arXiv:2605.08467v1 [cs.LG] 8 May 2026

CUDAHercules: Benchmarking Hardware-Aware Expert-level CUDA Optimization for LLMs

Shiyang Li
University of Minnesota
li004074@umn.edu

Zijian Zhang
University of Minnesota
zha00175@umn.edu

Guangyan Sun
University of Minnesota
sun01158@umn.edu

Yuebo Luo
University of Minnesota
luo00466@umn.edu

Winson Chen
University of Minnesota
chen9619@umn.edu

Yanzhi Wang
Northeastern University
yanz.wang@northeastern.edu

Mingyi Hong
University of Minnesota
mhong@umn.edu

Caiwen Ding
University of Minnesota
dingc@umn.edu

Abstract

Large language models show promise for automated CUDA programming, however even the strongest coding models (e.g., Claude-Opus-4.6) may still fall short of expert-level, architecture-aware optimization. We introduce **CUDAHercules**, a benchmark that evaluates generated CUDA against end-to-end human-expert SOTA systems. It spans single kernels, module-level operators, full applications, and unsolved challenge tasks across Ampere, Hopper, and Blackwell GPUs, with end-to-end tasks gated by domain-specific semantic validators. Evaluating models such as Claude-Opus-4.6 and GPT-5.4 shows a large gap between runnable CUDA and expert CUDA engineering: models often compile and pass tests, but rarely recover the optimization strategies needed to match expert performance. Application semantics further reduce success, and iterative or tool-augmented feedback can improve correctness while drifting toward slow fallback implementations. These results show that automated CUDA programming remains far from fully solved and requires stronger hardware reasoning, better tool use, and training objectives that connect code understanding to hardware architecture-grounded intelligence.

1 Introduction

Large language model (LLM)-based systems have recently shown substantial potential for automated CUDA programming [3, 28, 12, 11, 2]. Given a PyTorch operator [17, 9] or natural-language specification [13], current models can often generate compilable CUDA code and sometimes outperform functional baseline implementations. This progress raises an important question for GPU systems research: can LLMs move beyond functional code generation and perform the kind of deep optimization required in expert CUDA engineering?

Existing evaluation benchmarks do not fully answer this question. Most current benchmarks measure whether a model can translate an operator into CUDA or produce a correct kernel under a fixed interface [17, 9]. This is useful for measuring code-generation reliability, but it misses a central property of CUDA programming: high-performance kernels are the result of extreme hardware-software co-design. Effective implementations depend on optimization decisions that are tightly coupled to both workload structure and target GPU architecture, and these decisions are not stable

Preprint.

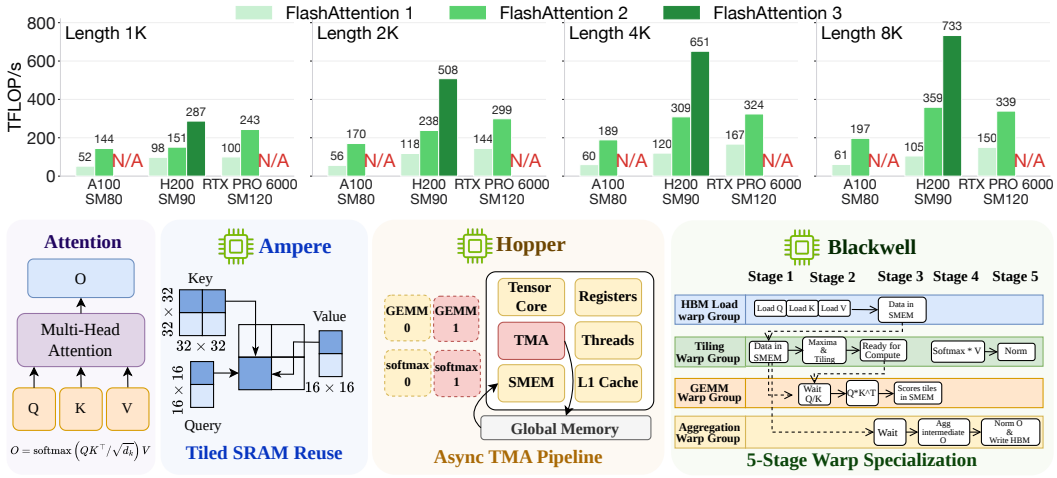


Figure 1: **Top:** throughput (TFLOP/s) of FA1, FA2, and FA3 on causal forward attention (head dim 128) across A100, H200, and RTX PRO 6000 Blackwell at sequence lengths 1K–8K; N/A marks unsupported configurations. **Bottom:** SOTA optimization strategies for self-attention vary across GPU architectures, motivating architecture-aware kernel design.

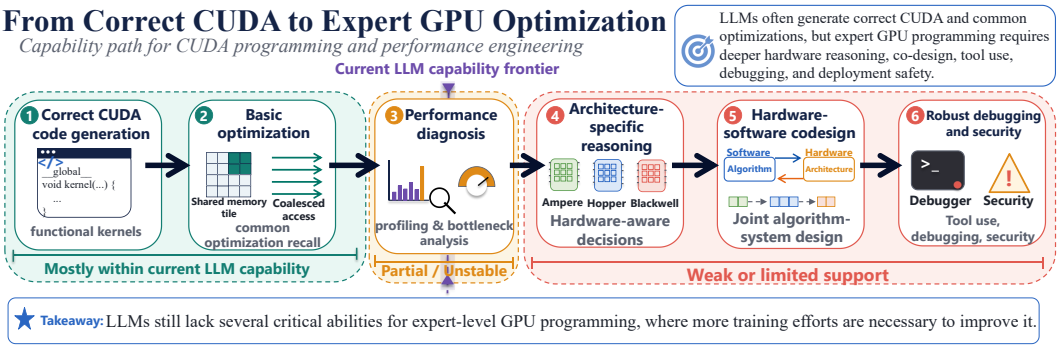


Figure 2: Capability path from correct CUDA generation to expert-level GPU optimization. Current LLMs are strongest at functional kernel generation and common optimization recall, partially reliable at performance diagnosis, but have limited ability in architecture-specific reasoning, hardware-software co-design, and robust debugging or deployment security.

across GPU generations. Even within a single workload family, the top panel of Figure 1 shows that the throughput of FlashAttention variants changes substantially across FlashAttention V1 [6] to FlashAttention V2 [5] and FlashAttention V3 [20] on different GPU architectures. The bottom panel illustrates why: the optimization strategies of FlashAttention move from tiled shared-memory reuse on Ampere to asynchronous pipelines on Hopper and warp-specialized execution on Blackwell, with each variant designed for a specific hardware architecture.

Another limitation is reference quality. Many existing benchmarks compare generated kernels against functional baselines rather than human-expert CUDA implementations. A model can therefore appear successful by producing a correct solution that beats a weak baseline, even if it remains far from human-expert performance and does not recover any architecture-specific optimization strategy. Such evaluation can overestimate progress toward automated CUDA engineering, because it conflates functional synthesis with real deep optimization.

Together, these limitations point to a capability gap rather than a simple correctness gap. Figure 2 decomposes CUDA programming into a path from functional generation and common optimization recall to bottleneck diagnosis, architecture-specific implementation, hardware-software co-design, and robust debugging. **CUDAHercules** targets these later stages: whether an LLM-based system can produce CUDA implementations that are correct, competitive with human-expert references, specialized to the target GPU, and valid under the semantic constraints of real applications.

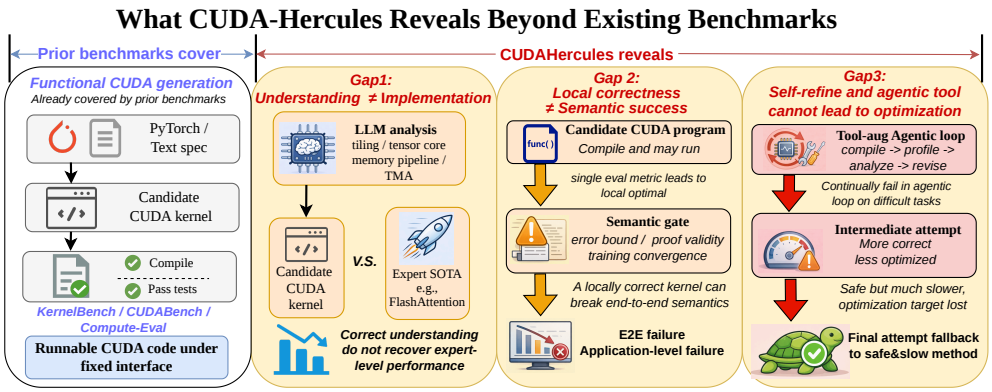


Figure 3: What **CUDAHercules** reveals beyond existing CUDA-generation benchmarks. Prior benchmarks primarily cover functional CUDA generation under fixed interfaces, while **CUDAHercules** exposes another three ability gaps for current LLMs.

We introduce **CUDAHercules**, a benchmark for measuring expert-referenced, architecture-aware CUDA optimization by LLM-based systems. Tasks are drawn from curated state-of-the-art CUDA systems beyond ML inference, e.g., cryptographic proving and scientific computing [14, 9, 8, 16, 27, 26, 22, 7, 10, 24, 25, 18, 23]. Each task specifies a target architecture, solution interface, build and execution harness, correctness or semantic validator, expert-level reference, and performance parser through a unified `task.yaml` metadata schema. Candidate implementations are evaluated against human-expert references rather than naive baselines; architecture-specific subsets target Ampere, Hopper, and Blackwell GPUs; and application tasks require domain-specific semantic validity.

We evaluate three frontier models, GPT-5.4 [15], Claude-Opus-4.6 [1], and Qwen3.5-122B-A10B [19], one domain-specific RL model Kevin32B [2], and two multi-agent methods, CUDAForge [28] and StitchCUDA [11], on **CUDAHercules**. As summarized in Figure 3, existing CUDA-generation benchmarks mainly cover functional code generation under fixed interfaces, while **CUDAHercules** exposes three deeper gaps. First, current models can often describe expert-level strategies but fail to realize comparable implementations in code. Second, locally correct kernels can fail under application-level semantic validators such as bounded error or proof verification. Third, agentic refinement can improve correctness while reducing optimization quality, because the search drifts toward slow fallback implementations. These findings suggest that the primary bottleneck is not only CUDA code generation, but also reliably turning optimization knowledge into deployable kernels.

Our main contributions are summarized as follows:

- **An expert-referenced benchmark for LLM-based CUDA programming.** We release 195 tasks spanning single kernels, modules, full application workloads, and unsolved challenges across multiple domains, with expert-quality references from SOTA systems such as CUTLASS library and FlashAttention-3.

- **Architecture-aware and semantics-gated evaluation.** We evaluate generated CUDA code against expert references under architecture-specific targets across Ampere, Hopper, and Blackwell GPUs, and require application-level semantic validity, including bounded error, convergence, and proof correctness, before assigning performance credit.

- **Architecture-grounded kernel intelligence.** Through evaluation of frontier models and agentic systems, **CUDAHercules** identifies the need for stronger code understanding with hardware architecture-aware implementation, and the risk of degenerative optimization behavior during agentic search. It further argues that effective CUDA acceleration requires joint software-hardware co-design tailored to the practical properties of real GPU operations.

Benchmark	Setting	Tasks	Multi-domain coverage	Human expert reference	Application-level semantics	Architecture-aware
KernelBench	PyTorch → CUDA	250	✗	✗	✗	✗
CUDABench	Text → CUDA	500	✓	✗	✗	✗
TritonBench-T	PyTorch → Triton	166	✗	✗	✗	✗
ComputeEval	Text → CUDA	566	✓	✗	✗	✗
CUDAHercules	Text → CUDA	195	✓	✓	✓	✓

Table 1: Comparison with other GPU kernel generation benchmarks. **CUDAHercules** is distinguished by expert-level references, architecture-aware evaluation, and application-level semantic validation.

2 Related Work

GPU kernel generation benchmarks. KernelBench [17] measures PyTorch-to-CUDA replacement, and recent systems report strong scores on it [28, 11, 2, 12, 4]. CUDABench [29] expands text-to-CUDA synthesis with multiple prompt levels and automated correctness checks. TritonBench [9] evaluates PyTorch-to-Triton generation, and ComputeEval [13] provides lightweight text-to-CUDA evaluation. These benchmarks establish reproducible functional evaluation, but their reference targets are primarily functional baseline implementations. They do not evaluate generated code against expert-level implementations, and they do not jointly test architecture-specific optimization and application-level semantic validity.

LLM-based automated CUDA programming systems. Existing systems use LLMs, agentic search, and tool feedback to generate or optimize CUDA code. CUDAForge [28] incorporates profiling feedback into iterative kernel improvement. StitchCUDA [11] proposes a multi-agent pipeline with rubric-based single-turn reinforcement learning (RL) for end-to-end GPU programming. Both CUDAForge and StitchCUDA note that iterative feedback can produce unstable or degenerative optimization behavior, but their evidence is primarily tied to KernelBench-style functional CUDA tasks, which limits how clearly such behavior can be evaluated against expert-level optimization targets. Kevin32B [2] and CUDA Agent [4] improve the CUDA programming ability of LLM with multi-turn RL. These systems motivate the evaluation settings in **CUDAHercules**: one-shot evaluation measures direct generation, while tool-augmented and iterative-refinement settings measure whether execution feedback and agentic search close the gap to expert implementations.

Positioning. Table 1 positions **CUDAHercules** by evaluation axes. Existing benchmarks primarily test whether a model can produce a runnable CUDA or Triton implementation under a fixed interface. **CUDAHercules** instead evaluates the stages that remain after basic code generation: matching expert-level references and architecture-specific requirements, preserving application-level semantics, and reporting compilation, correctness, speed, and agentic-search behavior as separate diagnostics.

3 Benchmark Design

This section specifies the task organization, execution protocol, and scoring metrics used by **CUDAHercules**. Each task is a self-contained evaluation unit with a target architecture(SM version), reference solution interface, build and execution harness, correctness or semantic validator, source code of baselines for open-sourced references or binary for closed-sourced references, performance parser, and anti-cheating metadata. Here, anti-cheating metadata refers to validity rules that prevent benchmark-specific shortcuts such as direct reference reuse, disallowed library calls, hard-coded outputs, or timing manipulation.

CUDAHercules follows four design requirements:

1. **Use expert references.** Candidate performance is measured against optimized implementations from human-expert level CUDA systems rather than functional naive baselines. FlashAttention-3 [20], for example, provides an expert Hopper reference for self-attention.
2. **Encode target hardware.** Task metadata records GPU requirements because optimization strategies depend on architecture-specific instructions, memory pipelines, and scheduling primitives.

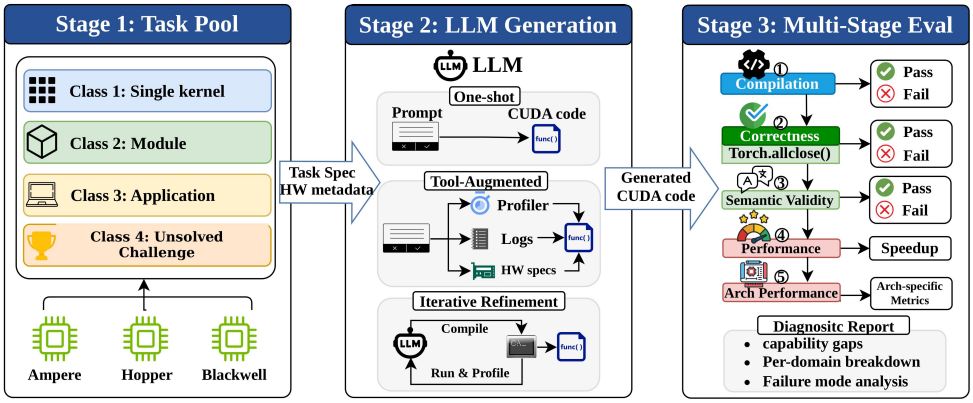


Figure 4: The test workflow of **CUDAHercules**.

Class	Unit of optimization	Tasks	Architecture coverage
Class 1	Single kernel	63	20 general, 21 Hopper, 22 Blackwell
Class 2	Module or kernel family	119	43 general, 64 Hopper, 12 Blackwell
Class 3	Full application workload	10	Blackwell evaluation
Class 4	Unsolved challenge task	3	Blackwell evaluation

Table 2: Task classes in **CUDAHercules**. Counts report released tasks; architecture buckets apply to Classes 1–2 in the current release.

In the Blackwell_FP8_GEMM task, the target is an SM100 GPU, and the expert implementation relies on Blackwell FP8 tensor-core execution, where a generic CUDA GEMM is not equivalent.

3. **Cover multiple domains.** The released task set spans ML training and inference, HPC, graph analytics, imaging, GPU compression, formal verification, cryptography, and communication-heavy multi-GPU workloads.

4. **Require semantic validation.** Application tasks use domain-specific checks, such as bounded reconstruction error for GPU lossy compression, stable loss decrease for LLM training, and valid proof verification for GPU-based zero-knowledge proving.

3.1 Task Classes

We categorize our tasks by the scope of the optimization goal, ranging from local kernel level to application-wise, with specific hardware and software environment setup , rather than solely the difficulty of the task. The difference stems from our observations that previous benchmarks can become agnostic to detailed HW/SW environments. Because the CUDA implementations behind high-level abstraction (e.g., PyTorch API) vary across HW/SW environments. This can lead to manifest performance discrepancies of reference baselines on the same hardware architecture with different PyTorch versions, or the same PyTorch version on different hardware.

CUDAHercules instead fixes both the target hardware and the expert reference. Class 1 isolates single expert kernels, Class 2 covers module-level or kernel-family optimization, Class 3 evaluates full applications with semantic validators, and Class 4 contains unsolved challenge tasks. Thus, measured speedup reflects the gap to human-expert, architecture-aware CUDA on a specified platform. Detailed task inventories and representative case studies for each class are provided in Appendix A.

Overall, the released task set contains 195 tasks organized by optimization scope, as summarized in Table 2. Within Classes 1–2, tasks are additionally bucketed by target architecture. Here, “general” denotes tasks that do not require a special architecture-specific instruction subset. Class 3 are evaluated under the NVIDIA RTX PRO 6000 GPU (Blackwell SM120).

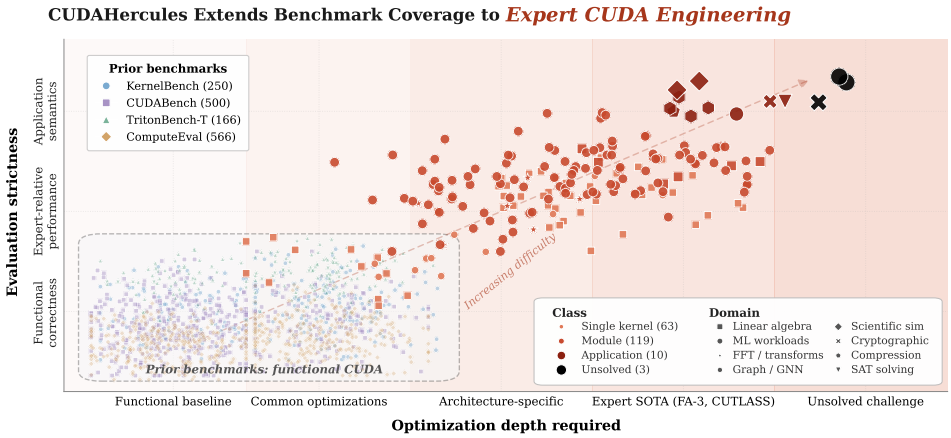


Figure 5: Domain coverage of the **CUDAHercules** task set.

3.2 Domain Coverage and Task Construction

A benchmark that only covers ML inference can overfit to one optimization profile. **CUDAHercules** therefore spans workload groups chosen to exercise qualitatively different optimization regimes. Figure 5 summarizes the domain coverage in the released task set. Detailed descriptions of tasks are provided in Appendix A.

Each task is packaged with a unified metadata file (`task.yaml`) that specifies hardware requirements, runner backend, build and execute commands, correctness mode, timing parser, source provenance, and anti-cheating rules; Appendix B gives a concrete metadata and prompt example. This separates task specification from execution and lets a unified runner dispatch to backend-specific pipelines. The implementation supports four backends: Makefile-based single-kernel tasks, Python-definition module tasks, full-application tasks, and unsolved challenge tasks.

3.3 Evaluation Protocol and Metrics

We evaluate each model in three settings. **One-shot** gives the model the task specification, hardware metadata, and few-shot exemplars without execution feedback; an illustrative prompt is shown in Appendix B. **Iterative refinement** uses CUDAForge [28] to provide bounded compile, run, and execution feedback over multiple turns. **Tool-augmented** evaluation exposes build logs, hardware specifications, and GPU profiling tools under the refinement budget, with tool calls selected by the model through LangChain [21]. We also run strategy-enriched prompt diagnostics, adding high-level expert guidance such as tiling, memory movement, tensor-core use, pipeline organization, or fusion choices while withholding reference code, to test whether failures reflect missing guidance or inability to realize that guidance in CUDA.

As shown in Figure 4, the runner checks the target GPU requirement, creates an isolated workspace, builds the generated solution, runs the task validator, and measures speed only after validation succeeds. Correctness is backend-defined rather than model-reported: Class 1 uses compiled CUDA harnesses over multiple problem sizes, Class 2 compares generated and reference modules on randomized inputs with task tolerances, and Classes 3–4 use application drivers where validity can depend on semantic checks such as error bounds, convergence criteria, energy tolerance, SAT/UNSAT verdicts, or proof verification. This separation lets the benchmark distinguish compilation, correctness, semantic validity, and performance failures.

For a task set T, the compilation rate is the fraction that builds, and the pass rate is the fraction that satisfies the task validator. For a valid task t, the expert-relative speed is

$$s_t = \frac{r_t^{\text{expert}}}{r_t^{\text{gen}}},$$

where both runtimes are parsed by the task's performance parser; invalid tasks receive zero performance credit. We report mean speed as

$$|T|^{-1} \sum_{t \in T} \mathbf{1}\{\text{valid}(t)\} s_t$$

and threshold success as

$$\texttt{fast\_p}(p) = |T|^{-1} \sum_{t \in T} \mathbf{1}\{\text{valid}(t) \wedge s_t \geq p\}.$$

We use `fast_p`(1.05) as the primary threshold because a 5% margin reduces sensitivity to system-level timing noise and better reflects a meaningful improvement over the expert reference.

3.4 Reference Quality and Anti-Cheating Controls

Because many expert references come from public high-performance CUDA systems, **CUDAHercules** records source provenance and applies static and procedural anti-cheating controls. In the current release, prompts do not expose the target expert solution, generated code is checked for disallowed library calls and reference reuse, scoring runs in separate workspaces, and passing solutions can be audited for shortcut behavior such as hard-coded outputs or timing manipulation. Audited violations are counted as failures; detailed control rules are provided in Appendix D.

4 Experiments

We evaluate GPT-5.4 [15], Claude-Opus-4.6 [1], Qwen3.5-122B-A10B [19], and Kevin32B [2] on **CUDAHercules**. For Classes 1 and 2, we report one-shot pass@1/pass@3, iterative self-refinement with a 10-round revision budget (self-refine@10), and tool-augmented search. The tool-augmented runs are implemented in the CUDAForge framework [28]. The "general" tasks without special architecture-specific instruction dependencies are measured on RTX PRO 6000 by default, with the Ampere-instruction-dependent subset evaluated on A100; Hopper-targeted tasks are evaluated on H200; and Blackwell-targeted tasks are evaluated on RTX PRO 6000 or B200 depending on the required arch-specific CUDA instruction. Class 3 tasks are evaluated under the StitchCUDA [11], an end-to-end CUDA programming agentic framework, and all current Class 3 runs are measured on RTX PRO 6000. We do not report pass@N for Class 3 because no evaluated LLM achieves success on any Class 3 task by one-shot generation.

4.1 Headline Results

Table 3 summarizes the results on general subsets of Class 1 (20 tasks) and Class 2 (43 tasks). The strongest one-shot model is GPT-5.4: at pass@3 it reaches 70% correctness on Class 1 and 81% on Class 2, with mean expert-relative speed ratios of $0.257\times$ and $0.463\times$, respectively. Claude-Opus-4.6 is consistently second-best in one-shot correctness, while Qwen3.5-122B-A10B falls off sharply on the Class 2 tasks: at pass@3 it compiles 74% of tasks but solves none of them correctly. Kevin32B solves a small fraction of Class 1 tasks and reaches 20% correctness on Class 2 under self-refine@10, but its mean speed remains very low.

Correctness does not translate into expert-level performance. The best one-shot mean speedups are only $0.257\times$ on Class 1 and $0.463\times$ on Class 2, while `fast_p`(1.05) remains low. The `fast_p` thresholds in Table 3 quantify this distance from human experts: $p = 0.5$ is half-speed, $p = 0.8$ is near-expert speed, and $p = 1.05$ exceeds the expert reference by 5%. And results show that even `fast_p`(0.5) is still low for all evaluated models, indicating the CUDA optimization ability of LLMs is still far from human experts.

Architecture-specific stress tests are harder still. On Hopper H200, only a few settings reach non-zero `fast_p`(1.05), peaking at 14% on Class 1 under Claude Opus 4.6 self-refine@10 and 13% on Class 2 under Claude Opus 4.6 tool augmentation. On Blackwell B200, Claude Opus 4.6 tool augmentation reaches 45% Class 1 pass but only $0.053\times$ mean speed, no setting achieves non-zero `fast_p`(1.05), and Class 2 remains unsolved. Full results are in Appendix C.

		Class 1 general subset (20)						Class 2 general subset (43)					
		Outcome			`fast_p` threshold			Outcome			`fast_p` threshold		
Setting	Model	Compiled	Pass	Mean speed	0.5	0.8	1.05	Compiled	Pass	Mean speed	0.5	0.8	1.05
pass@1	GPT-5.4	70%	45%	0.231×	25%	10%	5%	79%	72%	0.425×	37%	23%	14%
pass@3	GPT-5.4	95%	70%	0.257×	30%	15%	5%	86%	81%	0.463×	37%	28%	14%
pass@1	Claude Opus 4.6	55%	45%	0.108×	5%	0%	0%	74%	65%	0.276×	19%	19%	9%
pass@3	Claude Opus 4.6	70%	50%	0.124×	10%	0%	0%	84%	58%	0.300×	42%	26%	9%
pass@1	Qwen3.5-122B-A10B	70%	20%	0.055×	5%	0%	0%	40%	2%	0.005×	0%	0%	0%
pass@3	Qwen3.5-122B-A10B	85%	35%	0.093×	5%	5%	0%	74%	0%	0.000×	0%	0%	0%
pass@1	Kevin32B	30%	10%	0.047×	5%	0%	0%	23%	0%	0.000×	0%	0%	0%
pass@3	Kevin32B	40%	15%	0.084×	5%	0%	0%	28%	0%	0.000×	0%	0%	0%
Self-refine@10	GPT-5.4	85%	60%	0.309×	25%	20%	15%	67%	56%	0.298×	33%	21%	7%
Self-refine@10	Claude Opus 4.6	90%	65%	0.272×	20%	15%	5%	70%	67%	0.451×	44%	30%	14%
Self-refine@10	Qwen3.5-122B-A10B	100%	90%	0.237×	20%	10%	5%	81%	2%	0.001×	0%	0%	0%
Self-refine@10	Kevin32B	70%	20%	0.105×	0%	0%	0%	30%	20%	0.001×	0%	0%	0%
Tool-augmented	GPT-5.4	100%	85%	0.323×	25%	15%	5%	100%	2%	0.020×	2%	2%	0%
Tool-augmented	Claude Opus 4.6	100%	90%	0.403×	35%	20%	0%	72%	47%	0.138×	16%	5%	2%
Tool-augmented	Qwen3.5-122B-A10B	95%	85%	0.171×	15%	0%	0%	93%	2%	0.007×	0%	0%	0%
Tool-augmented	Kevin32B	30%	10%	0.125×	0%	0%	0%	15%	0%	0.000×	0%	0%	0%

Table 3: Results on the general subsets of **CUDAHercules**. Compiled reports successfully; Pass reports correctness. Mean speed is the expert runtime divided by the generated runtime. The `fast_p` threshold columns report the fraction of tasks with valid speedup at or above the listed threshold. Self-refine@10 uses a 10-round revision budget with compile and execution feedback; tool-augmented Class 1/2 evaluations use a 20-round budget.

4.2 Understanding-to-Implementation Gap

The main result in Table 3 is not a ranking among models, but a gap between functional CUDA generation and expert implementation. The strongest models often compile and sometimes pass correctness checks, yet their expert-relative speed remains far below the reference: GPT-5.4 reaches high pass@3 correctness on both Class 1 and Class 2, but its mean speed stays below $0.5\times$, and `fast_p`(1.05) remains low. Qwen3.5-122B-A10B gives an even sharper example on Class 2: it compiles most pass@3 candidates, but none pass correctness. These results show that compilation, API-level behavior, and local correctness are not enough to measure expert-level CUDA ability.

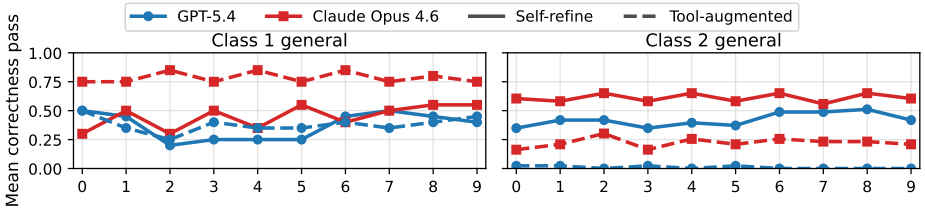
Kevin32B is especially diagnostic. It is a CUDA-specialized model that has already been RL-tuned on KernelBench and reported strong performance there [2, 17]. On **CUDAHercules**, however, it solves only a small fraction of Class 1 tasks, reaches only 20% correctness on Class 2 under Self-refine@10, and its mean speed remains near zero. This contrast suggests that prior benchmarks can teach models benchmark-specific CUDA generation patterns without training the transferable skills needed to write expert-level, architecture-aware CUDA.

Strategy-enriched prompting also does not close the gap. We tested prompts that inject expert-level optimization guidance, such as intended tiling, memory movement, tensor-core use, and fusion strategy, while still withholding the reference implementation. Models can often restate these strategies or produce plausible design rationales, but the generated code still misses low-level scheduling details, uses the wrong memory hierarchy, fails to coordinate fused stages, or falls back to simpler, correct-but-slow kernels. The bottleneck is therefore not prompt underspecification alone; models must learn to execute high-level optimization plans into correct, architecture-specific, high-performance CUDA code.

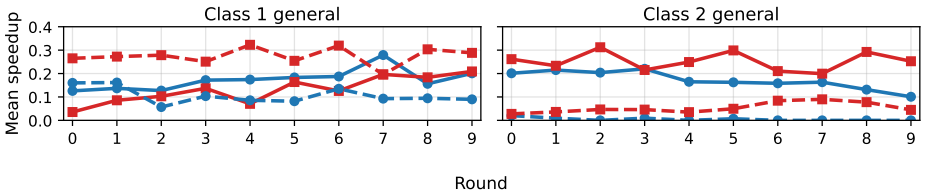
4.3 Agentic Search Helps Inconsistently

Table 3 shows that feedback-based workflows can improve Class 1 correctness, but still do not close the performance gap. Claude Opus 4.6 reaches 90% pass under tool augmentation and Qwen3.5-122B-A10B reaches 90% under self-refine@10, yet the best Class 1 mean speed ratio is only $0.403\times$ and the best `fast_p`(1.05) is 15%. Agentic search can therefore repair many functional failures without recovering expert schedules.

Class 2 shows the opposite pattern: tool use often degrades outcomes. Claude Opus 4.6 drops from 67% pass under Self-refine@10 to 47% under tool augmentation, and GPT-5.4 drops to 2%



(a) Mean correctness pass.



(b) Mean expert-relative speedup.

Figure 6: Per-round correctness and expert-relative speedup on the Class 1 and Class 2 general subsets under Self-refine@10 and tool augmentation. Incorrect candidates contribute zero.

under tool augmentation. Figure 6 shows the same behavior round by round: correctness and speed are non-monotonic, and the Class 2 tool-augmented trajectories often collapse. This degenerative behavior was noted by prior agentic systems [28, 11], but **CUDAHercules** makes the gap sharper because the search is judged against expert references. This suggests that current models have limited capability to leverage CUDA profiling tools and execution feedback for deep optimization; they often overfit to local diagnostics or preserve compilation while breaking correctness or performance.

4.4 Application-Level Semantic Evaluation

Class 3 is evaluated in the StitchCUDA setting, which is an agentic framework for end-to-end GPU programming with access to tools such as Nsys, NCU, file retrieval, and file editing. The main empirical result is that application-level semantic gating sharply reduces success. Across the current Class 3 task set, Claude-Opus-4.6 reaches $0.35\times$ on `cuszp`, $0.74\times$ on `llmc`, and $0.52\times$ on `Liberator`. GPT-5.4 only reaches $0.43\times$ on `Liberator`. Qwen3.5-122B-A10B and Kevin32B do not solve any Class 3 task under the full semantic contract.

The dominant failure mode is not the inability to produce executable CUDA, but the inability to preserve application semantics. GPT-5.4 and Claude-Opus-4.6 both execute full workloads for `gpumd` and `exachem_ccsd_t`, yet still fail byte-level force checks or energy tolerances. Both models also fail every `icicle_zk` round despite repeated successful builds. The resulting picture is stark: once end-to-end semantic validity is enforced, both success rate and performance drop substantially. Class 4 unsolved challenge tasks all remain unsolved in the current evaluation. No evaluated model produces a semantics-valid implementation for any Class 4 task.

5 Conclusion

CUDAHercules evaluates LLM-based CUDA generation against expert-level implementations across single kernels, module-level operators, full applications, and unsolved challenge tasks on Ampere, Hopper, and Blackwell GPUs. The results show a persistent ability gap for LLMs between runnable CUDA and expert-level CUDA engineering: frontier models can often compile code and sometimes pass correctness checks, but they rarely recover the hardware-specific optimization strategies needed to match expert references. The gap widens on architecture-specific and application-level tasks. Our results also show that enriched prompts and iterative or tool-augmented feedback do not close this gap. Reliable automated CUDA programming, therefore, requires progress in hardware reasoning, tool-mediated optimization, robust debugging, and application-level semantic validation.

Task	Model			
	GPT-5.4	Claude Opus 4.6	Qwen3.5 122B-A10B	Kevin32B
TC-GNN	✗	✗	✗	✗
cuSZp	✗	✓ 0.35×	✗	✗
llmc (774M)	✗	✓ 0.74×	✗	✗
Icicle-ZK	✗	✗	✗	✗
GPUMD	✗	✗	✗	✗
ExaChem	✗	✗	✗	✗
ParaFROST	✗	✗	✗	✗
Liberator	✓ 0.43×	✓ 0.52×	✗	✗
MGG-AGNN	✗	✗	✗	✗
MGG-GCN	✗	✗	✗	✗
Tool calls	223	252	196	183

Table 4: Task-level Class 3 results under the StitchCUDA framework with full tool access. Numeric entries denote semantics-valid expert-relative speedup. '✗' means evaluated, but no semantics-valid solution was found. Tool calls count read, write, profile, and planning actions across the Class 3 runs.

References

[1] Anthropic. Claude-opus-4.6, 2026. URL https://claude.ai.

[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] Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. Avo: Agentic variation operators for autonomous evolutionary search, 2026. URL https://arxiv.org/abs/2603.24517.

[4] Weinan Dai, Hanlin Wu, Qiying Yu, Huan ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation, 2026. URL https://arxiv.org/abs/2602.24286.

[5] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

[7] Yafan Huang, Sheng Di, Guanpeng Li, and Franck Cappello. Gpu lossy compression for hpc can be versatile and ultra-fast. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–20, New York, USA, 2025. Association for Computing Machinery. ISBN 9798400714665. doi: 10.1145/3712285.3759817. URL https://doi.org/10.1145/3712285.3759817.

[8] ICICLE. Icicle, 2026. URL https://dev.ingonyama.com.

[9] Jianwei Li, Shilin Li, Zhen Gao, Qian Shi, Yufei Li, Zhen Wang, Jiawei Huang, Haojie Wang, Jiayi Wang, Xinyu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066, 2025.

[10] Shiyang Li, Ruiqi Tang, Jingyu Zhu, Ziyi Zhao, Xiaoli Gong, Wenwen Wang, Jin Zhang, and Pen-Chung Yew. Liberator: a data reuse framework for out-of-memory graph computing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 34(6):1954–1967, 2023. doi: 10.1109/TPDS.2023.3268662.

[11] Shiyang Li, Zijian Zhang, Winson Chen, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Stitchcuda: An automated multi-agents end-to-end gpu programing framework with rubric-based agentic reinforcement learning, 2026. URL https://arxiv.org/abs/2603.02637.

[12] Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations, 2026. URL https://arxiv.org/abs/2602.05885.

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

[14] NVIDIA. Cutlass. GitHub repository, 2026. URL https://github.com/NVIDIA/cutlass.

[15] OpenAI. Chatgpt, 2026. URL https://chatgpt.com.

[16] Muhammad Osama, Anton Wijs, and Armin Biere. Certified sat solving with gpu accelerated inprocessing. *Formal Methods in System Design*, 62(1):79–118, 2024.

[17] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

[18] Ajay Panyala, Niri Govind, Karol Kowalski, Nicholas Bauman, Bo Peng, Himadri Pathak, Erdal Mutlu, Daniel Mejia Rodriguez, Sotiris Xantheas, and Sriram Krishnamoorthy. Exachem/exachem. [Computer Software] https://doi.org/10.11578/dc.20230628.1, jun 2023. URL https://doi.org/10.11578/dc.20230628.1.

[19] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

[20] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

[21] LangChain Team. LangChain, 2026. URL https://github.com/langchain-ai/langchain.

[22] ThunderKittens Authors. Thunderkittens. GitHub repository, 2026. URL https://github.com/HazyResearch/ThunderKittens.

[23] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Ang Li, Kevin Barker, and Yufei Ding. Mgg: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-gpu platforms. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2023.

[24] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. Tc-gnn: Accelerating sparse graph neural network computation via dense tensor core on gpus. In *USENIX Annual Technical Conference*, 2023.

[25] Ke Xu, Hekai Bu, Shuning Pan, Eric Lindgren, Yongchao Wu, Yong Wang, Jiahui Liu, Keke Song, Bin Xu, Yifan Li, Tobias Hainer, Lucas Svensson, Julia Wiktor, Rui Zhao, Hongfu Huang, Cheng Qian, Shuo Zhang, Zezhu Zeng, Bohan Zhang, Benrui Tang, Yang Xiao, Zihan Yan, Jiuyang Shi, Zhixin Liang, Junjie Wang, Ting Liang, Shuo Cao, Yanzhou Wang, Penghua Ying, Nan Xu, Chengbing Chen, Yuwen Zhang, Zherui Chen, Xin Wu, Wenwu Jiang, Esme Berger, Yanlong Li, Shunda Chen, Alexander J. Gabourie, Haikuan Dong, Shiyun Xiong, Ning Wei, Yue Chen, Jianbin Xu, Feng Ding, Zhimei Sun, Tapio Ala-Nissila, Ari Harju, Jincheng Zheng, Pengfei Guan, Paul Erhart, Jian Sun, Wengen Ouyang, Yanjing Su, and Zheyong Fan. Gpumd 4.0: A high-performance molecular dynamics package for versatile materials simulations with machine-learned potentials. *Materials Genome Engineering Advances*, 3(3): e70028, 2025. doi: https://doi.org/10.1002/mgea.70028. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/mgea.70028.

[26] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025. URL https://arxiv.org/abs/2501.01005.

[27] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026. URL https://arxiv.org/abs/2603.05451.

[28] Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, 2025. URL https://arxiv.org/abs/2511.01884.

[29] Jiace Zhu, Wentao Chen, Qi Fan, Zhixing Ren, Junying Wu, Xing Zhe Chai, Chotiwit Rungrueangwutthinon, Yehan Ma, and An Zou. Cudabench: Benchmarking llms for text-to-cuda generation, 2026. URL https://arxiv.org/abs/2603.02236.

A Task Catalog

This appendix provides a detailed description of the **CUDAHercules** task set. For each class, we describe the evaluation mechanics, list the full task inventory, and present representative case studies that illustrate the optimization challenges involved.

A.1 Class 1: Single-Kernel Tasks (63 tasks)

Structure. Each Class 1 task is a self-contained directory containing:

- `description.txt` — task specification including function signature, data types, memory layouts, and build instructions. The description is shown to the LLM.
- `solution.h` — a naive baseline implementation (e.g., a per-thread loop GEMM). The LLM replaces this file.
- `Makefile` — `make ref` compiles the expert reference; `make test` compiles the LLM solution.
- `task.yaml` — unified metadata: hardware requirements (`min_sm`), anti-cheat rules, timing parser regexes, and source provenance.

Evaluation compiles the solution via `make test`, runs it, and parses `Kernel time: / Ref time:` from stdout to compute speedup. Correctness is verified against the expert reference at multiple problem sizes (small, medium, large). All tasks require the solution to contain `__global__ void` and kernel launch syntax (`«<...»>`), blocking trivial host-only or library-call submissions.

Architecture distribution. Class 1 contains 20 general tasks, 21 Hopper (SM90) tasks, and 22 Blackwell (SM100+) tasks. Here, general means the task does not depend on a special architecture-specific instruction set, although many such tasks still require at least SM80 support and several are drawn from CUTLASS Ampere examples. Hopper tasks include warp-specialized GEMMs, FP8 variants, sparse GEMMs, and DeepGEMM/ThunderKittens kernels. Blackwell tasks cover narrow-precision GEMMs (FP4, NVFP4, MXFP8), MoE GEMMs, and low-latency GQA.

Case study: Fused GEMM + Softmax (`gemm_softmax`). This task requires implementing a batched GEMM followed by row-wise softmax in a single fused kernel using FP16 data with FP32 accumulation. The function signature takes matrices A (row-major), B (column-major), C, D, and `Softmax` output, along with batch strides. For each batch: (1) $D = \alpha \cdot A \cdot B + \beta \cdot C$, then (2) $\text{Softmax}[m,n] = \exp(D[m,n] - \max_n D[m,:]) / \sum_n \exp(D[m,n'] - \max_n D[m,:])$. The naive baseline computes these as separate passes. The expert CUTLASS reference fuses them in a single kernel with shared-memory tiling and online softmax reduction. Matching the reference requires fusing the epilogue with the GEMM mainloop, choosing correct tile shapes, and handling the numerical stability of online softmax within the tiled execution.

Case study: Hopper Warp-Specialized GEMM (`hopper_warp_specialized_gemm`). This task asks for a TF32 GEMM on Hopper using warp specialization. The naive baseline is a per-thread loop with 16×16 blocks. The expert CUTLASS 3.0 reference uses a persistent kernel with producer/consumer warp groups: producer warps issue TMA loads into shared memory via `cp.async`, while consumer warps execute WGMMA instructions on shared-memory operands. Matching this reference requires understanding Hopper's asynchronous execution model, TMA descriptor setup, and warp-group synchronization — none of which appear in pre-Hopper CUDA programming.

A.2 Class 2: Module-Level Tasks (119 tasks)

Structure. Each Class 2 task directory contains:

- `def.py` — Python definition specifying `FUNCTION_SIGNATURE`, `SCALAR_ARGS`, `TOLERANCES`, `DESCRIPTION`, and functions `get_inputs()`, `get_outputs()`, `reference_fn()`.

- `reference.cu` — expert CUDA reference calling into the original library (FlashAttention, cuFFT, etc.) via `extern "C"` linkage.
- `task.yaml` — metadata including anti-cheat blocked patterns (e.g., `#include "flash_attn"` is blocked).

The evaluation pipeline loads `def.py`, generates random inputs via `get_inputs()`, compiles both reference and solution as pybind11 modules via `torch.utils.cpp_extension.load()`, checks correctness via `torch.allclose` (3 trials), and measures performance via CUDA event timing with L2 cache clearing.

Task families. Class 2 tasks are organized into five kernel families:

- **Flash Attention (FA2/FA3):** Forward and backward passes across head dimensions (64, 96, 128, 192, 256) and precisions (FP16, BF16), with causal and non-causal variants. General tasks use FA2 (SM80); Hopper tasks use FA3 (SM90) with forward, backward, and split-forward variants.
- **LayerNorm:** Forward and backward passes, plus parallel variants, across 14 hidden dimensions (256–8192). Reference from flash-attention's layer_norm kernels.
- **FFT:** Complex-to-complex (C2C) and real-to-complex (R2C) transforms in 1D/2D/3D, across sizes (2^{10}–2^{18}) and precisions (FP32, FP64). Reference from VkFFT.
- **SageAttention:** Quantized attention with INT8 QK and FP16 PV (Hopper, SM90), and FP4 BlockScaled attention with NVFP4 MMA (Blackwell, SM120).
- **ThunderKittens:** Tile-primitive kernels including MHA, linear attention, Mamba2, and Hedgehog variants (Hopper/Blackwell).

Class 2 contains 43 general tasks, 64 Hopper tasks, and 12 Blackwell tasks.

Case study: Flash Attention Backward (`flash_attn_bwd_hdim128_bf16`). This task requires implementing the FA2 backward pass for BF16 with head dimension 128. Given upstream gradient dO and saved forward tensors (Q, K, V, O, lse), the kernel must compute dQ, dK, dV without materializing the full $S \times S$ attention matrix. The tiled algorithm iterates over K/V tiles nested with Q tiles, recomputing attention probabilities from saved LSE values. The expert FlashAttention reference achieves this with a multi-pass tiled kernel that fuses dQ, dK, dV accumulation. Tolerance is 5% (atol and rtol) due to BF16 numerical error in the backward pass. Matching the reference requires understanding online softmax gradients, tiled recomputation, and shared-memory staging for the double loop over Q and K/V tiles.

Case study: SageAttention INT8 (`sageattn_qk_int8_pv_fp16_hdim128`). This Hopper task implements quantized attention where Q and K are pre-quantized to INT8 with per-warp scale factors, while V remains in FP16. The kernel computes: (1) $S_{\text{int32}} = Q_{\text{int8}} \cdot K_{\text{int8}}^T$ using INT8 tensor core MMA, (2) dequantize $S_{\text{float}} = S_{\text{int32}} \cdot q_{\text{scale}} \cdot k_{\text{scale}}$, (3) softmax, (4) $O = P \cdot V$ using FP16 MMA. The challenge is coordinating two different MMA datapaths (INT8 for QK, FP16 for PV) within a single fused kernel while managing per-warp quantization scales across shared memory.

Case study: 1D FFT (`fft_c2c_1d_1024_fp32`). This task requires a 1D complex-to-complex FFT of size 1024 batched over 1024 independent transforms. The reference uses cuFFT, a runtime code-generated FFT library that emits architecture-specific kernels. Anti-cheat rules block both `cufftExec` and `VkFFTAppend` calls. The LLM must implement the Cooley–Tukey butterfly from scratch, choosing radix decomposition, shared-memory bank-conflict avoidance, and thread-to-element mapping without access to any FFT library.

A.3 Class 3: Application-Level Tasks (10 tasks)

Class 3 tasks are drawn from complete application codebases. The LLM optimizes all kernel files in the application; correctness is verified via application-level checks (loss convergence, output matching, error bounds) rather than tensor comparison alone. A model must therefore produce architecture-appropriate optimizations for the same application codebase under specific hardware profiles. All current Class 3 evaluations are run on RTX PRO 6000.

Task	Domain	Description	Semantic constraint	Files
`cuszp`	Lossy compression	Error-bounded GPU compression/decompression across 1D/2D/3D and FP32/FP64	Every decompressed value within prescribed error bound	6
`exachem`	Quantum chemistry	CCSD(T) perturbative triples with FP64 tensor cores	Energy within 10^{-6} relative tolerance	3
`gpumd`	Molecular dynamics	End-to-end MD across Lennard-Jones, Tersoff, and Coulomb/Ewald force models	Energy drift < 0.05 over simulation	1
`icicle_zk`	ZK cryptography	NTT and MSM on BN254 for zero-knowledge proofs	Exact match against CPU reference	1 dir
`liberator`	Graph processing	BFS, CC, SSSP, PageRank on Friendster graph under GPU memory limits	Correct output at all memory limits	1
`llmc`	LLM training	GPT-2 774M training pipeline (attention, layernorm, GELU, AdamW)	Loss convergence: final loss < 8.0	8
`mgg_agnn`	GNN training	Attention-based GNN with cosine-similarity edge weights	Training accuracy convergence	1
`mgg_gcn`	Multi-GPU GNN	4-GPU GCN with NVSHMEM communication-computation overlap	Correct aggregation across GPUs	1
`parafrost_sat`	SAT solving	GPU-accelerated CDCL SAT solver with inprocessing	Correct SAT/UNSAT verdict	13
`tcgnn_gcn`	GNN training	Tensor Core SpMM for GCN neighbor aggregation	Training accuracy convergence	1

Table 5: Class 3 application-level tasks. "Files" indicates the number of solution files the LLM must optimize.

Task overview. Table 5 summarizes all 10 Class 3 tasks.

Case study: GPT-2 Training (`llmc`). This task optimizes end-to-end GPT-2 774M training. The LLM receives 8 kernel modules: attention (permute, causal softmax, unpermute), layernorm (forward, fused residual+layernorm), GELU, token/position embedding, fused cross-entropy classifier, AdamW optimizer, gradient norm, and cuBLAS matmul wrappers. Training runs 20 steps on TinyShakespeare with BF16 precision. Correctness requires that loss decreases monotonically and the final loss is below 8.0. Performance is total training time. Unlike Class 2 tasks where each kernel is evaluated independently, here the LLM must co-optimize 8 interacting kernel modules under a global training correctness constraint.

Case study: Error-Bounded Compression (`cuszp`). This task optimizes the cuSZp GPU lossy compressor across 6 variants (1D/2D/3D $\times$ FP32/FP64) at three error-bound levels (relative error 10^{-2}, 10^{-3}, 10^{-4}). Correctness is not just "output matches reference" but a hard guarantee: every decompressed value must lie within the prescribed error bound of the original. The test suite runs 54 configurations (6 variants $\times$ 3 modes $\times$ 3 error bounds) on 2 GB datasets. A kernel that is fast but violates even one error bound fails the entire configuration.

Case study: GPU SAT Solver (`parafrost_sat`). This task provides 13 kernel and support files from the ParaFROST SAT solver, implementing GPU-accelerated inprocessing for conflict-driven clause learning. The GPU kernels handle occurrence table construction, variable elimination (3-phase: candidate computation, prefix scan, resolvent generation), clause subsumption, blocked clause elimination, and memory recycling. Correctness is binary: the solver must produce the correct SAT/UNSAT verdict on each test instance. This is one of the most structurally complex tasks in the benchmark, requiring the LLM to reason about multi-kernel dataflow, dynamic memory management, and formal correctness of clause-learning invariants.

A.4 Class 4: Unsolved Challenge Tasks (3 tasks)

Class 4 tasks target problems where strong implementations exist in higher-level abstractions but no equivalent hand-written CUDA has been demonstrated. These are reported as a separate unsolved challenge track.

FlashAttention-4 Forward (`flash_attn4_fwd`). This task asks for a hand-written CUDA implementation of the FA4 forward pass on Blackwell (SM100), without using the CuTe DSL. The FA4 algorithm introduces four key innovations over FA3: (1) conditional softmax rescaling that skips $\sim 90\%$ of rescaling operations by only triggering when the row-max exceeds the previous maximum by $\tau = \log_2(256) = 8.0$; (2) software exponential emulation using a degree-3 polynomial on FMA units, reserving the hardware EX2 unit for the ~ 10–25% of entries needing higher precision; (3) 5-way warp specialization with concurrent load, MMA, softmax, correction, and epilogue warps; and (4) Blackwell-specific hardware including TMEM (256 KB programmer-managed L1), `tcgen05.mma` (fully asynchronous 128×128 MMA), and 2-CTA MMA mode.

The task provides the full FA4 paper, the official CuTe DSL reference implementation (2842 lines), and helper modules for TMA, WGMMA, and TMEM. The LLM must translate these abstractions into raw CUDA/PTX. Evaluation tests multiple configurations (head dimensions 64/128/256, sequence lengths 1K–32K, causal/non-causal, GQA) and reports TFLOPS against the CuTe baseline. This task requires SM100 hardware.

FlashAttention-4 Backward (`flash_attn4_bwd`). This task asks for a hand-written CUDA/PTX implementation of the FA4 backward pass on Blackwell (SM100), without using CuTe DSL or existing FlashAttention backward kernels. Given Q, K, V, saved forward output O, upstream gradient dO, and saved log-sum-exp values, the implementation must compute dQ, dK, and dV for BF16 attention with FP32 accumulation.

The task requires recomputing tiled attention scores without materializing the full attention matrix, reconstructing softmax probabilities from saved LSE, computing $dV = P^T dO$ and $dS = P \cdot (dP - \mathrm{rowsum}(dO \cdot O))$, and accumulating dQ and dK through tiled QK recomputation. It must support causal and non-causal masking, GQA, head dimensions 64/128/256, and sequence lengths up to 32K in the full challenge setting. Correctness is checked against PyTorch SDPA autograd with numerical tolerances, and performance is measured as total backward time. This task requires SM100 features including TMA, `tcgen05.mma`, TMEM accumulators, and warp-specialized producer/consumer pipelines.

Groth16 ZK Prover (`groth16_prover`). This task asks for a complete GPU Groth16 zero-knowledge proof prover in hand-written CUDA. Groth16 is the most widely deployed zkSNARK (used in Zcash, Filecoin, many L2 rollups). The proving algorithm requires: (1) 6–7 NTTs (forward and inverse) over the BN254 scalar field on domains of size 2^{20}–2^{26}; (2) 5 multi-scalar multiplications on BN254 G1 and G2 curves; and (3) full 254-bit modular and elliptic curve arithmetic (Montgomery multiplication, projective point addition/doubling, quadratic extension field operations for G2).

No hand-written CUDA Groth16 prover exists; all current implementations (Icicle, cuZK, bellman) rely on library primitives. The LLM must implement NTT, MSM (Pippenger bucket method), and BN254 arithmetic from scratch, then orchestrate them into the full proving pipeline. Correctness is verified by checking that the generated proof passes the Groth16 pairing-based verification equation. Performance is compared against the Icicle GPU prover. This task requires SM80+ hardware.

B Task Metadata and Prompt Examples

This section shows the concrete artifacts used to instantiate a task. The example is adapted from the Class 1 `gemm_softmax` task and omits only repository-local path details that are not shown to the model. The task name is shown without the internal numeric prefix used in early development.

Example `task.yaml` metadata. The metadata file records the target GPU requirement, runner backend, build and execution commands, correctness policy, performance parser, provenance, and anti-cheating checks. The benchmark runner reads this file before constructing a model prompt or executing a generated solution.

```
schema_version: 1
task_id: class1/general/gemm_softmax
name: gemm_softmax
task_class: 1
domain: ml
tags:
  - gemm
```

```
  - softmax
  - fusion
hardware:
  min_sm: 80
  tested_sms: [80]
  required_features: []
runner:
  backend: class1_make
  workdir: tasks/class1/general/gemm_softmax
  timeout_sec: 180
  env:
    KH_BENCHMARK: "20"
  solution_file: solution.h
build:
  cmd: make test
  clean_cmd: make clean
execute:
  cmd: ./test
  success:
    exit_code: 0
    stdout_regex: Passed
correctness:
  mode: stdout_or_exit
  tolerances:
    atol: 0.01
    rtol: 0.01
performance:
  enabled: true
  parser:
    kernel_time_ms_regex: "Kernel time:\\s*([0-9.]+)\\s*ms"
    ref_time_ms_regex: "Ref time:\\s*([0-9.]+)\\s*ms"
  metric: speedup_ref_over_sol
source:
  repo: NVIDIA/cutlass
  files:
    - examples/35_gemm_softmax
  commit: "<recorded commit hash>"
  license: BSD-3-Clause
anti_cheat:
  blocked_patterns: []
  required_patterns:
    - "__global__\\s+void"
    - "<<<.*>>>"
```

Example one-shot prompt. The one-shot prompt combines the task description, hardware metadata, solution interface, and build and scoring rules. When few-shot exemplars are enabled, they are appended after the task-specific block shown below. The prompt does not expose the expert implementation that is used for scoring.

```
System:
You are an expert CUDA programmer. Write a correct and high-performance CUDA
implementation for the requested task. Return only the replacement source file.

User:
Task: gemm_softmax
Class: single-kernel CUDA optimization
Target GPU: NVIDIA SM80 or newer
Source file to replace: solution.h

Goal:
Implement a CUDA solution for a fused batched GEMM followed by row-wise softmax.
The evaluator will compile your solution with `make test`, run `./test`, check
correctness against an expert CUDA reference, and parse runtime from stdout.

Interface:
  You must provide the function and kernel definitions expected by solution.h.
  The solution must include at least one __global__ kernel and launch it with
  CUDA <<<...>>> syntax. Do not change the public function signature.

Inputs and semantics:
  For each batch, compute D = alpha * A * B + beta * C, then compute
  Softmax[m, n] = exp(D[m, n] - max(D[m, :])) /
                  sum_j exp(D[m, j] - max(D[m, :])).
  A is row-major, B is column-major, and the output softmax tensor is row-major.

Hardware and optimization requirements:
  The target GPU supports SM80 features. A strong solution should exploit
  tiled matrix multiplication, shared-memory reuse, numerically stable softmax
  reduction, and fusion between the GEMM epilogue and softmax computation.
```

```
Correctness and scoring:
  The program must print "Passed" under the benchmark harness. Correctness is
  checked at multiple problem sizes with atol=1e-2 and rtol=1e-2. Performance
  is scored as expert_runtime / generated_runtime, using the harness-reported
  "Ref time:" and "Kernel time:" values. Incorrect or non-compiling solutions
  receive zero performance credit.

Restrictions:
  Do not call cuBLAS, CUTLASS, Thrust, or any external GEMM/softmax library.
  Do not hard-code outputs, inspect timing loops, bypass the test harness, or
  depend on fixed pointer identities. The solution must be general for the
  problem sizes used by the evaluator.

Return:
  Only the complete contents of solution.h.
```

C Detailed Architecture Stress-Test Results

Table 6 gives the full breakdown for the architecture-specific Hopper and Blackwell stress tests summarized in the main text. We report pass count, pass rate, mean expert-relative speed, and `fast_p`(1.05) after applying the benchmark's validity checks. The Hopper rows combine the architecture-specific Class 1 and Class 2 slices evaluated on H200.

On Hopper Class 1, refinement is the most effective strategy: GPT-5.4 reaches 57% pass rate under both refine@10 and tool augmentation, while Claude Opus 4.6 peaks at 48% under tool augmentation. Claude Opus 4.6 reaches the highest Class 1 `fast_p`(1.05), with 14% under refine@10 and 5% under pass@3. On Hopper Class 2, GPT-5.4 is stronger under single-sample and few-sample settings (33% pass@1, 34% pass@3), whereas Claude Opus 4.6 benefits substantially from tool augmentation, reaching 47% pass rate with a mean speed of 0.323× and `fast_p`(1.05) of 13%. Across both Hopper slices, Qwen3.5-122B-A10B and Kevin32B solve no tasks.

The Blackwell rows separate the 22-task Class 1 SM100a subset from the 12-task Class 2 subset. On Blackwell Class 1, Claude Opus 4.6 obtains the highest pass rate, reaching 45% under tool augmentation, while GPT-5.4 reaches 41% under refinement. However, both remain far below the expert references: all `fast_p`(1.05) values are 0, and the best mean speed is only $0.053\times$. The Blackwell Class 2 subset is fully unsolved across all models.

D Reference Quality and Anti-Cheating Controls

CUDAHercules uses the following controls to reduce contamination, library bypass, direct reference reuse, and benchmark-specific shortcut behavior.

- **Source provenance and prompt construction.** Each task records source repository, file, commit, and license metadata when available. Prompts are built from task descriptions, interfaces, metadata, solution templates, and selected context files, rather than by directly handing the model the expert kernel as the target solution.
- **Static anti-cheat checks.** Task metadata and global rules block disallowed library calls, reference includes, shortcut APIs, or required-kernel violations before a solution is scored.
- **Workspace and reference separation.** Evaluators run submitted code in temporary workspaces and separate scoring references or private application baselines from the generated solution path, so a candidate is measured through the task harness rather than by directly reusing the scorer.
- **Post-evaluation hacking audit.** Passing solutions can be audited for benchmark-specific shortcut behavior such as pointer-identity caches, reference re-imports, hard-coded outputs, timing-loop manipulation, or suspicious speedups; audited hacking cases are counted as failures in the reported results.

Target	Model	Setting	Tasks	Pass	Pass rate	Mean speed	fast_p(1.05)
Hopper H200 C1	GPT-5.4	pass@1	21	3	14%	0.012×	0%
		pass@3	21	2	10%	0.004×	0%
		refine@10	21	12	57%	0.053×	0%
		tool-augmented	21	12	57%	0.081×	0%
	Claude Opus 4.6	pass@1	21	5	24%	0.045×	0%
		pass@3	21	5	24%	0.076×	5%
		refine@10	21	8	38%	0.090×	14%
		tool-augmented	21	10	48%	0.031×	0%
	Qwen3.5-122B-A10B	all evaluated	21	0	0%	0.000×	0%
	Kevin32B	all evaluated	21	0	0%	0.000×	0%
Hopper H200 C2	GPT-5.4	pass@1	64	21	33%	0.196×	6%
		pass@3	64	22	34%	0.166×	6%
		refine@10	64	15	23%	0.122×	5%
		tool-augmented	64	16	25%	0.109×	5%
	Claude Opus 4.6	pass@1	64	11	17%	0.053×	0%
		pass@3	64	20	31%	0.211×	11%
		refine@10	64	7	11%	0.098×	2%
		tool-augmented	64	30	47%	0.323×	13%
	Qwen3.5-122B-A10B	all evaluated	64	0	0%	0.000×	0%
	Kevin32B	all evaluated	64	0	0%	0.000×	0%
Blackwell B200 C1	GPT-5.4	pass@1	22	0	0%	0.000×	0%
		pass@3	22	1	5%	0.005×	0%
		refine@10	22	9	41%	0.052×	0%
		tool-augmented	22	1	5%	0.009×	0%
	Claude Opus 4.6	pass@1	22	9	41%	0.015×	0%
		pass@3	22	9	41%	0.018×	0%
		refine@10	22	9	41%	0.043×	0%
		tool-augmented	22	10	45%	0.053×	0%
	Qwen3.5-122B-A10B	all evaluated	22	0	0%	0.000×	0%
	Kevin32B	all evaluated	22	0	0%	0.000×	0%
Blackwell B200 C2	GPT-5.4	all evaluated	12	0	0%	0.000×	0%
	Claude Opus 4.6	all evaluated	12	0	0%	0.000×	0%
	Qwen3.5-122B-A10B	all evaluated	12	0	0%	0.000×	0%
	Kevin32B	all evaluated	12	0	0%	0.000×	0%

Table 6: Hopper and Blackwell architecture stress-test results. Hopper C1 and C2 report the Class 1 and Class 2 architecture-specific slices separately. Blackwell C1 reports the 22-task SM100a. Blackwell C2 reports the 12-task SM100a subset.