CUDABench: Benchmarking LLMs for Text-to-CUDA Generation

Jiace Zhu **Wentao Chen** **Qi Fan** **Zhixing Ren** **Junying Wu** **Xing Zhe Chai**
Chotiwit Rungrueangwutthinon **Yehan Ma** **An Zou***
Shanghai Jiao Tong University
an.zou@sjtu.edu.cn

Abstract

Recent studies have demonstrated the potential of Large Language Models (LLMs) in generating GPU kernels. Current benchmarks focus on the translation of high-level languages into CUDA, overlooking the more general and challenging task of text-to-CUDA generation. Furthermore, given the hardware-specific and performance-critical features of GPU programming, accurately assessing the performance of LLM-generated GPU programs is nontrivial. In this work, we introduce **CUDABench**, a comprehensive benchmark designed to evaluate the text-to-CUDA capabilities of LLMs. First, we construct **CUDABench-Set**, which covers Breadth-Depth-Difficulty evaluation space in diverse application domains, including artificial intelligence, scientific computing, and data analytics, etc. Furthermore, we propose **CUDABench-Score** and **Generative Verification Pipeline** that assess (1) compilation correctness, (2) functional consistency through execution-based verification, and (3) a novel roofline-based metric, Performance-Score. Benchmarking state-of-the-art LLMs reveals insightful findings and challenges of text-to-CUDA, such as a notable mismatch between high compilation success rates and low functional correctness, domain-dependent challenges in algorithm retrieval and synthesis, and limited roofline-normalized performance. Our benchmark is available at `https://github.com/CUDA-Bench/CUDABench`.

1 Introduction

The rapid advancement of Large Language Models (LLMs) has gained increasing interest in their application to code generation [3, 12], including GPU programming. Recent studies demonstrate that LLMs are capable of generating correct and executable GPU programs in CUDA [26, 4], thereby lowering the barrier for developers to leverage the computational power of modern GPUs. Despite these promising early results, a comprehensive benchmarking of LLMs for GPU program generation remains limited. Existing benchmarks and evaluations focus primarily on the translation of high-level programming languages, such as PyTorch, into CUDA [26, 15, 31]. This setting evaluates LLMs in a code-to-code translation regime, where the program structure and computational intent are explicitly specified in the input code. As a result, such benchmarks fail to capture the more general and challenging scenario of text-to-CUDA generation, which requires covering the breadth of diverse domains beyond native PyTorch Machine Learning (ML) workloads (e.g., Scientific Simulation) at production-level input scales, while forcing LLMs to infer algorithmic intent and implementation details directly from natural language.

Moreover, CUDA kernel is inherently hardware-specific and performance-critical [13, 29]. Writing a high-performance CUDA kernel requires careful consideration of architectural details, specifically balancing computational throughput and memory bandwidth to improve performance facing both

*Corresponding Author.

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

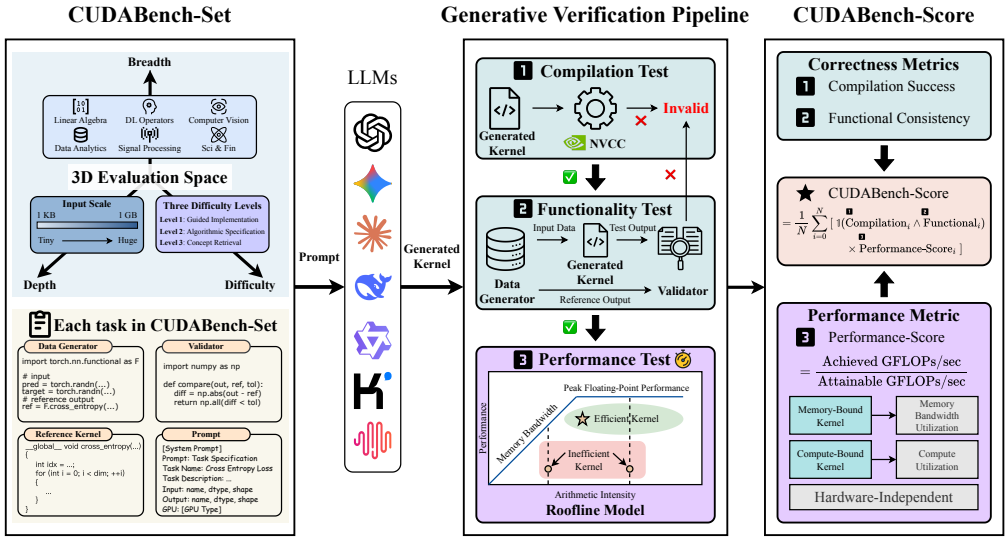


Figure 1: Overview of CUDABench.

Table 1: Difference between CUDABench and existing benchmarks.

Kernel Generation Benchmark	Paradigm	Application Domain Coverage		Evaluation Infrastructure		Evaluation Metrics		
		Machine Learning Workloads	Holistic Domain Coverage	Reference Implementations	Realistic Input Sizes	Correctness	Execution Time	Memory Bandwidth & Compute Utilization
KernelBench [26]	PyTorch-to-CUDA	✓	✗	✗	✗	✓	✓	✗
TritonBench [15]	PyTorch-to-Triton	✓	✗	✗	✗	✓	✓	✗
TritonGym [10]	PyTorch-to-Triton	✓	✗	✗	✗	✓	✓	✗
BabelTower [31]	C-to-CUDA	✓	✗	✗	✗	✓	✓	✗
ComputeEval [19]	Text-to-CUDA	✗	✓	✗	✗	✓	✗	✗
CUDABench (*Ours*)	Text-to-CUDA	✓	✓	✓	✓	✓	✓	✓

compute and memory bounds [11]. Consequently, evaluating LLM-generated kernels solely based on functional correctness is insufficient. A kernel that produces correct output may still suffer from severe performance degradation compared to optimized implementations [13, 29]. Therefore, accurately assessing kernel performance and understanding how close it comes to expert-level performance are nontrivial.

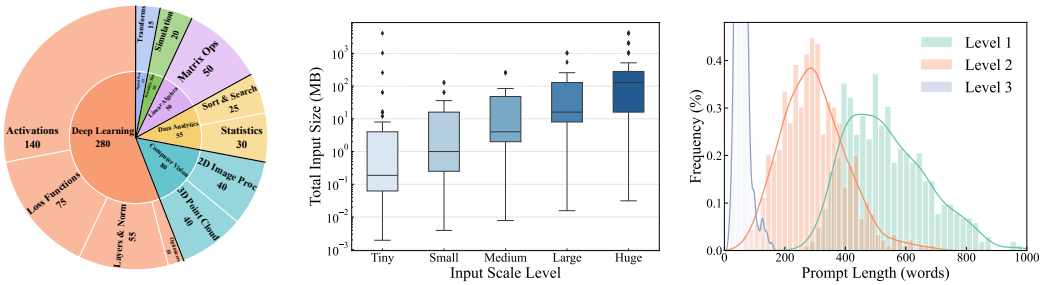
These challenges highlight the necessity for benchmarks and evaluation methodologies that (1) target text-to-CUDA generation directly, (2) provide coverage across multiple application domains, and (3) incorporate performance metrics that go beyond raw execution time and better reflect the practical efficiency of GPU programming.

Therefore, in this work, we present **CUDABench** (as shown in Fig. 1 and Tab. 1), a comprehensive benchmark to systematically evaluate the text-to-CUDA program generation capabilities of LLMs across diverse domains, including but not limited to PyTorch ML workloads kernels. The main contributions are as follows.

- We construct **CUDABench-Set**, a text-to-CUDA dataset that covers a Breadth-Depth-Difficulty three-dimensional evaluation space, encompassing multiple application domains, different input scales, and varying difficulty levels of prompt details.

- We propose **Generative Verification Pipeline**, which evaluates compilation and functional correctness through strict execution-based verification, and **CUDABench-Score**, a novel hardware-independent roofline-based metric that measures generated kernels by their utilization of the target GPU's attainable performance rather than raw execution time alone.

- We conduct extensive evaluations of state-of-the-art LLMs using CUDABench, revealing their strengths, limitations, and several insightful findings when generating CUDA kernels.



(a) Breadth: Tasks distribution (b) Depth: Input size distribution (c) Difficulty: Task prompt length

Figure 2: Statistical distribution of CUDABench-Set dimensions. (a) **Breadth**: Distribution of tasks across domains. (b) **Depth**: Input size distribution across scale levels. (c) **Difficulty**: Task prompt length distribution across difficulty levels.

2 Related Works

Developing high-performance CUDA kernels is a time-consuming process that requires substantial expertise in parallel programming, memory hierarchies, and GPU architectures [11, 5]. This challenge has motivated recent efforts to leverage LLMs to automate CUDA kernel generation. To evaluate the ability of LLMs to produce efficient and correct CUDA kernels, KernelBench [26] introduces a benchmarking framework together with the fast$_p$@k metric, which jointly measures functional correctness and performance. As one of the most widely adopted benchmarks in this area, KernelBench has inspired a growing body of work on CUDA kernel generation and optimization [2, 16, 8, 14]. However, KernelBench is limited to PyTorch-to-CUDA code translation and is largely restricted to PyTorch ML workloads. Moreover, it does not provide reference CUDA implementations.

ComputeEval [19] introduces a natural-language-based dataset that broadens the scope of CUDA kernel generation tasks. Nevertheless, as shown in Tab. 1, compute-eval lacks reference implementations and relies on relatively small-scale test cases. Crucially, it lacks performance evaluation, limiting its ability to assess practical performance and real-world engineering relevance.

For performance evaluation, most existing benchmarks and methods rely on execution time as the main metric [26, 31]. However, runtime is insufficient as the primary metric for from-scratch CUDA generation: generated kernels may differ in algorithmic structure, parallelization strategy, memory access pattern, and synchronization behavior, while raw runtime also entangles hardware capability, input scale, and system noise. SOL-ExecBench [17] similarly adopts a roofline-based approach to performance evaluation, but focuses on optimization of production deep-learning kernels and subgraphs. In contrast, CUDABench evaluates from-scratch text-to-CUDA generation across diverse GPU-computing domains.

To address these limitations, CUDABench evaluates compilation correctness, functional correctness, and performance using the roofline-based CUDABench-Score. This score further supports hardware-independent evaluation by applying the same target-GPU-normalized scoring formulation across CUDA-capable GPUs. Furthermore, we present a comprehensive evaluation of state-of-the-art LLMs using CUDABench.

3 Text-to-CUDA Dataset: CUDABench-Set

We construct **CUDABench-Set**, a text-to-CUDA dataset designed to assess LLM capabilities in CUDA kernel generation. Each evaluation instance consists of a fixed system prompt and a task prompt. The system prompt is shared across all tasks and levels, while the task prompt contains the task name, task description, I/O specifications, data types, and target GPU. Each task is further equipped with a generative verification pipeline, comprising a data generator and a validator. Moreover, we provide reference CUDA kernel codes for all tasks. On top of these components, CUDABench-Set constructs a **Breadth-Depth-Difficulty** evaluation space.

Breadth. CUDABench-Set provides comprehensive coverage of six representative GPU computing domains and a broad range of real-world workloads, including: (1) **Fundamental Linear Algebra**, including GEMM, transposition, element-wise operations, and least-squares solvers; (2) **Deep Learning Operators**, including activations, losses, normalization layers, and optimizers; (3) **Computer Vision and Image Processing**, including 2D image operators and 3D point-cloud kernels; (4) **Data Analytics**, including sorting, Top-K, histograms, scan, and reductions; (5) **Signal Processing**, including FIR filtering and wavelet transforms; and (6) **Scientific Simulation and Finance**, including Monte Carlo methods, PDE solvers, and Black-Scholes.

All the tasks are constructed from widely used, open-source CUDA codebases, including deep learning operator libraries [27, 28], NVIDIA official CUDA Samples [22], and other production-level GPU repositories. The number and distribution of the tasks is shown in Fig. 2a.

Depth. CUDABench-Set ensures comprehensive performance evaluation through hierarchical test input data design. We define five progressive input scale levels (Tiny, Small, Medium, Large, Huge) for each kernel to mimic a wide spectrum of production-level environments. For instance, in the Haar Wavelet Transform task, the input size scales exponentially across the five levels, with 1D signal lengths ranging from 512 to 8,192. As depicted in the box plot (Fig. 2b), the dataset covers a vast range of data sizes, scaling from a few kilobytes in the "Tiny" category to over 1 GB in the "Huge" category. By systematically saturating the scale hierarchy and GPU compute units, this design ensures sufficient verification depth, enabling the evaluation of kernel performance under hardware stress.

Note that every input size for each task is paired with a dedicated reference kernel, data generator, and validator, since both algorithmic implementation and validation criteria vary significantly across different scales of input. This granular alignment ensures that the reference output and validation logic are strictly adapted to the computational characteristics of each specific input data size.

Difficulty. To explore the LLM capacities from CUDA syntax generation to domain-knowledge retrieval, we construct each benchmark task with prompts of three different difficulty levels. These levels share the same system prompt and structured task prompt, including I/O specifications and target GPU, but progressively remove contextual and implementation details in the task description:

- **Level 1 (Guided Implementation)**, providing the task name, a detailed algorithm description, and explicit CUDA implementation guidelines (e.g., memory hierarchy usage and thread-mapping strategies). This level evaluates LLM capacity to generate architecturally and syntactically correct CUDA code.
- **Level 2 (Algorithmic Specification)**, including the task name and algorithm description, but omits all hardware-specific guidance. At this level, the model must autonomously map scalar algorithmic logic onto a parallel GPU execution model.
- **Level 3 (Concept Retrieval)**, supplying only the task name. This extreme setting assesses the ability of LLM to internally retrieve the required domain knowledge (e.g., inferring the mathematical formulation of Black-Scholes) and synthesize a correct solution without algorithmic explanations or CUDA implementation hints.

As shown in Fig. 2c, prompts at different levels exhibit distinct length distributions, reflecting the progressive reduction in information density. This hierarchical design allows for a granular evaluation of specific LLM capabilities, ranging from syntax translation to autonomous domain knowledge retrieval.

In summary, CUDABench-Set comprises a total of 1,500 evaluation instances, derived from 500 tasks paired with three difficulty levels. Each task is equipped with a verified reference CUDA kernel, which serves as the ground truth for the accompanying data generator and validator, supporting the subsequent evaluation pipeline.

4 Text-to-CUDA Metric: CUDABench-Score

We first define correctness and performance metrics. Then, we unify these metrics into the **CUDABench-Score** and detail the **Generative Verification Pipeline** to automate this end-to-end evaluation framework.

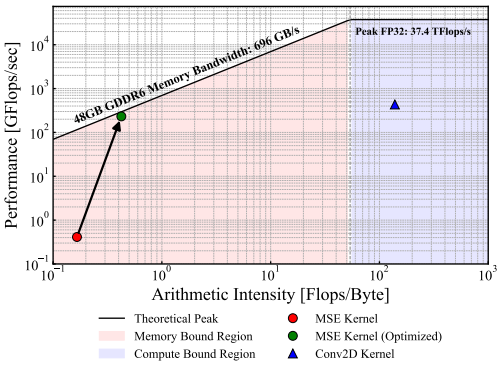
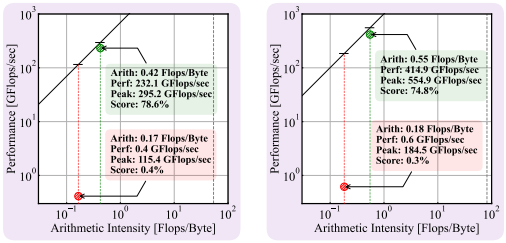


Figure 3: The roofline model for an NVIDIA A40 GPU.



(a) NVIDIA A40 (b) NVIDIA RTX 4090

Figure 4: Comparison of roofline models for the MSE kernel executed on different GPUs. **Arith** denotes Arithmetic Intensity, **Perf** refers to the Achieved GFLOPs/sec, **Peak** represents the Attainable GFLOPs/sec, and **Score** corresponds to the calculated Performance-Score.

4.1 Correctness Metrics

We enforce a strict two-stage filter that a generated kernel is considered valid only if it satisfies: (1) **Compilation Success:** The kernel compiles without error using the standard NVIDIA CUDA Compiler Driver (NVCC) [24], (2) **Functional Consistency:** The kernel executes successfully across generated test input data, producing results that match the reference output. Only **valid** kernels are eligible for the subsequent performance benchmarking in Sec. 4.2. The complete verification protocol is detailed in Appendix C.1.

4.2 Performance Metrics

Most existing benchmarks [26, 15, 31] and optimization [4] approaches primarily evaluate CUDA kernels using execution time as the performance indicator [26, 15]. However, execution time is sensitive to hardware configurations, memory bandwidth, system load, and other factors. Therefore, we introduce a roofline-based performance metric that evaluates the performance of LLM-generated kernel performance relative to its theoretically attainable performance. Meanwhile, we use hardware-independent evaluation to mean that the same roofline-based scoring formulation can be instantiated on any CUDA-capable GPU using its own peak compute throughput, peak memory bandwidth, and profiling measurements.

4.2.1 Performance-Score

The roofline model is a visual performance model used to provide performance evaluation of a given compute kernel on multi-core, many-core, or accelerator processor architectures [32]. To characterize a kernel on a roofline model, three key quantities must be collected: the execution time, the total number of floating-point operations (FLOPs) performed, and the total volume of data movement. These quantities jointly determine the arithmetic intensity (Arith), which is defined as the ratio of the total number of FLOPs executed by the application to the total number of bytes transferred to support those operations.

The roofline model defines attainable performance as the theoretical upper bound of performance that a specific kernel can achieve on a given architecture. Formally, this limit is determined by the minimum of the hardware's peak computational throughput and its memory bandwidth capacity, as expressed in Eq. (1) [32],

$$\begin{aligned}\text{Attainable GFLOPs/sec} = \min\{&\text{Peak Floating-Point Performance},\\ &\text{Peak Memory Bandwidth} \times \text{Arithmetic Intensity}\}.\end{aligned} \tag{1}$$

For instance, we present the roofline model for an NVIDIA A40 GPU in Fig. 3. The horizontal line shows the peak floating-point performance of the GPU. The actual floating-point performance of a floating-point kernel cannot be higher than this line. The slanted line bounds the maximum floating-point performance that the memory bandwidth of the GPU can support for a given arithmetic

intensity. We evaluate three cases: a Mean Squared Error (MSE) kernel (in the Breadth dimension of the Deep Learning Operators domain, and the Depth dimension with an input size of 2^{10}), a Conv2D kernel (in the Breadth dimension of the Deep Learning Operators domain, and the Depth dimension with inputs of a $(128, 128)$ matrix and $(24, 24)$ kernel), and an optimized MSE kernel refined by LLMs. As observed in Fig. 3, these kernels have different arithmetic intensities and different achieved performance.

By mapping the specific arithmetic intensity of a kernel to Eq. (1), we establish its attainable performance as the theoretical hardware limit. Based on this, we introduce the **Performance-Score**, measuring how closely its achieved performance approaches the theoretical peak performance. Performance-Score is given by:

$$\text{Performance-Score} = \frac{\text{Achieved GFLOPs/sec}}{\text{Attainable GFLOPs/sec}}. \tag{2}$$

This formulation provides a normalized evaluation of kernel performance, enabling fair comparison across kernels with fundamentally different performance bottlenecks.

Fig. 4 illustrates how the same hardware-independent scoring procedure is instantiated on two GPUs with different compute throughput and memory bandwidth. Although the raw achieved GFLOPs/sec differs substantially between A40 and RTX 4090, Performance-Score evaluates each kernel relative to the attainable roofline of the corresponding target GPU. The similar scores observed in this example suggest that roofline normalization reduces the dominance of raw hardware scale in performance reporting.

4.2.2 Memory-Bound & Compute-Bound

The roofline model maps the arithmetic intensity of each kernel to its attainable performance, distinguishing whether a workload is limited by memory bandwidth or computational throughput. For memory-bound kernels, substituting the corresponding term from Eq. (1) into Eq. (2) yields:

$$\begin{aligned} \text{Performance-Score} &= \frac{\text{Achieved GFLOPs/sec}}{\text{Attainable GFLOPs/sec}} = \frac{\text{Arith} \times \text{Achieved Memory Bandwidth}}{\text{Arith} \times \text{Peak Memory Bandwidth}} \\ &= \text{Memory Bandwidth Utilization}. \end{aligned} \tag{3}$$

which means the Performance-Score effectively reduces to an evaluation of memory bandwidth utilization. Therefore, the large score improvement indicates that the kernel substantially increases effective memory bandwidth usage, moving much closer to the memory roof. This behavior is expected and desirable, as memory bandwidth is the dominant performance factor for low arithmetic intensity kernels.

Symmetrically, for compute-bound kernels operating to the right of the ridge point, the same metric naturally transitions to measuring compute utilization,

$$\begin{aligned} \text{Performance-Score} &= \frac{\text{Achieved GFLOPs/sec}}{\text{Attainable GFLOPs/sec}} = \frac{\text{Achieved GFLOPs/sec}}{\text{Peak GFLOPs/sec}} \\ &= \text{Compute Utilization}. \end{aligned} \tag{4}$$

In this sense, Performance-Score provides a unified performance metric that adapts to the kernel's bottleneck regime, quantifying bandwidth utilization for memory-bound workloads and compute efficiency for compute-bound workloads.

4.3 CUDABench-Score

We define **CUDABench-Score**, a unified metric that integrates compilation correctness, functional consistency, and Performance-Score into a scalar value, which is given by:

$$\text{CUDABench-Score} = \frac{1}{N} \sum_{i=1}^{N} \left[\mathbb{1}(\text{Compilation}_i \wedge \text{Functional}_i) \times \text{Performance-Score}_i \right]. \tag{5}$$

where N represents the total number of tasks in the CUDABench-Set, and i denotes the index of the i-th task. Consequently, CUDABench-Score reflects the capability of LLMs to generate reliable, high-performance CUDA kernels within CUDABench-Set.

4.4 Generative Verification Pipeline

To derive CUDABench-Score for CUDABench-Set, we build a generative verification pipeline as part of CUDABench. The pipeline is designed as a fully automated, end-to-end framework that validates and evaluates LLM-generated kernels. The framework integrates **Data Generators** and **Validators** to execute a sequential evaluation pipeline comprising four key stages: (1) test data preparation; (2) kernel compilation; (3) execution-based functional verification; and (4) performance evaluation.

For each task, the data generator produces random input data and corresponding reference outputs. To guarantee robust verification, we ensure the input values are large enough to avoid underflow, making it easier to verify the correctness within the error tolerance. Functional validation follows a predefined numerical tolerance policy: discrete outputs require exact agreement, while floating-point tolerances range from 10^{-5} to 10^{-1} according to arithmetic depth and expected accumulation error. Quantized image outputs use separate per-channel tolerances. The complete tolerance policy is provided in Appendix C.2.

Then the validator compiles and executes the generated code, checking its functional consistency against reference outputs. Kernels that pass correctness verification are subsequently profiled using NVIDIA Nsight Compute [23] to measure the execution time, the number of FLOPs, and the volume of data movement, and calculate the CUDABench-Score. Utilizing the real-time generation strategy, the pipeline synthesizes and consumes data without requiring persistent storage. This approach eliminates the heavy disk space occupancy typically associated with large-scale static benchmarks.

5 Experiment

5.1 Experimental Setup

Model. We evaluate CUDABench using the latest LLMs, including several models with specialized reasoning modes. Specifically, we include: *GPT-5.2 (High)* [25], *Claude 4.5 Sonnet* [1], *Gemini 3 Flash* [9, 6], *DeepSeek-V3.2* [7], *Kimi K2 Thinking* [30], *Qwen3-Max* [33], and *MiniMax-M2.1* [18]. All models are accessed via their official APIs to guarantee that the evaluation reflects their most current capabilities.

Hardware. We primarily conduct experiments on the NVIDIA A40 GPU, based on the Ampere architecture [21]. The A40 GPU is equipped with 48 GB of GDDR6 memory providing 696 GB/s of bandwidth and consists of 84 Streaming Multiprocessors (SMs) delivering a peak FP32 throughput of 37.4 TFLOPS. To illustrate the cross-GPU applicability of our hardware-independent evaluation framework, we provide supplementary results using the NVIDIA GeForce RTX 4090 GPU (Ada Lovelace architecture [20]). While the RTX 4090 GPU has a smaller memory capacity (24 GB GDDR6X), it offers significantly higher computational density with 128 SMs, achieving a peak FP32 performance of 82.6 TFLOPS and a memory bandwidth of 1,008 GB/s.

Metrics. We evaluate the generated kernels using **Compilation Success**, **Functional Correctness**, and **CUDABench-Score**, with **Pass@k** ($k = \{1, 3\}$) [3], that estimates the probability that at least one valid or high-performance CUDA kernel is generated within k sampling attempts.

5.2 Evaluation Results

Tab. 2 presents the comprehensive evaluation results of different LLMs on **CUDABench**. We report results across three difficulty levels. In terms of overall performance, Claude 4.5 Sonnet demonstrates superior capability in correctness, consistently achieving the highest metrics in both compilation and functional consistency. Specifically, under the Level 1 (Pass@1) setting, it attains a near-perfect compilation rate of 99.8% and a leading functional accuracy of 85.8%. Conversely, regarding kernel performance, GPT-5.2 (High) has a slight advantage, securing the top CUDABench-Score of 40.9% at Level 1 and maintaining this lead across higher difficulty levels. While other model also delivers competitive results, a deeper examination of these metrics reveals critical limitations shared across all models. Analyzing the results across different models and levels, we derive the following findings:

Finding 1: CUDABench poses significant challenges for LLMs. The tested LLMs exhibit distinguishable performance on CUDABench. Taking GPT-5.2 (High) as an example, under the

Table 2: Results from the latest LLMs on CUDABench. Performance comparison of state-of-the-art LLMs across three difficulty levels. We report Pass@1 and Pass@3 rates for compilation correctness, functional consistency, and performance score. Best results are **bolded**, and second-best are underlined. Specific model versions evaluated include: `gpt-5.2-2025-12-11`, `gemini-3-flash-preview` (2025-12-17), `claude-sonnet-4-5-20250929`, `deepseek-reasoner` (2025-12-01), `qwen3-max-2025-09-23`, `kimi-k2-thinking` (2025-11-06), and `MiniMax-M2.1` (2025-12-23).

Dataset	Model	Pass@1			Pass@3		
		Compilation	Function	Score	Compilation	Function	Score
Level 1	GPT-5.2 (High) [25]	93.4%	79.8%	**40.9%**	99.6%	88.6%	**45.4%**
	Gemini 3 Flash [6]	97.6%	83.0%	40.1%	**100.0%**	88.6%	44.2%
	Claude 4.5 Sonnet [1]	**99.8%**	**85.8%**	40.2%	**100.0%**	**90.0%**	42.9%
	DeepSeek-V3.2 [7]	96.0%	65.2%	31.6%	99.6%	77.6%	38.9%
	Qwen3 Max [33]	93.8%	76.8%	32.1%	98.6%	85.2%	36.5%
	Kimi K2 Thinking [30]	82.6%	69.8%	35.1%	95.6%	85.6%	41.3%
	MiniMax-M2.1 [18]	87.0%	55.2%	26.7%	98.8%	77.0%	36.9%
Level 2	GPT-5.2 (High) [25]	93.4%	75.2%	37.9%	99.0%	85.0%	**44.5%**
	Gemini 3 Flash [6]	98.6%	77.6%	37.6%	**100.0%**	83.2%	41.7%
	Claude 4.5 Sonnet [1]	**99.2%**	**82.8%**	**39.4%**	**100.0%**	**86.6%**	42.6%
	DeepSeek-V3.2 [7]	94.2%	63.4%	31.4%	99.6%	77.6%	37.9%
	Qwen3 Max [33]	92.6%	73.8%	30.2%	98.4%	80.4%	35.2%
	Kimi K2 Thinking [30]	64.8%	53.8%	29.1%	84.4%	73.0%	37.2%
	MiniMax-M2.1 [18]	89.6%	57.6%	26.6%	99.8%	76.0%	38.5%
Level 3	GPT-5.2 (High) [25]	92.4%	60.8%	**29.8%**	99.4%	**70.6%**	**35.5%**
	Gemini 3 Flash [6]	96.8%	59.4%	28.4%	**100.0%**	68.6%	33.6%
	Claude 4.5 Sonnet [1]	**99.0%**	**63.2%**	28.4%	**100.0%**	69.8%	32.3%
	DeepSeek-V3.2 [7]	96.4%	49.8%	22.5%	**100.0%**	61.4%	28.7%
	Qwen3 Max [33]	92.0%	50.6%	22.5%	99.2%	61.0%	26.8%
	Kimi K2 Thinking [30]	76.4%	49.0%	23.5%	92.4%	63.2%	30.5%
	MiniMax-M2.1 [18]	86.6%	42.6%	20.1%	99.6%	59.8%	28.8%

Pass@1 setting, its functional accuracy is only 79.8% at Level 1 and drops significantly to 60.8% at Level 3, indicating increasing difficulty as kernel complexity grows.

This performance stands in contrast to the strong results previously reported on general-purpose code generation benchmarks such as HumanEval [3], where even earlier models (e.g., GPT-5) achieved accuracies above 90%. This discrepancy suggests that strong general code generation capabilities do not directly generalize to the domain of CUDA kernel generation, which requires specialized knowledge of parallel programming, memory hierarchies, and performance constraints. Also, we observe a notable improvement in Pass@3 compared to Pass@1. This indicates that LLMs are capable of generating correct CUDA kernels given multiple attempts, suggesting substantial headroom for future improvements through better prompting, sampling strategies, or domain-specific training.

Finding 2: Generated kernels achieve high compilation rates but low functional correctness. We observe a substantial divergence between syntactic validity and semantic accuracy. As illustrated in Fig. 5, functional errors are the primary source of failure, accounting for the majority of incorrect outputs and substantially exceeding compilation errors across nearly all evaluated models. While model like Claude 4.5 Sonnet achieve a perfect 100% compilation rate across all difficulty levels in Tab. 2 , their ability to generate functionally accurate code declines sharply. This trend is most pronounced in DeepSeek-V3.2 at Level 3, where functional errors account for approximately 47% of generations, whereas compilation faults remain 3.6%. This distribution indicates that modern LLMs have effectively mastered CUDA syntax and API conventions, yet they frequently generate superficially valid but logically flawed code, failing to handle complex semantics such as thread synchronization and memory boundary conditions.

Finding 3: LLMs lack domain-specific knowledge and CUDA implementation expertise. In the Level 3 evaluation, the task description contains only the task name, without algorithmic explanations or CUDA implementation hints. Consequently, we observe a sharp performance decline on Level 3 across all LLMs compared to Levels 1 and 2. This degradation highlights a critical retrieval generation

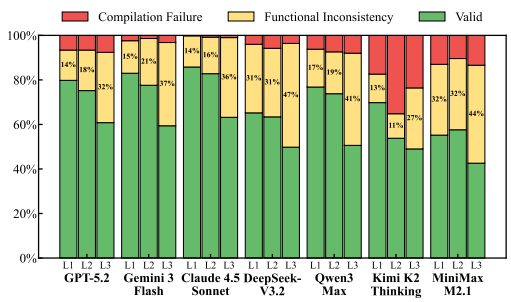


Figure 5: The proportion of Compilation Failure, Functional Inconsistency, and Valid kernel for each model with different levels (Pass@1).

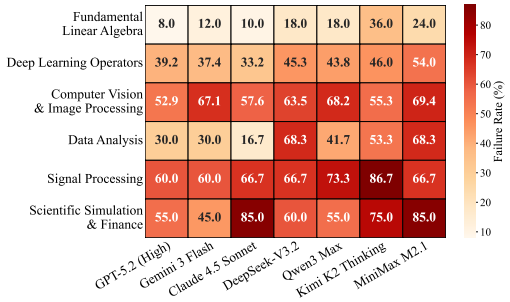


Figure 6: Analysis of failure rates on Level 3 tasks (Pass@1), accounting for kernels that either compilation failure or function inconsistency.

bottleneck: without explicit external guidance, models struggle to retrieve the correct algorithmic primitives and implementation patterns from their internal weights. This evidence strongly suggests that current LLMs suffer from a fundamental scarcity of domain-specific knowledge and lack the intrinsic expertise required for CUDA implementation.

As shown in Fig. 6, further analysis of failure rates on Level 3, across multiple domains, supports this finding through a more granular analysis. While models exhibit relative proficiency in foundational tasks like "Linear Algebra" (e.g., GPT-5.2 (High) shows a low failure rate of 8.0%), they struggle significantly in more comprehensive domains. Specifically, in the "Scientific Simulation & Finance" category, failure rates surge, with Claude 4.5 Sonnet reaching a staggering 85.0% failure rate. This domain-level disparity underscores that LLMs' internal knowledge is skewed towards general mathematical concepts, while remaining critically deficient in the complex, niche algorithms required for signal processing and scientific computing.

Finding 4: Generated kernels demonstrate limited roofline-normalized performance. To evaluate performance, we utilize the CUDABench-Score, a roofline-based metric that quantifies the proximity of a kernel's performance to the hardware's theoretical peak (Sec. 4.3). The results in Tab. 2 reveal a common inefficiency across all evaluated models. Even at Level 1, the performance of top-tier models is capped at a mediocre level. Notably, this performance bottleneck is systemic rather than model-specific, as evidenced by the negligible performance gap among leading models. For instance, GPT-5.2 (High), Claude 4.5 Sonnet, and Gemini 3 Flash achieve nearly identical performance scores of 40.9%, 40.2%, and 40.1%, respectively. This challenge suggests that while current LLMs are capable of generating syntactically correct code, they cannot universally implement critical hardware-aware optimizations, resulting in implementations that fail to fully leverage the massive capacity of modern GPUs.

To assess ranking robustness, we additionally evaluate model rankings using the reference-based $\text{fast}_1@k$ metric [26]. Despite their distinct evaluation perspectives, the two metrics yield broadly consistent model rankings, suggesting that the observed model-level trends are not specific to the proposed scoring formulation. Unlike reference-based speedup, CUDABench-Score measures roofline-normalized hardware utilization without relying on the performance of reference implementations. Detailed comparisons are provided in Appendix A.1.

5.3 Supplementary Results on RTX 4090 GPU

To illustrate the cross-GPU applicability of our hardware-independent evaluation framework, we instantiate the same evaluation pipeline on NVIDIA RTX 4090, which has substantially different peak compute throughput and memory bandwidth from NVIDIA A40.

As shown in Tab. 3, the results on NVIDIA RTX 4090 follow the same trends as the main results on NVIDIA A40 in Tab. 2. Across models, both functional correctness and CUDABench-Score generally decrease from Level 1 to Level 3. For example, Qwen3 Max drops from 76.8% functional correctness and 28.0% score at Level 1 to 54.8% and 19.7% at Level 3 under Pass@1, matching the degradation pattern observed on NVIDIA A40.

Table 3: Supplementary results on NVIDIA RTX 4090 GPU.

Dataset	Model	Pass@1		Pass@3	
		Function	Score	Function	Score
Level 1	DeepSeek-V3.2	64.0%	28.2%	78.9%	34.1%
	Qwen3 Max	76.8%	28.0%	85.9%	31.1%
	Kimi K2 Thinking	51.5%	25.5%	69.7%	34.4%
	MiniMax-M2.1	56.2%	24.0%	79.4%	33.9%
Level 2	DeepSeek-V3.2	62.8%	27.1%	75.9%	33.4%
	Qwen3 Max	75.6%	27.6%	82.6%	30.6%
	Kimi K2 Thinking	56.4%	26.9%	73.5%	34.6%
	MiniMax-M2.1	55.3%	22.9%	76.4%	32.8%
Level 3	DeepSeek-V3.2	51.1%	20.0%	63.0%	26.2%
	Qwen3 Max	54.8%	19.7%	61.0%	21.6%
	Kimi K2 Thinking	50.1%	23.7%	63.0%	28.9%
	MiniMax-M2.1	42.6%	18.0%	60.0%	26.1%

These results show that the main findings from A40 are preserved on RTX 4090. By normalizing performance relative to the target GPU's attainable roofline, CUDABench-Score provides a hardware-independent evaluation formulation beyond raw execution time.

6 Conclusion

We introduced CUDABench, a comprehensive benchmark with a hardware-independent evaluation framework that establishes a Breadth-Depth-Difficulty evaluation space and uses the roofline-based CUDABench-Score to systematically evaluate text-to-CUDA generation. Our extensive evaluation of state-of-the-art LLMs reveals that despite achieving high compilation success rates, current models struggle with a significant disparity in functional correctness, domain-dependent difficulties in algorithm retrieval and synthesis, and limited roofline-normalized performance. CUDABench provides a foundation for future research on text-to-CUDA generation.

Limitations. First, CUDABench currently targets the NVIDIA ecosystem, relying on NVCC and Nsight-based profiling, which limits its applicability to other hardware platforms. Second, the benchmark focuses on FP32 and single-kernel generation, without covering mixed-precision formats (e.g., FP16, MX-FP8, and NV-FP4), kernel fusion, or end-to-end multi-kernel workflows. Third, as with other code generation benchmarks, training-data contamination cannot be completely ruled out, although we mitigate this risk through task rewriting, new natural-language prompts, and manually implemented benchmark instances.

Acknowledgments and Disclosure of Funding

This work was supported in part by NSFC 62473254.

References

[1] Anthropic. Claude-Sonnet 4-5 System Card. `https://assets.anthropic.com/m/12f214efcc2f457a/original/Claude-Sonnet-4-5-System-Card.pdf`, 2025. Accessed: 2025-12-09.

[2] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL `https://arxiv.org/abs/2507.11948`.

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

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

[5] Wentao Chen, Jiace Zhu, Xing Zhe Chai, Zeng Qu, Qiaoling Xiao, Liucheng Duan, and An Zou. From tokens to regions: Cuda-sensitive instruction tuning for gpu kernel generation. *arXiv preprint arXiv:2606.16231*, 2026.

[6] DeepMind. Gemini 3 Flash. `https://deepmind.google/models/gemini/flash/`, 2025. Accessed: 2026-01-15.

[7] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

[8] 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.

[9] Google. Gemini 3. `https://blog.google/products/gemini/gemini-3/`, 2025. Accessed: 2025-12-09.

[10] Yue Guan, Yichen Lin, Xu Zhao, Jianzhu Yao, Xinwei Qiang, Zhongkai Yu, Pramod Viswanath, Yufei Ding, and Adnan Aziz. Tritongym: A benchmark for agentic llm workflows in triton gpu code generation. In *Forty-third International Conference on Machine Learning*, 2026.

[11] Design Guide. Cuda c++ programming guide. *NVIDIA, July*, 2020.

[12] Sathvik Joel, Jie Wu, and Fatemeh Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*, 2024.

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

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

[15] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066, 2025.

[16] 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.

[17] Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, et al. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026.

[18] MiniMax. MiniMax M2.1: Significantly Enhanced Multi-Language Programming, Built for Real-World Complex Tasks. `https://www.minimax.io/news/minimax-m21`, 2025. Accessed: 2026-01-15.

[19] NVIDIA. Computeeval: Evaluating large language models for cuda code generation. `https://github.com/NVIDIA/compute-eval`, 2025. GitHub repository.

[20] NVIDIA. NVIDIA ADA GPU ARCHITECTURE. `https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf`, 2026. Accessed: 2026-01-28.

[21] NVIDIA. NVIDIA AMPERE GA102 GPU ARCHITECTURE. `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`, 2026. Accessed: 2026-01-28.

[22] NVIDIA. CUDA Samples. `https://github.com/NVIDIA/cuda-samples`, 2026. Accessed: 2026-01-28.

[23] NVIDIA. NVIDIA Nsight Compute. `https://developer.nvidia.com/nsight-compute`, 2026. Accessed: 2026-01-15.

[24] NVIDIA. NVIDIA CUDA Compiler Drive. `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/`, 2026. Accessed: 2026-01-28.

[25] OpenAI. GPT-5 System Card. Technical report, OpenAI, August 2025. URL `https://cdn.openai.com/gpt-5-system-card.pdf`. Accessed: 2025-12-07.

[26] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[28] PyTorch. PyTorch Documentation Reference API: torch.nn.functional. `https://docs.pytorch.org/docs/stable/nn.functional.html`, 2026. Accessed: 2026-01-28.

[29] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[30] Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[31] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xin Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR, 2022.

[32] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[33] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[34] 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.

A Additional Experimental Analysis

A.1 Fixed-Reference Metric and Ranking Stability

The roofline-normalized score uses each candidate's measured arithmetic intensity, so redundant work or memory traffic can affect its bound. A complementary fixed-reference metric compares generated code with the same executable reference for each task. Let V_{ij} indicate that attempt j compiles, executes successfully, and passes functional validation. For reference time t_i^{ref} and generated time t_{ij}, define

$$s_i^{(k)} = \begin{cases} \dfrac{t_i^{\text{ref}}}{\min_{j \le k:\, V_{ij}=1} t_{ij}}, & \exists j \le k : V_{ij} = 1, \\ 0, & \text{otherwise}, \end{cases} \qquad \text{fast}_p@k = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}\left[s_i^{(k)} > p \right], \quad p > 0. \tag{6}$$

Thus $\text{fast}_1@k$ is the fraction of all tasks with at least one correct generation faster than the fixed reference. Timing sums Nsight Compute's `gpu__time_duration.sum` over kernel invocations, excluding host-side data generation and I/O. We retain the main-table denominator $N = 500$ for each level; The five tasks absent from the legacy model outputs contribute zero. Failed timing attempts are conservatively excluded from the set of usable candidates.

Table 4: Fixed-reference results on A40 GPU. Each cell reports $\text{fast}_1@k$ and the rank change from CUDABench-Score to the fixed-reference metric. Ties receive average ranks. P@k uses up to k independent generations.

Model	Level 1 P@1	Level 1 P@3	Level 2 P@1	Level 2 P@3	Level 3 P@1	Level 3 P@3
GPT-5.2 (High)	59.80% (1→1)	70.80% (1→1)	53.20% (2→2)	65.40% (1→2)	40.00% (1→1)	52.40% (1→1)
Gemini 3 Flash	50.40% (3→3)	67.60% (2→2)	47.00% (3→3)	62.60% (3→3)	38.20% (2.5→2)	50.00% (2→2)
Claude 4.5 Sonnet	52.80% (2→2)	66.00% (3→3)	54.40% (1→1)	67.40% (2→1)	35.80% (2.5→3)	49.60% (3→3)
DeepSeek-V3.2	37.20% (6→6)	58.60% (5→5)	40.20% (4→5)	57.60% (5→4)	26.00% (5.5→5.5)	42.20% (6→6)
Qwen3 Max	39.40% (5→5)	51.80% (7→7)	41.80% (5→4)	52.40% (7→7)	26.00% (5.5→5.5)	40.80% (7→7)
Kimi K2 Thinking	48.00% (4→4)	65.60% (4→4)	34.20% (6→6)	54.40% (6→5)	31.00% (4→4)	48.40% (4→4)
MiniMax-M2.1	30.40% (7→7)	52.40% (6→6)	31.40% (7→7)	54.20% (4→6)	24.80% (7→7)	43.80% (5→5)
Spearman ρ	1.000	1.000	0.964	0.857	0.991	1.000

Tab. 4 gives Spearman correlations of 0.857–1.000. Three settings preserve the exact ranking, while the remaining differences are mainly local rank swaps with relatively small performance differences. Even in the lowest-correlation setting, Level 2 Pass@3, the leading models remain largely unchanged. This suggests that our model-level comparisons are robust to the fixed-reference metric.

A.2 Reference-Kernel Anchors and Iterative Refinement

The supplied reference kernels provide executable ground truth for task construction and functional validation, rather than optimized performance ceilings. Evaluated using the same scoring pipeline, they achieve a mean CUDABench-Score of 33.53% across all tasks. However, direct comparison with model scores conflates functional correctness and kernel performance, as models solve different subsets of tasks.

To control for task coverage, we construct *correctness-matched reference anchors*. Let R_i denote the reference score for task i, and $M_i^{(m,l,k)}$ indicate whether model m solves task i at difficulty level l within k attempts. We define

$$\text{MatchedRef}_{m,l,k} = \frac{1}{500} \sum_{i=1}^{500} M_i^{(m,l,k)} R_i. \tag{7}$$

This applies each model's correctness mask to the reference scores while retaining normalization over all tasks, unlike the conditional score in Appendix A.3. Task-level masking also accounts for differences in solved-task subsets that aggregate accuracy cannot capture.

As shown in Tab. 5, the 42 matched anchors range from 16.99% to 32.12%. Model-generated kernels consistently outperform their matched references by 2.08–14.00 percentage points, indicating higher aggregate performance under identical correctness coverage.

Table 5: Correctness-matched reference CUDABench-Scores on A40 GPU. Each cell reports scores under the corresponding model's Pass@1/Pass@3 correctness masks.

Model	Level 1	Level 2	Level 3
GPT-5.2 (High)	29.53% / 31.54%	27.87% / 30.50%	22.03% / 24.81%
Gemini 3 Flash	30.94% / 32.03%	29.15% / 30.75%	21.81% / 24.84%
Claude 4.5 Sonnet	31.79% / 32.12%	30.77% / 31.23%	22.53% / 23.79%
DeepSeek-V3.2	27.68% / 30.22%	26.03% / 29.15%	20.08% / 23.53%
Qwen3 Max	29.09% / 31.38%	28.12% / 29.59%	19.00% / 22.70%
Kimi K2 Thinking	28.44% / 30.96%	24.16% / 29.14%	19.37% / 22.86%
MiniMax-M2.1	23.29% / 29.26%	23.64% / 29.74%	16.99% / 22.03%

Supplementary iterative-refinement baseline. While independent generation is the primary evaluation protocol, we additionally evaluate a basic iterative-refinement baseline [34] using DeepSeek-V4, achieving a CUDABench-Score of 55.04%.

Table 6: Summary of reference, independent-generation, and iterative-refinement scores. The median and best independent-generation results summarize 42 model/level/Pass@k configurations.

Evaluation	CUDABench-Score
Reference kernels	33.53%
Correctness-matched reference anchors	16.99%–32.12%
Median independent-generation result	35.15%
Best independent-generation result	45.40%
Supplementary iterative refinement (DeepSeek-V4)	55.04%

Tab. 6 summarizes the evaluation results. Independent-generation scores range from 20.10% to 45.40%, with a median of 35.15%, while iterative refinement reaches 55.04%. These results highlight CUDABench's ability to distinguish model generation performance and evaluate further optimization through refinement.

A.3 Conditional Performance Score

CUDABench-Score is designed to jointly reflect both correctness and performance. Averaging over all tasks is a deliberate design choice: kernels that fail to compile or produce incorrect outputs receive zero contribution to the final score, which is important in an end-to-end generation setting. Nevertheless, it is also useful to separate the performance of valid kernels from the overall success rate of a model. To this end, we report an additional conditional score that measures the average Performance-Score among kernels that pass both compilation and functional verification.

From Eq. (5), the conditional score can be derived as:

$$\begin{aligned}\text{Conditional Score} &= \frac{\sum_{i=0}^{N}\left[\mathbb{1}(\text{Compilation}_i \wedge \text{Functional}_i) \times \text{Performance-Score}_i\right]}{\sum_{i=0}^{N}\mathbb{1}(\text{Compilation}_i \wedge \text{Functional}_i)} \\ &= \frac{\text{CUDABench-Score}}{\frac{1}{N}\sum_{i=0}^{N}\mathbb{1}(\text{Compilation}_i \wedge \text{Functional}_i)} \\ &= \frac{\text{CUDABench-Score}}{\text{Functional Correctness}}.\end{aligned} \tag{8}$$

Tab. 7 provides examples on Level 1 and Level 2 under the Pass@1 setting. The decomposition reveals different model behaviors behind similar overall scores. Qwen3 Max achieves higher functional correctness, indicating stronger reliable generation, while Kimi K2 Thinking obtains a higher conditional score once a valid kernel is generated. This suggests that Kimi K2 Thinking tends to produce more performance-oriented kernels among its correct generations, whereas Qwen3 Max favors more reliable generation. Therefore, similar CUDABench-Scores may arise from different trade-offs between correctness and performance.

Table 7: Conditional Score analysis on Level 1 and Level 2 under Pass@1. Functional denotes functional correctness, Score denotes CUDABench-Score, and Conditional Score is computed as Score divided by functional correctness.

Level	Model	Functional	Score	Conditional Score
1	Qwen3 Max	76.8%	32.1%	41.8%
1	Kimi K2 Thinking	69.8%	35.1%	50.3%
2	Qwen3 Max	73.8%	30.2%	40.9%
2	Kimi K2 Thinking	53.8%	29.1%	54.1%

A.4 Error Analysis

Table 8: Fine-grained error analysis of invalid LLM-generated kernels. The four categories are: (1) Unsupported API/Primitive Usage, (2) Execution Scope Error, (3) Resource Violation, and (4) Host-Device Semantic Mismatch.

Model	Level	1	2	3	4
DeepSeek-V3.2	Level 1	10.39%	61.69%	20.13%	7.79%
	Level 2	14.70%	51.18%	26.47%	7.65%
	Level 3	7.51%	56.92%	23.71%	11.86%
GPT-5.2 (High)	Level 1	28.13%	43.75%	14.58%	13.54%
	Level 2	26.05%	42.86%	21.85%	9.24%
	Level 3	17.28%	59.69%	13.09%	9.95%

To better understand why generated kernels fail, we further categorize invalid LLM-generated kernels into four types: (1) Unsupported API/Primitive Usage, (2) Execution Scope Error, (3) Resource Violation, and (4) Host-Device Semantic Mismatch. Tab. 8 reports the breakdown for DeepSeek-V3.2 and GPT-5.2 (High) across three difficulty levels.

Across models and prompt levels, three patterns emerge. First, Execution Scope Error is the dominant failure mode in nearly all settings, indicating that current models struggle most with CUDA's hierarchical execution semantics, including indexing, grid-stride loops, and warp/shared-memory reasoning. Second, Unsupported API/Primitive Usage is another major source of failure, especially for GPT-5.2 (High) under weaker prompt settings, suggesting limited API grounding. Third, Resource Violations remain nontrivial, while Host-Device Semantic Mismatch is less frequent but still recurrent.

These fine-grained results complement Fig. 6 by showing that current models fail in systematic ways, especially in execution-scope reasoning, API grounding, and hardware-resource reasoning.

A.5 Additional Cross-GPU Case Study

In addition to the A40 GPU and RTX 4090 GPU results discussed in Sec. 4, we further evaluate the MSE kernel and its optimized version on RTX 3090 GPU and RTX 5080 GPU. These GPUs cover different architectures and hardware scales, providing an additional case study for the cross-GPU applicability of our hardware-independent scoring formulation.

As shown in Tab. 9, raw achieved performance varies substantially across GPUs, reflecting differences in peak compute throughput, memory bandwidth, and architecture. In contrast, CUDABench-Score remains interpretable because each kernel is evaluated relative to the attainable roofline performance of the target GPU. The original MSE kernel consistently achieves very low normalized utilization, while the optimized version achieves much higher utilization across all tested GPUs.

These results provide an additional case study showing that the same hardware-independent scoring formulation can be instantiated on different GPUs using each target GPU's peak compute throughput, peak memory bandwidth, and measured profiling statistics. The goal is not to require identical scores across hardware, but to provide a normalized performance evaluation that is more informative than raw execution time alone.

Table 9: Additional cross-GPU case study for the MSE kernel and its optimized version. Arith denotes arithmetic intensity, Perf denotes achieved GFLOPs/sec, Peak denotes attainable GFLOPs/sec under the target GPU roofline, and Score denotes Performance-Score.

Kernel	GPU	Arch.	Arith	Perf	Peak	Score
MSE	RTX 5080	Blackwell	0.18	0.53	172.0	0.3%
MSE	RTX 3090	Ampere	0.19	0.47	172.1	0.3%
MSE	RTX 4090	Ada Lovelace	0.18	0.60	184.5	0.3%
MSE	A40	Ampere	0.17	0.40	115.4	0.4%
MSE Opt.	RTX 5080	Blackwell	0.55	383.0	518.9	73.8%
MSE Opt.	RTX 3090	Ampere	0.48	322.3	435.4	74.0%
MSE Opt.	RTX 4090	Ada Lovelace	0.55	414.9	554.9	74.8%
MSE Opt.	A40	Ampere	0.42	232.1	295.2	78.6%

A.6 Applicability to Kernel Optimization

Although CUDABench focuses on from-scratch text-to-CUDA generation, the proposed roofline-based CUDABench-Score is not limited to this setting. It can also serve as a principled metric for evaluating CUDA kernel optimization quality, since it measures how closely a valid kernel approaches the attainable performance of the target GPU.

As illustrated in Fig. 4, CUDABench-Score clearly distinguishes the original MSE kernel from its optimized version. The optimized kernel achieves a substantially higher score on both A40 and RTX 4090, indicating that the metric captures whether an improved implementation moves closer to the target GPU's roofline limit. Therefore, while optimization-oriented kernel generation is not the main focus of this benchmark, CUDABench-Score provides a natural evaluation basis for future studies on LLM-based CUDA kernel optimization.

Moreover, each task in CUDABench-Set includes a manually implemented and functionally correct baseline CUDA kernel, `bench.cu`, together with its data generator and validator. These components provide a potential starting point for future optimization-oriented extensions, where models are given an existing CUDA implementation and asked to improve its efficiency under the same correctness constraints.

A.7 Lexical Overlap and Contamination Analysis

We examine whether the reduced guidance in Level 3 leads to increased lexical copying of reference kernels. The audit covers seven models, three difficulty levels, 495 instances (99 base tasks at five input scales), and three attempts, totaling 31,185 generation slots. We analyze only code within `__global__` and `__device__` functions, excluding host/I/O scaffolding, comments, strings, and preprocessor directives. To filter common coding patterns, we remove n-grams appearing in at least five distinct reference-task families, counting frequency by base task rather than input scale.

For generated device code G, its task-matched reference R, and the common-boilerplate set C_n, we define the rare n-gram overlap as

$$O_n(G, R) = \frac{|(\mathcal{N}_n(G) \setminus C_n) \cap (\mathcal{N}_n(R) \setminus C_n)|}{|\mathcal{N}_n(G) \setminus C_n|}. \tag{9}$$

Outputs without extractable n-grams are treated as missing rather than zero. The numbers of valid outputs at Levels 1/2/3 are 10,138/9,867/10,030, respectively.

Beyond aggregate overlap, we calibrate unusually high rare-10 similarity using mismatched references. Each generated program is compared against 98 different-task references at the same input scale. A matched overlap is flagged if it reaches the 99-th percentile of mismatched overlaps within the same generated-code-length decile. This identifies unusually high task-specific lexical similarity, rather than establishing training-data contamination.

Tab. 10 shows that Level 3 has the lowest upper-quartile overlap across all three n-gram sizes. In particular, its rare-10 Q3 is 2.548%, compared with 3.614% at Level 1 and Level 2. Its calibrated high-tail rate is also lowest at 28.91%, versus 32.59% and 34.59%.

Table 10: Rare device-code n-gram overlap. Values report median $[Q1, Q3]$; The final row shows the calibrated rare-10 high-tail rate among valid outputs.

Metric	Level 1	Level 2	Level 3
Rare-8	0.996% $[0, 4.715\%]$	1.163% $[0, 4.673\%]$	0.373% $[0, 3.366\%]$
Rare-10	0% $[0, 3.614\%]$	0% $[0, 3.614\%]$	0% $[0, 2.548\%]$
Rare-12	0% $[0, 2.353\%]$	0% $[0, 2.652\%]$	0% $[0, 1.714\%]$
Rare-10 q99 tail	32.59%	34.59%	28.91%

To assess the robustness of these differences, we conduct a post-hoc sensitivity analysis across $n = 8, 10, 12$, using 10,000 cluster-bootstrap replicates (seed `20260727`). We resample the 99 base tasks while retaining all associated models, scales, and attempts within each cluster.

Table 11: Post-hoc n-gram sensitivity analysis. Entries report median bootstrap differences in Q3 overlap and 95% percentile confidence intervals, in percentage points.

Rare n	Level 3 – Level 1	Level 3 – Level 2
8	-1.227 $[-1.904, -0.667]$	-1.179 $[-2.016, -0.585]$
10	-0.962 $[-1.533, -0.378]$	-1.006 $[-1.690, -0.339]$
12	-0.568 $[-1.117, -0.159]$	-0.856 $[-1.516, -0.293]$

As shown in Tab. 11, all six confidence intervals for the Level 3 – Level 1/Level 2 Q3 differences exclude zero, confirming that the lower upper-quartile overlap at Level 3 is consistent across neighboring n-gram sizes. Rare-8 and rare-12 serve as post-hoc sensitivity checks rather than independent confirmatory tests.

Overall, we find no evidence of increased reference-specific lexical copying at Level 3. Although lexical overlap cannot rule out training-data exposure or non-verbatim memorization, these results suggest that reduced prompt guidance does not lead to greater verbatim reproduction of reference kernels.

B Responsible Discussion

B.1 Scope of Text-to-CUDA Generation

CUDABench focuses on from-scratch text-to-CUDA generation, which we view as a broad CUDA engineering paradigm. Although optimizing existing kernels is important, especially for AI workloads, many CUDA applications start from mathematical or algorithmic descriptions rather than existing kernel implementations, such as PDE solvers, signal processing kernels, and scientific computing workloads.

This scope is particularly relevant because GPUs are used far beyond AI, and the overlap between AI-related and non-AI kernels in CUDABench-Set is only 18.2%. Many non-AI workloads therefore require domain-specific CUDA implementations instead of optimization of standard AI kernels. Even within AI, newly emerging operators also naturally fit the text-to-CUDA setting. Thus, CUDABench targets a broad from-scratch generation scenario while treating kernel optimization as a complementary direction.

B.2 Broader Impacts

CUDABench is intended to support reproducible evaluation of LLMs for CUDA kernel generation. Its positive impact is to provide a systematic benchmark for understanding the correctness and efficiency limitations of current LLM-generated GPU code. A potential negative impact is that stronger CUDA generation systems could be misused to produce inefficient, unsafe, or hard-to-audit low-level code. CUDABench mitigates this risk by focusing on benchmark evaluation rather than deploying an autonomous code-generation system, and by using execution-based correctness verification.

B.3 Asset Documentation and Release

CUDABench-Set is a newly introduced benchmark asset. Each benchmark instance includes a task prompt, a manually implemented reference CUDA kernel, a data generator, and a validator. Existing repositories and documentation, including PyTorch APIs and NVIDIA CUDA Samples, are used only as references for task construction rather than directly copied as benchmark kernels. The released dataset and code include documentation for task structure, prompt format, validation scripts, and reproduction instructions.

C Evaluation and Verification Details

C.1 Generation and Correctness-Verification Protocol

The main evaluation uses independent, from-scratch generations. Models receive the fixed system prompt and the level-specific task specification, including I/O fields and target GPU. They do not receive reference kernels, generated test values, validator outcomes, profiler measurements, or optimization feedback. Pass@3 aggregates independent attempts rather than sequentially refining a kernel. The supplementary iterative-refinement baseline in Appendix A.2 is a separate setting.

Every task/scale has its own generator, reference kernel, and validator. The generator produces randomized inputs with the prescribed shapes and data types and corresponding expected outputs; inputs are chosen to avoid numerical underflow during validation. The generated CUDA program must compile with NVCC, execute successfully on these inputs, and pass `compare.py` under the predefined tolerance policy. Only then is it profiled. Compilation, execution, and functional failures contribute zero through the indicator in Eq. (5), with the denominator retained over all tasks. This protocol removes the feedback channel available in iterative optimization.

C.2 Numerical Tolerance Policy

CUDABench uses a standardized tolerance policy based on numerical semantics rather than task-specific author preference. Discrete outputs, including integer labels, indices, histogram bins, sampled point indices, nearest-neighbor indices, max-subarray sums, and stereo disparity outputs, are compared exactly. Packed 8-bit image outputs are compared per channel after unpacking: a one-level channel difference is allowed for direct float-to-uint8 image filtering, and a two-level difference is allowed for recursive image filters where roundoff may accumulate before quantization.

For floating-point outputs, thresholds are grouped by expected forward error sources. Float32 kernels without substantial accumulation use a tolerance of 10^{-5}, covering pointwise activations, simple data movement, reshaping, selection, sorting, max/min pooling, and max/min reductions. Kernels with short local reductions or moderate arithmetic depth use 10^{-4}, including small filters, color conversion, pairwise distances, and row-wise variance. Kernels involving reductions, normalization, interpolation, wavelet transforms, probabilistic losses, or normalized nonlinear operations use 10^{-3}. Kernels with large or order-dependent accumulation use 10^{-2}, including matrix products, convolutions, attention, optimizers, graph convolution, prefix sums, dot products, and scientific simulations. Repeated matrix powers use 10^{-1} because repeated float32 matrix products can amplify absolute roundoff error more than single-pass dense linear algebra.

These thresholds are intentionally grouped by numerical error sources, including quantization, exact discrete semantics, float32 library differences, and accumulation order. The policy avoids per-instance tuning based on observed generated outputs.

C.3 Nsight Compute Measurement and Score Calculation

The implementation is `eval_eff_only` in `CUDABench/Evaluate/evaluator_core.py`. For each kernel name, metric values are averaged over its Nsight Compute records. The counters and their roles are:

Table 12: Nsight Compute counters used for FP32 roofline scoring.

Counter	Role
`smsp__sass_thread_inst_executed_op_fadd_pred_on.sum.per_cycle_elapsed`	r_{add}
`smsp__sass_thread_inst_executed_op_fmul_pred_on.sum.per_cycle_elapsed`	r_{mul}
`smsp__sass_thread_inst_executed_op_ffma_pred_on.sum.per_cycle_elapsed`	r_{fma}
`smsp__cycles_elapsed.avg.per_second`	f_{smsp}
`sm__sass_thread_inst_executed_op_ffma_pred_on.sum.peak_sustained`	$r_{\text{fma}}^{\text{peak}}$
`sm__cycles_elapsed.avg.per_second`	f_{sm}
`dram__bytes.sum.per_second`	B
`dram__bytes.sum.peak_sustained`	Peak bytes/cycle
`dram__cycles_elapsed.avg.per_second`	f_{dram}

FADD and FMUL count as one FP32 FLOP, and FFMA as two. The `pred_on` counters exclude predicated-off operations. Using the measured SMSP, SM, and DRAM frequencies,

$$F = (r_{\text{add}} + r_{\text{mul}} + 2r_{\text{fma}}) f_{\text{smsp}}, \qquad P_{\text{peak}} = 2 r_{\text{fma}}^{\text{peak}} f_{\text{sm}}, \tag{10}$$

$$B_{\text{peak}} = \texttt{dram__bytes.sum.peak_sustained}\, f_{\text{dram}}, \qquad \text{AI} = F/B. \tag{11}$$

F is achieved FLOP/s, and B is achieved off-chip DRAM bytes/s. The resulting score is

$$\text{Performance-Score} = \begin{cases} B/B_{\text{peak}}, & |F| < 10^{-9}, \\ F/\min(P_{\text{peak}}, \text{AI}B_{\text{peak}}), & \text{otherwise.} \end{cases} \tag{12}$$

Thus arithmetic intensity reflects the operations and traffic realized for the profiled input, including data-dependent control flow and dynamic memory access; It is not statically assigned from the algorithm. This permits scoring individual dynamic kernels without claiming coverage of complete multi-kernel MoE or runtime-sparse pipelines. If a program exposes multiple kernel names, the current implementation selects the one with the highest measured FLOP/s, or highest DRAM bandwidth if all have zero FP32 FLOP/s. Accordingly, its roofline score is not an end-to-end multi-kernel score; the fixed-reference timing in Appendix A.1 instead sums kernel-invocation durations.

D Prompt Template and Difficulty Examples

D.1 System and Task Prompts

System Prompt

You are an expert CUDA High-Performance Computing (HPC) engineer. Your goal is to write a standalone, compilable, and highly efficient CUDA program based on the user's task description.
CRITICAL INSTRUCTIONS

1. **Output Format**:
   - You must output **ONLY** a single code block (starting with "'cpp and ending with "').
   - Do NOT include any introductory text, explanations, or concluding remarks.
   - The content inside the block must be the complete C++ source code.
2. **Mandatory Helper Functions & Headers**:
   - You MUST start your code with the exact boilerplate provided below.
   - **Do NOT add any other headers** (like `<algorithm>` or `<cmath>`) outside this block; they are already included.
   - Do NOT modify the `read_binary` or `write_binary` functions.

// — BEGIN REQUIRED BOILERPLATE —

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cmath>
#include <algorithm>
#include <cfloat>
#include <cuda_runtime.h>

void read_binary(const std::string& filename, float* data, size_t size)
{{
    std::ifstream in(filename, std::ios::binary);
    if (!in) {{
        std::cerr << "Can not open: " << filename << std::endl;
        exit(1);
    }}
    in.read(reinterpret_cast<char*>(data), size * sizeof(float));
    in.close();
}}

void write_binary(const std::string& filename, const float* data,
size_t size) {{
    std::ofstream out(filename, std::ios::binary);
    if (!out) {{
        std::cerr << "Can not write: " << filename << std::endl;
        exit(1);
    }}
    out.write(reinterpret_cast<const char*>(data), size * sizeof(float));
    out.close();
}}
```

// — END REQUIRED BOILERPLATE —

3. **Implementation Rules**:
    - **Single Kernel Entry**: Implement the main logic in a single `__global__` function. You may define `__device__` helper functions if needed.
    - **Precision**: Use `float` (float32) for all data.
    - **Math Functions**: Do NOT use `std::max`, `std::min`, or `std::abs` in device code. ALWAYS use intrinsics: `fmaxf`, `fminf`, `fabsf`, `sqrtf`, `expf`, etc.
    - **Safety**: Always perform boundary checks.
4. **Strict Data & Parameter Binding** (CRITICAL):
    - **Variable Names**: You MUST match the variable names in the `main` function to the names provided in the Input/Output section (e.g., Input `matA` -> Host pointer `h_matA` / Device pointer `d_matA`).
    - **Filenames**: You MUST assume all input and output files are located in the `data/` directory. Construct the file path by prepending `data/` and appending `.bin` to the variable name (e.g., Input `matA` -> read from `"data/matA.bin"`), unless a full path is explicitly provided.
    - **Shapes & Sizes**:
        - Respect the explicit shape given (e.g., `(1024, 4096)`).
        - Calculate total elements as the product of dimensions.
        - Hardcode these dimensions as `const int` or `#define` if they are fixed numbers in the prompt. Do NOT use arbitrary default sizes.
    - **Data Types**: Even if the input says `int` or `double`, you must treat them as `float` to comply with the mandatory boilerplate, or cast them appropriately if logic requires.
5. **Main Function Logic**:
    - Define file paths based on the Rule #4.
    - Allocate Host memory (`new float[...]`) and Device memory (`cudaMalloc`) using the EXACT sizes from the task description.
    - Read input using `read_binary`.
    - Copy Host -> Device (`cudaMemcpy`).
    - **Launch Configuration**: Calculate grid and block dimensions dynamically based on input size. Optimize for the {gpu}.
    - Launch the kernel.
    - Copy Device -> Host.
    - Write output using `write_binary`.
    - Free all memory (`cudaFree`, `delete[]`).

Task Prompt

TASK SPECIFICATION
Task Name: {task_name}
Task Description: {task_description}
Input: {input_spec}
Output: {output_spec}
GPU: {gpu}
CUDA program:

D.2 Difficulty-Level Example

Here shows an example for the different level task description of Haar Wavelet Transform kernel (in the Breadth dimension of Signal Processing domain, and the Depth dimension with an input size of 8,192).

Level 1

Implement a CUDA kernel for the Haar wavelet transform (DWT_Haar). The input is a signal tensor of 8192 float32 values. The kernel must recursively decompose the signal into detail and approximation coefficients. Each thread block processes 1024 elements using 512 threads. The computation involves pairwise operations: detail coefficients are calculated as $(x - y) * 1/\sqrt{2}$ and approximation coefficients as $(x + y) * 1/\sqrt{2}$. Detail coefficients from all decomposition levels must be stored in the 8192-length output tensor, with level-1 details placed in the second half. The final approximation coefficient per block must be stored in an 8-length output tensor. Constraints: fixed input size (8192), 8 blocks, and 512 threads per block.

Level 2

Compute a Haar wavelet transform on an 8192-element signal. Divide the signal into 8 blocks. For each block, recursively apply pairwise operations: details = $(x - y)/\sqrt{2}$ and approximations = $(x + y)/\sqrt{2}$. Store all detail coefficients in an 8192-output array (level-1 details in indices 4096-8191). Store the final approximation per block in an 8-element array.

Level 3

Compute the Haar wavelet transform for an 8192-element signal, generating detail coefficients and block-wise approximations.

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: The main claims and contributions are stated in the Abstract and Sec. 1, with scope and limitations clarified in Appendix **??**.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: The paper discusses key limitations in Sec. **??**, including NVIDIA-specific tooling, single-kernel scope, and contamination risk.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

Answer: [N/A]

Justification: The paper does not present theoretical results or formal proofs; it defines evaluation metrics and validates them empirically in Sec. 4 and Sec. 5.

Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: Sec. 3–5 describe the dataset construction, evaluation pipeline, metrics, models, hardware, and Pass@k setting needed to reproduce the main experimental results.

   Guidelines:

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

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

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

Answer: [Yes]

Justification: The dataset, evaluation scripts, and reproduction instructions are provided in the supplemental material.

Guidelines:

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: Sec. 5 specifies the evaluated models, hardware, metrics, and Pass@k setting; dataset and prompt details are described in Sec. 3 and Appendix D.1–D.2.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [Yes]

   Justification: The paper reports Pass@k results over multiple samples.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: Sec. 5 reports the GPUs used for evaluation, including A40 and RTX 4090 specifications; profiling is performed with NVIDIA Nsight Compute as described in Sec. 4.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: The research follows the NeurIPS Code of Ethics and preserves anonymity in the submission.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

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

    Answer: [Yes]

    Justification: Appendix **??** discusses limitations and potential misuse considerations, including the current single-kernel scope and lack of coverage for misuse scenarios.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.

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

11. **Safeguards**

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

    Answer: [N/A]

    Justification: The released asset is a CUDA benchmark dataset rather than a model or scraped dataset with high misuse risk.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: Existing assets used for task construction, including PyTorch documentation and NVIDIA CUDA Samples, are cited in the paper and used only as references for rewritten benchmark tasks.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

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

    Answer: [Yes]

    Justification: The paper introduces CUDABench-Set as a new benchmark asset, documented in Sec. 3 with task construction, prompt levels, generators, validators, and reference kernels.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

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

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

    Answer: [N/A]

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

    Guidelines:

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

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

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

    Answer: [Yes]

    Justification: LLMs are the evaluated subjects of CUDABench, and their usage, prompting setup, and evaluated model versions are described in Sec. 3 and Sec. 5.

    Guidelines:

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