generation, we introduce ***CuKe***, an augmented dataset that is substantially improved upon the ConCuR dataset (Kong et al., 2025). We enhance the original dataset in the following two dimensions:

High-Performance Samples. Although the cudaLLM SFT dataset[3] contains numerous PyTorch-CUDA pairs, our analysis shows that only 86% of its kernels outperform their PyTorch equivalents (Appendix A.5). Speedups close to $1.0\times$ can be sensitive to kernel launch overhead, cache state, and timing jitter. To favor kernels with clear performance gains, we apply a $2.0\times$ speedup threshold relative to the PyTorch reference. This filter yields 1,425 PyTorch-CUDA pairs of CuKe.

Structural Diversity. To extend CuKe beyond isolated operations, we incorporate diverse structures from mainstream LLMs, such as attention submodules and MLP blocks. We vary their input dimensions because CUDA optimization strategies can change with tensor shape. For each of the 291 curated structural tasks, we use Mercury Coder (Labs et al., 2025) to generate up to five candidate CUDA kernels. We retain implementations that pass functional validation and run faster than their PyTorch references. The resulting 36 pairs, together with 4,842 ConCuR pairs and 1,425 high-performance pairs, form the 6,303-sample CuKe dataset.

[3] https://huggingface.co/datasets/ByteDance-Seed/cudaLLM-data/blob/main/sft_cuda_llm_r1.parquet

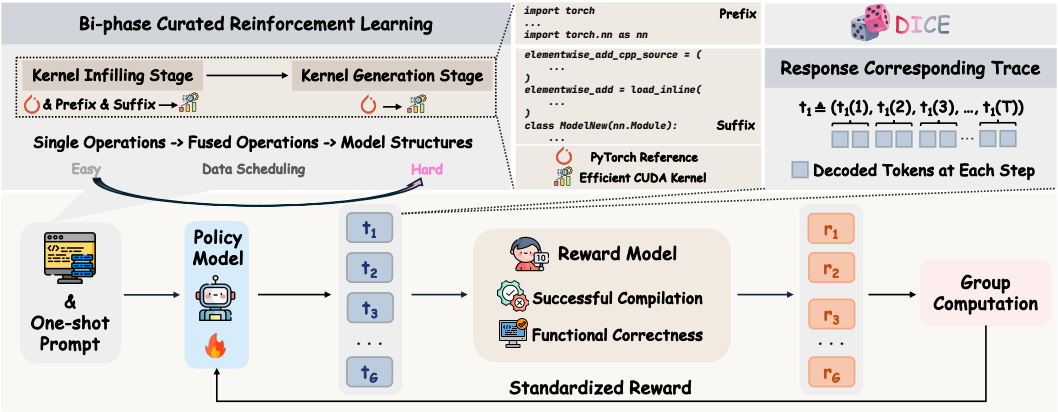


Figure 3: Overview of **DICE**. The **Bi-phase Curated Reinforcement Learning (BiC-RL)** framework first trains the model to complete the CUDA kernel core within a fixed scaffold, then to generate the whole kernel. Across both stages, **Data Scheduling** increases task complexity from single operations through fused operations to whole model structures. Together, these components aim to improve the functional correctness and runtime performance of generated CUDA kernels.

4 DICE

In this section, we formalize bi-phase curated reinforcement learning (***BiC-RL***). Leveraging this hierarchical paradigm and data scheduling, we develop ***DICE***, a series of dLLMs specially designed for CUDA kernel generation. An overview of the end-to-end framework is illustrated in Figure 3.

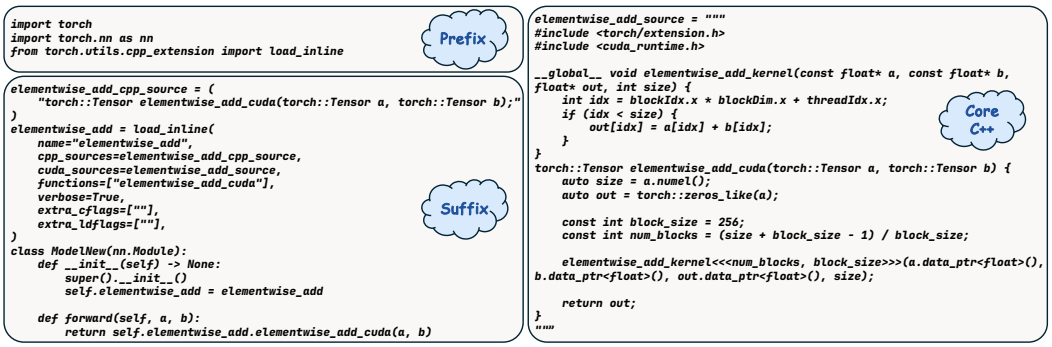
4.1 Supervised Fine-tuning

Empirical results indicate that even state-of-the-art reasoning models, such as Qwen3 (Yang et al., 2025) and Seed-Coder-Reasoning (Seed et al., 2025), struggle to generate functionally correct CUDA kernels. This performance gap highlights the difficulty of this task. Consequently, incorporating supervised fine-tuning as a cold-start initialization phase is essential to establish the necessary foundational alignment. Specifically, we optimize the model with the objective defined in Equation 2.

4.2 BiC-RL

To improve CUDA kernel generation with DICE and reduce reward hacking, we introduce BiC-RL, building on TraceRL (Wang et al., 2026). BiC-RL organizes training along two dimensions. In the data dimension, we leverage Mercury Coder (Labs et al., 2025) to categorize examples into difficulty levels from basic operations to complex model structures. In the training dimension, the model first fills in the core implementation of a CUDA kernel, then generates the complete kernel end to end. This sequence makes the model focus on core kernel logic before tackling the full generation task.

Kernel Infilling Stage. Recent studies (Baronio et al., 2026; Li et al., 2025c; Woo et al., 2025) have highlighted reward hacking in kernel generation. We focus on three common patterns in which generated kernels bypass the intended CUDA implementation during training. ① The model reuses the prompt-provided example kernel but performs the forward computation using high-level PyTorch operations. ② The model generates a custom CUDA extension but neither stores it as an attribute of `ModelNew` nor invokes its kernels in `forward`. ③ The model generates a custom CUDA extension and stores it as an attribute of `ModelNew`, but never invokes its kernels in `forward`. To constrain these shortcuts during training, we introduce a kernel infilling stage with a fixed scaffold that provides the surrounding invocation logic and calls the generated CUDA implementation. Representative examples of these patterns are provided in Appendix A.7.1.




```
import torch
import torch.nn as nn
from torch.utils.cpp_extension import load_inline
```

```
elementwise_add_cpp_source = (
    "torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b);"
)
elementwise_add = load_inline(
    name="elementwise_add",
    cpp_sources=elementwise_add_cpp_source,
    cuda_sources=elementwise_add_source,
    functions=["elementwise_add_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)
class ModelNew(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.elementwise_add = elementwise_add

    def forward(self, a, b):
        return self.elementwise_add.elementwise_add_cuda(a, b)
```

```
elementwise_add_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void elementwise_add_kernel(const float* a, const float* b,
float* out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        out[idx] = a[idx] + b[idx];
    }
}
torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b) {
    auto size = a.numel();
    auto out = torch::zeros_like(a);

    const int block_size = 256;
    const int num_blocks = (size + block_size - 1) / block_size;

    elementwise_add_kernel<<<num_blocks, block_size>>>(a.data_ptr<float>(),
b.data_ptr<float>(), out.data_ptr<float>(), size);

    return out;
}
"""
```

Figure 4: Our defined CUDA kernel components: the prefix, the suffix, and the core implementation, which is a C++ snippet. Each component varies according to the exact CUDA kernels.

Specifically, we decompose the CUDA kernel into a tripartite structure: the prefix, the core C++ snippet, and the suffix. As illustrated in Figure 4, the prefix establishes the execution environment and global dependencies, primarily consisting of library imports that define the necessary namespace. The suffix provides the function declarations, compilation and binding logic, and the PyTorch module that invokes the generated CUDA implementation. The core implementation part is the main logic of the CUDA kernel that the model needs to generate. This scaffold narrows the generation task to the kernel core and restricts opportunities to bypass the intended CUDA implementation.

Kernel Generation Stage. After kernel infilling, we proceed to the end-to-end CUDA generation stage of BiC-RL. Given a high-level PyTorch reference, the model must generate a complete CUDA implementation, including both the core snippet computation and all necessary invocation logic. Rather than completing the kernel within a supplied code context, the model now learns to construct the implementation as a whole. Training continues from the infilling, building on the previously acquired implementation skills while broadening the scope of the generation task.

Reward Function. Both training stages use a binary correctness reward. For each problem, we sample G candidate kernels $\{t_i\}_{i=1}^{G}$ and assign $s_i = \mathbb{1}[\text{correct}(t_i)]$, where correctness requires successful compilation and execution with outputs equivalent to the PyTorch reference. Rewards are standardized within each group as $r_i = \frac{s_i - \mu_G}{\sigma_G}$, where μ_G and σ_G are the mean and standard deviation over the group. SFT on performance-curated CuKe provides examples of accelerated CUDA implementations. Building on this initialization, BiC-RL aims to improve functional correctness without directly rewarding the execution speedup of generated kernels.

Data Scheduling. We notice the optimization plateaus in baseline RL, as shown in the right panel of Figure 1. We assume the main reason is that there is a large exploration space during the RL training, and the difficulty of each group of problems varies greatly. Encountering difficult problems in the early stages of training will lead to fewer effective rewards and lower training efficiency. Hence, inspired by curriculum learning (Bengio et al., 2009), we propose data scheduling, transitioning training data from fundamental operations to complex model structures during the two RL stages.

5 Experiments

5.1 Training and Evaluation Setups

Training Details. We initialize our training from SDAR (Cheng et al., 2025), utilizing our curated CuKe dataset for the SFT phase. For the RL phase, we sample PyTorch code from the cudaLLM RL dataset[4]: 992 programs are selected for the kernel infilling stage, while 4,000 programs are utilized for the end-to-end kernel generation stage. Training and evaluation experiments are conducted on 8 NVIDIA A100 GPUs. Detailed training configurations are provided in Appendix A.4.

Evaluation Benchmark and Metrics. We evaluate DICE on KernelBench (Ouyang et al., 2025), which contains 250 PyTorch-to-CUDA tasks across three levels: 100 single operators, 100 fusion

[4] https://huggingface.co/datasets/ByteDance-Seed/cudaLLM-data/blob/main/rl_cuda_llm_0424.parquet

Table 1: Main results on KernelBench (Ouyang et al., 2025) across 8B and similar scale models. We report execution correctness (Exec) and speedup ($fast_1$ and $fast_2$) as percentages (%). We compare Autoregressive and Diffusion LLMs, featuring general, code, and reasoning models. The best and second-best results are highlighted in **bold** and underline, respectively. Gray results are for reference.

Model	Type	#Params	LEVEL1		LEVEL2		LEVEL3	
			Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
Autoregressive LLMs								
Gemini-3-Pro (Google DeepMind, 2025)	General	-	14.0	6.0 / 5.0	14.0	7.0 / 4.0	8.0	4.0 / 2.0
Claude-Sonnet-4[5]	Code	-	44.0	10.0 / 6.0	48.0	24.0 / 18.0	4.0	0.0 / 0.0
CUDA-L1 (Li et al., 2026)	Kernel (CUDA)	671B	65.0	62.0 / 60.0	16.0	13.0 / 5.0	28.0	22.0 / 16.0
Kevin (Baronio et al., 2026)	Kernel (CUDA)	32B	47.0	18.0 / 9.0	63.0	22.0 / 11.0	44.0	10.0 / 6.0
Deepseek-Coder-Instruct-v1.5 (Guo et al., 2024)	Code	7B	13.0	2.0 / 0.0	7.0	2.0 / 1.0	0.0	0.0 / 0.0
Qwen2.5-Coder-Instruct (Hui et al., 2024)	Code	7B	15.0	1.0 / 1.0	0.0	0.0 / 0.0	4.0	0.0 / 0.0
Seed-Coder-Reasoning (Seed et al., 2025)	Code	8B	0.0	0.0 / 0.0	2.0	0.0 / 0.0	0.0	0.0 / 0.0
Qwen3 (Yang et al., 2025)	Reasoning (General)	8B	8.0	1.0 / 0.0	25.0	11.0 / 6.0	4.0	2.0 / 2.0
cudaLLM[1]	Reasoning (CUDA)	8B	46.0	12.0 / **7.0**	34.0	12.0 / **6.0**	6.0	4.0 / 0.0
Diffusion LLMs								
Mercury-Coder-Small (Labs et al., 2025)	Code	-	26.0	16.0 / 4.0	44.0	22.0 / 9.0	24.0	12.0 / 10.0
LLaDA-Instruct (Nie et al., 2025)	General	8B	2.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
LLaDA-1.5 (Zhu et al., 2025)	General	8B	1.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Dream-v0-Instruct (Ye et al., 2025)	General	7B	2.0	0.0 / 0.0	0.0	0.0 / 0.0	2.0	2.0 / 2.0
SDAR-Chat (Cheng et al., 2025)	General	8B	6.0	2.0 / 1.0	2.0	0.0 / 0.0	0.0	0.0 / 0.0
WeDLM (Liu et al., 2025)	General	8B	14.0	1.0 / 1.0	1.0	0.0 / 0.0	0.0	0.0 / 0.0
DiffuCoder-Instruct (Gong et al., 2026)	Code	7B	1.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Dream-Coder-v0-Instruct (Xie et al., 2025b)	Code	7B	1.0	1.0 / 0.0	8.0	8.0 / 5.0	6.0	0.0 / 0.0
Stable-DiffCoder (Fan et al., 2026)	Code	8B	27.0	7.0 / 4.0	5.0	4.0 / 1.0	10.0	6.0 / 2.0
TraDo-Instruct (Wang et al., 2026)	Code & Math	8B	4.0	2.0 / 0.0	2.0	0.0 / 0.0	6.0	4.0 / 0.0
DICE	Kernel (CUDA)	8B	**47.0**	**15.0** / 6.0	**35.0**	**19.0** / 1.0	**14.0**	**6.0 / 6.0**

Table 2: Main results on KernelBench (Ouyang et al., 2025) across 4B and similar scale models. We report execution correctness (Exec) and speedup ($fast_1$ and $fast_2$) as percentages (%). We compare Autoregressive and Diffusion LLMs, featuring general, code, and reasoning models. The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Type	#Params	LEVEL1		LEVEL2		LEVEL3	
			Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
Llama-3.2-Instruct	General	3B	1.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
SDAR-Chat (Cheng et al., 2025)	General	4B	5.0	0.0 / 0.0	1.0	0.0 / 0.0	4.0	0.0 / 0.0
TraDo-Instruct (Wang et al., 2026)	Code & Math	4B	1.0	1.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Qwen2.5-Coder-Instruct (Hui et al., 2024)	Code	3B	0.0	0.0 / 0.0	1.0	1.0 / 1.0	0.0	0.0 / 0.0
Phi-4-Mini-Reasoning (Xu et al., 2025)	Reasoning	4B	0.0	0.0 / 0.0	0.0	0.0 / 0.0	2.0	0.0 / 0.0
Qwen3 (Yang et al., 2025)	Reasoning	4B	3.0	2.0 / 2.0	**29.0**	14.0 / 4.0	**14.0**	2.0 / 0.0
DICE	Kernel (CUDA)	4B	**27.0**	**13.0 / 4.0**	24.0	**15.0 / 9.0**	12.0	**6.0 / 4.0**

patterns, and 50 full model architectures. We also report results on the CUDA backend of MultiKernelBench (Wen et al., 2025). While the original benchmark comprised 285 tasks across 14 functional categories, we evaluate its updated version, which adds 15 attention tasks and contains 300 tasks. For KernelBench, we report execution correctness (Exec) and $fast_p$, the fraction of tasks yielding correct kernels with speedup greater than p over PyTorch. On MultiKernelBench, Compilation@1, Pass@1, and $SpeedUp_1$@1 measure compilation success, functional correctness, and the fraction of tasks achieving at least $1\times$ speedup over PyTorch, respectively.

Evaluation Details. We compare DICE with general, code, and reasoning models from both AR and diffusion families at 1.7B, 4B, and 8B scales. CUDA-specialized baselines include CUDA-L1 (Li et al., 2026), Kevin-32B (Baronio et al., 2026), and cudaLLM[1]. Commercial baselines include Gemini-3-Pro (Google DeepMind, 2025), Claude-Sonnet-4[5], and Mercury-Coder-Small (Labs et al., 2025). We retain default inference configurations, adjusting only maximum output lengths. AR reasoning models receive a 32,768-token budget, whereas general and code models receive 4,096 tokens. For fixed-length dLLMs, including LLaDA (Nie et al., 2025; Zhu et al., 2025), Dream (Ye et al., 2025; Xie et al., 2025b), and DiffuCoder (Gong et al., 2026), we use a 1,024-token limit. For dLLMs with adaptive stopping, we use a 4,096-token limit. These include SDAR (Cheng et al., 2025), TraDo (Wang et al., 2026), WeDLM (Liu et al., 2025), Stable-DiffCoder (Fan et al., 2026), and DICE. Commercial models use the default configurations specified in their official documentation. More evaluation details are provided in Appendix A.4.3.

[5] https://www.anthropic.com/news/claude-4

Table 3: Main results on KernelBench (Ouyang et al., 2025) across 1.7B and similar scale models. We report execution correctness (Exec) and speedup ($fast_1$ and $fast_2$) as percentages (%). We compare Autoregressive and Diffusion LLMs, featuring general, code, and reasoning models. The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Type	#Params	LEVEL1		LEVEL2		LEVEL3	
			Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
Gemma-2-It (Team et al., 2024)	General	2B	0.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
SDAR-Chat (Cheng et al., 2025)	General	1.7B	0.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Qwen2.5-Coder-Instruct (Hui et al., 2024)	Code	1.5B	0.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Deepseek-Coder-Instruct (Guo et al., 2024)	Code	1.3B	0.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Qwen3 (Yang et al., 2025)	Reasoning	1.7B	0.0	0.0 / 0.0	1.0	0.0 / 0.0	0.0	0.0 / 0.0
DICE	Kernel (CUDA)	1.7B	**24.0**	**12.0 / 3.0**	**6.0**	**3.0 / 2.0**	**10.0**	**6.0 / 4.0**

Table 4: Evaluation results on MultiKernelBench (Wen et al., 2025) across 8B and similar scale models. We report Compilation@1 (Comp@1), Pass@1, and $SpeedUp_1$@1 (SU_1@1) as percentages (%). The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Type	#Params	Comp@1	Pass@1	SU_1@1
Deepseek-Coder-Instruct-v1.5 (Guo et al., 2024)	Code	7B	58.3	1.3	0.7
Qwen3 (Yang et al., 2025)	Reasoning (General)	8B	47.7	0.7	0.7
cudaLLM[1]	Reasoning (CUDA)	8B	**96.0**	0.3	0.0
LLaDA-Instruct (Nie et al., 2025)	General	8B	38.0	2.0	1.3
SDAR-Chat (Cheng et al., 2025)	General	8B	82.0	1.7	0.3
Stable-DiffCoder (Fan et al., 2026)	Code	8B	80.7	2.7	2.7
DICE	Kernel (CUDA)	8B	93.3	**3.3**	**3.0**

5.2 MAIN RESULTS

Results on KernelBench. Tables 1, 2, and 3 compare DICE with AR and diffusion LLMs at comparable parameter scales. DICE-8B and DICE-1.7B achieve the highest execution correctness across all three levels within their respective scale groups. Notably, DICE-1.7B reaches 24.0% execution correctness on level 1, where all evaluated baselines score 0.0%, demonstrating functional kernel generation at a compact scale. DICE-4B leads in level 1 execution correctness and $fast_2$ across all three levels. On level 2, DICE-8B exceeds cudaLLM in execution correctness and $fast_1$. Appendix A.2 provides pass@10 results and further comparisons with cudaLLM, respectively.

Results on MultiKernelBench. On MultiKernelBench (Table 4), DICE-8B achieves the highest Pass@1 (3.3%) and SU_1@1 (3.0%) among the evaluated models, compared with 2.7% on both metrics for Stable-DiffCoder (Fan et al., 2026). Its compilation success rate is 93.3%. These results provide additional evidence for the effectiveness of DICE beyond KernelBench.

5.3 ABLATION STUDIES

Analysis of Data Selection Speedup Threshold. We examine how the speedup threshold used to curate CuKe affects downstream kernel generation performance. We fine-tune SDAR-Chat-4B (Cheng et al., 2025) on dataset variants filtered at 1.5×, 2.0×, and 2.5× speedup, containing 7,940, 6,303, and 5,685 samples, respectively. As shown in Table 5, the 2.0× variant achieves stronger overall results on levels 1 and 2, whereas the alternative thresholds yield better level 3 results. These findings support 2.0× as a practical threshold for CuKe curation.

Analysis of Supervised Fine-tuning Dataset. We compare supervised fine-tuning of SDAR-Chat-8B (Cheng et al., 2025) using the cudaLLM SFT dataset[4], the original ConCuR dataset (Kong et al., 2025), and our curated CuKe dataset. Table 6 shows that CuKe yields stronger overall performance than the two alternative datasets. These results support the effectiveness of our data curation strategy and highlight the value of high-performance training examples for CUDA kernel generation.

Effectiveness of BiC-RL Framework. The training trajectories in the right panel of Figure 1 show sustained improvement with BiC-RL compared with the RL baseline. The kernel infilling stage prepares the model for end-to-end CUDA kernel generation, supporting progressive learning through the two-stage process. Tables 7 and 13 demonstrate the benefits of BiC-RL across model scales. SFT

Table 5: Performance comparison of SDAR-Chat-4B trained on CuKe dataset variants selected by different speedup thresholds (1.5×, 2.0×, and 2.5×). The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Threshold	Size	Level 1		Level 2		Level 3	
			Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
SDAR-Chat-4B	1.5×	7,940	26.0	14.0 / 3.0	9.0	5.0 / 2.0	**18.0**	**6.0 / 4.0**
	2.0×	6,303	**29.0**	**17.0 / 6.0**	**21.0**	**16.0 / 5.0**	10.0	2.0 / 0.0
	2.5×	5,685	27.0	14.0 / 5.0	17.0	13.0 / 5.0	16.0	6.0 / 2.0

Table 6: Performance comparison of SDAR-Chat-8B trained on the SFT dataset of cudaLLM, ConCuR, and our curated CuKe datasets. The best and second-best results are highlighted in **bold** and underline, respectively.

Dataset	Size	Level 1		Level 2		Level 3	
		Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
cudaLLM	8,920	24.0	15.0 / **5.0**	7.0	5.0 / 1.0	2.0	2.0 / 0.0
ConCuR	4,842	31.0	15.0 / 2.0	16.0	**12.0 / 9.0**	12.0	0.0 / 0.0
CuKe	6,303	**34.0**	**16.0** / 4.0	**18.0**	11.0 / 8.0	**16.0**	**2.0** / 0.0

Table 7: Ablation results for SDAR-Chat-8B (Cheng et al., 2025) under different training configurations. The best and second-best results are highlighted in **bold** and underline, respectively. All metrics are reported as percentages (%).

Model	Level 1		Level 2		Level 3	
	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
SDAR-Chat-8B	6.0	2.0 / 1.0	2.0	0.0 / 0.0	0.0	0.0 / 0.0
+ SFT	34.0	16.0 / 4.0	18.0	11.0 / **8.0**	16.0	2.0 / 0.0
+ SFT → RL (baseline)	31.0	13.0 / 4.0	18.0	9.0 / 6.0	18.0	6.0 / 4.0
+ SFT → RL (*with data scheduling*)	41.0	**16.0** / 5.0	21.0	14.0 / 2.0	**18.0**	**8.0** / 4.0
+ SFT → kernel infilling	37.0	13.0 / 4.0	16.0	10.0 / 2.0	16.0	6.0 / 4.0
DICE-8B	**47.0**	15.0 / **6.0**	**35.0**	**19.0** / 1.0	14.0	6.0 / **6.0**

establishes basic kernel generation capabilities, particularly for the 1.7B model, which produces no functionally correct kernels before SFT. The ablations support the contributions of kernel infilling and data scheduling, with the complete DICE models outperforming the RL baseline on most metrics.

Comparison of Generative Paradigms. To examine the choice of generation paradigm, we apply the same training pipeline to the autoregressive Qwen3 series. Both families are fine-tuned on CuKe (Appendix A.1). For RL, Qwen3 is optimized with GRPO (Shao et al., 2024) from TRL (von Werra et al., 2020), while all other BiC-RL settings are kept identical. As shown in Table 8, DICE achieves higher execution correctness than its Qwen3 counterpart in 8 of the 9 scale–level settings, with the largest margins on level 2. These results support the suitability of dLLMs for CUDA kernel generation under matched training conditions.

6 Related Work

Diffusion Large Language Models. The paradigm of large language models has been dominated by autoregressive (Achiam et al., 2023; Liu et al., 2024; Yang et al., 2025) for a long time, but the emergence of diffusion large language models (Nie et al., 2025; Zhu et al., 2025; DeepMind, 2025; Labs et al., 2025; Ye et al., 2025; Wang et al., 2026; Bie et al., 2025; Liu et al., 2025) has broken this, offering a new choice for language models. Unlike autoregressive methods, which use a strict left-to-right, token-by-token decoding process, dLLMs first generate multiple masks, then gradually denoise the masked sequence during decoding, which allows for simultaneous sampling and decoding of multiple tokens. Moreover, dLLMs employ a bidirectional attention mechanism to more comprehensively understand the context. LLaDA (Nie et al., 2025) is an 8B-parameter diffusion language model trained from scratch, whereas Dream (Ye et al., 2025) is a 7B-parameter model initialized from pre-trained autoregressive weights. SDAR (Cheng et al., 2025) synergistically combines the advantages of diffusion and autoregressive. TraDo (Wang et al., 2026) proposes TraceRL, deriving the first long-CoT dLLM, which achieves strong performance over models of comparable parameter size. Recently, WeDLM (Liu et al., 2025) represents the first dLLM to surpass industrial autoregressive engines while maintaining high performance on reasoning tasks.

dLLMs for Code Generation. Diffusion-based approaches to code generation have been explored across different model scales and training strategies. CodeFusion (Singh et al., 2023) is an early diffusion-based model for natural-language-to-code generation, applying Gaussian diffusion to continuous code embeddings in a 75M-parameter architecture. More recent code-focused dLLMs,

Table 8: Comparison of Qwen3 (Yang et al., 2025) and SDAR-Chat (Cheng et al., 2025) on KernelBench (Ouyang et al., 2025) after SFT training on CuKe and subsequent BiC-RL. DICE denotes SDAR-Chat after the full training pipeline. All metrics are reported as percentages (%). The higher value within each parameter scale and training stage is highlighted in **bold**.

#Params	Training	Model	Level 1		Level 2		Level 3	
			Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
1.7B	SFT	Qwen3	6.0	2.0 / 2.0	6.0	1.0 / 1.0	8.0	2.0 / 0.0
		SDAR-Chat	**16.0**	**7.0 / 3.0**	**6.0**	**3.0 / 1.0**	**8.0**	**4.0 / 2.0**
	SFT + BiC-RL	Qwen3	17.0	12.0 / **4.0**	5.0	1.0 / 0.0	6.0	4.0 / 0.0
		DICE	**24.0**	**12.0** / 3.0	**6.0**	**3.0 / 2.0**	**10.0**	**6.0 / 4.0**
4B	SFT	Qwen3	9.0	0.0 / 0.0	13.0	3.0 / 0.0	10.0	**6.0** / 0.0
		SDAR-Chat	**29.0**	**17.0 / 6.0**	**21.0**	**16.0 / 5.0**	**10.0**	2.0 / 0.0
	SFT + BiC-RL	Qwen3	21.0	10.0 / 3.0	11.0	8.0 / 4.0	8.0	6.0 / 2.0
		DICE	**27.0**	**13.0 / 4.0**	**24.0**	**15.0 / 9.0**	**12.0**	**6.0 / 4.0**
8B	SFT	Qwen3	25.0	13.0 / 3.0	13.0	7.0 / 4.0	16.0	**6.0** / 0.0
		SDAR-Chat	**34.0**	**16.0 / 4.0**	**18.0**	**11.0 / 8.0**	**16.0**	2.0 / 0.0
	SFT + BiC-RL	Qwen3	34.0	13.0 / **7.0**	14.0	11.0 / **6.0**	**18.0**	**10.0** / 0.0
		DICE	**47.0**	**15.0** / 6.0	**35.0**	**19.0** / 1.0	14.0	6.0 / **6.0**

including DiffuCoder (Gong et al., 2026), Dream-Coder (Xie et al., 2025b), and Stable-DiffCoder (Fan et al., 2026), instead adopt masked discrete diffusion over tokens. Open-dCoder (Peng et al., 2025) provides an open 0.5B-scale model and training pipeline. Commercial models, including Gemini Diffusion (DeepMind, 2025), Mercury Coder (Labs et al., 2025), and Seed Diffusion (Song et al., 2025), report competitive coding results and high generation throughput. Despite substantial progress in general code generation, tailoring dLLMs for CUDA kernel generation has received little attention.

LLM Kernel Generation. The advancement of LLMs has shifted the focus from general programming to hardware-specific tasks, kernel generation. Central to this progress is the development of specialized benchmarks, such as KernelBench (Ouyang et al., 2025), MultiKernelBench (Wen et al., 2025), TritonBench (Li et al., 2025b), and FlashInfer-Bench (Xing et al., 2026), which provide standardized environments for evaluating functional correctness and execution efficiency. Beyond evaluation, recent research emphasizes improving generation quality through sophisticated training paradigms and reasoning alignment. Specifically, Kevin (Baronio et al., 2026), TritonRL (Woo et al., 2025), and AutoTriton (Li et al., 2025c) employ reinforcement learning to navigate models to effectively generate high-performance GPU kernels. Recently, agentic methods have become increasingly popular, ranging from iterative refinement and search-based systems such as Astra (Wei et al., 2025), CudaForge (Zhang et al., 2025), and KernelEvolve (Liao et al., 2025) to agentic RL systems like CUDA Agent (Dai et al., 2026). To address the scarcity of high-quality kernels, Kong et al. (2025) introduce ConCuR, an SFT dataset for CUDA kernel generation. Despite these advances, adapting dLLMs to this challenging task remains underexplored.

7 Conclusion

We introduce ***DICE***, a family of dLLMs for CUDA kernel generation at 1.7B, 4B, and 8B parameter scales. DICE combines supervised fine-tuning on our curated ***CuKe*** dataset with ***BiC-RL***, a reinforcement learning framework that progresses from kernel infilling to end-to-end kernel generation with data scheduling. Experiments demonstrate competitive performance against autoregressive and diffusion baselines at comparable parameter scales. Dataset comparisons further show that CuKe yields stronger overall performance than the original ConCuR and cudaLLM SFT datasets, supporting the effectiveness of targeted data curation. Comparisons under the same training pipeline further show higher execution correctness than Qwen3 in most evaluated settings. Together, these findings support the potential of dLLMs for CUDA kernel generation through the combination of curated training data and progressive reinforcement learning.

AI USE STATEMENT

LLMs are used in this work as follows. (1) As part of the method: Mercury Coder is used to generate candidate CUDA kernels for the structural samples in CuKe (Section 3) and to categorize RL training examples by difficulty (Section 4.2). (2) As writing assistance: LLMs are used to polish grammar and wording. LLMs are not used for research ideation, experimental design, or result analysis. The authors review all LLM-assisted content and take full responsibility for this work.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Marianne Arriola, Aaron Gokaslan, Justin T Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *ICLR*, 2025.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. In *ICLR*, 2026.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *ICML*, 2009.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, et al. Llada2. 0: Scaling up diffusion language models to 100b. *arXiv preprint arXiv:2512.15745*, 2025.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, et al. Sdar: A synergistic diffusion-autoregression paradigm for scalable sequence generation. *arXiv preprint arXiv:2510.06303*, 2025.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Google DeepMind. Gemini diffusion, 2025. URL https://blog.google/technology/google-deepmind/gemini-diffusion.

Chenghao Fan, Wen Heng, Bo Li, Sichen Liu, Yuxuan Song, Jing Su, Xiaoye Qu, Kai Shen, and Wei Wei. Stable-diffcoder: Pushing the frontier of code diffusion large language model. *arXiv preprint arXiv:2601.15892*, 2026.

Nima Fathi, Torsten Scholak, and Pierre-Andre Noel. Unifying autoregressive and diffusion-based sequence generation. In *COLM*, 2025.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm: Making kernel development more accessible, 2025. URL https://huggingface.co/facebook/KernelLLM.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. In *ICLR*, 2026.

Google DeepMind. Gemini 3 pro model card. Model card, Google DeepMind, November 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

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.

Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. Ssd-lm: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. In *ACL*, 2023.

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

Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. Concur: Conciseness makes state-of-the-art kernel generation. *arXiv preprint arXiv:2510.07356*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023.

Inception Labs, Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, et al. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*, 2025.

Chengze Li, Yitong Zhang, Jia Li, Liyi Cai, and Ge Li. Beyond autoregression: An empirical study of diffusion large language models for code generation. *arXiv preprint arXiv:2509.11252*, 2025a.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *ACL Findings*, 2025b.

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*, 2025c.

Tianyi Li, Mingda Chen, Bowei Guo, and Zhiqiang Shen. A survey on diffusion language models. *arXiv preprint arXiv:2508.10875*, 2025d.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. In *ICLR*, 2026.

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.

Aiwei Liu, Minghua He, Shaoxun Zeng, Sijun Zhang, Linhao Zhang, Chuhan Wu, Wei Jia, Yuan Liu, Xiao Zhou, and Jie Zhou. Wedlm: Reconciling diffusion language models with standard causal attention for fast inference. *arXiv preprint arXiv:2512.22737*, 2025.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Yuxin Ma, Lun Du, Lanning Wei, Kun Chen, Qian Xu, Kangyu Wang, Guofeng Feng, Guoshan Lu, Lin Liu, Xiaojing Qi, et al. dinfer: An efficient inference framework for diffusion language models. *arXiv preprint arXiv:2510.08666*, 2025.

Jinjie Ni, Qian Liu, Longxu Dou, Chao Du, Zili Wang, Hang Yan, Tianyu Pang, and Michael Qizhe Shieh. Diffusion language models are super data learners. *arXiv preprint arXiv:2511.03276*, 2025a.

Jinjie Ni, Qian Liu, Chao Du, Longxu Dou, Hang Yan, Zili Wang, Tianyu Pang, and Michael Qizhe Shieh. Training optimal large diffusion language models. *arXiv preprint arXiv:2510.03280*, 2025b.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. In *NeurIPS*, 2025.

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

Zhangzhi Peng, Shuibai Zhang, Alex Tong, and contributors. Open-dllm: Open diffusion large language models, 2025. URL https://github.com/pengzhangzhi/Open-dLLM.

Mihir Prabhudesai, Mengning Wu, Amir Zadeh, Katerina Fragkiadaki, and Deepak Pathak. Diffusion beats autoregressive in data-constrained settings. In *NeurIPS*, 2025.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

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

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. Codefusion: A pre-trained diffusion model for code generation. In *EMNLP*, 2023.

Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, et al. Seed diffusion: A large-scale diffusion language model with high-speed inference. *arXiv preprint arXiv:2508.02193*, 2025.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020. URL https://github.com/huggingface/trl.

Yinjie Wang, Ling Yang, Bowen Li, Ye Tian, Ke Shen, and Mengdi Wang. Revolutionizing reinforcement learning framework for diffusion large language models. In *ICLR*, 2026.

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. In *NeurIPS Workshop*, 2025.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025.

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.

Zhihui Xie, Jie Chen, Liyu Chen, Weichao Mao, Jingjing Xu, and Lingpeng Kong. Teaching language models to critique via reinforcement learning. In *ICML*, 2025a.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-coder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025b.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, et al. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *arXiv preprint arXiv:2601.00227*, 2026.

Haoran Xu, Baolin Peng, Hany Awadalla, Dongdong Chen, Yen-Chun Chen, Mei Gao, Young Jin Kim, Yunsheng Li, Liliang Ren, Yelong Shen, et al. Phi-4-mini-reasoning: Exploring the limits of small reasoning language models in math. *arXiv preprint arXiv:2504.21233*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 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.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *ACL*, 2024.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, et al. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. *arXiv preprint arXiv:2505.19223*, 2025.

A APPENDIX

Here, we provide further details that are not discussed in the main paper and include extra experimental results. The appendix is structured as follows:

A.1 Data Efficiency of dLLM 14
A.2 Additional Evaluation Results 14
A.2.1 Pass@10 Results on KernelBench 14
A.2.2 Comparison with cudaLLM 15
A.2.3 Additional Ablation Results 16
A.3 Output Inspection for Reliable Evaluation 16
A.4 Experimental Details 17
A.4.1 Supervised Fine-tuning Details 17
A.4.2 Reinforcement Learning Training Details 17
A.4.3 Evaluation Details 17
A.5 Dataset Analysis 19
A.6 One-shot Prompt Templates 19
A.6.1 Kernel Infilling Stage 19
A.6.2 Kernel Generation Stage and Evaluation 20
A.7 Case Study 21
A.7.1 Examples of Kernel-Generation Shortcuts 21
A.7.2 Generated CUDA Kernels with Speedup 25
A.7.3 Failure Case Analysis 32
A.8 Limitations and Future Work 33

A.1 DATA EFFICIENCY OF DLLM

We perform supervised fine-tuning on the Qwen3 (Yang et al., 2025) and SDAR (Cheng et al., 2025) model families using the same LLaMAFactory (Zheng et al., 2024) pipeline and the CuKe dataset. We keep the training hyperparameters identical across model families, while restricting the differences to architecture-specific modifications required. We report the results on KernelBench (Ouyang et al., 2025) in Table 8. We observe that the SDAR family generally benefits from SFT, whereas the Qwen3 family, especially the 4B and 8B models, even exhibits degradation on several metrics after fine-tuning. Figure 5 further shows that SDAR consistently achieves larger average gains than Qwen3 after supervised fine-tuning on CuKe. Although our setting focuses on task-specific fine-tuning rather than pre-training, this trend is qualitatively consistent with prior evidence that diffusion language models are more data-efficient under limited-data conditions (Ni et al., 2025a).

A.2 ADDITIONAL EVALUATION RESULTS

This section supplements pass@10 results on KernelBench (Ouyang et al., 2025) across three model scales. The latter assesses whether at least one of ten generated candidates is correct, complementing the single-sample evaluation in the main text. We also compare DICE with cudaLLM under different output token limits and extend the BiC-RL ablations to the 1.7B and 4B models.

A.2.1 PASS@10 RESULTS ON KERNELBENCH

The pass@10 results further support the effectiveness of DICE when multiple candidate kernels are generated for each task. At 8B, DICE achieves the highest execution correctness across all three

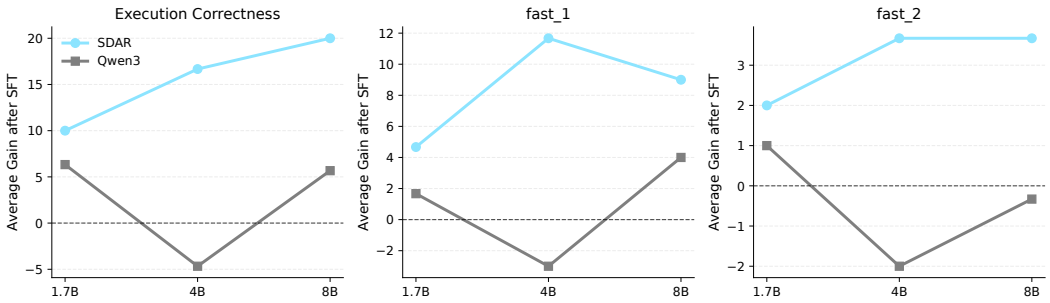


Figure 5: SFT gains of SDAR and Qwen3 on CuKe, measured in percentage points relative to each base model and averaged over the three KernelBench levels.

Table 9: Pass@10 results on KernelBench (Ouyang et al., 2025) across 8B and similar scale models. We report execution correctness (Exec) percentages (%). We compare Autoregressive and Diffusion LLMs, featuring general, code, and reasoning models. The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Type	#Params	Level 1	Level 2	Level 3
Autoregressive LLMs					
Deepseek-Coder-Instruct-v1.5 (Guo et al., 2024)	Code	7B	19.0	9.0	8.0
Qwen2.5-Coder-Instruct (Hui et al., 2024)	Code	7B	26.0	4.0	10.0
Seed-Coder-Reasoning (Seed et al., 2025)	Code	8B	5.0	8.0	4.0
Qwen3 (Yang et al., 2025)	Reasoning (General)	8B	12.0	<u>35.0</u>	6.0
Diffusion LLMs					
LLaDA-1.5 (Zhu et al., 2025)	General	8B	3.0	10.0	16.0
Dream-v0-Instruct (Ye et al., 2025)	General	7B	9.0	0.0	14.0
SDAR-Chat (Cheng et al., 2025)	General	8B	12.0	3.0	0.0
DiffuCoder-Instruct (Gong et al., 2026)	Code	7B	7.0	0.0	0.0
Dream-Coder-v0-Instruct (Xie et al., 2025b)	Code	7B	13.0	18.0	<u>20.0</u>
Stable-DiffCoder (Fan et al., 2026)	Code	8B	<u>28.0</u>	5.0	14.0
TraDo-Instruct (Wang et al., 2026)	Code & Math	8B	14.0	1.0	14.0
DICE	Kernel (CUDA)	8B	**53.0**	**37.0**	**26.0**

levels, reaching 53.0%, 37.0%, and 26.0%, respectively (Table 9). The advantage also extends to the 1.7B scale, where DICE leads every level and achieves 26.0% and 14.0% on levels 1 and 3, for which all evaluated baselines score zero (Table 11). At 4B, DICE achieves the highest level 1 score and ranks second on levels 2 and 3, behind Qwen3 (Table 10). Across all three scales, DICE also improves over the corresponding SDAR-Chat backbones at every difficulty level. These results show that the benefits of specializing dLLMs for CUDA kernel generation remain evident when multiple attempts are available to obtain a correct solution.

A.2.2 Comparison with cudaLLM

We conduct a multi-dimensional comparison between cudaLLM and DICE, as summarized in Table 12. cudaLLM is a reasoning model, we set a substantial maximum generation length to 32,768 tokens to accommodate its extensive Chain-of-Thought (CoT) contents. Furthermore, cudaLLM utilizes high-volume training data, leveraging 8,920 samples for supervised fine-tuning and 71,996 samples for reinforcement learning. In contrast, DICE operates within a more constrained generation window of 4,096 tokens and employs a compact training set of only 6,303 SFT and 4,992 RL samples. Despite using fewer training samples and a shorter output token limit, DICE achieves higher execution correctness across all three levels. As shown in Table 12, cudaLLM performs worse under a 4,096-token limit, motivating our use of the 32,768-token setting to avoid this performance loss.

Table 10: Pass@10 results on KernelBench (Ouyang et al., 2025) across 4B and similar scale models. We report execution correctness (Exec) as percentages (%). We compare Autoregressive and Diffusion LLMs, featuring general, code, and reasoning models. The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Type	#Params	Level 1	Level 2	Level 3
Llama-3.2-Instruct	General	3B	10.0	2.0	0.0
SDAR-Chat (Cheng et al., 2025)	General	4B	13.0	2.0	8.0
TraDo-Instruct (Wang et al., 2026)	Code & Math	4B	12.0	0.0	0.0
Qwen2.5-Coder-Instruct (Hui et al., 2024)	Code	3B	16.0	2.0	0.0
Phi-4-Mini-Reasoning (Xu et al., 2025)	Reasoning	4B	6.0	5.0	4.0
Qwen3 (Yang et al., 2025)	Reasoning	4B	14.0	**31.0**	**24.0**
DICE	Kernel (CUDA)	4B	**30.0**	26.0	14.0

Table 11: Pass@10 results on KernelBench (Ouyang et al., 2025) across 1.7B and similar scale models. We report execution correctness (Exec) as percentages (%). We compare Autoregressive and Diffusion LLMs, featuring general, code, and reasoning models. The best and second-best results are highlighted in **bold** and underline, respectively.

Model	Type	#Params	Level 1	Level 2	Level 3
Gemma-2-It (Team et al., 2024)	General	2B	0.0	0.0	0.0
SDAR-Chat (Cheng et al., 2025)	General	1.7B	0.0	0.0	0.0
Qwen2.5-Coder-Instruct (Hui et al., 2024)	Code	1.5B	0.0	0.0	0.0
Deepseek-Coder-Instruct (Guo et al., 2024)	Code	1.3B	0.0	0.0	0.0
Qwen3 (Yang et al., 2025)	Reasoning	1.7B	0.0	6.0	0.0
DICE	Kernel (CUDA)	1.7B	**26.0**	**7.0**	**14.0**

A.2.3 Additional Ablation Results

Table 13 presents the ablation results for the 1.7B and 4B models using the same training variants. At 1.7B, BiC-RL increases execution correctness over the single-stage RL baseline from 4.0% to 6.0% on level 2 and from 8.0% to 10.0% on level 3. At 4B, it raises fast_2 from 6.0% to 9.0% on level 2 and from 2.0% to 4.0% on level 3. Together with the 8B results, these comparisons support the effectiveness of BiC-RL across the evaluated parameter scales.

A.3 Output Inspection for Reliable Evaluation

Automated evaluation can accept outputs that reproduce reference results without executing the intended CUDA implementation. The KernelBench v0.1 blog[6] therefore recommends inspecting high-scoring outputs even after automated checks. Motivated by this concern, we examine how additional output inspection affects the reported results. We inspect generated outputs for the patterns described in Section 4.2 and compare evaluation results before and after this inspection. This additional inspection is denoted as *Check* in Table 14. Outputs exhibiting these patterns are counted as unsuccessful, with the total number of evaluated tasks unchanged. Unless explicitly identified as pre-inspection results, all KernelBench results reported incorporate this additional check.

The additional inspection reveals score reductions among both AR and diffusion LLMs, particularly on levels 2 and 3. SDAR-Chat (Cheng et al., 2025), the base model of DICE, exhibits reductions across all three levels. On level 3, its execution correctness drops from 22.0% to 0.0% after inspection. In contrast, DICE-8B retains unchanged execution correctness and speedup metrics on levels 1 and 2. Its score adjustments are confined to level 3, where execution correctness decreases from 20.0% to 14.0%. After inspection, DICE-8B still achieves the highest execution correctness across all three levels among the models included in this comparison. The stability on levels 1 and 2 supports the robustness of DICE-8B against the examined shortcuts. The comparison thus shows that the advantage of DICE-8B persists when outputs exhibiting the examined shortcuts are counted as

[6] https://scalingintelligence.stanford.edu/blogs/kernelbenchv01/

Table 12: Comparison results of cudaLLM and DICE on KernelBench (Ouyang et al., 2025). Tokens indicate the maximum output tokens, SFT and RL represent the training data volume in each stage.

Model	Language	#Params	#Tokens	SFT	RL	Level 1		Level 2		Level 3	
						Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
cudaLLM	CUDA	8B	4,096	8,920	71,996	27.0	10.0 / 2.0	5.0	5.0 / 5.0	2.0	0.0 / 0.0
cudaLLM	CUDA	8B	32,768	8,920	71,996	46.0	12.0 / **7.0**	34.0	12.0 / **6.0**	6.0	4.0 / 0.0
DICE	CUDA	8B	4,096	6,303	4,992	**47.0**	**15.0** / 6.0	**35.0**	**19.0** / 1.0	**14.0**	**6.0 / 6.0**

Table 13: Ablation results for SDAR-Chat-4B and SDAR-Chat-1.7B (Cheng et al., 2025) under different training configurations. The best and second-best results are highlighted in **bold** and underline in the respective parameter scales. All metrics are reported as percentages (%).

Model	Level 1		Level 2		Level 3	
	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
SDAR-Chat-4B	5.0	0.0 / 0.0	1.0	0.0 / 0.0	4.0	0.0 / 0.0
+ SFT	29.0	17.0 / 6.0	21.0	**16.0** / 5.0	10.0	2.0 / 0.0
+ SFT → RL (baseline)	**29.0**	**17.0** / 6.0	20.0	11.0 / 6.0	10.0	4.0 / 2.0
+ SFT → RL (*with data scheduling*)	26.0	12.0 / **6.0**	24.0	14.0 / 6.0	10.0	4.0 / 0.0
+ SFT → kernel infilling	27.0	13.0 / 4.0	23.0	14.0 / 7.0	12.0	6.0 / 4.0
DICE-4B	27.0	13.0 / 4.0	**24.0**	15.0 / **9.0**	**12.0**	**6.0 / 4.0**
SDAR-Chat-1.7B	0.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
+ SFT	16.0	7.0 / 3.0	6.0	3.0 / 1.0	8.0	4.0 / 2.0
+ SFT → RL (baseline)	23.0	11.0 / 2.0	4.0	2.0 / 1.0	8.0	6.0 / 2.0
+ SFT → RL (*data scheduling*)	24.0	12.0 / 3.0	5.0	**5.0 / 3.0**	8.0	6.0 / 4.0
+ SFT → kernel infilling	**25.0**	**13.0** / 3.0	5.0	3.0 / 2.0	6.0	6.0 / 2.0
DICE-1.7B	24.0	12.0 / **3.0**	**6.0**	3.0 / 2.0	**10.0**	**6.0 / 4.0**

failures. This finding is consistent with the intended role of kernel infilling in encouraging the use of generated CUDA implementations.

A.4 Experimental Details

A.4.1 Supervised Fine-tuning Details

We choose the SDAR model family (Cheng et al., 2025) as our base models, which contain models across three parameter scales (1.7B, 4B, and 8B). We conduct supervised fine-tuning on our curated CuKe dataset for 3 epochs using 8 NVIDIA A100 GPUS, with a learning rate of 1.0×10^{-5}.

A.4.2 Reinforcement Learning Training Details

During the reinforcement learning stage, we employ a block size of 4 and the dynamic decoding strategy introduced in SDAR (Cheng et al., 2025), with a threshold $\tau = 0.9$. The inference hyperparameters are set at $top - p = 1.0$, $top - k = 0$, and $temperature = 1.0$ during RL sampling. We train for 20 steps in the kernel infilling phase and 100 steps during the end-to-end kernel generation phase. For each step, we sample 64 problems, each with 16 generated responses. The policy network is optimized with a learning rate of 1×10^{-6}, with $\epsilon = 0.2$ and $\beta = 0.01$.

A.4.3 Evaluation Details

In this section, we provide details on evaluation configurations to ensure a rigorous, transparent comparison across dLLMs. As LLaDA (Nie et al., 2025; Zhu et al., 2025), Dream (Ye et al., 2025; Xie et al., 2025b), and Diffucoder (Gong et al., 2026) series models are constrained to fixed-length outputs, the generation time explodes as the number of output tokens increases. Setting a longer output length leads to repetitive outputs (to meet the output length setting), which can degrade model performance, so we chose 1,024 as the output length setting for these models. We use a

Table 14: KernelBench results for baseline models and DICE-8B with and without additional output checks. All metrics are reported as percentages (%).

Model	Check	Level 1		Level 2		Level 3	
		Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
Seed-Coder-Reasoning	✗	2.0	2.0 / 2.0	4.0	0.0 / 0.0	2.0	0.0 / 0.0
	✓	0.0	0.0 / 0.0	2.0	0.0 / 0.0	0.0	0.0 / 0.0
LLaDA-Instruct	✗	2.0	0.0 / 0.0	13.0	8.0 / 3.0	26.0	0.0 / 0.0
	✓	2.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
LLaDA-1.5	✗	2.0	0.0 / 0.0	10.0	2.0 / 0.0	28.0	8.0 / 2.0
	✓	1.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
Dream-v0-Instruct	✗	2.0	0.0 / 0.0	3.0	2.0 / 1.0	10.0	6.0 / 6.0
	✓	2.0	0.0 / 0.0	0.0	0.0 / 0.0	2.0	2.0 / 2.0
DiffuCoder-Instruct	✗	12.0	6.0 / 4.0	46.0	25.0 / 13.0	44.0	24.0 / 0.0
	✓	1.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
SDAR-Chat	✗	9.0	3.0 / 2.0	5.0	1.0 / 0.0	22.0	2.0 / 0.0
	✓	6.0	2.0 / 1.0	2.0	0.0 / 0.0	0.0	0.0 / 0.0
DICE	✗	47.0	15.0 / 6.0	35.0	19.0 / 1.0	20.0	8.0 / 6.0
	✓	**47.0**	**15.0 / 6.0**	**35.0**	**19.0 / 1.0**	**14.0**	**6.0 / 6.0**

1,024-token limit for these fixed-length models. Table 15 reports a length sensitivity comparison for LLaDA-Instruct and LLaDA-1.5 using 1,024 and 4,096 tokens.

Table 15: Results comparison of LLaDA-Instruct (Nie et al., 2025) and LLaDA-1.5 (Zhu et al., 2025) when setting the maximum generated tokens to 1,024 and 4,096 on KernelBench.

Model	#Tokens	Level 1		Level 2		Level 3	
		Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$	Exec	$fast_1$ / $fast_2$
LLaDA-Instruct	1,024	2.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
	4,096	0.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
LLaDA-1.5	1,024	1.0	0.0 / 0.0	0.0	0.0 / 0.0	0.0	0.0 / 0.0
	4,096	0.0	0.0 / 0.0	1.0	0.0 / 0.0	0.0	0.0 / 0.0

- For LLaDA-Instruct (Nie et al., 2025) and LLaDA-1.5 (Zhu et al., 2025), we set a maximum of 1,024 generated tokens, with a block size of 32 and a temperature of 0.0. The overall steps are equal to the generation sequence length.
- For Dream-v0-Instruct (Ye et al., 2025) and Dream-Coder-v0-Instruct (Xie et al., 2025b), we set a maximum of 1,024 generated tokens, with temperatures of 0.2 and 0.1. The overall steps are equal to the generation sequence length.
- For DiffuCoder-Instruct (Gong et al., 2026), we set a maximum of 1,024 generated tokens, with a temperature of 0.3. The overall steps are equal to the generation sequence length.
- For SDAR-Chat (Cheng et al., 2025), TraDo-Instruct (Wang et al., 2026), and DICE models, we employ the static decoding introduced in SDAR. We use a threshold $top - p = 1.0$, $top - k = 1$, and $temperature = 1.0$, with a default block size of 4 and a maximum of 4,096 generated tokens.
- For WeDLM (Liu et al., 2025), we set a maximum of 4,096 generated tokens, leveraging the optimized wedlm engine to accelerate inference, with a temperature of 0.2.
- For Stable-DiffCoder (Fan et al., 2026), we set a maximum of 4,096 generated tokens, with a block size of 4 and a temperature of 0.0.

A.5 Dataset Analysis

Table 16 reports the speedup distribution of the cudaLLM SFT dataset relative to the PyTorch reference implementations. We observe that only 86% of the samples achieve speedup, while the majority of these data are concentrated in the 1.0×–1.5× range.

Table 16: Speedup distribution of the cudaLLM SFT dataset.

Speedup	$< 1.0\times$	$1.0 - 1.5\times$	$1.5 - 2.0\times$	$2.0 - 2.5\times$	$2.5 - 3.0\times$	$> 3.0\times$
Percentage (%)	14.0	52.0	18.0	6.9	3.5	5.6

A.6 One-shot Prompt Templates

A.6.1 Kernel Infilling Stage

The one-shot prompt used during the kernel-infilling RL stage is shown below. It includes a complete worked example that pairs a PyTorch reference implementation with its corresponding generated CUDA kernel. This pairing demonstrates the expected transformation from a high-level operator definition to the missing core CUDA C++ implementation. It also clarifies the required output granularity and encourages semantic consistency with the provided prefix and suffix. We reuse the element-wise addition example from KernelBench (Ouyang et al., 2025) without modification. Specifically, `example_pytorch_reference` and `example_generated_cuda_kernel` denote the PyTorch reference implementation and CUDA kernel supplied by this example, respectively.

````
You write custom CUDA kernels to replace the pytorch operators in the
given architecture to get speedups.

You will be provided with a reference pytorch implementation, which
serves as the ground truth for logical behavior, along with a partial
CUDA kernel skeleton (prefix and suffix).

Your objective is to generate the core missing C++ code of custom CUDA
 kernels within the skeleton to ensure the custom CUDA kernel is
functionally equivalent to the pytorch reference. You must adhere
strictly to the provided kernel configuration.

Here's an example to show you the syntax for generating C++ code based
 on a given prefix and suffix: The example given architecture is:
```
{example_pytorch_reference}
```

The example give prefix is:
```
{example_pytorch_reference_prefix}
```

The example give suffix is:
```
{example_pytorch_reference_suffix}
```

The example new arch with custom CUDA kernels looks like this (combine
 your generated C++ code with given prefix and suffix):
```
{example_generated_cuda_kernel}
```

You are given the following architecture:
````

```
{given_pytorch_code}
```

The corresponding CUDA kernel prefix:
```
{given_prefix}
```

The corresponding CUDA kernel suffix:
```
{given_ suffix}
```

Generate the core C++ code of this custom CUDA kernel to make your optimized output architecture complete. Output the new code in codeblocks. Please generate real code, NOT pseudocode, make sure the code compiles and is fully functional. Just output the new model code, no other text, and NO testing code!

A.6.2 KERNEL GENERATION STAGE AND EVALUATION

For the subsequent end-to-end kernel-generation stage, we use the one-shot prompt shown below. Its structure follows the KernelBench specification, including the reference program, implementation constraints, and required output format. This alignment ensures that the reinforcement learning objective targets the same task definition used by the standard KernelBench evaluation protocol. We use an identical prompt template during training and evaluation, changing only the task-specific problem and reference implementation.

```
You write custom CUDA kernels to replace the pytorch operators in the given architecture to get speedups.

You have complete freedom to choose the set of operators you want to replace. You may make the decision to replace some operators with custom CUDA kernels and leave others unchanged. You may replace multiple operators with custom implementations, consider operator fusion opportunities (combining multiple operators into a single kernel, for example, combining matmul+relu), or algorithmic changes (such as online softmax). You are only limited by your imagination.

Here's an example to show you the syntax of inline embedding custom CUDA operators in torch: The example given architecture is:
```
{example_pytorch_reference}
```
The example new arch with custom CUDA kernels looks like this:
```
{example_generated_cuda_kernel}
```

You are given the following architecture:
```
{given_pytorch_code}
```

Optimize the architecture named Model with custom CUDA operators! Name your optimized output architecture ModelNew. Output the new code in codeblocks. Please generate real code, NOT pseudocode, make sure the code compiles and is fully functional. Just output the new model code, no other text, and NO testing code!
```

A.7 Case Study

A.7.1 Examples of Kernel-Generation Shortcuts

In this section, we present representative examples of shortcuts. In the following generated CUDA kernel, the model replicates the element-wise addition example provided in the prompt, while the actual tensor multiplication is performed by the high-level `torch.matmul` function within the forward pass. Although the model is tasked with generating a specialized matrix multiplication kernel, it exhibits superficial structural mimicry without implementing the requisite low-level logic.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.cpp_extension import load_inline

# Define the custom CUDA kernel for element-wise addition
elementwise_add_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void elementwise_add_kernel(const float* a, const float* b,
     float* out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        out[idx] = a[idx] + b[idx];
    }
}

torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor b) {
    auto size = a.numel();
    auto out = torch::zeros_like(a);

    const int block_size = 256;
    const int num_blocks = (size + block_size - 1) / block_size;

    elementwise_add_kernel<<<num_blocks, block_size>>>(a.data_ptr<
        float>(), b.data_ptr<float>(), out.data_ptr<float>(), size);

    return out;
}
"""

elementwise_add_cpp_source = (
    "torch::Tensor elementwise_add_cuda(torch::Tensor a, torch::Tensor
         b);"
)

# Compile the inline CUDA code for element-wise addition
elementwise_add = load_inline(
    name="elementwise_add",
    cpp_sources=elementwise_add_cpp_source,
    cuda_sources=elementwise_add_source,
    functions=["elementwise_add_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)

class ModelNew(nn.Module):
    def __init__(self) -> None:
```

```
        super().__init__()
        self.elementwise_add = elementwise_add

    def forward(self, A, B):
        """
        Performs 3D tensor-matrix multiplication.

        Args:
            A (torch.Tensor): Input 3D tensor of shape (N, M, K).
            B (torch.Tensor): Input matrix of shape (K, L).

        Returns:
            torch.Tensor: Output tensor of shape (N, M, L), resulting
                from the multiplication of A and B along the last
                dimension of A.
        """
        return torch.matmul(A, B)
```

In the following case, the model generates a custom CUDA kernel but fails to instantiate it within the `ModelNew` constructor, opting instead to utilize PyTorch operators for the final implementation.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.cpp_extension import load_inline

# Define the custom CUDA kernel for element-wise convolution
elementwise_convolution_source = """
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void elementwise_convolution_kernel(const float* input,
    const float* kernel, float* output, int depth, int height, int
    width, int kernel_size, int stride, int padding) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int dx = blockIdx.x * stride + idx;
    int dy = blockIdx.y * stride + padding;
    int dz = blockIdx.z * stride + padding;

    if (dx < depth && dy < height && dz < width) {
        float sum = 0.0;
        for (int i = 0; i < kernel_size; ++i) {
            for (int j = 0; j < kernel_size; ++j) {
                sum += input[dx * width + dy * kernel_size + dz + i] *
                     kernel[i * kernel_size + j];
            }
        }
        output[dx * width + dy * kernel_size + dz] = sum;
    }
}

torch::Tensor elementwise_convolution_cuda(torch::Tensor input, torch
    ::Tensor kernel, int depth, int height, int width, int kernel_size
    , int stride, int padding) {
    auto size = input.numel();
    auto output = torch::zeros_like(input);

    const int block_size = 256;
    const int num_blocks = (size + block_size - 1) / block_size;
```

```
    elementwise_convolution_kernel<<<num_blocks, block_size>>>(input.
        data_ptr<float>(), kernel.data_ptr<float>(), output.data_ptr<
        float>(), depth, height, width, kernel_size, stride, padding);

    return output;
}
"""

elementwise_convolution_cpp_source = (
    "torch::Tensor elementwise_convolution_cuda(torch::Tensor input,
        torch::Tensor kernel, int depth, int height, int width, int
        kernel_size, int stride, int padding);"
)

# Compile the inline CUDA code for element-wise convolution
elementwise_convolution = load_inline(
    name="elementwise_convolution",
    cpp_sources=elementwise_convolution_cpp_source,
    cuda_sources=elementwise_convolution_source,
    functions=["elementwise_convolution_cuda"],
    verbose=True,
    extra_cflags=[""],
    extra_ldflags=[""],
)

class ModelNew(nn.Module):
    """
    Model that performs a 3D convolution, applies Group Normalization,
         minimum, clamp, and dropout.
    """
    def __init__(self, in_channels, out_channels, kernel_size, groups,
         min_value, max_value, dropout_p):
        super(ModelNew, self).__init__()
        self.conv = nn.Conv3d(in_channels, out_channels, kernel_size,
            stride=1, padding=0)
        self.norm = nn.GroupNorm(groups, out_channels)
        self.dropout = nn.Dropout(dropout_p)

    def forward(self, x):
        x = self.conv(x)
        x = self.norm(x)
        x = torch.min(x, torch.tensor(min_value, device=x.device))
        x = torch.clamp(x, min=min_value, max=max_value)
        x = self.dropout(x)
        return x
```

In the following case, the model generates a custom CUDA kernel and initializes it in the `ModelNew` constructor, but it is not called in the forward function, utilizing high-level PyTorch operators for the final implementation. Formally, these instances constitute a failure to fulfill the core CUDA kernel generation objective; however, due to the limitations of current evaluation frameworks, such outputs are erroneously categorized as functionally correct. Consequently, rigorous verification is essential to exclude these false positives and ensure the accuracy of the evaluation results.

```
import torch
import torch.nn as nn
from torch.utils.cpp_extension import load_inline

cuda_source = r"""
```

```
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void depthwise_conv_kernel(const float* input, const float*
     weights, const float* bias, float* output, int batch_size, int
    in_channels, int out_channels, int height, int width, int
    kernel_size, int stride, int padding, int dilation) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= batch_size * in_channels * height * width)
        return;

    int w = idx % width;
    int h = (idx / width) % height;
    int c = (idx / (width * height)) % in_channels;
    int n = idx / (in_channels * width * height);

    float sum = 0.0;
    for (int ky = 0; ky < kernel_size; ++ky) {
        for (int kx = 0; kx < kernel_size; ++kx) {
            int input_h = h + ky * dilation - padding;
            int input_w = w + kx * dilation - padding;
            if (input_h >= 0 && input_h < height && input_w >= 0 &&
                input_w < width) {
                int input_idx = n * in_channels * height * width + c *
                     height * width + input_h * width + input_w;
                float in_val = input[input_idx];
                int weight_idx = c * kernel_size * kernel_size + ky *
                    kernel_size + kx;
                float weight_val = weights[weight_idx];
                sum += in_val * weight_val;
            }
        }
    }
    if (bias) {
        sum += bias[c];
    }
    int output_idx = n * in_channels * height * width + c * height *
        width + h * width + w;
    output[output_idx] = sum;
}

torch::Tensor depthwise_conv_forward(torch::Tensor input, torch::
    Tensor weights, torch::Tensor bias, int kernel_size, int stride,
    int padding, int dilation) {
    TORCH_CHECK(input.is_cuda(), "Input must be a CUDA tensor");
    TORCH_CHECK(weights.is_cuda(), "Weights must be a CUDA tensor");
    int batch_size = input.size(0);
    int in_channels = input.size(1);
    int height = input.size(2);
    int width = input.size(3);
    int out_channels = weights.size(0);

    auto output = torch::empty({batch_size, in_channels, height, width
        }, input.options());

    int num_elements = batch_size * in_channels * height * width;
    int block_size = 256;
    int num_blocks = (num_elements + block_size - 1) / block_size;
```

```
    depthwise_conv_kernel<<<num_blocks, block_size>>>(
        input.data_ptr<float>(),
        weights.data_ptr<float>(),
        bias.defined() ? bias.data_ptr<float>() : nullptr,
        output.data_ptr<float>(),
        batch_size, in_channels, out_channels, height, width,
            kernel_size, stride, padding, dilation
    );

    return output;
}
"""

cpp_source = """
torch::Tensor depthwise_conv_forward(torch::Tensor input, torch::
    Tensor weights, torch::Tensor bias, int kernel_size, int stride,
    int padding, int dilation);
"""

depthwise_conv_ops = load_inline(
    name="depthwise_conv_cuda",
    cpp_sources=[cpp_source],
    cuda_sources=[cuda_source],
    functions=["depthwise_conv_forward"],
    verbose=False
)

class ModelNew(nn.Module):
    def __init__(self, in_channels: int, out_channels: int,
        kernel_size: int, stride: int = 1, padding: int = 0, dilation:
         int = 1, bias: bool = False):
        super(ModelNew, self).__init__()
        self.depthwise = nn.Conv2d(in_channels, in_channels,
            kernel_size, stride=stride, padding=padding, dilation=
            dilation, groups=in_channels, bias=bias)
        self.pointwise = nn.Conv2d(in_channels, out_channels,
            kernel_size=1, bias=bias)
        self.depthwise_conv = depthwise_conv_ops

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.depthwise(x)
        x = self.pointwise(x)
        return x
```

A.7.2 Generated CUDA Kernels with Speedup

We present representative generated CUDA kernels on the level 3 task that are not only functionally correct, but also achieve measurable speedup over their PyTorch reference.

`NetVladWithGhostClusters` with 2.1143x speedup generated by DICE-1.7B:

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch as th
from torch.utils.cpp_extension import load_inline

cuda_source = r"""
```

```
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <float.h>

__global__ void softmax_forward_kernel_vec4(const float* __restrict__
    input,
                                            float* __restrict__ output
                                                ,
                                            const int num_features) {
    extern __shared__ float sdata[];

    const int row = blockIdx.x;
    const int tid = threadIdx.x;
    const int stride = blockDim.x;

    const float* row_in = input + row * num_features;
    float* row_out = output + row * num_features;

    const float4* in4 = reinterpret_cast<const float4*>(row_in);
    float4* out4 = reinterpret_cast<float4*>(row_out);

    const int num_vec = num_features >> 2;
    const int rem = num_features & 3;

    float thread_max = -FLT_MAX;
    for (int vec_idx = tid; vec_idx < num_vec; vec_idx += stride) {
        float4 v = in4[vec_idx];
        thread_max = fmaxf(thread_max, v.x);
        thread_max = fmaxf(thread_max, v.y);
        thread_max = fmaxf(thread_max, v.z);
        thread_max = fmaxf(thread_max, v.w);
    }

    if (tid == 0 && rem) {
        int base = num_vec << 2;
        for (int i = 0; i < rem; ++i) {
            thread_max = fmaxf(thread_max, row_in[base + i]);
        }
    }

    sdata[tid] = thread_max;
    __syncthreads();
    for (int offset = stride >> 1; offset > 0; offset >>= 1) {
        if (tid < offset) {
            float other = sdata[tid + offset];
            if (other > sdata[tid])
                sdata[tid] = other;
        }
        __syncthreads();
    }
    float row_max = sdata[0];

    float thread_sum = 0.0f;
    for (int vec_idx = tid; vec_idx < num_vec; vec_idx += stride) {
        float4 v = in4[vec_idx];
        float4 ev = {__expf(v.x - row_max),
                     __expf(v.y - row_max),
                     __expf(v.z - row_max),
                     __expf(v.w - row_max)};
```

```
        out4[vec_idx] = ev;
        thread_sum += ev.x + ev.y + ev.z + ev.w;
    }

    if (tid == 0 && rem) {
        int base = num_vec << 2;
        for (int i = 0; i < rem; ++i) {
            float val = __expf(row_in[base + i] - row_max);
            row_out[base + i] = val;
            thread_sum += val;
        }
    }

    sdata[tid] = thread_sum;
    __syncthreads();
    for (int offset = stride >> 1; offset > 0; offset >>= 1) {
        if (tid < offset) {
            sdata[tid] += sdata[tid + offset];
        }
        __syncthreads();
    }
    float sum_exp = sdata[0] + 1e-8f;

    for (int vec_idx = tid; vec_idx < num_vec; vec_idx += stride) {
        float4 ev = out4[vec_idx];
        ev.x /= sum_exp;
        ev.y /= sum_exp;
        ev.z /= sum_exp;
        ev.w /= sum_exp;
        out4[vec_idx] = ev;
    }

    if (tid == 0 && rem) {
        int base = num_vec << 2;
        for (int i = 0; i < rem; ++i) {
            row_out[base + i] /= sum_exp;
        }
    }
}

torch::Tensor softmax_forward_cuda(torch::Tensor input) {
    TORCH_CHECK(input.is_cuda(), "input must be a CUDA tensor");
    TORCH_CHECK(input.is_contiguous(), "input must be contiguous");
    TORCH_CHECK(input.scalar_type() == torch::kFloat32, "only float
        tensors are supported");
    TORCH_CHECK(input.dim() == 2, "input must be 2-D (batch_size,
        num_features)");

    const int batch_size = input.size(0);
    const int num_features = input.size(1);

    auto output = torch::empty_like(input);

    const int threads = 1024;
    const int shared_mem_sz = threads * sizeof(float);

    softmax_forward_kernel_vec4<<<batch_size, threads, shared_mem_sz
        >>>(
        input.data<float>(),
        output.data<float>(),
```

```
        num_features);

    return output;
}
"""

cpp_source = "torch::Tensor softmax_forward_cuda(torch::Tensor input);
    "

softmax_ops = load_inline(
    name="row_softmax_cuda",
    cpp_sources=[cpp_source],
    cuda_sources=[cuda_source],
    functions=["softmax_forward_cuda"],
    verbose=False
)

class ModelNew(nn.Module):
    def __init__(self, cluster_size, feature_size, ghost_clusters):
        super(ModelNew, self).__init__()
        self.feature_size = feature_size
        self.cluster_size = cluster_size
        self.ghost_clusters = ghost_clusters

        init_sc = (1 / math.sqrt(feature_size))
        clusters = cluster_size + ghost_clusters

        # The `clusters` weights are the `(w,b)` in the paper
        self.clusters = nn.Parameter(init_sc * th.randn(feature_size,
            clusters))
        self.batch_norm = nn.BatchNorm1d(clusters)
        # The `clusters2` weights are the visual words `c_k` in the
            paper
        self.clusters2 = nn.Parameter(init_sc * th.randn(1,
            feature_size, cluster_size))
        self.out_dim = self.cluster_size * feature_size

    def forward(self, x, mask=None):
        max_sample = x.size()[1]
        x = x.view(-1, self.feature_size)  # B x N x D -> BN x D

        if x.device != self.clusters.device:
            msg = f"x.device {x.device} != cluster.device {self.
                clusters.device}"
            raise ValueError(msg)

        assignment = th.matmul(x, self.clusters)  # (BN x D) x (D x (K
            +G)) -> BN x (K+G)
        assignment = self.batch_norm(assignment)

        # Apply custom softmax
        original_shape = assignment.shape
        assignment_2d = assignment.view(-1, original_shape[1])
        softmaxed = softmax_ops.softmax_forward_cuda(assignment_2d)
        assignment = softmaxed.view(original_shape)

        assignment = assignment[:, :self.cluster_size]
        assignment = assignment.view(-1, max_sample, self.cluster_size
            )  # -> B x N x K
```

```
        a_sum = th.sum(assignment, dim=1, keepdim=True)  # B x N x K
            -> B x 1 x K
        a = a_sum * self.clusters2

        assignment = assignment.transpose(1, 2)  # B x N x K -> B x K
            x N

        x = x.view(-1, max_sample, self.feature_size)  # BN x D -> B x
             N x D
        vlad = th.matmul(assignment, x)  # (B x K x N) x (B x N x D)
            -> B x K x D
        vlad = vlad.transpose(1, 2)  # -> B x D x K
        vlad = vlad - a

        # L2 intra norm
        vlad = F.normalize(vlad)

        # flattening + L2 norm
        vlad = vlad.reshape(-1, self.cluster_size * self.feature_size)
              # -> B x DK
        vlad = F.normalize(vlad)
        return vlad  # B x DK
```

`ReLUSelfAttention` with 4.0756x speedup generated by DICE-4B:

```
import torch
import torch.nn as nn
from torch.utils.cpp_extension import load_inline
import math

# CUDA implementation of a vectorized (float4) row-wise ReLU
cuda_source = r"""
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void relu_forward_kernel_vec4(const float* __restrict__
    input,
                                         float* __restrict__ output,
                                         const int num_features) {
    const int  row    = blockIdx.x;      // Each block handles one row
    const int  tid    = threadIdx.x;
    const int  stride = blockDim.x;

    const float* row_in  = input  + row * num_features;
    float*       row_out = output + row * num_features;

    const int num_vec = num_features >> 2;      // num_features / 4
    const int rem     = num_features & 3;       // leftover elements
        (0-3)

    for (int i = tid; i < num_vec; i += stride) {
        float4 in4  = reinterpret_cast<const float4*>(row_in)[i];
        float4 out4 = {fmaxf(in4.x, 0.0f),
                       fmaxf(in4.y, 0.0f),
                       fmaxf(in4.z, 0.0f),
                       fmaxf(in4.w, 0.0f)};

        reinterpret_cast<float4*>(row_out)[i] = out4;
```

```
    }

    if (tid == 0 && rem) {
        int base = num_vec << 2;         // num_vec * 4
        for (int i = 0; i < rem; ++i) {
            row_out[base + i] = fmaxf(row_in[base + i], 0.0f);
        }
    }
}

torch::Tensor relu_forward_cuda(torch::Tensor input) {
    TORCH_CHECK(input.is_cuda(), "input must be a CUDA tensor");
    TORCH_CHECK(input.is_contiguous(), "input must be contiguous");
    TORCH_CHECK(input.scalar_type() == torch::kFloat32, "only float
        tensors are supported");
    TORCH_CHECK(input.dim() == 2, "input must be 2-D (batch_size,
        num_features)");

    const int batch_size   = input.size(0);
    const int num_features = input.size(1);

    auto output = torch::empty_like(input);

    const int threads        = 1024;
    const int shared_mem_sz  = threads * sizeof(float);

    relu_forward_kernel_vec4<<<batch_size, threads, shared_mem_sz>>>(
        input.data_ptr<float>(),
        output.data_ptr<float>(),
        num_features);

    return output;
}
"""

cpp_source = "torch::Tensor relu_forward_cuda(torch::Tensor input);"

# Compile and load the CUDA extension
relu_ops = load_inline(
    name="relu_cuda",
    cpp_sources=[cpp_source],
    cuda_sources=[cuda_source],
    functions=["relu_forward_cuda"],
    verbose=False,
)

class ModelNew(nn.Module):
    def __init__(self, n_embd, n_head, max_seqlen):
        super(ModelNew, self).__init__()
        assert n_embd % n_head == 0
        self.c_attn = nn.Linear(n_embd, 3 * n_embd)
        self.c_proj = nn.Linear(n_embd, n_embd)
        self.register_buffer("bias", torch.tril(torch.ones(max_seqlen,
             max_seqlen)).view(1, 1, max_seqlen, max_seqlen))
        self.n_head = n_head
        self.n_embd = n_embd
        self.relu_fn = relu_ops

    def forward(self, x):
        B, T, C = x.size()
```

```
        q, k, v = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1,
            2)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1,
            2)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1,
            2)

        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1))
            )
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf')
            )

        # Apply custom ReLU
        original_shape = att.shape
        att_2d = att.contiguous().view(-1, original_shape[-1])
        att = self.relu_fn.relu_forward_cuda(att_2d).view(
            original_shape)

        y = att @ v
        y = y.transpose(1, 2).contiguous().view(B, T, C)
        return y
```

`RegNet` with 3.5430x speedup generated by DICE-8B:

```
import torch
import torch.nn as nn
from torch.utils.cpp_extension import load_inline

# CUDA kernel for global average pooling
cuda_source = r"""
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void global_avg_pool_kernel(const float* input, float*
    output, int N, int C, int H, int W) {
    int n = blockIdx.x;
    int c = threadIdx.x;
    if (c >= C) return;
    float sum = 0;
    for (int h = 0; h < H; ++h) {
        for (int w = 0; w < W; ++w) {
            sum += input[n * C * H * W + c * H * W + h * W + w];
        }
    }
    output[n * C + c] = sum / (H * W);
}

torch::Tensor global_avg_pool_cuda(torch::Tensor input) {
    TORCH_CHECK(input.is_cuda(), "Input must be a CUDA tensor");
    TORCH_CHECK(input.dim() == 4, "Input must be 4D (N, C, H, W)");
    int N = input.size(0);
    int C = input.size(1);
    int H = input.size(2);
    int W = input.size(3);
    auto output = torch::empty({N, C}, input.options());
    const int block_size = C;
    const int grid_size = N;
```

```
    global_avg_pool_kernel<<<grid_size, block_size>>>(
        input.data_ptr<float>(),
        output.data_ptr<float>(),
        N, C, H, W
    );
    return output;
}
"""

cpp_source = "torch::Tensor global_avg_pool_cuda(torch::Tensor input);
    "

# Compile and load the CUDA extension
gap_ops = load_inline(
    name="gap_cuda",
    cpp_sources=[cpp_source],
    cuda_sources=[cuda_source],
    functions=["global_avg_pool_cuda"],
    verbose=False
)

class ModelNew(nn.Module):
    def __init__(self, input_channels, stages, block_widths,
        output_classes):
        super(ModelNew, self).__init__()
        self.stages = stages
        self.block_widths = block_widths
        layers = []
        current_channels = input_channels
        for i in range(stages):
            layers.append(self._make_stage(current_channels,
                block_widths[i]))
            current_channels = block_widths[i]
        self.feature_extractor = nn.Sequential(*layers)
        self.fc = nn.Linear(block_widths[-1], output_classes)
        self.gap_fn = gap_ops

    def _make_stage(self, in_channels, out_channels):
        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3,
                padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=3,
                padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

    def forward(self, x):
        x = self.feature_extractor(x)
        x = self.gap_fn.global_avg_pool_cuda(x)
        x = self.fc(x)
        return x
```

A.7.3 Failure Case Analysis

We analyze failure cases of DICE-generated CUDA kernels in Figure 6, which summarizes execution failures, functional mismatches and generation failures, alongside successful outputs. These failure

patterns suggest several directions for future improvement, including stronger structural constraints, more explicit rewards for valid operator replacement during training, etc.

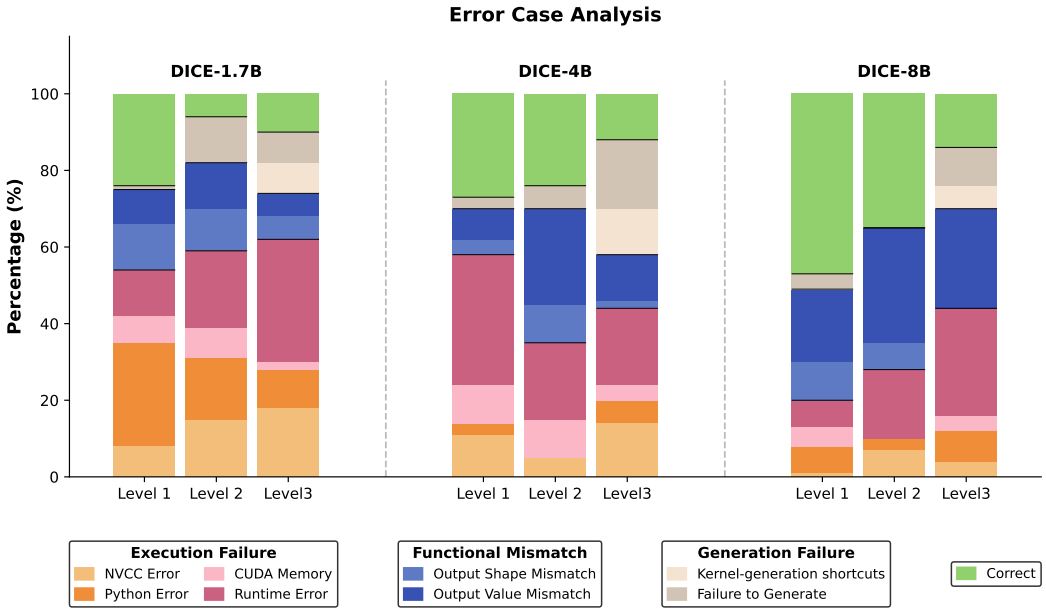


Figure 6: We categorize failure modes of DICE-generated CUDA kernels into execution failures, functional mismatches, and generation failures.

A.8 Limitations and Future Work

Our evaluation focuses on CUDA kernel generation under the hardware and workload configurations considered in this study. Kernel performance can vary across GPU architectures and execution settings. Systematic evaluation across additional devices, input shapes, and numerical precisions is therefore needed. Future work could investigate hardware-conditioned generation and profiling-guided adaptation to improve performance portability.