From Tokens to Regions: CUDA-Sensitive Instruction Tuning for GPU Kernel Generation

Wentao Chen[1], **Jiace Zhu**[1], **Zeng Qu**[2], **Qiaoling Xiao**[2], **Liucheng Duan**[2], **An Zou**[1*]
[1]Shanghai Jiao Tong University [2]Biren Technology
{wentaochen, zhujiace, an.zou}@sjtu.edu.cn
{zqu, qlxiao, aduan}@birentech.com

Abstract

High-performance CUDA kernels are essential for scalable AI systems, while Large Language Models (LLMs) still struggle to generate correct kernels due to strict and implicit execution constraints. Existing LLM-based approaches either rely on costly agentic or reinforcement-learning (RL) pipelines, or adopt supervised fine-tuning (SFT) objectives that fail to explicitly model CUDA sensitivity, namely code tokens or regions tightly coupled with execution constraints. In this work, we investigate CUDA sensitivity from the perspective of token confidence patterns, showing that CUDA sensitivity appears at both token and region levels, where most CUDA-sensitive tokens are predicted with high confidence, while a smaller low-confidence subset forms regions corresponding to execution-critical structures. These findings suggest that effective CUDA kernel generation should both leverage high-confidence CUDA-sensitive tokens and preserve low-confidence CUDA-sensitive regions. Building on these insights, we propose **CUDA-Sensitive Instruction Tuning (CuSeT)**, a low-cost post-training method within a simple SFT framework. CuSeT follows the principle of "from tokens to regions" by combining *adaptive token-level masking* with *region-aware sample reweighting*. Experiments show that CuSeT consistently improves both functional correctness and kernel performance across multiple model families and scales, outperforming standard SFT and advanced SFT variants, while achieving competitive performance against frontier CUDA kernel generation models with substantially lower inference cost.

1 Introduction

High-performance GPU kernels are a cornerstone of scalable AI systems (Dao et al., 2022; Ye et al., 2025; Miao et al., 2025), yet developing such kernels remains challenging due to strict hardware-level execution constraints. While Large Language Models (LLMs) show promise in automating GPU kernel generation (Ouyang et al., 2025), existing approaches struggle to produce correct and executable CUDA code. These approaches include agentic frameworks (Chen et al., 2025; Wei et al., 2025; Dong et al., 2025; Saba et al., 2026), post-training methods such as supervised fine-tuning (SFT) (Fisches et al., 2025; Lv et al., 2025; Kong et al., 2025), reinforcement learning (RL) (Li et al., 2025c; Baronio et al., 2025; Liu et al., 2026b), as well as hybrid training pipelines that combine multiple stages (Woo et al., 2025; Cao et al., 2026b; Li et al., 2025b). As summarized in Tab. 1, existing paradigms generally lack CUDA sensitivity awareness, rely on execution feedback rather than CUDA-sensitive structural signals, or provide supervision only at token-level or whole-code-level granularity, often incurring additional inference or training cost.

The fundamental reason behind this limitation is the mismatch between existing methods and CUDA programming. Unlike general-purpose code generation (Wei et al., 2024; Roziere et al., 2023), CUDA kernel generation is a system-constrained conditional generation problem. A generated kernel must satisfy not only syntactic and semantic correctness, but also implicit execution constraints (Cook, 2012; Kirk and Wen-Mei, 2016), including memory access patterns, thread indexing, synchronization, and launch configuration. We define *CUDA sensitivity* to describe code tokens or regions whose correctness is tightly coupled with these execution constraints. Such CUDA-sensitive tokens are concentrated in execution-critical locations and regions, where small deviations may lead to compilation failures or incorrect execution, while many other tokens contribute far less to functional correctness.

This token-level and region-level sensitivity poses a fundamental challenge to existing CUDA

*Corresponding Author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 43505–43519
October 24-29, 2026 ©2026 Association for Computational Linguistics

Table 1: Comparison of LLM-based GPU kernel generation methods.

Paradigm		CUDA Sensitivity Awareness	CUDA-Sensitive Structural Signal	Supervision Granularity
Agentic		No	Execution Feedback	Whole-code level
Post-training	Standard SFT	No	No	Uniform token-level
	Improved SFT	No	No	Token-level
	RL	No	Execution Feedback	Whole-code level
	Hybrid	No	Execution Feedback	Whole-code level
	CuSeT (Ours)	**Yes**	**Region Confidence**	**Token-to-region**

kernel generation paradigms. Agentic methods refine kernels through iterative feedback, but they often require long optimization loops and substantial inference cost. RL-based methods optimize execution-level objectives, but their rewards are typically sparse and available only after execution. Hybrid pipelines combine multiple signals but introduce additional training and system complexity. Standard SFT remains an attractive foundation due to its simplicity, stability, and computational efficiency. However, it applies uniform supervision across all target tokens (Ouyang et al., 2022) and does not explicitly account for token-level and region-level CUDA sensitivity, which limits its effectiveness in supervising execution-critical code. Existing SFT improvements partially address token importance by reweighting token losses using predicted probabilities, entropy, or other uncertainty signals (Wu et al., 2025; Diao et al., 2026; Zhu et al., 2025), but they remain insufficient for CUDA kernel generation as they are usually CUDA-agnostic and overlook regional structure. Consequently, an effective SFT-based approach for CUDA kernel generation should address CUDA sensitivity at two complementary granularities: strengthening supervision for execution-critical tokens while emphasizing samples containing low-confidence CUDA-sensitive regions.

To better understand how CUDA sensitivity manifests in model predictions, we analyze token confidence patterns in CUDA code, revealing that CUDA sensitivity occurs at both token and region levels: most CUDA-sensitive tokens are predicted with high confidence, while a smaller low-confidence subset forms regions corresponding to execution-critical structures. This finding motivates a supervision strategy from tokens to regions for CUDA kernel generation: leveraging high-confidence CUDA-sensitive tokens while preserving low-confidence CUDA-sensitive regions.

Building on these insights, we propose **CUDA-Sensitive Instruction Tuning (CuSeT)**, a low-cost post-training method within a simple SFT framework that explicitly strengthens supervision over CUDA-sensitive tokens and regions. CuSeT implements the principle of "from tokens to regions" via two complementary designs: (1) *adaptive token-level masking*, which filters noisy low-confidence CUDA-neutral tokens while preserving CUDA-sensitive tokens, and (2) *region-aware sample reweighting*, which emphasizes samples containing low-confidence CUDA-sensitive regions. In this way, token-level masking determines which tokens contribute to supervision, while region-aware sample reweighting identifies difficult regions that require greater optimization emphasis. By explicitly modeling CUDA sensitivity at both token and region levels, CuSeT improves both functional correctness and kernel performance while maintaining computational efficiency, without relying on costly RL or complex multi-stage pipelines. Our contributions are summarized as follows:

- We introduce **CUDA sensitivity** and analyze its token-level and region-level patterns, showing that most CUDA-sensitive tokens are high-confidence, while a small subset of low-confidence tokens forms regions critical for execution.
- We propose **CUDA-Sensitive Instruction Tuning (CuSeT)**, a low-cost SFT-based post-training method that combines **adaptive token-level masking** and **region-aware sample reweighting** to strengthen supervision over CUDA-sensitive tokens and regions.
- Extensive experiments demonstrate that CuSeT significantly improves both functional correctness and kernel performance across multiple model families and scales, outperforming standard SFT and improved SFT methods, while remaining competitive with

frontier CUDA kernel generation models at substantially lower inference cost.

2 Related Work

2.1 LLMs for GPU Kernel Generation

Recently, LLMs have demonstrated remarkable general capabilities in software development tasks (Guo et al., 2024; Roziere et al., 2023), but applying LLMs to GPU programming remains extremely challenging. Ouyang *et al.* introduced KernelBench (Ouyang et al., 2025), showing that current LLMs struggle to generate both correct and performant GPU code, highlighting the gap between general code generation and domain-specific high-performance programming. Following works have investigated a variety of approaches to automate GPU kernel generation and optimization using LLMs. These mainly include leveraging agentic systems with self-refinement and iterative optimization (Chen et al., 2025; Lange et al., 2025; Dong et al., 2025; Lei et al., 2025). While effective in certain settings, these approaches incur substantial computational costs, requiring long iterative cycles or complex multi-stage pipelines. Instead, more works have shifted towards training more powerful models capable of handling GPU kernel programming. These include supervised fine-tuning (SFT) methods (Fisches et al., 2025; Kong et al., 2025; Ke et al., 2025), where pre-trained LLMs are adapted to GPU kernel generation tasks by fine-tuning on a domain-specific dataset. Additionally, reinforcement learning (RL)-based methods have also been explored (Baronio et al., 2025; Li et al., 2025c; Liu et al., 2026b), which focus on optimizing the kernel generation process by rewarding the model for generating correct and efficient kernels. Some works (Li et al., 2025b; Woo et al., 2025; Cao et al., 2026b) have combined SFT with RL to leverage the advantages of both techniques. However, despite these advancements, these methods often overlook the unique characteristics of CUDA code, including implicit execution constraints, as well as token-level and region-level CUDA sensitivity, limiting their ability to generate functionally correct and executable GPU kernels.

2.2 Post-training LLMs

Post-training methods, including SFT and RL, are widely applied to align pre-trained LLMs with specific tasks. Standard SFT minimizes token-level negative log-likelihood (Ouyang et al., 2022), assuming uniform token importance, while RL updates the model based on its generated outputs using task-specific reward signals (Shao et al., 2024; Schulman et al., 2017). However, in CUDA code, execution-critical tokens are unevenly distributed, and uniform supervision fails to capture the contributions of CUDA-sensitive tokens and regions. Token-level improvements using prediction probabilities (Wu et al., 2025; Li et al., 2025a) or entropy (Diao et al., 2026; Wang et al., 2026) provide uncertainty-aware supervision but remain token-centric and CUDA-agnostic. Similarly, RL-based approaches (Baronio et al., 2025; Li et al., 2025b,c) also face complementary challenges since sparse rewards, often available only at the final execution result, fail to provide fine-grained feedback on critical CUDA-sensitive regions. In this work, we analyze CUDA sensitivity from tokens to regions and propose **CUDA-Sensitive Instruction Tuning (CuSeT)**, a low-cost SFT-based post-training method that explicitly strengthens supervision over execution-critical CUDA-sensitive tokens and regions, improving both functional correctness and kernel performance in CUDA kernel generation.

3 Motivation

To investigate how CUDA sensitivity is reflected in model predictions, we select representative CUDA code samples from NVIDIA TensorRT (NVIDIA Corporation) and label tokens as *CUDA-sensitive* or *CUDA-neutral* based on their dependence on execution constraints, such as memory access patterns or thread indexing, as shown in Fig. 2a. We then compute each token's predicted probability using Qwen3-8B (Yang et al., 2025), referred to as its *token confidence*. Our analysis reveals two key observations.

At the token level, CUDA-sensitive and CUDA-neutral tokens exhibit clearly different confidence distributions. As shown in Fig. 1, most CUDA-sensitive tokens are concentrated in high-confidence regions, suggesting that token confidence can serve as a reliability signal for supervising execution-critical code. In contrast, CUDA-neutral tokens are more dispersed and have higher density in low-confidence regions, indicating less reliable supervision signals. A two-sample Kolmogorov-Smirnov (KS) test (Massey Jr, 1951) confirms that the confidence distributions are statistically distinct[1]. This observation suggests that

[1]The null hypothesis assumes that CUDA-sensitive and

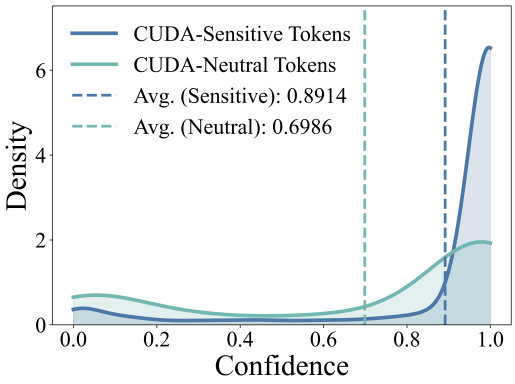


Figure 1: Token confidence density distributions of CUDA-sensitive and CUDA-neutral tokens in CUDA code, computed as the model's next-token prediction probabilities. CUDA-sensitive tokens encode execution-critical logic and concentrate in high-confidence regions, while CUDA-neutral tokens are more dispersed with higher density in low-confidence regions. The two distributions are statistically distinct under a two-sample KS test.

token confidence can serve as a useful reliability signal for token-level supervision.

At the region level, a small subset of low-confidence CUDA-sensitive tokens forms regions corresponding to execution-critical structures, such as shared memory allocation or launch configuration. Fig. 2 illustrates one representative example. Unlike low-confidence tokens in general code generation tasks, which often reflect stylistic variation or interchangeable expressions with limited semantic impact, these regions indicate structured uncertainty about execution constraints. Properly supervising these regions may therefore be important for learning difficult CUDA patterns.

Together, these observations indicate that supervision in CUDA kernel generation should go beyond token-level confidence: high-confidence CUDA-sensitive tokens provide reliable guidance, whereas low-confidence CUDA-sensitive tokens should be preserved through region-level supervision because they often form execution-critical structures. This motivates a supervision strategy from tokens to regions, leveraging reliable high-confidence tokens while preserving low-confidence CUDA-sensitive regions. Such a strategy serves as the basis for our method, **CuSeT**, which explicitly incorporates CUDA sensitivity at both token and region levels.

CUDA-neutral tokens are drawn from the same confidence distribution. The test yields $p = 4.6 \times 10^{-39}$, leading to a significant rejection of the null hypothesis.

4 CUDA-Sensitive Instruction Tuning

4.1 Overall

Fig. 3 illustrates the overall architecture of CuSeT. Consider a dataset $\mathcal{D} = \{(x_n, y_n^*, s_n)\}_{n=1}^N$ consisting of N prompt-response pairs with associated implicit system and execution constraints, where each x_n is the input prompt and $y_n^* = (y_{n,1}^*, y_{n,2}^*, \ldots, y_{n,T_n}^*)$ is the target CUDA code. The goal is to generate y^* from x using a policy π_θ parameterized by θ, under implicit system and execution constraints s, including memory access patterns, thread synchronization, and indexing. CuSeT extends standard SFT by explicitly modeling CUDA-sensitive structures from tokens to regions, via adaptive token-level masking and region-aware sample reweighting.

4.2 Adaptive Token-level Masking

Confidence Mask. Motivated by the confidence distribution analysis in Sec. 3, we use token confidence as a dynamic signal to guide token-level supervision. For the t-th token $y_{n,t}^*$ in the n-th sample, its confidence is defined as the model's next-token prediction probability:

$$c_{n,t} = \pi_\theta(y_{n,t}^* \mid x_n, y_{n,<t}^*), \tag{1}$$

Using this confidence measure, we define a confidence mask $\mathcal{M}_{n,t}^{\text{conf}}$ as a binary indicator:

$$\mathcal{M}_{n,t}^{\text{conf}} = \mathbb{1}\left[\text{sg}\left(\pi_\theta(y_{n,t}^* \mid x_n, y_{n,<t}^*) > \tau\right)\right], \tag{2}$$

where $\mathbb{1}[\cdot]$ is the indicator function, $\text{sg}(\cdot)$ denotes stop-gradient, and $\tau \in [0, 1]$ is a static threshold. During training, the confidence mask selects high-confidence tokens for supervision while suppressing low-confidence tokens, thereby reducing the influence of potentially unreliable token-level supervision. This also aligns with the interpretation that low-probability tokens can induce larger gradients and interfere with effective optimization (Liu et al., 2026a).

CUDA Mask. While the confidence mask preserves high-confidence tokens, it may suppress low-confidence tokens in CUDA-sensitive code structures that are nevertheless essential for correct CUDA execution. To preserve supervision in such execution-critical locations, we introduce a CUDA mask:

$$\mathcal{M}_{n,t}^{\text{CUDA}} = \lambda \cdot \mathbb{1}\{y_{n,t}^* \in \text{kernel body or launch configuration tokens of sample } n\}, \tag{3}$$

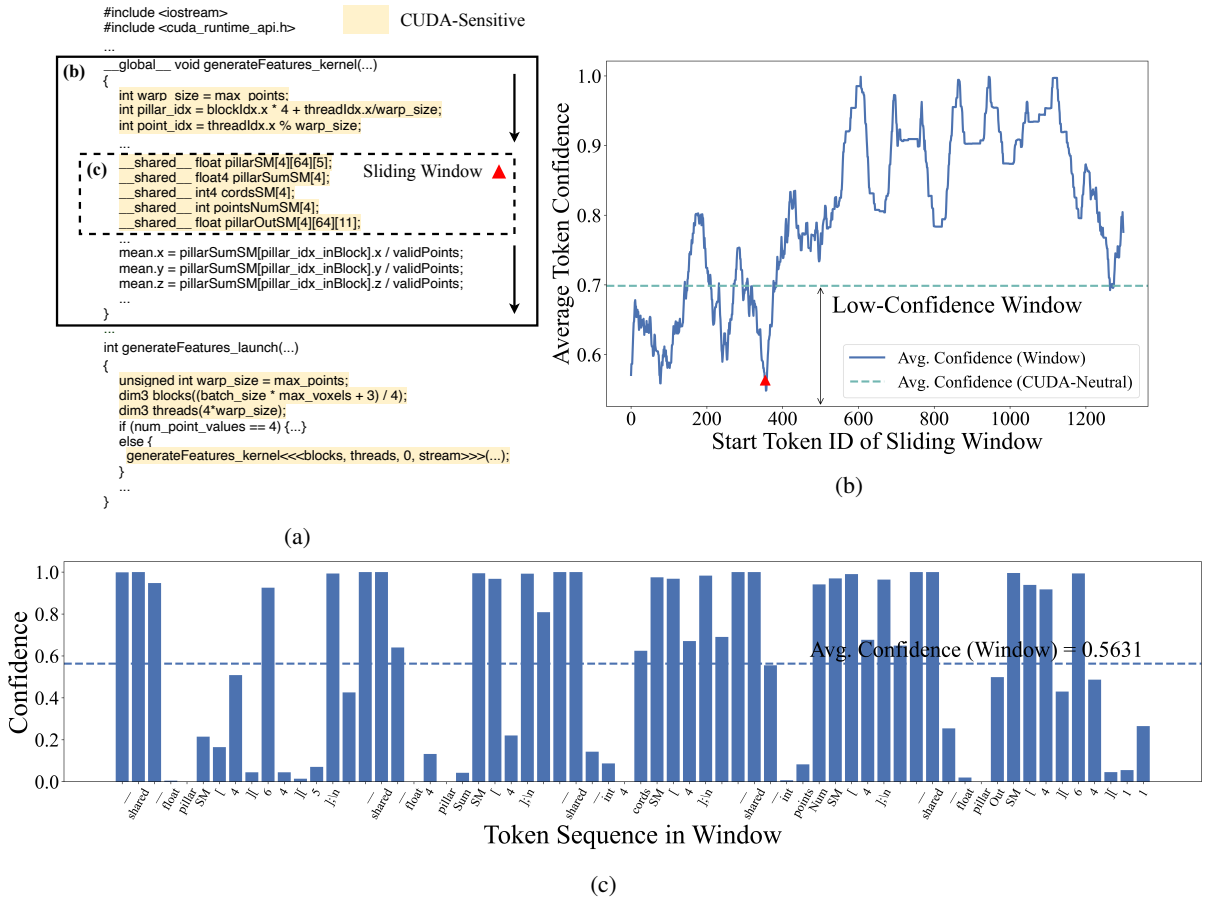


Figure 2: (a) A sample CUDA code with CUDA-sensitive tokens highlighted; (b) Average token confidence over sliding windows within the solid-box region in (a), using a window size of 64 tokens and a stride of 1 token. Low-confidence windows are defined as sliding windows whose average token confidence is below the average confidence of CUDA-neutral tokens in Fig. 1; (c) Zoom-in view of a representative low-confidence window (▲), showing individual token confidences. Across all samples, **74.47%** contain at least one low-confidence window, and **91.18%** of these low-confidence windows correspond to CUDA-sensitive code structures, indicating that a low-confidence subset of CUDA-sensitive tokens forms regions associated with execution-critical structures.

where $\lambda > 0$ controls the weight assigned to tokens in execution-critical code locations. In practice, we construct the CUDA mask through lightweight character-level span annotation: CUDA syntax markers such as `__global__` declarations and `<<<...>>>` launch configurations are used to identify kernel body and launch configuration spans, which are then aligned with tokenizer offsets. This mask ensures that tokens in kernel bodies and launch configurations retain a basic level of supervision even when their confidence is low.

Hybrid Mask. We construct a hybrid token-level mask by combining the confidence mask and the CUDA mask:

$$\mathcal{M}_{n,t}^{\text{hybrid}} = \mathcal{M}_{n,t}^{\text{conf}} + \mathcal{M}_{n,t}^{\text{CUDA}}. \tag{4}$$

The hybrid mask selects high-confidence tokens through the confidence mask while retaining low-confidence tokens in execution-critical code locations through the CUDA mask. Meanwhile, low-confidence CUDA-neutral tokens receive reduced supervision. In this way, $\mathcal{M}_{n,t}^{\text{hybrid}}$ incorporates CUDA sensitivity into token-level supervision without treating all target tokens uniformly.

4.3 Region-aware Sample Reweighting

While the hybrid mask strengthens token-level supervision, it does not explicitly account for low-confidence CUDA-sensitive regions. As observed in Sec. 3, a small subset of CUDA-sensitive tokens forms low-confidence regions corresponding to execution-critical structures, such as shared memory allocation, which are critical for compilation and correct execution. Fig. 2b shows the sliding window confidence curve of a representative CUDA kernel, and Fig. 2c illustrates a low-

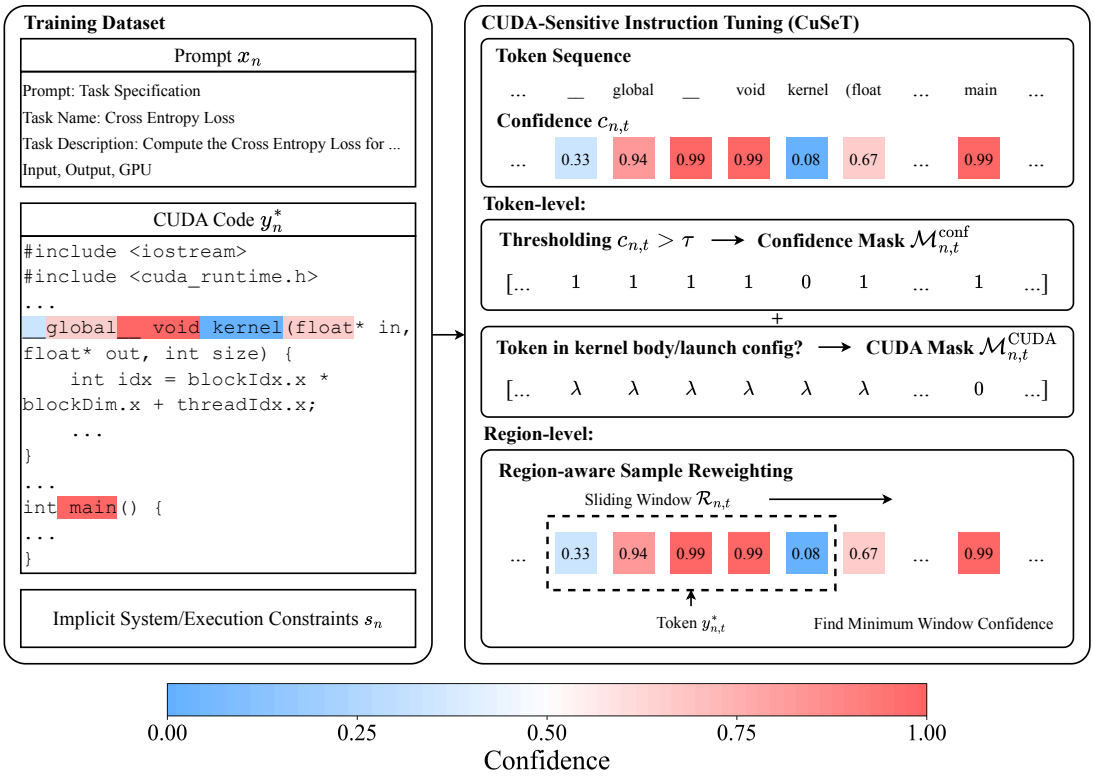


Figure 3: Overview of CuSeT, a CUDA-sensitive SFT framework. For each target token, CuSeT computes token confidence and derives a confidence mask, which is combined with a CUDA mask to form a hybrid token-level mask for execution-critical tokens. Sliding window analysis identifies low-confidence CUDA-sensitive regions, enabling sample reweighting. This pipeline focuses supervision on CUDA-sensitive tokens and regions critical for reliable CUDA kernel generation.

confidence window corresponding to shared memory CUDA code.

These observations indicate that supervision should extend beyond individual tokens to the region level, since low-confidence regions often correspond to CUDA-sensitive code structures that encode complex execution constraints spanning multiple tokens. To incorporate this structure, we model a region as a contiguous sliding window over the target token sequence and propose region-aware sample reweighting based on window confidence analysis, which assigns higher weights to samples containing such low-confidence CUDA-sensitive regions. Specifically, for each token $y_{n,t}^*$, following the design principles in (Fu et al., 2025; Tang et al., 2025), we associate it with a sliding window $\mathcal{R}_{n,t}$ of size l centered at position t:

$$\mathcal{R}_{n,t} = \left\{ y_{n,j}^* \;\middle|\; \max\left(1, t - \left\lfloor \frac{l}{2} \right\rfloor\right) \le j \le \min\left(T_n, t + \left\lceil \frac{l}{2} \right\rceil - 1\right) \right\}, \tag{5}$$

The window is clipped to valid token positions at sample sequence boundaries, and if $T_n < l$, it covers the entire sequence. We define the window confidence as the average token confidence within each sliding window:

$$C_{n,t} = \frac{1}{|\mathcal{R}_{n,t}|} \sum_{y_{n,j}^* \in \mathcal{R}_{n,t}} c_{n,j}, \tag{6}$$

For each sample, the most difficult region is identified by the minimum window confidence:

$$\mathcal{C}_n = \min_{1 \le t \le T_n} \{C_{n,t}\}, \tag{7}$$

The sample-level weight is then defined as:

$$w_n = \text{sg}\left(1 - \mathcal{C}_n\right). \tag{8}$$

This assigns higher weights to samples containing low-confidence CUDA-sensitive regions, encouraging the model to focus on challenging CUDA-sensitive structures during optimization.

4.4 Training Objective

We combine the token-level hybrid mask and the sample-level weight to define the final CuSeT ob-

jective. The model is trained with a likelihood objective that incorporates adaptive token-level masking and region-aware sample reweighting:

$$\mathcal{L}_{\text{CuSeT}}(\theta) = \mathbb{E}_{\{(x_n, y_n^*, s_n)\}_{n=1}^{N}} \left[-w_n \sum_{t=1}^{T_n} \mathcal{M}_{n,t}^{\text{hybrid}} \log c_{n,t} \right]. \tag{9}$$

This objective implements CUDA-sensitive supervision from tokens to regions. At the token level, $\mathcal{M}_{n,t}^{\text{hybrid}}$ preserves high-confidence tokens while retaining low-confidence CUDA-sensitive tokens in execution-critical code locations. At the sample level, w_n increases the training emphasis on samples containing low-confidence regions identified via sliding window analysis. Overall, CuSeT strengthens supervision on execution-critical tokens while prioritizing samples containing low-confidence CUDA-sensitive regions, thereby enhancing the model's focus on CUDA-sensitive structures and improving functional correctness.

5 Evaluation

5.1 Experimental Setup

Data Gathering Pipeline and Benchmarks. We construct a diverse CUDA kernel dataset from multiple sources, including open-source repositories such as GitHub and Hugging Face, as well as benchmark datasets (Gong et al., 2025; Wen et al., 2022) and PyTorch modules (Ouyang et al., 2025; Paliskara and Saroufim, 2025). To ensure functional correctness, we use an LLM (e.g., Qwen3-Coder (Yang et al., 2025)) to generate five sets of input cases for each kernel and derive corresponding reference outputs for validation. Each kernel is compiled and executed on a GPU, and only those passing both compilation and functional correctness checks are retained. We further perform deduplication and contamination checks to remove duplicate or highly similar samples between the training data and evaluation benchmarks. This results in a dataset of 6,278 high-quality prompt-response pairs covering diverse CUDA kernel patterns and execution constraints. For evaluation, we primarily use CUDABench Level 1 benchmark (Zhu et al., 2026). Additional results on the more challenging Level 2 are provided in Appendix D to evaluate generalization across difficulty levels.

Baselines and Metrics. We compare CuSeT against standard SFT. Additionally, we include several improved SFT methods, including DFT (Wu et al., 2025), EAFT (Diao et al., 2026), ASFT (Zhu et al., 2025), and ProFit (Liu et al., 2026a). We evaluate the generated kernels using **Compilation Success**, **Functional Correctness**, and **CUDABench-Score** (Zhu et al., 2026). Compilation Success and Functional Correctness measure whether the generated kernels can be successfully compiled and produce correct outputs, respectively, while CUDABench-Score further evaluates their execution performance beyond correctness. We report all metrics under **Pass@k** ($k = \{1, 3\}$) (Chen, 2021), following the evaluation protocol of CUDABench.

Table 2: Main results on CUDABench Level 1 benchmark. We report the compilation success, functional correctness, and CUDABench-Score of the vanilla baseline, standard SFT, and varying improved SFT strategies (DFT, EAFT, ASFT, ProFit) on multiple model families. Best results are **bolded**, and second-best are underlined.

Method	Pass@1			Pass@3		
	Comp.	Func.	Score	Comp.	Func.	Score
Qwen2.5-Coder-7B-Instruct						
Vanilla	69.0	35.2	15.67	83.6	49.2	19.68
SFT	92.2	55.0	19.18	96.6	61.0	21.57
DFT	93.0	60.2	22.47	93.8	62.0	22.86
EAFT	91.4	56.8	19.06	97.2	65.2	22.36
ASFT	77.8	50.4	20.15	81.6	54.4	21.61
ProFit	**94.2**	59.6	19.02	97.0	64.0	21.64
CuSeT	93.6	**65.6**	**22.68**	**98.8**	**74.8**	**25.50**
DeepSeek-Coder-6.7B-Instruct						
Vanilla	80.2	42.4	16.40	87.8	48.4	18.43
SFT	90.4	54.6	21.01	96.2	62.2	24.33
DFT	88.0	53.0	21.29	90.6	55.0	21.93
EAFT	89.4	53.4	19.78	94.0	60.2	22.95
ASFT	89.4	51.8	19.76	93.2	56.0	21.42
ProFit	91.8	56.8	21.80	97.0	63.6	24.60
CuSeT	**95.0**	**63.2**	**22.35**	**99.0**	**70.6**	**25.26**
Seed-Coder-8B-Instruct						
Vanilla	4.0	2.8	1.34	9.0	6.6	2.56
SFT	93.4	61.2	21.24	97.6	68.8	24.41
DFT	97.0	66.4	22.55	98.0	67.6	23.39
EAFT	94.6	62.6	20.84	99.2	69.8	24.14
ASFT	88.8	59.0	21.65	91.4	64.8	22.78
ProFit	**97.6**	67.6	23.34	98.4	69.6	24.20
CuSeT	94.6	**67.8**	**23.42**	**98.8**	**75.8**	**26.27**

Models and Implementation Details. To verify the generalizability of CuSeT across model families, we conduct experiments on a diverse set of code LLMs, including Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024), and Seed-Coder-8B-Instruct (Seed et al., 2025). In the training stage, we utilize the LLaMAFactory (Zheng et al., 2024) framework and perform LoRA (Hu et al., 2022) fine-tuning

Table 3: Comparison of frontier models and CuSeT-tuned models on the CUDABench Level 1 benchmark. Avg. Consumed Tokens denotes the average number of tokens generated per sample during inference, serving as a measure of generation efficiency.

Model	Type	#Params	Pass@1		Pass@3		Avg. Consumed Tokens
			Comp.	Func.	Comp.	Func.	
Frontier Models							
GLM-5.1	Reasoning	40B/744B	16.8	15.6	32.2	29.2	8,220
DeepSeek-V3.2-Thinking	Reasoning	685B	93.8	66.6	99.4	80.6	3,717
Qwen3-Coder-Next	Code	80B	86.0	63.4	98.0	80.8	1,273
Kevin	Reasoning	32B	96.0	68.8	**99.6**	78.2	5,392
KernelCoder	Reasoning	32B	**96.8**	70.6	**99.6**	79.8	5,379
Models Fine-tuned with CuSeT							
Qwen2.5-Coder-7B-Instruct	Code	7B	93.6	65.6	98.8	74.8	743
DeepSeek-Coder-6.7B-Instruct	Code	6.7B	95.0	63.2	99.0	70.6	1,018
Seed-Coder-8B-Instruct	Code	8B	94.6	67.8	98.8	75.8	839
Qwen3-14B	General	14B	92.6	66.8	97.6	76.0	**722**
Qwen2.5-Coder-32B-Instruct	Code	32B	96.0	**75.6**	98.8	**81.0**	726

with LoRA rank $r = 8$, $\alpha = 16$, and a LoRA dropout rate of 0.05 for all linear layers. The maximum sequence length is set to 4,096, with a batch size of 1 per GPU. The model is fine-tuned with a learning rate of 1×10^{-4} for 3 epochs on 8 NVIDIA RTX 3090 Ti GPUs. For the hyperparameters of CuSeT, we set the confidence mask threshold to $\tau = 0.1$ and the CUDA mask scaling factor to $\lambda = 1.0$ in adaptive token-level masking, ensuring that tokens in kernel bodies and launch configurations receive basic supervision even under low confidence. In region-aware sample reweighting, we use a sliding window of size $l = 256$ to compute window confidence. A detailed sensitivity analysis of the CuSeT hyperparameters is provided in Appendix C.

5.2 Performance Analysis

The main results are reported in Tab. 2. CuSeT consistently outperforms standard SFT and representative improved SFT methods across all model families in terms of functional correctness, while also achieving the highest CUDABench-Score and maintaining consistently high compilation success. For instance, on Qwen2.5-Coder-7B-Instruct, CuSeT achieves 65.6% functional correctness under Pass@1, exceeding the best baseline by over 5%, while maintaining a high compilation success rate. Similar patterns are observed on DeepSeek-Coder-6.7B-Instruct and Seed-Coder-8B-Instruct, where CuSeT consistently improves functional correctness by up to 6.4% over the strongest baselines. Beyond correctness, CuSeT also achieves the highest CUDABench-Score across all three model families. These results indicate that combining adaptive token-level masking and region-aware sample reweighting improves the model's ability to capture CUDA-sensitive structures, leading to higher functional correctness while maintaining strong kernel performance.

5.3 Comparison with SOTA Methods

We compare CuSeT-tuned models against two groups of baselines for CUDA kernel generation in terms of correctness and inference efficiency, including frontier general-purpose and code models such as GLM-5.1 (Z.ai, 2026), DeepSeek-V3.2-Thinking (Liu et al., 2025) and Qwen3-Coder-Next (Cao et al., 2026a), as well as CUDA-specialized models such as Kevin (Baronio et al., 2025) and KernelCoder (Kong et al., 2025). Tab. 3 summarizes the results on CUDABench Level 1 benchmark. CuSeT-tuned models consistently achieve competitive or superior functional correctness while significantly reducing inference cost measured by average generated tokens. In particular, Qwen2.5-Coder-32B-Instruct with CuSeT achieves 75.6% (Pass@1) and 81.0% (Pass@3) functional correctness, while using only 726 tokens on average, substantially fewer than frontier baselines. These results demonstrate a favorable trade-off between functional correctness and inference efficiency, indicating that explicitly modeling CUDA-sensitive tokens and regions enables effective kernel generation without relying on costly multi-stage training or long-context reasoning pipelines.

5.4 Scalability

To evaluate the scalability of CuSeT, we conduct experiments on models of different sizes, including Qwen3-4B (Yang et al., 2025) and Qwen2.5-Coder-32B-Instruct (Hui et al., 2024). Tab. 4 reports the results compared with the vanilla pretrained models, standard SFT, and CuSeT. CuSeT consistently improves functional correctness over standard SFT across both small and large models, while also achieving higher CUDABench-Scores. These results demonstrate that CuSeT generalizes well across model scales and architectures, indicating that it can effectively capture CUDA-sensitive code structures across different backbone capacities. Overall, CuSeT exhibits strong scalability and robustness, delivering consistent gains in both functional correctness and kernel performance without additional training complexity or data overhead.

Table 4: Scalability evaluation of CuSeT across models of different sizes.

Method	Pass@1			Pass@3		
	Comp.	Func.	Score	Comp.	Func.	Score
Qwen3-4B						
Vanilla	70.6	23.0	9.80	82.2	33.6	13.21
+SFT	89.6	51.2	18.75	**98.4**	65.0	23.19
+CuSeT	**91.6**	**57.8**	**20.61**	97.6	**66.2**	**23.41**
Qwen2.5-Coder-32B-Instruct						
Vanilla	92.6	60.2	21.21	97.6	70.0	26.76
+SFT	94.4	68.0	21.23	98.6	79.6	27.23
+CuSeT	**96.0**	**75.6**	**23.23**	**98.8**	**81.0**	**27.32**

5.5 Ablation Study

We conduct ablation studies in Tab. 5 to evaluate each component of CuSeT. The results show consistent improvements from each design choice. (1) Introducing the confidence mask improves functional correctness over standard SFT, indicating that token-level confidence-based reweighting enhances supervision by focusing on high-confidence CUDA-sensitive tokens. (2) Adding the CUDA mask further improves functional correctness, demonstrating that explicitly preserving supervision for execution-critical tokens in kernel bodies and launch configurations provides complementary gains beyond confidence-based reweighting. (3) Incorporating region-aware sample reweighting leads to additional improvements in functional correctness, suggesting that emphasizing samples containing low-confidence CUDA-sensitive regions helps the model better learn difficult execution-critical patterns. (4) Combining all components yields the best overall functional correctness and CUDABench-Score, confirming that token-level and region-level CUDA sensitivity modeling are complementary and jointly improve CUDA kernel generation performance.

Table 5: Ablation of CuSeT components: (a) confidence mask, (b) CUDA mask, and (c) region-aware sample reweighting.

(a)	(b)	(c)	Pass@1			Pass@3		
			Comp.	Func.	Score	Comp.	Func.	Score
✘	✘	✘	92.2	55.0	19.18	96.6	61.0	21.57
✔	✘	✘	94.2	59.6	19.02	97.0	64.0	21.64
✔	✔	✘	**96.6**	62.8	21.61	**99.4**	70.4	23.98
✔	✘	✔	94.6	61.8	21.35	98.0	68.0	22.76
✔	✔	✔	93.6	**65.6**	**22.68**	98.8	**74.8**	**25.50**

6 Conclusion

In this work, we present **CUDA-Sensitive Instruction Tuning (CuSeT)**, a low-cost post-training method for improving LLM-based CUDA kernel generation. Motivated by the observation that CUDA sensitivity appears at both token and region levels, CuSeT enhances supervision over CUDA-sensitive structures within a simple SFT framework. Specifically, it integrates token-level confidence with a CUDA mask to preserve supervision on execution-critical tokens while maintaining coverage of critical code locations. At the same time, region-aware sample reweighting emphasizes training on samples containing low-confidence CUDA-sensitive regions. Extensive experiments across multiple model families and scales demonstrate that CuSeT consistently improves functional correctness and delivers strong kernel performance while maintaining high compilation success. CuSeT outperforms standard SFT and improved SFT variants, remains competitive with frontier CUDA kernel generation models, and significantly reduces inference token cost. Overall, CuSeT provides a simple yet effective way to incorporate CUDA sensitivity into post-training, enabling LLMs to better capture execution-constrained code structures and generate more reliable CUDA kernels.

Limitations

Although CuSeT consistently improves CUDA kernel generation across multiple model families and

scales, our evaluation is primarily conducted on CUDA and single kernel benchmarks. Its generalization to other GPU programming backends, and larger end-to-end systems remains to be further explored. In addition, CuSeT currently focuses on CUDA-sensitive supervision rather than directly optimizing execution performance. Future work can incorporate performance feedback and evaluate CuSeT in broader hardware and system-level settings.

References

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, et al. 2026a. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, et al. 2026b. Ascendkernelgen: A systematic study of llm-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*.

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

Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. 2025. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*.

Shane Cook. 2012. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

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

Muxi Diao, Lele Yang, Wuxuan Gong, Yutong Zhang, Zhonghao Yan, Yufei Han, Kongming Liang, Weiran Xu, and Zhanyu Ma. 2026. Entropy-adaptive fine-tuning: Resolving confident conflicts to mitigate forgetting. *arXiv preprint arXiv:2601.02151*.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. 2025. Stark: Strategic team of agents for refining kernels. *arXiv preprint arXiv:2510.16996*.

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

Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. 2025. Deep think with confidence. *arXiv preprint arXiv:2508.15260*.

Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. 2025. From large to small: Transferring cuda optimization expertise via reasoning graph. *arXiv preprint arXiv:2510.19873*.

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

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

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

Changxin Ke, Rui Zhang, Shuo Wang, Li Ding, Guangli Li, Yuanbo Wen, Shuoming Zhang, Ruiyuan Xu, Jin Qin, Jiaming Guo, et al. 2025. Qimeng-mupa: Mutual-supervised learning for sequential-to-parallel code translation. *arXiv preprint arXiv:2506.11153*.

David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

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

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. 2025. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. Technical report, Technical report, Sakana AI, 02 2025.

Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2025. Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization. *arXiv preprint arXiv:2511.06345*.

Gaotang Li, Ruizhong Qiu, Xiusi Chen, Heng Ji, and Hanghang Tong. 2025a. Beyond log likelihood: Probability-based objectives for supervised fine-tuning across the model capability continuum. *arXiv preprint arXiv:2510.00526*.

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

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025c. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Tao Liu, Taiqiang Wu, Runming Yang, Shaoning Sun, Junjie Wang, and Yujiu Yang. 2026a. Profit: Leveraging high-value signals in sft via probability-guided token selection. *arXiv preprint arXiv:2601.09195*.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. 2026b. Dr. kernel: Reinforcement learning done right for triton kernel generations. *arXiv preprint arXiv:2602.05885*.

Jiaqi Lv, Xufeng He, Yanchen Liu, Xu Dai, Aocheng Shen, Yinghao Li, Jiachen Hao, Jianrong Ding, Yang Hu, and Shouyi Yin. 2025. Hpctranscompile: An ai compiler generated dataset for high-performance cuda transpilation and llm preliminary exploration. *arXiv preprint arXiv:2506.10401*.

Frank J Massey Jr. 1951. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. 2025. Towards efficient generative large language model serving: A survey from algorithms to systems. *ACM Computing Surveys*, 58(1):1–37.

NVIDIA Corporation. Nvidia tensorrt. https://developer.nvidia.com/tensorrt/.

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

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Sahan Paliskara and Mark Saroufim. 2025. Kernelbook.

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

Tara Saba, Anne Ouyang, Xujie Si, and Fan Long. 2026. Cutegen: An llm-based agentic framework for generation and optimization of high-performance gpu kernels using cute. *arXiv preprint arXiv:2604.01489*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

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

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

Zhenchao Tang, Fang Wang, Haohuai He, Jiale Zhou, Tianxu Lv, Jun Zhu, Shouzhi Chen, Minghao Yang, Yu Wang, Jiayang Wu, et al. 2025. Aligning llms with biomedical knowledge using balanced fine-tuning. *arXiv preprint arXiv:2511.21075*.

Zecheng Wang, Deyuan Liu, Chunshan Li, Yupeng Zhang, Zhengyun Zhao, Dianhui Chu, Bingning Wang, and Dianbo Sui. 2026. Gradients must earn their influence: Unifying sft with generalized entropic objectives. *arXiv preprint arXiv:2602.11424*.

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

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, pages 52632–52657.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. 2022. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR.

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

Yongliang Wu, Yizhou Zhou, Zhou Ziheng, Yingzhe Peng, Xinyu Ye, Xinting Hu, Wenbo Zhu, Lu Qi, Ming-Hsuan Yang, and Xu Yang. 2025. On the generalization of sft: A reinforcement learning perspective with reward rectification. *arXiv preprint arXiv:2508.05629*.

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

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. 2025. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7.

Z.ai. 2026. Glm-5.1: Towards long-horizon tasks. https://z.ai/blog/glm-5.1. Official release blog.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pages 400–410.

He Zhu, Junyou Su, Peng Lai, Ren Ma, Wenjia Zhang, Linyi Yang, and Guanhua Chen. 2025. Anchored supervised fine-tuning. *arXiv preprint arXiv:2509.23753*.

Jiace Zhu, Wentao Chen, Qi Fan, Zhixing Ren, Junying Wu, Xing Zhe Chai, Chotiwit Rungrueangwutthinon, Yehan Ma, and An Zou. 2026. Cudabench: Benchmarking llms for text-to-cuda generation. *arXiv preprint arXiv:2603.02236*.

A Error Type Analysis

To better understand where CuSeT improves over standard SFT, we conduct a fine-grained analysis of functionally incorrect generated kernels. We categorize common failure modes observed during CUDA kernel generation into four types:

(1) Unsupported API/Primitive Usage,

(2) Execution Scope Error,

(3) Resource Violation,

(4) Host-Device Semantic Mismatch.

Tab. 6 summarizes the distribution of these error types in invalid kernels generated by representative frontier models.

We further compare standard SFT and CuSeT across the four error categories on three base models, using three generated kernels per task following the Pass@3 evaluation setting. Tab. 7 reports the number of errors and the corresponding reduction ratio (RR) achieved by CuSeT.

These results show that CuSeT consistently reduces errors across all four categories and model families. Notably, substantial reductions are observed in Execution Scope Error and Resource Violation for Qwen2.5-Coder-7B-Instruct and DeepSeek-Coder-6.7B-Instruct, while consistent improvements are also observed for Seed-Coder-8B-Instruct. These error types are closely related to execution-critical CUDA structures, including thread indexing, memory access, synchronization, and resource usage. This is consistent with the motivation of CuSeT, which strengthens supervision on CUDA-sensitive tokens and regions associated with such execution constraints.

B Case Study

We present a case study on the BatchNorm task to further analyze how CuSeT improves functional correctness, as shown in Fig. 4. The task requires computing column-wise batch normalization over an input tensor `matA` of shape $(8192, 2048)$, followed by affine transformation using parameters stored in `matB`.

The SFT-generated program in Fig. 4b is syntactically valid but fails to implement the correct execution semantics. Specifically, it assigns each thread to an individual (n, c) element and computes normalization statistics over a spatial dimension, which is inconsistent with the required column-wise reduction. Consequently, the resulting mean and variance do not correspond to the statistics of each column over the batch dimension, leading to incorrect outputs.

In contrast, as shown in Fig. 4c, the CuSeT-generated program maps each thread to a column and performs reduction over the batch dimension. It therefore correctly computes the mean and variance for each column before applying the affine transformation, matching the intended task semantics.

Overall, this example illustrates a typical Type (2) Execution Scope Error: an SFT model may generate syntactically valid CUDA code while incorrectly mapping threads and reduction dimensions to the intended computation. CuSeT better captures execution-critical CUDA-sensitive patterns, including thread mapping, reduction scope, and parameter layout, resulting in functionally correct execution.

C Sensitivity Analysis of Hyperparameters

We analyze the sensitivity of CuSeT to hyperparameters in this section. All experiments are conducted on Qwen2.5-Coder-7B-Instruct to ensure fair comparison.

Table 6: Distribution of four error types in invalid LLM-generated CUDA kernels.

Model	Type (1)	Type (2)	Type (3)	Type (4)
DeepSeek-V3.2-Thinking	10.86%	56.6%	23.44%	9.1%
GPT-5.2	23.81%	48.77%	16.51%	10.91%

Table 7: Error counts of standard SFT and CuSeT, together with the reduction ratio (RR) achieved by CuSeT.

Error Type	**SFT**	**CuSeT**	**RR**
Qwen2.5-Coder-7B-Instruct			
Unsupported API/Primitive Usage	67	54	19.40
Execution Scope Error	381	279	26.77
Resource Violation	96	70	**27.08**
Host–Device Semantic Mismatch	137	117	14.60
DeepSeek-Coder-6.7B-Instruct			
Unsupported API/Primitive Usage	52	46	11.54
Execution Scope Error	359	290	19.22
Resource Violation	96	77	**19.79**
Host–Device Semantic Mismatch	161	137	14.91
Seed-Coder-8B-Instruct			
Unsupported API/Primitive Usage	47	36	**23.40**
Execution Scope Error	332	268	19.28
Resource Violation	79	71	10.13
Host–Device Semantic Mismatch	123	100	18.70

Analysis of Confidence Mask Threshold τ. To study the effect of the confidence mask threshold τ, we vary it from 0.1 to 0.9 and report results in Fig. 5a. We observe that relatively low thresholds (e.g., $\tau = 0.1$ and $\tau = 0.3$) achieve the best overall performance. This indicates that a moderate confidence mask effectively removes low-confidence CUDA-neutral tokens while preserving CUDA-sensitive tokens. In contrast, larger thresholds lead to consistent performance degradation, as overly strict filtering removes important CUDA-sensitive tokens with lower confidence. These results suggest that a relatively lower threshold provides a better balance between suppressing low-confidence CUDA-neutral tokens and retaining CUDA-sensitive tokens.

Analysis of CUDA Mask Scale λ. We further study the effect of the CUDA mask scale λ, which controls the supervision strength applied to kernel bodies and launch configurations. As shown in Fig. 5b, a moderate value achieves the best performance, with $\lambda = 1$ yielding the highest functional correctness under both Pass@1 (65.6%) and Pass@3 (74.8%). When λ is too small, the CUDA mask provides insufficient additional supervision, limiting its ability to preserve execution-critical information. In contrast, overly large values overemphasize execution-critical code location and disrupt the balance between general code modeling and CUDA sensitivity learning. For example, increasing λ to 4 reduces Pass@1 functional correctness from 65.6% to 58.8%. These results suggest that a properly balanced CUDA mask scale is important for effectively enhancing supervision on execution-critical code locations while maintaining overall modeling stability.

Analysis of Sliding Window Size l. We further study the impact of the sliding window size l used for computing window confidence. As shown in Fig. 5c, the performance is sensitive to the choice of l. When the window size is too small (e.g., $l = 64$), the computed confidence mainly reflects short-range token fluctuations and provides a limited view of low-confidence CUDA-sensitive regions, leading to lower functional correctness. Increasing the window size to $l = 128$ and $l = 256$ improves performance, indicating that a moderate window is better at capturing execution-critical patterns in low-confidence CUDA-sensitive regions, such as shared memory allocation, indexing, and launch configuration. However, overly large windows degrade performance. When $l = 512$, Pass@1 functional correctness drops from 65.6% to 61.8%, suggesting that large windows over-smooth the confidence signal and mix low-confidence CUDA-sensitive regions with surrounding CUDA-neutral code. Overall, $l = 256$ achieves the best performance, suggesting that CuSeT benefits from a window size that is sufficiently large to capture execution-critical patterns at the region level, while avoiding excessive interference from surrounding CUDA-neutral code.

D Generalization Evaluation

To further validate the generalization capability of CuSeT, we conducted additional experiments on CUDABench Level 2 benchmark using Qwen2.5-Coder-7B-Instruct as the base model. As shown in Tab. 8, CuSeT consistently improves functional correctness over standard SFT and also achieves a higher CUDABench-Score. These results demon-

BatchNorm

Implement a CUDA kernel for batch normalization. The kernel takes two input tensors: `matA` of shape $(8192, 2048)$ and `matB` of shape $(2048, 2048)$, both of float32 type. The output tensor `matC_out` has shape $(8192, 2048)$ and float32 type. For each column of `matA`, compute the mean and variance of all elements in that column. Then, for each element in the column, normalize it by subtracting the mean and dividing by the square root of the variance plus $1e^{-5}$. Then, scale the normalized value by a γ parameter and add a β parameter. The γ and β parameters for column index j are stored in `matB`: γ at the j-th element of the first 2,048 elements (flattened) of `matB`, and β at the j-th element of the next 2,048 elements (flattened) of `matB`. The kernel must be implemented to handle the given tensor dimensions.

(a) Task description

```
__global__ void batch_norm_kernel(float* output, const float* input, const float* gamma, const float* beta,
                                  int N, int C, int H, int W, float eps) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < N * C; i += stride) {
        int n = i / C;
        int c = i % C;

        float mean = 0.0f;
        float var = 0.0f;

        for (int hw = 0; hw < H * W; ++hw) {
            int input_idx = n * C * H * W + c * H * W + hw;
            mean += input[input_idx];
        }
        mean /= (H * W);

        for (int hw = 0; hw < H * W; ++hw) {
            int input_idx = n * C * H * W + c * H * W + hw;
            float diff = input[input_idx] - mean;
            var += diff * diff;
        }
        var /= (H * W);

        float x_hat = (input[n * C * H * W + c * H * W] - mean) / sqrtf(var + eps);
        output[n * C * H * W + c * H * W] = gamma[c] * x_hat + beta[c];
    }
}
```

(b) SFT

```
__global__ void batch_norm_kernel(float* output, const float* input, const float* params,
                                  int N, int C, float eps) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= C) return;

    // Compute mean
    float mean = 0.0f;
    for (int i = 0; i < N; ++i) {
        mean += input[i * C + idx];
    }
    mean /= N;

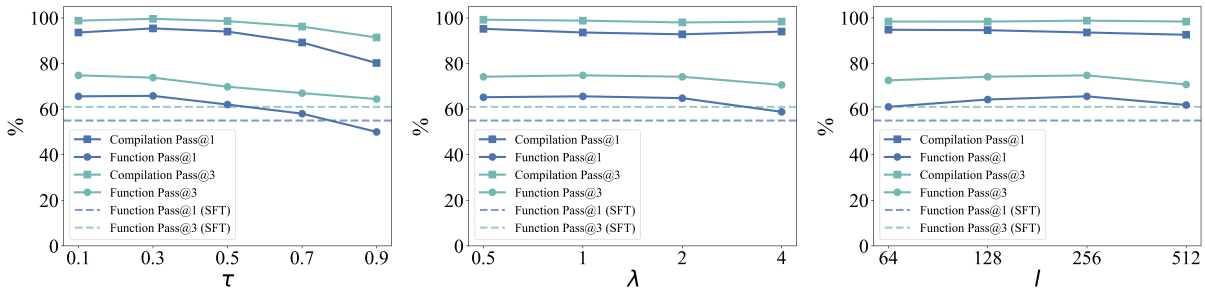
    // Compute variance
    float var = 0.0f;
    for (int i = 0; i < N; ++i) {
        float diff = input[i * C + idx] - mean;
        var += diff * diff;
    }
    var /= N;

    // Normalize and apply gamma/beta
    float inv_std = 1.0f / sqrtf(var + eps);
    float gamma = params[idx];
    float beta = params[C + idx];

    for (int i = 0; i < N; ++i) {
        output[i * C + idx] = gamma * (input[i * C + idx] - mean) * inv_std + beta;
    }
}
```

(c) CuSeT

Figure 4: Case study on the BatchNorm task showing that SFT fails to capture correct execution semantics, leading to incorrect normalization scope, while CuSeT produces correct column-wise reduction and functional correct outputs.



(a) (b) (c)

Figure 5: Ablation analysis of hyper-parameters: (a) confidence mask threshold τ, (b) CUDA mask scale λ, and (c) sliding window size l.

strate that the effectiveness of CuSeT generalizes beyond the original CUDABench Level 1 setting to more challenging CUDA kernel generation tasks.

Table 8: Generalization evaluation of CuSeT on the CUDABench Level 2 benchmark.

Method	**Pass@1**			**Pass@3**		
	Comp.	**Func.**	**Score**	**Comp.**	**Func.**	**Score**
Vanilla	71.2	35.6	16.77	84.4	48.8	21.06
+SFT	91.2	54.6	19.19	97.6	68.6	25.30
+CuSeT	**93.8**	**62.8**	**21.93**	**98.4**	**70.6**	**25.56**