# CUDA-HARNESS: HARNESSING AGENTIC CUDA KERNEL GENERATION AND OPTIMIZATION FROM NATURAL LANGUAGE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

The powerful code generation capabilities of Large Language Models (LLMs) have advanced LLM-based CUDA kernel generation, a task that inherently demands specialized expertise. However, existing methods mainly focus on transpilation from high-level frameworks like PyTorch to CUDA (Torch2CUDA) rather than the general Text2CUDA, and remain vulnerable to reward hacking due to reliance on predefined test inputs. To address these issues, we propose CUDA-Harness, a framework for harnessing agentic CUDA kernel generation and optimization from natural language by integrating Intermediate-Structured Generation, Synthesis-Based Verification, and Feedback-Adaptive Evolution. Finally, through extensive experiments, we demonstrate the effectiveness of CUDA-Harness, with further evaluations illustrating generalization across LLMs, hardware platforms, and to C-to-CUDA transpilation.

## 1 INTRODUCTION

With the advancement of high-performance computing, CUDA has become the dominant platform for parallel computing, powering applications from artificial intelligence (Paszke et al., 2019; Shah et al., 2024) to quantum computing (Guo et al., 2025) and scientific simulation (Xie et al., 2025). However, developing high-performance kernels remains challenging, as it requires not only concise knowledge of algorithm implementation and testing details, but also specialized understanding of hardware architectures and parallel optimization. These requirements create a substantial expertise barrier that limits the accessibility of GPU programming. Hence, generating CUDA kernels directly from natural language (Text2CUDA) is essential to bridge the gap between high-level application needs and low-level parallel implementation.

Large Language Models (LLMs) have demonstrated remarkable capabilities in general-purpose code generation (Zheng et al., 2025), prompting a series of studies (Yu et al., 2026) that leverage LLMs to produce high-performance CUDA kernels. Nevertheless, existing works (Ouyang et al., 2025; Dong et al., 2025) primarily focus on transpilation from high-level frameworks (e.g., Torch2CUDA, which transpiles PyTorch to CUDA) rather than Text2CUDA. The distinction is fundamental. Transpilation starts from programs whose algorithmic details are already specified, whereas Text2CUDA starts from natural language descriptions that can be incomplete and underspecified. Hence, Text2CUDA requires the model to more deeply understand the high-level intended semantics while simultaneously handling low-level kernel implementation and validation, as illustrated in Fig. 1. Therefore, a Text2CUDA framework that systematically addresses input semantic understanding is essential.

Existing LLM-based CUDA kernel generation approaches can be broadly categorized into two types: training-based and agent-based. Training-based approaches (Li et al., 2025; Dai et al., 2026; Du et al., 2026) enhance the model's CUDA kernel generation capability through supervised fine-tuning or reinforcement learning. Nevertheless, they heavily rely on high-quality kernel implementations and optimization trajectories to build the training data (Zhou et al., 2023), which is scarce and expensive in practice. Agent-based approaches (Dong et al., 2025; Zhang et al., 2025; Chen et al., 2025a) leverage multi-agent collaboration and self-refinement (Madaan et al., 2023) to achieve training-free kernel generation and optimization. However, they typically depend on predefined test inputs for validation and feedback construction during iterative refinement, which may lead to re-



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.

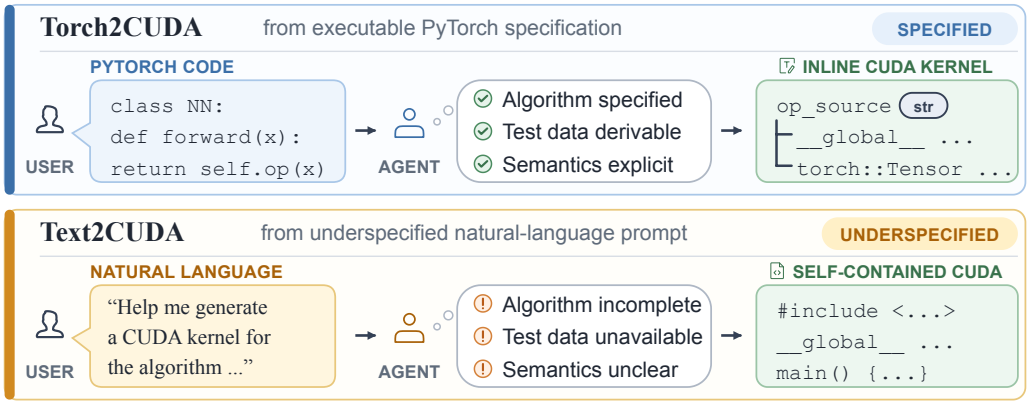



Figure 1: Text2CUDA vs. Torch2CUDA. Torch2CUDA starts from detailed PyTorch code, while Text2CUDA starts from an underspecified natural-language prompt with no available test data.

ward hacking (Denison et al., 2024) and optimization over flawed implementations. Therefore, harnessing agentic CUDA generation is critical for generating correct and efficient CUDA kernels.

In this paper, we propose CUDA-Harness, a framework for harnessing agentic CUDA kernel generation and optimization from natural language. With three components for kernel generation, verification, and optimization, it connects input semantic understanding with verification-guided refinement to improve correctness and efficiency. Empirically, CUDA-Harness shows that natural language can serve as a practical interface for correct and high-performance CUDA kernel development. The contributions are fourfold.

- We introduce Intermediate-Structured Generation, bridging the gap between high-level semantic understanding and low-level kernel synthesis for Text2CUDA.
- We construct Synthesis-Based Verification, providing isolated test data synthesis and progressive validation to dilute reward hacking in Text2CUDA.
- We propose Feedback-Adaptive Evolution, prioritizing correctness to avoid error accumulation during optimization.
- We conduct extensive experiments demonstrating the effectiveness of CUDA-Harness, while extended evaluations illustrate its generalizability across LLMs, hardware platforms, and to C-to-CUDA transpilation.

## 2 Related Work

Developing high-performance CUDA kernels remains challenging, as it requires jointly handling algorithmic correctness, verification, hardware-specific constraints, and parallel optimization. Motivated by these challenges and the strong capabilities of LLMs, LLM-based kernel generation has gained significant attention, which can be broadly divided into training-based and agent-based approaches.

Training-based methods (Li et al., 2025; Kong et al., 2025; Zhu et al., 2026b; Dai et al., 2026; Du et al., 2026) enhance the model's capability to generate high-performance CUDA kernels through large-scale post-training, including supervised fine-tuning (SFT) and reinforcement learning (RL). However, these methods rely on expensive high-quality kernel implementations and optimization traces in practice. Agent-based methods (Chen et al., 2025a; Dong et al., 2025; Gong et al., 2025; Zhang et al., 2025) generate and optimize kernels at test time via multi-agent collaboration and iterative refinement (Madaan et al., 2023). Nevertheless, due to dependence on predefined test inputs to guide optimization, they are typically vulnerable to reward hacking. Although existing works have achieved notable success, they primarily focus on transpilation such as Torch2CUDA rather than the more general Text2CUDA. For Text2CUDA, existing efforts (Chen et al., 2025b; Zhu et al., 2026a) remain constrained by the limitations of existing LLM-based kernel generation methods. Therefore, a framework for harnessing agentic Text2CUDA is essential.

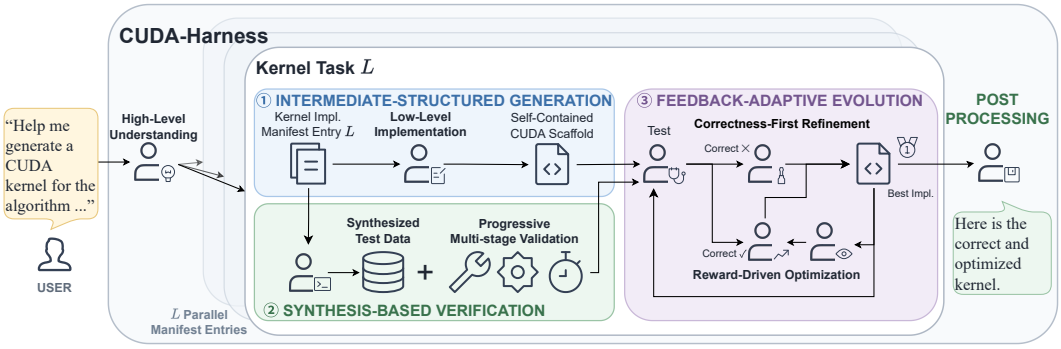



Figure 2: The overview of the CUDA-Harness.

## 3 Overview

As the self-contained CUDA program serves as the basic unit providing the complete context to compile, execute, and validate a kernel, we analyze its composition and execution model and present our CUDA-Harness in this section.

### 3.1 Anatomy of Self-Contained CUDA

A self-contained CUDA program typically comprises necessary headers, device-side kernels and host-side code. Fig. 3 in Appendix. A showcases a simplified example of the self-contained CUDA program. Specifically, the device-side kernels perform memory accesses and execute the parallel computation, while the host-side code prepares the inputs, manages the memory, and launches kernels with appropriate launch configurations.

Beyond the structure, the execution depends on well-prepared test inputs for validation and further performance measurement. Additionally, as problem scale and input dimensions directly influence the performance bottlenecks observed in CUDA kernels at runtime, dummy inputs are insufficient.

In Torch2CUDA, the input PyTorch code already reveals algorithmic detail and naturally provides a basis for test construction. However, Text2CUDA takes only high-level but often underspecified natural language intents as input, lacking the test data needed for validation.

### 3.2 From Anatomy to CUDA-Harness

According to the anatomy of the self-contained CUDA program, the challenges of Text2CUDA are:

***Challenge 1 [Bridging Intent and Implementation]****: How to enable the agent to deeply understand the high-level natural language intents and complete the missing details is the first challenge. In addition, how to use the completed information to guide the agent towards easily testable kernel implementations is another challenge.*

***Challenge 2 [Missing Test Data for Validation]****: How to synthesize test data for kernel validation is the first challenge. Furthermore, how to avoid reward hacking of kernel implementation due to test data synthesis is another challenge.*

We propose the CUDA-Harness to address the above challenges. The overview of the CUDA-Harness is presented in Fig. 2. ① Intermediate-Structured Generation introduces two intermediate blueprints, namely the kernel implementation manifest and the self-contained CUDA scaffold. Given a natural language input, the agent first produces the implementation manifest to externalize semantic understanding and implementation planning. For each entry in the manifest, the agent then instantiates the CUDA scaffold and focuses on the concrete implementation of the corresponding kernel. ② Synthesis-Based Verification decouples test data synthesis from kernel generation, allowing the agent to construct test inputs in an isolated environment for validation. Based on the feedback from the progressive validation, ③ Feedback-Adaptive Evolution optimizes the kernel for better performance while preserving correctness. Once all manifest entries are implemented,

CUDA-Harness collects the best-performing kernels together with the corresponding kernel launchers for downstream post-processing.

# 4 CUDA-HARNESS

This section presents the components of CUDA-Harness.

## 4.1 INTERMEDIATE-STRUCTURED GENERATION

Within Intermediate-Structured Generation, the kernel implementation manifest enhances high-level semantic understanding, while the self-contained CUDA scaffold supports low-level kernel implementation.

### 4.1.1 KERNEL IMPLEMENTATION MANIFEST

The kernel implementation manifest is a structured per-kernel specification, where each entry stands alone. Each entry records the essential information to implement a kernel, including the kernel name, the input and output descriptions of the kernel, a functional description that reveals algorithmic detail, and a language-agnostic reference code snippet. Additionally, each entry carries detailed test information, encompassing the shapes and data types of the test inputs and outputs.

When the agent produces the manifest from the natural language input, it does not focus on writing code for concrete kernel implementation. Therefore, the agent can concentrate on understanding the high-level semantics and on completing details that the original input leaves underspecified.

### 4.1.2 SELF-CONTAINED CUDA SCAFFOLD

The self-contained CUDA scaffold is a code template that tells the agent how to implement the self-contained CUDA program based on the manifest entry. Fig. 4 in Appendix. B illustrates a simplified example of the self-contained CUDA scaffold. The scaffold includes placeholders for the components of the self-contained CUDA program illustrated in Sec. 3.1.

Within the scaffold, we aggregate the operations related to kernel launch, including memory management, data transfer, and launch configuration definition, into a dedicated launcher function rather than scattering it across the host-side code. With this arrangement, the agent can concentrate on the kernel and its launcher and ignore the remaining host-side details. However, because the launch configuration determines the memory-access pattern of the kernel, the order in which kernel and launcher are generated matters. During initial kernel generation, the agent assumes a reasonable memory access pattern. However, when generating the launcher, the launch configuration is specified. When the implicitly assumed memory access pattern in the kernel conflicts with the explicitly defined launch configuration in the launcher, the agent will reflect (Shinn et al., 2023) and retrospectively edit the already generated kernel, producing invalid CUDA programs. Therefore,

Moreover, as an initial step towards mitigating reward hacking, the scaffold prohibits the use of dummy inputs and provides helper functions for reading the well-prepared test inputs. The test input paths are passed as command-line arguments, and the calling convention is emitted at compile time. Thus, the agent remains unaware of the details of the test inputs when implementing the kernel and cannot tailor an overfitted implementation to them.

## 4.2 SYNTHESIS-BASED VERIFICATION

To address the lack of test data in Text2CUDA, Synthesis-Based Verification enables decoupled test data synthesis and evaluates the correctness and performance through progressive multi-stage validation. Rather than theoretically eliminating the risk of reward hacking, the purpose of Synthesis-Based Verification is to provide a trustworthy correctness signal for kernel refinement when benchmark test data is unavailable.

#### 4.2.1 DECOUPLED TEST DATA SYNTHESIS

Although the self-contained CUDA scaffold eliminates a pathway for reward hacking by using well-prepared test inputs, there remain risks of reward hacking during test data synthesis. If test data synthesis shares context with kernel generation, the agent may adapt test inputs and reference outputs to the kernel implementation rather than to the intended algorithm. Therefore, we decouple test data synthesis and kernel generation. Specifically, the test inputs and reference outputs are produced through standard numerical libraries (e.g., NumPy and PyTorch) in an isolated environment, where context isolation is enforced. Namely, kernel generation and test data synthesis use separate contexts to avoid cross-contamination.

However, decoupling test data synthesis is insufficient, as reward hacking can still arise through the validation rule due to how the test data is synthesized. The functional validation for a kernel $\mathcal{K}$ under test, with test inputs $x$ and reference outputs $y$, is defined as

$$\mathcal{V}(\mathcal{K}, x, y) = \mathbb{I}\Big[|\mathcal{K}(x) - y| \leqslant \epsilon_{\text{atol}} + \epsilon_{\text{rtol}}|y|\Big] \tag{1}$$

where $\mathcal{K}(x)$ denotes the kernel outputs, $\mathbb{I}$ is the indicator function, $\epsilon_{\text{atol}}$ and $\epsilon_{\text{rtol}}$ are the absolute and relative tolerances, respectively. According to Eq. equation 1, when the test inputs concentrate on small-magnitude values, the output magnitude $|y|$ also tends to be small, causing $\epsilon_{\text{rtol}}|y|$ to diminish and $\epsilon_{\text{atol}}$ to dominate the criterion. In such cases, a numerically incorrect kernel may still pass the validation, leading to reward hacking when this feedback is used for optimization.

Hence, we require the agent to draw test inputs from a broad numeric range. For instance, test inputs following a Gaussian distribution with a mean of 5.0 and a standard deviation of 10.0 are required to induce distribution shift and cover broader numeric ranges than dummy or small-magnitude inputs. However, larger-magnitude inputs may amplify floating-point accumulation error, which makes validation stricter and can increase false negatives (i.e., rejecting correct kernels). Nevertheless, we adopt this trade-off deliberately to reduce false positives (i.e., accepting incorrect kernels), since accepting an incorrect kernel would mislead subsequent optimization.

#### 4.2.2 PROGRESSIVE MULTI-STAGE VALIDATION

Given synthesized test data, CUDA-Harness employs a progressive multi-stage protocol for kernel validation. As validation stages are naturally ordered, with performance mattering only after a kernel compiles and produces correct outputs, CUDA-Harness rejects invalid kernels early to avoid unnecessary runs on target hardware. Specifically, CUDA-Harness follows a fail-fast order that checks compilation first, then functional correctness, and finally runtime performance.

To support the validation protocol, CUDA-Harness exposes a validation toolkit based on the Model Context Protocol (MCP) (Anthropic, 2024). These tools provide a uniform interface to the agent, while their concrete execution remains bound to the environment where the toolkit is deployed, enabling hardware-scalable validation. The toolkit comprises tools that handle compilation, functional validation, and performance profiling.

The compilation tool checks whether a kernel $\mathcal{K}$ can be built successfully under a given compilation setting, which can be represented as

$$\mathcal{C}(\mathcal{K}; \phi) = \mathbb{I}\Big[\mathcal{K} \text{ compiles under } \phi\Big] \tag{2}$$

where $\phi$ is the compilation parameters, such as `"-arch=sm_80"`. Given the test inputs $x$ and reference outputs $y$, the functional validation tool verifies whether the kernel outputs match the reference, producing the result $\mathcal{V}(\mathcal{K}, x, y)$. Utilizing profilers such as Nsight Systems and Nsight Compute, the performance profiling tool can measure the runtime performance $\mathcal{P}(\mathcal{K}, x)$, where a larger $\mathcal{P}(\cdot)$ indicates a better performance. Taking latency as an example, the performance can be defined as

$$\mathcal{P}_{\text{latency}}(\mathcal{K}, x) = 1 \,/\, \text{Latency}(\mathcal{K}, x) \tag{3}$$

Integrating these tools, the agent first invokes the compilation tool and is informed of the calling convention emitted by the scaffold via a compile-time `#pragma message` as described in Sec. 4.1.2. When the kernel passes compilation, the agent calls the functional validation tool based on the calling convention, followed by the performance profiling tool upon successful validation.

### 4.3 FEEDBACK-ADAPTIVE EVOLUTION

To optimize the kernel while preserving correctness, Feedback-Adaptive Evolution combines correctness-first refinement and reward-driven optimization to perform test-time kernel evolution.

#### 4.3.1 CORRECTNESS-FIRST REFINEMENT

As performance only becomes meaningful after correctness is in place, the agent must care about whether the generated kernel compiles and produces the correct outputs. However, the agent tends to introduce advanced and aggressive optimization strategies before the implementation is fully stable and the correctness is guaranteed. When optimization is built on a kernel that looks high-performance but is functionally flawed, subsequent refinement typically remains trapped in the same faulty pattern. Therefore, CUDA-Harness introduces correctness-first refinement into test-time kernel evolution.

The refinement branches based on which validation stage reports the failure. If the generated kernel fails to compile, the agent is steered towards a specific repair based on the error details provided by the compilation tool. When the kernel fails functional validation, the agent is instructed to retreat to a more conservative implementation, prioritizing correctness above all else.

Through correctness-first refinement, once the kernel is reset to a simpler implementation that compiles and passes functional validation, correctness is initially guaranteed. Therefore, subsequent performance optimization can proceed from a trustworthy and well-behaved base.

#### 4.3.2 REWARD-DRIVEN OPTIMIZATION

Building on correctness-first refinement, CUDA-Harness performs reward-driven optimization for the test-time kernel evolution. Benefiting from the progressive multi-stage validation in Sec. 4.2.2, there is a natural and verifiable reward signal characterizing kernel quality. Since the signal comes from compilation, functional validation, and profiling on the target hardware, it is measurable and trustworthy rather than estimated. Therefore, we employ Reinforcement Learning with Verifiable Reward (RLVR) (Lambert et al., 2024) to drive optimization. With compilation parameters $\phi$, test inputs $x$, and reference outputs $y$ fixed, the results of compilation, functional validation, and performance profiling for a generated kernel $\mathcal{K}$ can be simplified as $\mathcal{C}(\mathcal{K})$, $\mathcal{V}(\mathcal{K})$ and $\mathcal{P}(\mathcal{K})$, respectively. Hence, the reward signal can be defined as

$$r(\mathcal{K}) = -1 + \mathcal{C}(\mathcal{K}) + \mathcal{C}(\mathcal{K})\mathcal{V}(\mathcal{K})\mathcal{P}(\mathcal{K}) \tag{4}$$

Concretely, a penalty is applied when the generated kernel fails to compile. If the kernel compiles and passes functional validation, the performance determines the reward, with higher performance yielding a higher reward.

However, there are no updatable parameters to drive the optimization. Hence, we introduce optimization insight summarization, in which the agent extracts and updates the experience from what changed and how the reward moved to steer the optimization. Given the pre-optimization kernel $\mathcal{K}_{\text{pre}}$ and the optimized kernel $\mathcal{K}_{\text{post}}$, the optimization insight summarization process is defined as

$$e = \mathcal{M}_e(\text{Diff}(\mathcal{K}_{\text{post}}, \mathcal{K}_{\text{pre}}), r(\mathcal{K}_{\text{post}}) - r(\mathcal{K}_{\text{pre}})) \tag{5}$$

where the agent $\mathcal{M}_e$ analyzes the difference in implementation and reward, and distills the optimization experience $e$, such as which optimization strategies are effective or ineffective.

Thus, given $N$ rounds of iterative optimization and $G$ generated kernels per round, and assuming the accumulated optimization insights at the $i$-th round are $E_{1:i}$, reward-driven optimization is written as the following objective:

$$\mathcal{J}(E_i) = \max \mathbb{E}_{\{\mathcal{K}_{i+1,j}\}_{j=1}^{G} \sim \mathcal{M}(\cdot|\mathcal{K}_i, E_{1:i})} \left[\Delta r_{i,j}\right] \tag{6}$$

where $\mathcal{M}$ is the optimization agent, $\mathcal{K}_i$ is the best kernel at the $i$-th round, $\mathcal{K}_{i+1,j}$ is the $j$-th generated kernel for round $i+1$, $\Delta r_{i,j} = r(\mathcal{K}_{i+1,j}) - r(\mathcal{K}_i)$ is the reward difference, and $E_{1:i} = \{E_1, \ldots, E_i\}$ accumulates the optimization insights from round 1 to $i$. The set $E_i$ collects the $G$ optimization insights from round $i$, namely $E_i = \{\mathcal{M}_e(\text{Diff}(\mathcal{K}_{i-1,j}, \mathcal{K}_{i,j}), r(\mathcal{K}_{i,j}) - r(\mathcal{K}_{i-1,j}))\}_{j=1}^{G}$.

Table 1: The comparison and ablation results. CUDA-ISG retains only Intermediate-Structured Generation, while CUDA-Harness* further removes Correctness-first Refinement from CUDA-Harness.

| Method | Overall | | | Level 1 | | | Level 2 | | | Level 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Comp. | Func. | RScore | Comp. | Func. | RScore | Comp. | Func. | RScore | Comp. | Func. | RScore |
| Baseline | 90.1 | 60.9 | 80.1 | 89.6 | 69.0 | 87.7 | 92.6 | 66.2 | 86.1 | 88.2 | 47.6 | 66.6 |
| OpenCode | 92.1 | 60.6 | 78.7 | 93.6 | 66.6 | 82.8 | 92.0 | 65.8 | 81.9 | 90.8 | 49.4 | 71.3 |
| Codex | 98.3 | 69.5 | 86.5 | 99.2 | 79.2 | 91.3 | 98.6 | 75.4 | 94.5 | 97.2 | 54.0 | 73.6 |
| KernelSkill | 99.4 | 63.7 | 86.1 | 99.0 | 72.0 | 94.4 | 100.0 | 70.4 | 94.3 | 99.2 | 48.6 | 69.7 |
| CudaForge | 99.8 | 67.7 | 87.9 | 99.6 | 75.2 | 92.7 | 99.8 | 73.2 | 93.7 | 100.0 | 54.8 | 77.4 |
| CUDA-ISG | 94.5 | 63.9 | 84.4 | 93.4 | 72.0 | 91.5 | 96.4 | 69.2 | 91.1 | 93.6 | 50.4 | 70.6 |
| CUDA-Harness* | 99.3 | 70.8 | 94.9 | 99.6 | 79.0 | 100.6 | 99.6 | 75.8 | 101.3 | 98.6 | 57.6 | 82.8 |
| **CUDA-Harness** | **99.7** | **80.5** | **110.7** | **99.4** | **86.2** | **116.2** | **100.0** | **84.8** | **115.6** | **99.8** | **70.4** | **100.4** |

# 5 EVALUATION

In this section, the experimental setup is first introduced. The comparison results and ablation studies are presented in Sec. 5.2. The generalizability of CUDA-Harness is evaluated in Sec. 5.3 under cross-LLM, cross-hardware, and C-to-CUDA transpilation scenarios.

## 5.1 EXPERIMENTAL SETUP

We evaluate on CUDABench (Zhu et al., 2026a), which comprises three difficulty levels of prompts that progressively remove details. As CUDA-Harness is decoupled from the underlying LLM, we use Seed2.0 Lite (`doubao-seed-2-0-lite-260215`) (ByteDance Seed Team, 2026) with thinking mode disabled as the invoked LLM. The model is fixed throughout the evaluation, isolating the effect of the harness from variation across LLMs. For each kernel, generation is performed only once, while repetitions are used solely during the performance-measurement stage. The experiments are conducted on an NVIDIA A40 GPU, which features the Ampere architecture and `sm_80` compute capability.

### 5.1.1 CUDA-HARNESS SETUP

The agent operates under the reasoning-action (ReAct) paradigm (Yao et al., 2022). The number of iterative optimization rounds is set to $N = 3$, with $G = 3$ kernels generated per round. Leveraging Nsight Systems, the performance profiling tool measures the execution time of the kernel over multiple runs and uses the average latency to compute the performance metric $\mathcal{P}(\cdot) = \mathcal{P}_{\text{latency}}(\cdot)$, as defined in Eq. equation 3. Following the evolution process detailed in Sec. 4.3, optimization thus proceeds towards a correct and fast kernel.

### 5.1.2 EVALUATION METRICS

Using the compilation parameters and test data provided by the benchmark, we report two correctness metrics, the compilation success rate (Comp.) and functional correctness rate (Func.). Since CUDA-Harness uses execution time to guide test-time evolution, for a fair comparison, we define the runtime performance score (RScore) as $1/$`execution time (ms)` computed over multiple measurements, which is not a percentage. The average RScore is reported, with kernels failing the correctness validation assigned a RScore of 0.

## 5.2 MAIN RESULTS

### 5.2.1 COMPARISON WITH BASELINES

We compare CUDA-Harness with the official CUDABench (Zhu et al., 2026a) evaluation procedure, which serves as the baseline harness. Additionally, we further compare the general harnesses, OpenCode (anomalyco, 2026) and Codex (openai, 2026), and agentic Torch2CUDA systems, KernelSkill (Sun et al., 2026) and CudaForge (Zhang et al., 2025). The results are presented in Tab. 1. Across all difficulty levels, CUDA-Harness outperforms the baseline in terms of compilation success rate, functional correctness rate, and the performance score, demonstrating the effectiveness of CUDA-Harness.

Table 3: The generalization results of CUDA-Harness across LLMs and across hardware platforms.

| Method | Metric | Cross-LLMs | | | Cross-Hardware | | |
|---|---|---|---|---|---|---|---|
| | | Seed 2.0 Lite | DeepSeek V3.2 | GLM 5.1 | A40 | 1660 SUPER | Jetson AGX Orin |
| Baseline | Comp. | 90.1 | 96.5 | 94.8 | 90.1 | 90.5 | 90.2 |
| | Func. | 60.9 | 57.8 | 60.7 | 60.9 | 60.3 | 59.9 |
| | RScore | 80.1 | 79.0 | 79.7 | 80.1 | 52.7 | 29.9 |
| **CUDA-Harness** | Comp. | **99.7** | **99.6** | **99.2** | **99.7** | **99.7** | **99.7** |
| | Func. | **80.5** | **76.3** | **101.8** | **80.5** | **79.8** | **79.4** |
| | RScore | **110.7** | **80.1** | **107.8** | **110.7** | **73.9** | **42.4** |

As the difficulty level increases, the performance gap between the baseline and CUDA-Harness grows, where higher levels indicate less information in the prompt. Since the baseline generates kernels directly from the raw prompt, it is forced to infer the high-level intents while simultaneously handling low-level implementation within a single generation pass. In contrast, CUDA-Harness introduces Intermediate-Structured Generation to complete underspecified details, allowing it to remain competitive even at Level 3.

### 5.2.2 Ablation Study

To isolate the contribution of each component, we incrementally enable them in Tab. 1.

CUDA-ISG retains only Intermediate-Structured Generation (Sec. 4.1). CUDA-ISG outperforms the baseline, demonstrating the effectiveness and the necessity of the proposed intermediate blueprints.

CUDA-Harness* adds Synthesis-Based Verification (Sec. 4.2) and Reward-Driven Optimization (Sec. 4.3.2) on top of CUDA-ISG while withholding Correctness-First Refinement (Sec. 4.3.1). Compared with CUDA-ISG, its higher compilation success rate and performance score shows that Reward-Driven Optimization is effective at repairing compilation failures and optimizing the generated kernels. However, the remaining gap to the full CUDA-Harness reveals why Correctness-First Refinement matters. Through Correctness-First Refinement, CUDA-Harness achieves a higher functional correctness rate than CUDA-Harness*, further confirming the importance of correctness-first refinement in preserving correctness throughout the optimization process.

### 5.2.3 Efficacy of the Synthesis-Based Verification

We evaluate whether the proposed Synthesis-Based Verification can provide a trustworthy correctness signal for CUDA-Harness. Since the benchmark test data is unavailable during test-time evolution, the synthesized data should lead to validation results that are consistent with the benchmark validation results. To examine the consistency, Tab. 2 compares the validation results obtained using synthesized test data with those obtained using benchmark test data.

As shown in Tab. 2, Synthesis-Based Verification is largely consistent with benchmark validation. Treating benchmark validation as the reference and Synthesis-Based Verification as the prediction, Table 2 shows an accuracy of 84.4%, with precision 89.4% and recall 91.4%. These results indicate that Synthesis-Based Verification provides a strong correctness signal for test-time evolution.

Table 2: The consistency between Synthesis-Based Verification and benchmark validation.

| Synthesis-based Verification | Benchmark Validation | |
|---|---|---|
| | Pass | Fail |
| Pass | **73.6%** | 8.7% |
| Fail | 6.9% | **10.8%** |

Furthermore, we manually analyze the false positives (8.7% of all cases) where Synthesis-Based Verification passes while benchmark validation fails. Among these false positive cases, 38.1% stem from prompt misinterpretation (e.g., interpreting $A^3$ as element-wise cubing), which misleads the subsequent test data synthesis and kernel generation. Another 30.5% arise from omitting steps during benchmark-specific post-processing (e.g., averaging the kernel outputs after execution). The remaining 31.4% are caused by a defect in the CUDABench template that loads test data as `float`, causing incorrect loading for underlying data types such as `uint8` and `int32`.

### 5.3 Generalization Analysis

#### 5.3.1 Generalization across LLMs

As CUDA-Harness is decoupled from the underlying LLM, we further evaluate whether it generalizes across different LLMs. Besides Seed2.0 Lite (ByteDance Seed Team, 2026), we invoke DeepSeek-V3.2 (Liu et al., 2025) and GLM-5.1 (Zeng et al., 2026) as the underlying LLMs, and compare CUDA-Harness with the baseline harness on CUDABench using an NVIDIA A40 GPU. As shown in Tab. 3, CUDA-Harness consistently outperforms the corresponding baseline across all three LLMs in terms of compilation success rate, functional correctness rate, and runtime performance score. These results substantiate the LLM-agnostic design of CUDA-Harness, as its improvements do not rely on a specific underlying LLM.

#### 5.3.2 Generalization across Hardware

We evaluate CUDA-Harness using Seed2.0 Lite on three GPUs with distinct architectures and compute capabilities, including the NVIDIA A40, the NVIDIA GeForce GTX 1660 SUPER, and the NVIDIA Jetson AGX Orin. The benchmark and the utilized LLM remain identical to Sec. 5.2. As shown in Tab. 3, the experimental results vary across hardware platforms due to differences in software (e.g., compilers) versions and hardware compute capabilities. However, CUDA-Harness maintains close compilation success rates and functional correctness rates across the three platforms. Although the performance score changes with the hardware compute capability, CUDA-Harness consistently outperforms the baseline on each platform. These results demonstrate that CUDA-Harness is not tailored to a single hardware environment and can generalize across different hardware platforms.

#### 5.3.3 Generalization to C-to-CUDA

We further evaluate under the C-to-CUDA transpilation scenario, where the input is sequential C code rather than natural language, and we evaluate on BabelTower (Wen et al., 2022; Ke et al., 2025). C code provides comprehensive semantics, including test inputs and outputs descriptions and algorithmic implementation details. For CUDA-Harness, the number of iterative optimization rounds is set to $N = 1$, with $G = 1$ kernels generated per round.

As shown in Tab. 4, CUDA-Harness outperforms the baseline, even though the evolution budget is relatively small, illustrating that CUDA-Harness can easily generalize to C-to-CUDA transpilation well. Since Intermediate-Structured Generation is not tied to natural language input, decoupling understanding and implementation helps the agent capture the C code and generate the CUDA kernel.

Table 4: The generalization results of CUDA-Harness to C-to-CUDA transpilation.

| Method | Comp. | Func. | Score |
|---|---|---|---|
| Baseline | 94.8 | 90.1 | 451.6 |
| **CUDA-Harness** | **98.7** | **97.1** | **514.8** |

## 6 Conclusion

In this paper, we propose CUDA-Harness, a framework for harnessing agentic CUDA kernel generation and optimization from natural language. To address the challenges in Text2CUDA, we introduce Intermediate-Structured Generation to bridge the gap between high-level intents and low-level implementation. We develop Synthesis-Based Verification, providing isolated test data synthesis and progressive validation. We propose Feedback-Adaptive Evolution to prioritize correctness while optimizing kernel performance during test-time evolution. Through extensive experiments, we demonstrate the effectiveness of CUDA-Harness. Additionally, we illustrate that CUDA-Harness generalizes across diverse hardware platforms and extends effectively to C-to-CUDA transpilation.

## AI use statement

In this work, we used generative AI tools to assist with translation. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims,

propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, support qualitative and thematic data analysis, or interpret results. Generating synthetic data sets, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and cleaning and reformatting datasets are not applicable to this work. Additionally, we used generative AI tools to check the grammatical errors and polish the language to improve readability. We have reviewed all AI-assisted work. We have checked LLM-translated content by manually consulting dictionaries to verify semantic accuracy. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES


anomalyco. opencode, 2026. URL https://github.com/anomalyco/opencode.

Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, 2024.

ByteDance Seed Team. Seed2.0 Model Card: Towards Intelligence Frontier for Real-World Complexity. https://lf3-static.bytednsdoc.com/obj/eden-cn/lapzild-tss/ljhwZthlaukjlkulzlp/seed2/0214/Seed2.0%20Model%20Card.pdf, 2026.

Jinwu Chen, Qidie Wu, Bin Li, Lin Ma, Xin Si, Yang Hu, Shouyi Yin, and Jun Yang. cupilot: A strategy-coordinated multi-agent framework for cuda kernel evolution. *arXiv preprint arXiv:2512.16465*, 2025a.

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

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.

Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, et al. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*, 2024.

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

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, et al. Kernel-smith: A unified recipe for evolutionary kernel optimization. *arXiv preprint arXiv:2603.28342*, 2026.

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

Ziqing Guo, Jan Balewski, and Ziwen Pan. Q-gear: Improving quantum simulation framework. In *Proceedings of the 54th International Conference on Parallel Processing*, pp. 638–647, 2025.

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

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

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

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

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

openai. codex, 2026. URL `https://github.com/openai/codex`.

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

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. Kernelskill: A multi-agent framework for gpu kernel optimization. *arXiv preprint arXiv:2603.10085*, 2026.

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

Jianbin Xie, Xingru Feng, Tianhai Gao, Changming Dong, Baoshu Yin, and Changmao Wu. Accelerating an implicit ocean model using cuda c. *Applied Ocean Research*, 163:104740, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, et al. Towards automated kernel generation in the era of llms. *arXiv preprint arXiv:2601.15727*, 2026.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

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.

Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, et al. Livecodebench pro: How do olympiad medalists judge llms in competitive programming? *arXiv preprint arXiv:2506.11928*, 2025.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36:55006–55021, 2023.

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

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, et al. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29168–29176, 2026b.

## A Self-Contained CUDA Program

```cuda
#include <cuda_runtime.h>

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

int main() {
    int n = 1024; // define number of elements
    ... // prepare dummy inputs or load test inputs
    ... // allocate device memory and copy inputs to device

    // prepare launch config and launch the kernel
    int blocksPerGrid = (n + 255) / 256; // launch config
    vecAddInplace<<<blocksPerGrid, 256>>>(a, b, n);

    ... // copy result to host and post-process if needed
    ... // verify results if needed and free memory
    return 0;
}
```

Figure 3: A simplified self-contained CUDA program.

## B Self-Contained CUDA Scaffold

```
[headers]
[kernel and launcher declaration]
[launcher]
[kernel]
T* read_binary_file(...) { ... }
T* write_binary_file(...) { ... }
int main(int argc, char **argv) {
    #pragma message("Calling convention: ...")
    [check arguments and read test inputs]
    [invoke the launcher function]
    [write kernel outputs]
}
```

Figure 4: A simplified self-contained CUDA scaffold.