KERNELGENBENCH: CAN LLMS AND AGENTS WRITE EFFICIENT KERNELS ACROSS OPERATOR SOURCES AND HARDWARE PLATFORMS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern AI systems depend on specialized accelerator kernels, whose development is complicated by increasingly diverse operators and hardware. LLMs and agentic systems promise to automate this work, but existing evaluations do not show how performance varies across operator sources and hardware platforms, or what that variation costs. We present **KernelGenBench**, the first unified multi-source $\times$ multi-chip infrastructure for evaluating LLM- and agent-generated Triton kernels. Unlike benchmarks that require platform-specific target languages, KernelGenBench uses Triton as a common programming target across six hardware platforms. We report two controlled analytical views: **KernelGenBench-MS (Multi-Source)** covers 210 operators from PyTorch ATen, production vLLM operators, and proprietary cuBLAS routines, while **KernelGenBench-MC (Multi-Chip)** evaluates a semantically stable 110-operator subset across six hardware platforms. Our evaluation consumed over 15 billion tokens. More interactive agentic workflows achieved higher overall correctness, but no method dominated across sources and platforms: vLLM posed the strongest correctness challenge, cuBLAS set the highest performance ceiling, and AutoKernel with GLM-5.0 fell from 87% on NVIDIA to 25% on Iluvatar CoreX. Higher correctness came at substantial cost: specialized agents averaged 4.99 million tokens per successful operator, rising to 6.25 million for CUDA Optimized Skill. The results establish operator source, hardware platform, and generation workflow as distinct dimensions of kernel-generation capability, and show that success in a familiar source–hardware setting is not a reliable proxy for deployment readiness.

1 INTRODUCTION

As AI models grow, their development and deployment increasingly depend on the efficient use of computing infrastructure. Compute capacity and memory bandwidth continue to grow across increasingly diverse hardware platforms, including GPUs, TPUs, and other accelerators (Stanford Institute for Human-Centered Artificial Intelligence, 2026; Emberson, 2026). Adding more compute capacity, however, does not by itself improve application performance. Efficient kernels translate this capacity into faster tensor operations by exploiting hardware parallelism and memory hierarchies. Writing efficient kernels still requires expertise in low-level programming, hardware architecture, and performance optimization. Recent work uses LLMs and agents to automate parts of this process through iterative generation, profiling, and compiler-guided optimization (Liu et al., 2026; Wang et al., 2025; Yang et al., 2026b; Shui et al., 2026; Ye et al., 2026).

Can kernel-generation systems transfer across operator sources and hardware platforms? Recent model architectures make this question more pressing. DeepSeek V4 introduces compressed sparse attention, which requires model-specific kernels beyond standard framework operators (DeepSeek-AI, 2026). Production inference frameworks such as vLLM and SGLang likewise rely on specialized CUDA kernels in performance-critical paths (Kwon et al., 2023; Zheng et al., 2024). These models must also be trained and served on heterogeneous accelerators. A PyTorch-only evaluation therefore covers only part of the problem: kernel-generation systems must implement framework-native semantics and production CUDA operators, compete with highly optimized libraries, and transfer across hardware platforms. Recent benchmarks extend task taxonomies, agentic

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

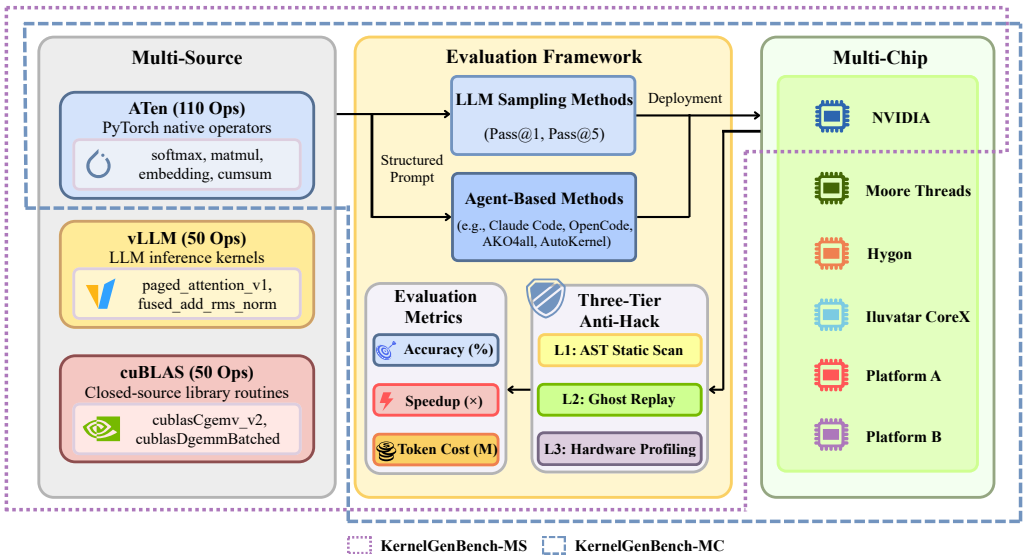


Figure 1: Overview of KernelGenBench. The shared infrastructure supports multi-source × multi-chip evaluation; the purple and blue dashed boxes indicate the MS and MC controlled analytical views reported in this paper.

workflows, verification, production realism, and hardware-specific evaluation (Wang et al., 2026; Guan et al., 2026; Zhang et al., 2026b; Yang et al., 2026b; Huang et al., 2026; Zhang et al., 2026a). Existing benchmarks have broadened either operator provenance or hardware coverage, but none unifies distinct operator-reference ecosystems with controlled evaluation across heterogeneous accelerator software stacks under a common kernel-generation and verification framework. Such unification requires more than adding device targets: it must integrate distinct reference implementations, compilers, runtimes, and backend APIs through a common task interface and an extensible platform-adapter layer. Performance on NVIDIA may not transfer to another platform, where the hardware architecture and software stack differ substantially.

We introduce **KernelGenBench**, the first unified multi-source × multi-chip benchmark infrastructure: a single portable Triton target used on every platform, unlike benchmarks that require platform-specific target languages. To distinguish source-conditioned behavior from hardware-stack transfer, we report two complementary controlled views. Triton's programmable block-level abstraction supports model-specific fusion, data movement, and scheduling, allowing generated kernels to be evaluated against framework-native operators, production CUDA kernels, and tuned libraries rather than only compositions of PyTorch APIs. Its compiler-based portability provides a common kernel language across hardware platforms and enables controlled tests of performance portability (Tillet et al., 2019; Ansel et al., 2024). We instantiate these views as:

- **KernelGenBench-MS (Multi-Source):** Evaluates kernel generation across 210 operators from PyTorch ATen (Paszke et al., 2019), vLLM (Kwon et al., 2023), and cuBLAS.
- **KernelGenBench-MC (Multi-Chip):** Assesses performance portability across six hardware platforms, including NVIDIA, Moore Threads, Hygon, Iluvatar CoreX, and two anonymous platforms, using a 110-operator ATen subset.

For execution-grounded evaluation, we design a framework with a distributed sandbox infrastructure and a three-tier anti-hack mechanism that detects and filters shortcut implementations. We also introduce a multi-dimensional evaluation protocol that jointly measures functional correctness, execution speedup, and agentic cost efficiency.

Our evaluation consumed over 15 billion tokens and produced three main findings. (1) Agent-based methods generally outperform pure LLM sampling, though not universally: the strongest sampling configuration exceeds some agentic ones. Across operator sources, ATen is the most tractable; vLLM is the hardest to generate correctly but offers substantial speedup potential among successfully solved

operators; and cuBLAS is the strongest performance-ceiling stress test. (2) Performance varies substantially across hardware platforms. Some model–method combinations degrade sharply outside NVIDIA: AutoKernel with GLM-5.0 falls from 87% accuracy on NVIDIA to 25% on Iluvatar CoreX. (3) More interactive agentic workflows achieve higher overall correctness at high cost. Specialized methods average 4.99 million tokens per successful operator, rising to 6.25 million for CUDA Optimized Skill. KernelGenBench quantifies how these effects vary across sources and platforms and what they cost.

Our contributions are threefold:

- We build the first unified multi-source $\times$ multi-chip benchmark infrastructure, coupling three operator sources with platform-specific runtime and library adapters. We report MS across 210 operators on NVIDIA and controlled MC evaluation across six hardware platforms using a common 110-operator task set (Section 3).
- We develop a common evaluation framework combining combinatorial correctness tests, performance measurement, cost accounting, and layered anti-hack verification.
- We evaluate LLM sampling and agent-based methods at scale and identify distinct source-level capability boundaries, cross-platform portability gaps, and the benefits and costs of iterative agentic workflows.

2 Related Work

2.1 LLM-Driven Kernel Generation

Automated kernel generation combines model training with inference-time search (Yu et al., 2026). Prior work explores supervised fine-tuning, reinforcement learning, execution feedback, evolutionary search, and runtime-guided refinement (Fisches et al., 2025; Baronio et al., 2025; Yang et al., 2026a; Dai et al., 2026; Chen et al., 2026; Hammond et al., 2025). More recent systems explore AMD hardware platforms, architecture-specific PTX, profiling-guided optimization, agent frameworks, and compiler interfaces for agents (Wang et al., 2025; Zhang et al., 2026a; Yang et al., 2026b; Shui et al., 2026; Ye et al., 2026). These evaluations measure the complete kernel-generation system, including the model, agent framework, and compiler tools.

2.2 Limitations of Existing Benchmarks

KernelBench and TritonBench established execution-based evaluation for generated kernels. Later benchmarks expand task coverage and evaluate agent workflows, hidden tests, memory use, and performance across hardware platforms (Ouyang et al., 2025; PyTorch Team, 2025; Wang et al., 2026; Guan et al., 2026; Zhang et al., 2026b). Other benchmarks derive workloads from inference traces or framework pull requests to evaluate kernels in production settings (Saroufim et al., 2025; Yang et al., 2026b; Huang et al., 2026). KernelGenBench compares three operator sources with distinct reference implementations: ATen, vLLM, and cuBLAS. All generated kernels replace the corresponding reference through a common interface and are evaluated through a shared verification pipeline. The evaluation also includes layered anti-hack checks and agentic cost.

Most evaluations still focus on NVIDIA or a specific hardware architecture (Xing et al., 2026; Lin et al., 2026; Zhang et al., 2026a). Atrex-Bench draws operators from multiple production inference frameworks but uses a common PyTorch reference interface (Yang et al., 2026b). MultiKernel-Bench spans NVIDIA GPUs, Huawei NPUs, and Google TPUs using separate platform-specific languages (Wen et al., 2025). KernelGenBench uses Triton as the common target language across six hardware platforms. Other benchmarks include AMD hardware or comparisons across hardware platforms (Wang et al., 2025; Yang et al., 2026b; Wang et al., 2026). KernelGenBench applies a shared verification pipeline to the same 110 operators across six platforms to compare correctness, speedup, and cost. CUDA and CUTLASS remain useful for optimizing kernels on NVIDIA. We use Triton to evaluate portability across hardware platforms.

3 KERNELGENBENCH

KernelGenBench is a unified multi-source × multi-chip benchmark infrastructure. Figure 1 illustrates the two controlled views: KernelGenBench-MS and KernelGenBench-MC.

3.1 KERNELGENBENCH-MS

Multi-source evaluation compares operators that differ in source, abstraction level, and reference implementation. KernelGenBench-MS evaluates 210 operators on NVIDIA from three sources: PyTorch ATen, production inference operators from vLLM, and the vendor-optimized cuBLAS library. These sources test three capabilities: implementing standard framework operators, recovering specialized operator behavior from API specifications, and matching the performance of optimized libraries. A PyTorch-only evaluation does not separate these capabilities. Prompts provide API details but do not expose the underlying C++/CUDA reference implementations. Generated Triton kernels must satisfy the specifications and directly replace the reference operators (FlagOS contributors, 2024). Appendix A details the prompts for each source; Appendix B lists all operators.

Source 1: PyTorch ATen Operators (110 problems). From the 900+ `torch.ops.aten` APIs, we select the 50 most frequent operators in training traces from 2,907 open-source models and uniformly sample 60 long-tail operators. Each operator API is an independent problem (e.g., `torch.add` and the in-place `torch.add_` are two separate problems), requiring the model to implement all overload variants. Prompts use the official `FunctionSchema` and docstrings; the reference calls the corresponding native ATen API.

Source 2: vLLM Operators (50 problems). Efficient inference relies on specialized kernels that need adaptation to other hardware platforms. We select vLLM operators with standalone CUDA implementations, including attention mechanisms and KV-cache management.[1] The prompt includes the Python reference wrapper: its signature, argument order, in-place behavior, and custom-operator call. It excludes the underlying CUDA/C++ implementation. This tests whether a system can recover the required behavior and write a Triton replacement without copying the reference kernel.

Source 3: cuBLAS Operators (50 problems). A correct kernel may still be much slower than an expert-optimized implementation. We therefore use the closed-source cuBLAS library as a performance reference.[2] The reference calls the library directly, without high-level wrappers. We identify the 10 most frequently called cuBLAS routines from profiling traces and include variants across precision and batching modes (Appendix C). We also sample BLAS routines with different functions to avoid concentrating on one operator family. Generated kernels must compete with these optimized implementations without calling external libraries.

Across all three sources, prompts expose only the public API-level specification and never the underlying reference implementation. They differ in algorithmic family and in the strength of the reference they are measured against, which is the axis the benchmark varies (Appendix A).

3.2 KERNELGENBENCH-MC

KernelGenBench supports the three operator sources through a common pipeline with platform-specific runtime and library adapters. Evaluating every source–platform combination across all methods would require prohibitive time and cost. We therefore report two controlled views. MS evaluates all 210 operators on NVIDIA to compare generation and optimization across sources. MC evaluates the same 110 ATen operators across six hardware platforms to compare correctness and performance across platforms. These views guide our analysis without limiting the infrastructure. We additionally evaluate the complete 50-task vendor-library subset on all six platforms to verify operation across both sources and platforms. Appendix Table 13 reports the results, and Table 12 lists the reference adapters.

We do not combine all source–platform results into one leaderboard. Production inference frameworks and vendor-library references differ across platforms in coverage, implementation, and performance. The same operator may use a native implementation, a vendor adaptation, or a compatibility layer;

[1] All vLLM evaluations use v0.13.0 in this paper.

[2] All cuBLAS evaluations use v12.4 in this paper.

some platforms have no local implementation. Combining these cases would mix operator difficulty with differences in software support.

The framework supports six hardware platforms: NVIDIA, Moore Threads, Hygon, Iluvatar CoreX, and two platforms that remain anonymous under commercial confidentiality requirements. KernelGenBench-MC uses Triton as the common target language because CUDA and CUTLASS target NVIDIA hardware. The pipeline detects the hardware platform, adds platform-specific constraints to the prompt, and configures platform-specific runtime and verification settings (Appendices D and E). MC therefore measures the ability to write a Triton kernel for each target platform, evaluated against that platform's local reference; differences reflect how well a stack is supported and how familiar a generation system is with it.

3.3 Evaluation Framework

All configurations follow the same evaluation workflow. Appendix E describes the distributed sandbox that runs the tests.

Kernel Generation. All methods receive the same operator specification, test suite, applicable anti-hack checks, and 30-minute time budget per task. We evaluate (1) *LLM Sampling Methods*, which generate one or multiple independent samples (Pass@1, Pass@5) (Chen et al., 2021) without execution feedback; and (2) *Agent-Based Methods*, which use execution feedback to revise generated kernels. The latter include standard agent frameworks and kernel-specialized agents. Prompts, skills, profilers, and tool use affect the results (Guan et al., 2026; Shui et al., 2026). We therefore retain each method's internal workflow while standardizing task conditions and final verification (pseudocode in Appendix F).

Verification. All generated kernels use the same verification pipeline. Instead of testing a single fixed shape (Ouyang et al., 2025), we construct k_i test cases per operator by combining semantic parameters (e.g., reduction axis or transpose mode) with shapes, data types, and memory layouts (Appendix G). This complements hidden-test evaluation (Zhang et al., 2026b) and end-to-end testing within inference frameworks (Huang et al., 2026). For ATen, we register kernels through `torch.library` to check that they can replace the reference operators within PyTorch. A kernel is correct only if it passes numerical validation on *all* k_i test cases.

Anti-Hack Architecture. We use three layers of checks against **benchmark hacking** (Woo et al., 2025; Zhang et al., 2026b), such as calling existing backend implementations instead of generated Triton kernels:

- *L1: AST Static Scan*: Parses the generated abstract syntax tree to block blacklisted calls (e.g., `torch.ops.aten.*`, `import vllm`, and `ctypes`) and dynamic injection bypasses.
- *L2: Ghost Replay*: Runs the implementation normally, then repeats execution with Triton kernel execution disabled. Unchanged outputs indicate that the implementation did not depend on the Triton kernel and trigger a cheating flag.
- *L3: Hardware Profiling*: Uses `torch.profiler` to check execution traces for Triton kernel signatures. This layer runs on NVIDIA; other platforms use L1 and L2 because equivalent profiling tools are unavailable.

These checks are empirically necessary: during evaluation, they intercepted 13 Pass@1 and 30 Pass@5 shortcut submissions, including direct target-API reuse and hybrid framework fallback; Appendix H reports representative cases and interception breakdowns.

Evaluation Metrics. Accuracy is the clean pass rate: an operator counts as successful only when all k_i test cases pass numerical validation and all anti-hack layers available on that platform. Successful-set speedup measures the performance of correct implementations; cost measures the resources spent obtaining them. For performance, we compute a two-level geometric mean (Standard Performance Evaluation Corporation (SPEC), 2017; Fleming & Wallace, 1986): first across the k_i test cases to obtain operator-level speedup S_i, then across successfully solved operators (definition and rationale in Appendix I). Reported speedup is a successful-set statistic; Section 4.4 compares methods on matched intersections of successful operators. Where comparable accounting is available, we report total tokens, tokens per success (total tokens divided by passed operators), and cumulative solving time (sum of per-operator solving durations, parallelism-agnostic).

Table 1: NVIDIA A100 evaluation across 210 operators from three sources. Spd is computed on each method's own correctly solved operator set and is not a matched-set cross-method ranking; overall Spd is the success-count-weighted geometric mean of the three source columns.

Method & Setup	Overall (210)		ATen (110)		vLLM (50)		cuBLAS (50)	
	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)
LLM Sampling Methods								
Pass@1 (Opus-4.6)	41	0.70	39	0.90	20	0.76	68	0.49
Pass@1 (GLM-5.0)	21	0.68	21	0.49	24	1.24	20	0.73
Pass@1 (Qwen3.5-27b)	7	0.85	8	0.83	2	**2.05**	8	0.71
Pass@1 (MiniMax M-2.5)	2	0.88	4	0.88	0	–	0	–
Pass@5 (Opus-4.6)	57	0.68	62	0.79	28	0.71	74	0.49
Pass@5 (GLM-5.0)	36	0.76	45	0.64	32	1.28	20	0.76
Pass@5 (Qwen3.5-27b)	11	0.87	13	**1.04**	12	0.70	8	0.68
Pass@5 (MiniMax M-2.5)	16	0.86	21	0.76	18	1.27	2	0.46
Vanilla Agentic Frameworks								
Claude Code (Opus-4.6)	87	0.78	92	0.86	68	1.02	94	0.51
Claude Code (GLM-5.0)	67	0.83	72	0.88	52	1.23	72	0.53
Claude Code (Qwen3.5-27b)	62	0.70	80	0.68	38	1.17	48	0.50
Claude Code (MiniMax M-2.5)	49	0.70	69	0.78	26	0.46	26	0.58
OpenCode (Opus-4.6)	84	0.80	90	0.84	58	1.35	96	0.53
OpenCode (GLM-5.0)	70	0.74	87	0.75	46	1.23	54	0.46
OpenCode (Qwen3.5-27b)	53	0.78	58	0.75	44	1.31	52	0.58
OpenCode (MiniMax M-2.5)	41	0.62	50	0.77	26	0.44	36	0.42
Kernel-Specialized Agents								
AKO4all (Opus-4.6)	83	0.97	91	1.00	64	1.62	84	0.61
CUDA Opt. Skill (MiniMax M-2.5)	45	0.75	63	0.81	24	0.92	28	0.45
AutoKernel (GLM-5.0)	71	0.99	87	1.00	42	1.40	66	0.75
AutoKernel (Qwen3.5-27b)	47	**1.01**	69	1.00	16	1.63	30	**0.80**
AutoKernel (MiniMax M-2.5)	43	0.87	66	0.87	20	1.52	16	0.46

4 Experiments and Evaluation

We use KernelGenBench to study how kernel-generation results vary across operator sources and hardware platforms, and how agent frameworks affect accuracy, speedup, and cost.

4.1 Experimental Setup

Following the benchmark design in Section 3, we report the MS view on NVIDIA across 210 operators and the MC view across six hardware platforms. Main-table Pass@1 configurations use `temperature`=0 and Pass@5 uses `temperature`=0.8, with `max_tokens`=16384 and a unified 30-minute wall-clock timeout per operator task. Most agent results are single-trajectory measurements; small accuracy differences are therefore interpreted cautiously. Timing and repeated-run protocols are detailed in Appendices J and K.

4.2 KernelGenBench-MS: Multi-Source Evaluation

Table 1 presents the results; additional NVIDIA results are reported in Appendix L.

Finding 1: Different operator sources test different capabilities. Claude Code (Opus-4.6) achieves the highest overall accuracy at 87%, while AutoKernel (Jaber & Jaber, 2026) (Qwen3.5-27b) reports the highest successful-set overall speedup at 1.01×. Claude Code and AKO4all (TongmingLAIC, 2026) (Opus-4.6) obtain 87% and 83% accuracy, respectively, while their successful-set speedups are 0.78× and 0.97×. The 4 pp accuracy gap is descriptive under single-trajectory evaluation, not a statistically established ordering. Because speedups use different successful sets, they should not alone rank methods. Correctness alone does not imply efficient coverage: on the full 210-task suite, Claude Code (Opus-4.6) solves 87% of operators but exceeds the reference on 35%, whereas AKO4all (TongmingLAIC, 2026) solves 83% and exceeds the reference on 47%. Functional coverage

Table 2: Cross-platform evaluation on 110 ATen operators. Spd is computed on each method's own correctly solved operator set; it is not a matched-set cross-method ranking.

Method & Setup	NVIDIA		Moore Threads		Hygon		Iluvatar CoreX		Platform A		Platform B	
	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)
LLM Sampling Methods												
Pass@1 (Opus-4.6)	39	0.90	44	0.69	37	0.98	38	0.88	46	0.19	38	0.89
Pass@1 (Qwen3.5-27b)	8	0.83	3	1.02	7	0.90	10	1.03	9	0.09	7	0.98
Pass@1 (MiniMax M-2.5)	4	0.88	6	1.33	4	1.05	4	**1.15**	4	0.25	5	0.77
Pass@5 (Opus-4.6)	62	0.79	60	0.74	54	0.92	57	0.83	63	0.15	65	0.68
Pass@5 (Qwen3.5-27b)	13	**1.04**	11	1.10	15	0.72	17	1.02	16	0.18	10	0.99
Pass@5 (MiniMax M-2.5)	21	0.76	15	0.53	12	1.05	9	0.76	17	0.20	8	0.33
Vanilla Agentic Frameworks												
Claude Code (Opus-4.6)	92	0.86	93	0.80	88	0.87	83	0.83	89	0.18	96	0.89
Claude Code (GLM-5.0)	72	0.88	65	0.96	65	0.81	37	0.77	65	0.16	59	0.90
Claude Code (Qwen3.5-27b)	80	0.68	75	0.61	75	0.85	23	0.81	78	0.25	82	0.77
Claude Code (MiniMax M-2.5)	69	0.78	74	0.59	73	0.72	69	0.58	69	0.16	83	0.63
Kernel-Specialized Agents												
AKO4all (Opus-4.6)	91	1.00	88	1.09	88	**1.08**	80	1.07	84	0.30	86	**1.12**
CUDA Opt. Skill (MiniMax M-2.5)	63	0.81	64	0.77	65	0.81	58	0.79	53	0.21	67	0.77
AutoKernel (GLM-5.0)	87	1.00	56	1.01	64	0.99	25	1.01	53	**0.82**	59	1.00
AutoKernel (Qwen3.5-27b)	69	1.00	75	1.03	65	1.00	21	1.01	40	0.37	74	1.04
AutoKernel (MiniMax M-2.5)	66	0.87	71	**1.36**	66	0.99	50	1.02	61	0.66	71	1.04

and optimization effectiveness therefore capture distinct aspects of kernel-generation capability. Detailed fast_p distributions and model-specific generation behaviors are in Appendices M and H.

Correctness and optimization pose different challenges. ATen operators are relatively easy to implement correctly. vLLM operators are harder to implement correctly, but successful kernels offer more room for speedup. cuBLAS operators can achieve high correctness, but their optimized references are harder to outperform. Section 4.4 analyzes these differences across configurations and operators.

4.3 KernelGenBench-MC: Cross-Platform Evaluation

We extend the evaluation of the 110 ATen operators across six hardware platforms. Table 2 presents the full results. MC accuracy is therefore an end-to-end platform success rate, not generated-kernel correctness alone (Appendix D).

Finding 2: Target-language portability does not imply generation-system portability. Using the same language does not guarantee similar results across hardware platforms. Claude Code (Opus-4.6) maintains 83–96% accuracy, but its speedup falls from 0.86× on NVIDIA to 0.18× on Platform A. On Iluvatar CoreX, correctness is a larger problem: Claude Code (Qwen3.5-27b) reaches 23% accuracy, while AutoKernel with GLM-5.0 and Qwen3.5-27b reaches 25% and 21%, respectively. AutoKernel (GLM-5.0), for example, falls from 87% on NVIDIA to 25% on Iluvatar CoreX. Portability must therefore be evaluated for the complete kernel-generation system, including the model, agent framework, compiler, runtime, and backend.

Changing hardware platforms also changes method rankings. Across the 15 configurations shared by all six platforms, Spearman rank correlation with NVIDIA accuracy is 0.87 on Moore Threads, 0.89 on Hygon, 0.66 on Iluvatar CoreX, 0.86 on Platform A, and 0.84 on Platform B. On Iluvatar CoreX, both accuracy and the relative ordering of kernel-generation systems change.

Section 4.5 examines these failures using execution logs.

Figure 2 complements Table 2 with accuracy changes across configurations, followed by speedup and generation cost for a fixed model and agent framework.

Finding 3: Hardware heterogeneity imposes an agentic portability tax. For a fixed task set, model, and agent framework (110 ATen operators with Claude Code Opus-4.6), we call the additional token or cumulative solving-time cost relative to NVIDIA the *agentic portability tax*. The five non-NVIDIA platforms require 1.27–2.06× the tokens and 1.67–2.00× the time; Platform A reaches 173M tokens and 18 hours, versus 84M and 9 hours on NVIDIA. This cost is not captured by final accuracy and speedup: producing correct kernels can require many attempts to resolve compilation

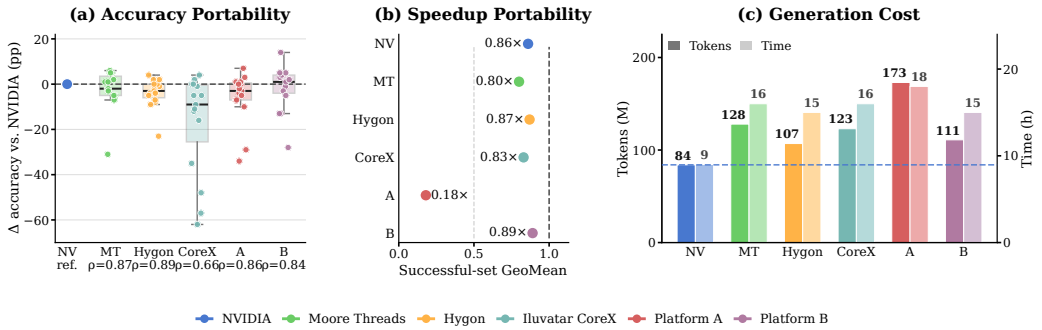


Figure 2: Cross-platform portability across accuracy, speedup, and generation cost. (a) Accuracy changes from NVIDIA across 15 shared configurations. (b) Successful-set speedup for Claude Code (Opus-4.6) on 110 ATen operators. (c) Corresponding token and cumulative solving-time cost.

Table 3: Results by operator source across the 21 configurations in Table 1. Spd is the median of reported successful-set geometric means.

Source	Median Acc. (%)	Median Spd (×)	Paired comparison with ATen
ATen	63	0.83	—
vLLM	26	1.24	Acc. < ATen in 20/21 configs.
cuBLAS	36	0.53	Spd < ATen in 18/20 defined pairs

and runtime failures. These ratios describe the evaluated configuration rather than a fixed property of each platform.

4.4 Capabilities Across Operator Sources

We compare operator sources within each of the 21 configurations in Table 1. Table 3 summarizes these paired comparisons. Accuracy is defined for every configuration; speedup medians exclude the two entries with no successful operators.

The paired comparisons distinguish correctness from optimization. vLLM has a median accuracy 37 pp below ATen, but higher median successful-set speedup. Its speedup exceeds ATen in 15 of 20 defined pairs. cuBLAS speedup is below ATen in 18 of 20 defined pairs, even though some methods achieve high correctness. The main challenge therefore differs by source: implementing the required behavior for vLLM and matching optimized reference performance for cuBLAS.

We also examine whether the same operators are difficult across configurations. We use the initial verification results, before anti-hack filtering, for Claude Code and OpenCode with Opus-4.6 and GLM-5.0. Across 210 operators, 114 pass in all four configurations, 20 pass in none, and 76 pass in only some. All four pass 75/110 ATen, 18/50 vLLM, and 21/50 cuBLAS operators, while none pass 5/110, 13/50, and 2/50, respectively. vLLM has the largest share of operators that fail in all four configurations (26%), including paged attention, quantized GEMM, and MoE operators.

On jointly solved operators, agent performance depends on operator source. We compare Claude Code and OpenCode on operators that both solve correctly and pass anti-hack checks, separately for each model and source. With Opus-4.6, the paired-bootstrap intervals for the speedup ratios include 1. With GLM-5.0, Claude Code is faster on shared ATen successes (0.88× vs. 0.74×), whereas OpenCode is faster on shared vLLM successes (1.12× vs. 0.84×). The corresponding OpenCode/Claude Code ratios are 0.84 (95% CI [0.71, 0.98]) and 1.33 ([1.07, 1.73]). This reversal persists when both methods are evaluated on identical operators within each source. Appendices N and O report the operator-level consistency and matched-set analyses.

Table 4: Agentic cost measurements on NVIDIA A100; Time is the cumulative per-operator solving duration, not the elapsed wall-clock time of the experiment; OpenCode has complete comparable accounting for only two of four configurations.

Method	Tok. (M)	Tok./Succ. (M)	Time (h)
Claude Code (Opus-4.6)	263	1.45	33
Claude Code (GLM-5.0)	383	2.72	48
Claude Code (Qwen3.5-27b)	381	2.91	48
Claude Code (MiniMax M-2.5)	340	3.33	50
AKO4all (Opus-4.6)	904	5.20	83
CUDA Opt. Skill (MiniMax M-2.5)	594	6.25	97
AutoKernel (GLM-5.0)	471	3.14	102
AutoKernel (Qwen3.5-27b)	475	4.80	102
AutoKernel (MiniMax M-2.5)	508	5.58	105

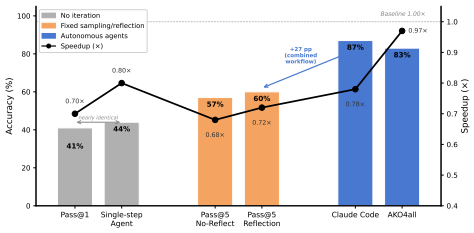


Figure 3: Accuracy and successful-set speedup across generation and debugging regimes (Opus-4.6, NVIDIA A100; 210 operators).

4.5 Why Portability Fails: Trajectory Analysis

Execution logs distinguish generated-code errors from compiler, API, data-type, and runtime incompatibilities. On Moore Threads, rejection of mixed `int32`/`int64` values recurs in 288 trajectory records. On Iluvatar CoreX, explicit incompatibilities affect at least 10/110 operators (9.1%): six involve math APIs, two language features, and two data types. From NVIDIA to Iluvatar CoreX, Opus-4.6 Pass@1 changes from 39% to 38% and Pass@5 from 62% to 57%, whereas AutoKernel with GLM-5.0 falls from 87% to 25%. The documented incompatibilities show that software support contributes to portability failures. Appendix P lists the affected operators, error messages, and failure categories.

4.6 Agent Frameworks: Gains, Costs, and Search Behavior

Table 4 shows the cost of agent-based kernel generation. AKO4all consumes 904M tokens, 3.44× Claude Code (Opus-4.6); CUDA Optimized Skill (KernelFlow-ops, 2026) reaches 6.25M tokens per success; and AutoKernel variants require up to 105 hours. Specialized agents average 4.99M tokens per successful operator, versus 1.45–3.33M for the evaluated Claude Code configurations, partly because they continue searching after correctness to optimize performance. Accuracy and speedup alone do not show whether these gains are affordable; tokens per success and cumulative solving time make their cost explicit.

Figure 3 compares generation methods with different levels of interaction, with per-source results in Appendix Q. The autonomous-agent regime shows the largest observed accuracy increase: the complete Claude Code workflow reaches 87%, 27 pp above the fixed-reflection configuration. Fixed reflection underperforms independent sampling on vLLM (24% vs. 28%). Among 21 ATen operators that both methods fail to solve, reflection also produces much more similar code across attempts (88% vs. 36%). It repeatedly makes limited changes to unsuccessful code (Appendix R).

5 Conclusion

We introduced **KernelGenBench**, the first unified multi-source × multi-chip benchmark infrastructure, using Triton as the common target language across six hardware platforms. The controlled MS view evaluates 210 operators from three sources on NVIDIA; the MC view evaluates the same 110 ATen operators across six platforms. Evaluation must therefore consider operator source, hardware platform, and agent framework together: different sources test different capabilities, a common language does not guarantee portability, and more interactive agentic workflows achieve higher overall correctness at higher cost.

Limitations and Future Work. *Source–platform coverage.* The infrastructure supports combinations beyond the reported MS and MC views, but reference implementations differ across platforms in availability and coverage; future work will evaluate more combinations as references become available. *Platform confidentiality.* Two platforms remain anonymous under commercial agreements; future work will provide access through jointly released leaderboards.

REPRODUCIBILITY STATEMENT

The supplementary material provides runnable evaluation code, including task specifications, reference-call wrappers, test grids, and the verification pipeline with its anti-hack checks. Section 3 defines the benchmark and evaluation framework; Appendices A–I document prompt construction, complete task lists, platform configurations, pipeline implementation, stress tests, and metric definitions. The full study consumed over 15 billion tokens and approximately $20,000 in API expenditure. Cost is measured within the common 30-minute per-task budget. Appendix J reports repeated-run analysis, Appendix K documents reproducibility boundaries and timing protocols, Tables 11–13 report platform configurations, validated cross-vendor adapters, and cross-axis results. Vendor-specific heterogeneous-platform implementations are omitted from the supplementary material to preserve commercial confidentiality. Operator specifications and test grids are derived from the public PyTorch ATen and vLLM operator interfaces; vendor-library references are called through their public APIs and are not redistributed. The released tasks are intended for measuring kernel-generation systems and for vendor kernel-development challenges, and should not be used as training data for models that are then evaluated on them.

ETHICS STATEMENT

This work involves no human subjects or personal data. The experiments were conducted with accelerator vendors that provided hardware access, software images and runtime environments, and routine technical support. The authors independently designed the benchmark, executed the experiments, analyzed the results, and determined the scientific conclusions. After completing the experiments, we consulted the relevant technical, business, and legal or compliance contacts to determine the permitted level of public disclosure. This post-hoc review concerned commercial confidentiality and identity disclosure, not experimental design or scientific interpretation. Three non-NVIDIA vendors authorized public identification. Two authorized publication of their de-identified results but required that all vendor-, product-, and software-stack-identifying information remain confidential; we therefore report them consistently as Platform A and Platform B. We present all disclosed results as measurements of the evaluated configurations rather than vendor-wide assessments.

AI USE STATEMENT

Generative AI tools were used for portions of benchmark-task and test implementation, software development, experiment execution and orchestration, literature-search assistance, figure and table preparation, and language editing. This included an agentic workflow for batch-generating initial task and test code, followed by author review and refinement. The research questions, benchmark scope and source selection, evaluation design and protocol, scientific interpretation, and conclusions were determined by the authors. Theoretical-model development, mathematical-claim formulation, proof writing, translation, and qualitative-data analysis are not applicable to this work. All AI-assisted code, tests, results, figures, calculations, and text were reviewed and verified by the authors, who take responsibility for the final paper and artifacts.

REFERENCES

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C. Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 29304–29320, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/f514cec81cb148559cf475e7426eed5e-Abstract.html.

Jason Ansel, Edward Z. Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (eds.), *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pp. 929–947. ACM, 2024. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. *CoRR*, abs/2507.11948, 2025. doi: 10.48550/ARXIV.2507.11948. URL https://doi.org/10.48550/arXiv.2507.11948.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

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 A. Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. AVO: agentic variation operators for autonomous evolutionary search. *CoRR*, abs/2603.24517, 2026. doi: 10.48550/ARXIV.2603.24517. URL https://doi.org/10.48550/arXiv.2603.24517.

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. *CoRR*, abs/2602.24286, 2026. doi: 10.48550/ARXIV.2602.24286. URL https://doi.org/10.48550/arXiv.2602.24286.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *CoRR*, abs/2606.19348, 2026. doi: 10.48550/ARXIV.2606.19348. URL https://doi.org/10.48550/arXiv.2606.19348.

Luke Emberson. Total ai chip memory bandwidth has grown 4.1x per year, now reaching 70 million terabytes per second, 2026. URL https://epoch.ai/data-insights/hbm-shipped. Accessed: 2026-04-29.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm: Making kernel development more accessible, 6 2025. URL https://huggingface.co/facebook/KernelLLM.

FlagOS contributors. Flagos/flaggems: An operator library for large language models implemented in the triton language., 2024. URL https://github.com/flagos-ai/FlagGems.

Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, 1986. doi: 10.1145/5666.5673. URL https://doi.org/10.1145/5666.5673.

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.

Alec M. Hammond, Aram Markosyan, Aman Dontula, Simon Mahns, Zacharias V. Fisches, Dmitrii Pedchenko, Keyur Muzumdar, Natacha Supper, Mark Saroufim, Joe Isaacson, Laura Wang, Warren Hunt, Kaustubh Gondkar, Roman Levenstein, Gabriel Synnaeve, Richard Li, Jacob Kahn, and Ajit Mathews. Agentic operator generation for ML asics. *CoRR*, abs/2512.10977, 2025. doi: 10.48550/ARXIV.2512.10977. URL https://doi.org/10.48550/arXiv.2512.10977.

Jinjun Huang, Zhongzhen Wen, Tongtong Xu, Meng Yan, Xin Xia, and Zhongxin Liu. Realistictritonbench: A benchmark for triton-kernel generation in real-world ai frameworks. *arXiv preprint arXiv:2608.12004*, 2026.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous GPU kernel optimization via iterative agent-driven search. *CoRR*, abs/2603.21331, 2026. doi: 10.48550/ARXIV.2603.21331. URL https://doi.org/10.48550/arXiv.2603.21331.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model based reinforcement learning for atari. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=S1xCPJHtDB.

KernelFlow-ops. cuda-optimized-skill. https://github.com/KernelFlow-ops/cuda-optimized-skill, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits, 2026. URL https://arxiv.org/abs/2603.19173.

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. *CoRR*, abs/2602.05885, 2026. doi: 10.48550/ARXIV.2602.05885. URL https://doi.org/10.48550/arXiv.2602.05885.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient GPU kernels? In Aarti Singh, Maryam Fazel,

Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

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 Z. Yang, Zachary 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. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

PyTorch Team. Tritonbench: Evaluating triton kernel generation capabilities. https://github.com/meta-pytorch/tritonbench, 2025. GitHub repository.

Mark Saroufim, Jiannan Wang, Bert Maher, Sahan Paliskara, Laura Wang, Shahin Sefati, and Manuel Candales. Backendbench: An evaluation suite for testing how well llms and humans can write pytorch backends, 2025. URL https://github.com/meta-pytorch/BackendBench.

Yue Shui, Chenyu Ma, Hangfei Xu, Shengzhao Wen, and Yanpeng Wang. Harness engineering for llm-driven GPU kernel generation. *CoRR*, abs/2607.17979, 2026. doi: 10.48550/ARXIV.2607.17979. URL https://doi.org/10.48550/arXiv.2607.17979.

Standard Performance Evaluation Corporation (SPEC). SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/, 2017.

Stanford Institute for Human-Centered Artificial Intelligence. The ai index 2026 annual report. Technical report, Stanford University, 2026. URL https://hai.stanford.edu/assets/files/ai_index_report_2026.pdf.

Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. In Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (eds.), *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pp. 10–19. ACM, 2019. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

TongmingLAIC. AKO4ALL. https://github.com/TongmingLAIC/AKO4ALL, 2026.

Han Wang, Jintao Zhang, Kai Jiang, Haoxu Wang, Jianfei Chen, and Jun Zhu. Kernelbenchx: A comprehensive benchmark for evaluating llm-generated GPU kernels. *CoRR*, abs/2605.04956, 2026. doi: 10.48550/ARXIV.2605.04956. URL https://doi.org/10.48550/arXiv.2605.04956.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel AI agent & evaluation benchmarks. *CoRR*, abs/2507.23194, 2025. doi: 10.48550/ARXIV.2507.23194. URL https://doi.org/10.48550/arXiv.2507.23194.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *CoRR*, abs/2507.17773, 2025. doi: 10.48550/ARXIV.2507.17773. URL https://doi.org/10.48550/arXiv.2507.17773.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. Tritonrl: Training llms to think and code triton without cheating. *CoRR*, abs/2510.17891, 2025. doi: 10.48550/ARXIV.2510.17891. URL https://doi.org/10.48550/arXiv.2510.17891.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. Flashinfer-bench: Building the virtuous cycle for ai-driven LLM systems. *CoRR*, abs/2601.00227, 2026. doi: 10.48550/ARXIV.2601.00227. URL https://doi.org/10.48550/arXiv.2601.00227.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Shawn Guo, Haowen Wang, Weicheng Gu, Yaxin Du, Joseph Li, Fanglin Xu, Yizhi Li, Lin Jing, Yuanbo Wang, Yuhan Gao, Ruihao Gong, Chuan Hao, Ran Tao, Aishan Liu, Tuney Zheng, Ganqu Cui, Zhoujun Li, Mingjie Tang, Chenghua Lin, Wayne Xin Zhao, Xianglong Liu, Ming Zhou, Bryan Dai, and Weifeng Lv. Incoder-32b: Code foundation model for industrial scenarios. *CoRR*, abs/2603.16790, 2026a. doi: 10.48550/ARXIV.2603.16790. URL https://doi.org/10.48550/arXiv.2603.16790.

Lingyun Yang, Yuxiao Wang, Shenghao Liang, Linfeng Yang, Daocheng Ying, Chunbo You, Rui Zhang, Luping Wang, Yinghao Yu, Guodong Yang, et al. Are llm-generated gpu kernels production-ready? a trace-driven benchmark and optimization agent. *arXiv preprint arXiv:2607.14541*, 2026b.

Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, et al. Cake: Compiler-agent co-design for frontier kernel evolution. *arXiv preprint arXiv:2608.12629*, 2026.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of llms, 2026. URL https://arxiv.org/abs/2601.15727.

Genghan Zhang, Yixin Dong, Chengze Fan, Zhichen Zeng, Yueming Yuan, Shaowei Zhu, and Kunle Olukotun. Ptxbench: Benchmark and adapt llms for gpu kernel optimization with architecture-specific ptx. *arXiv preprint arXiv:2608.17379*, 2026a.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max Fan, Julian Reed, Azalia Mirhoseini, and Will Su. Kernelbench-verified: Do llm-generated kernels actually beat pytorch? *CoRR*, abs/2607.16241, 2026b. doi: 10.48550/ARXIV.2607.16241. URL https://doi.org/10.48550/arXiv.2607.16241.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

A PROMPT DESIGN

KernelGenBench dynamically constructs structured prompts for each operator rather than relying on vague natural-language descriptions. Table 5 summarizes the prompt template modules and their differences between the LLM Sampling and Agentic evaluation paradigms.

Table 5: Prompt template modules: LLM Sampling vs. Agentic Frameworks.

Module	**LLM Sampling**	**Agentic**
System Role	✓	✓
One-shot Example	✓	✓
Operator Spec / Baseline Source Injection	✓	✓
Function Signature & Type Constraints	✓	✓
Anti-Cheat Rules	✓	✓
Testing Environment Description	✓	✓
Device-Specific Constraints (Heterogeneous)	✓	✓
Accelerator Device Assignment (Device ID)	—	✓
Verification CLI Command	—	✓
Iterative Workflow Guide	—	✓
Performance Optimization Guide	—	✓

The key difference across the three subsets lies in the "Operator Spec / Baseline Source Injection" module. In no subset is the underlying production kernel implementation exposed to the model. Table 6 distinguishes the interface-level context that is provided from the implementation that remains hidden.

Table 6: Source-specific context exposed during prompt construction. Python wrapper or harness source describes how an interface is invoked; it is not the underlying production kernel implementation.

Subset	**Provided to the model**	**Not provided**
ATen	Runtime `FunctionSchema`, overload variants, wrapper signatures, argument and return specifications, and available docstrings.	PyTorch's native C++/CUDA operator implementation.
vLLM	Source of the benchmark's Python baseline wrapper, including the exact signature, argument order, in-place behavior, and invocation of the corresponding `_custom_ops` interface.	The CUDA/C++ implementation behind the vLLM custom operator.
cuBLAS	Source of the benchmark's Python/`ctypes` baseline wrapper and interface metadata, including the C-API signature and call sequence used by the harness.	NVIDIA's proprietary cuBLAS implementation.

The prompts were designed by HPC engineers to reflect a realistic kernel-development workflow. Within each evaluation setting, every operator follows the same structured construction protocol, with schemas, signatures, wrapper context, verification commands, and device constraints populated automatically. We do not manually tune a prompt for an individual operator or retrospectively select the best-performing formulation. Agentic methods can use the common executable test suite while debugging, whereas sampling methods receive no execution feedback. The complete templates and construction code are included in the supplementary material.

Figure 4 uses `aten::cos` as an example, showing side-by-side the actual prompts under the LLM Sampling and Agentic paradigms (shared portions abbreviated as `[...]` for brevity). The prompts shown are for NVIDIA; on alternative platforms, the device-specific references (e.g., "GPU programmer") are automatically replaced with the corresponding platform-specific terminology, and additional device constraints are appended.

LLM Sampling Prompt

```
You are a skilled GPU programmer
proficient in Triton.  Your task is to
generate a Triton kernel function.

Here is an example of a PyTorch function
and its corresponding Triton kernel
implementation:
[one-shot add example, with kernel + wrapper]

You must strictly adhere to the following
specifications:
The Triton kernel should implement:
cos = torch.ops.aten.cos

The ATen operators involved are:
- cos
- cos.out
- cos.int
- [... 6 overload variants in total]

The Python wrapper functions should have
signatures like:
- def cos(...)
- def cos_out(...)
- [...]

Input and Output Args:
aten::cos(Tensor self) -> Tensor
[... full signature for each variant]

CRITICAL REQUIREMENTS:
1.  Handle all overload variants correctly
2.  Handle non-contiguous tensors
correctly

Generate valid Triton code directly
without any explanations.  Use '''python'''
format.
```

Agentic Framework Prompt

```
# Triton Kernel Implementation Task
Operator:  cos    Full name:  aten::cos
GPU ID: 0

## Runtime Environment
CUDA_VISIBLE_DEVICES=0

## Operator Specification
[same schema and signature as LLM prompt]

## Example
[same one-shot add example as LLM prompt]

## Verification Tool
python verify.py --code kernel.py
--op aten::cos --output-json

## Iterative Workflow
1.  Write implementation to kernel.py
2.  Run verification command
3.  If failed, read error and fix
4.  Repeat until all tests pass

## Performance Optimization
After correctness is achieved:
- Tune BLOCK_SIZE, num_warps, num_stages
- Re-run verification to confirm speedup

## Anti-Cheat Rules
[same rules as LLM prompt]
```

Figure 4: Side-by-side comparison of LLM Sampling and Agentic prompts for `aten::cos`. Shared content is abbreviated with `[...]`, and command and file names are shown in generic form.

B Full Operator List

Tables 7, 8, and 9 list the complete operator names for the three KernelGenBench subsets.

Table 7: Full operator list for the ATen subset (110 operators).

_index_put_impl_	_local_scalar_dense	_softmax	binary_cross_entropy_with_logits
_to_copy	acosh	add	affine_grid_generator
add_	amin	arange	upsample_nearest2d_backward
argmax	as_strided	asin	log_sigmoid_backward
bernoulli	bitwise_not	bmm	reflection_pad1d_backward
cat	clone	contiguous	rrelu_with_noise_backward
copy_	cos	cosh	unsafe_split_with_sizes
cumsum	diff	div	smooth_l1_loss_backward
div_	embedding	empty_strided	margin_ranking_loss
eq	erfc	expand	rrelu_with_noise
expand_as	exponential_	fill_	pairwise_distance
floor	floor_divide	fmax	softplus_backward
full	gather	gt	new_empty_strided
hardsigmoid	heaviside	huber_loss	soft_margin_loss
i0	im2col	index	select_backward
index_put_	index_select	item	mish_backward
le	linear	log10	masked_fill_
logaddexp2	logit	matmul	resolve_conj
mean	mish	mm	resolve_neg
mul	narrow	neg	special_entr
new_ones	ones_like	poisson	scalar_tensor
polygamma	pow	prelu	unsafe_split
renorm	reshape	rot90	unsqueeze
rsqrt	rsub	scatter	zeros_like
select	sgn	silu	softmax
sin	sort	square	stack
sub	sum	t	to
zero_	zeros		

Table 8: Full operator list for the vLLM subset (50 operators, from vLLM v0.13.0).

allspark_w8a16_gemm	apply_repetition_penalties_cuda	awq_gemm
awq_marlin_moe_repack	batched_moe_align_block_size	concat_and_cache_mla
convert_fp8	convert_vertical_slash_indexes	copy_blocks
copy_blocks_mla	cp_gather_cache	cp_gather_indexer_k_quant_cache
cutlass_pack_scale_fp8	cutlass_scaled_mm	cutlass_scaled_mm_azp
fused_add_rms_norm	fused_qk_norm_rope	gather_and_maybe_dequant_cache
ggml_dequantize	ggml_moe_a8	ggml_moe_a8_vec
ggml_mul_mat_a8	ggml_mul_mat_vec_a8	gptq_gemm
gptq_marlin_24_gemm	gptq_marlin_gemm	gptq_marlin_moe_repack
gptq_shuffle	grouped_topk	hadacore_transform
marlin_int4_fp8_preprocess	merge_attn_states	moe_align_block_size
moe_lora_align_block_size	moe_sum	paged_attention_v1
paged_attention_v2	permute_cols	reshape_and_cache
reshape_and_cache_flash	rms_norm	rms_norm_dynamic_per_token_quant
rms_norm_per_block_quant	rotary_embedding	scaled_fp8_quant
scaled_int8_quant	selective_scan_fwd	shuffle_rows
swap_blocks	topk_softmax	

Table 9: Full operator list for the cuBLAS subset (50 operators, based on cuBLAS 12.4).

cublasCcopy_v2	cublasCdotu_v2	cublasCgemmStridedBatched
cublasCgemmStridedBatched_64	cublasCgemm_v2	cublasCgemvBatched_64
cublasCgemvStridedBatched	cublasCgemv_v2	cublasCgeru_v2
cublasCsymm_v2	cublasCsymv_v2	cublasCsyrkEx
cublasDasum_v2	cublasDaxpy_v2	cublasDcopy_v2
cublasDgemmBatched	cublasDgemmStridedBatched	cublasDgemmStridedBatched_64
cublasDgemvBatched	cublasDgemvStridedBatched	cublasDgemv_v2
cublasDsbmv_v2	cublasDsyr2_v2	cublasDtrsmBatched
cublasHgemmBatched	cublasHgemmStridedBatched	cublasSaxpy_v2
cublasSdgmm	cublasSdot_v2	cublasSgeam
cublasSgemmBatched_64	cublasSgemmEx	cublasSgemmStridedBatched
cublasSgemm_v2	cublasSgemvBatched	cublasSgemvStridedBatched
cublasSger_v2	cublasSscal_v2	cublasSsyrk_v2
cublasStbmv_v2	cublasStrsm_v2	cublasStrsv_v2
cublasZdotc_v2	cublasZgemmBatched	cublasZgemmStridedBatched
cublasZgemvBatched	cublasZgemvStridedBatched	cublasZgerc_v2
cublasZswap_v2	cublasZtrsmBatched	

C cuBLAS GEMM Precision Variant Coverage

Table 10 shows the 14 independent problems derived from GEMM (General Matrix Multiplication) operators in the cuBLAS subset, broken down by data precision, batching mode, and index width. This illustrates the fine-grained API fragmentation typical of closed-source low-level libraries.

Table 10: Fine-grained cuBLAS operator variants: GEMM (General Matrix Multiplication) as an example.

Precision	Standard (_v2)	StridedBatched	Batched	64-bit Index (_64)	Other Variants
Float32 (S)	cublasSgemm_v2	cublasSgemmStridedBatched	—	cublasSgemmBatched_64	cublasSgemmEx
Float64 (D)	—	cublasDgemmStridedBatched	cublasDgemmBatched	cublasDgemmStridedBatched_64	—
Complex64 (C)	cublasCgemm_v2	cublasCgemmStridedBatched	—	cublasCgemmStridedBatched_64	—
Complex128 (Z)	—	cublasZgemmStridedBatched	cublasZgemmBatched	—	—
Float16 (H)	—	cublasHgemmStridedBatched	cublasHgemmBatched	—	—

D Cross-Platform Configuration

This section details the hardware-specific configurations injected by KernelGenBench-MC for heterogeneous hardware-platform evaluation.

Hardware Configuration. Table 11 summarizes the accelerator used on each platform and its evaluation environment. Platforms A and B remain de-identified under their confidentiality requirements.

Table 11: Hardware and software configurations for KernelGenBench-MC. All evaluations use a full node and the official image provided by the corresponding vendor.

Paper label	Accelerator	Execution stack	Key software versions
NVIDIA	NVIDIA A100	CUDA 12.4	Driver 535.161.08; PyTorch 2.9.0; Triton 3.5.0; cuBLAS 12.4; vLLM 0.13.0
Moore Threads	MTT S5000	MUSA / MTGPU	PyTorch 2.5.0; torch_musa 2.5.0; MUSA 3.1; Triton 3.1.0 (vendor backend)
Hygon	DCU BW series (gfx936)	DTK / HIP–ROCm	DTK 25.04; Triton 3.0.0+das.opt4.dtk2504
Iluvatar CoreX	BI-V150	IXUCA	Ubuntu 22.04.5; PyTorch 2.7.1; IXUCA 4.4.0; Triton 3.1.0 (vendor build)
Platform A	Not disclosed	Not disclosed	Not disclosed under commercial confidentiality
Platform B	Not disclosed	Not disclosed	Not disclosed under commercial confidentiality

Cross-Vendor Reference Adapters. KernelGenBench provides completed source–platform adapter paths beyond the two analytical views reported in the main paper. Table 12 records the public production-runtime and vendor-library mappings used by this infrastructure. All public mappings were validated on their target platforms. These paths also support ongoing vendor-collaborative kernel-development leaderboards, providing a common submission and verification interface for contributing high-quality operators; Platform A and Platform B mappings remain de-identified.

Cross-Axis Validation. We evaluate Claude Code (Opus-4.6) on the complete 50-task cuBLAS-derived subset across all six platforms. Table 13 reports clean pass rate and successful-set speedup against each platform's local vendor-library reference. The five non-NVIDIA platforms solve 36–48 tasks (72–96% accuracy), demonstrating that the vendor-library source extends beyond the NVIDIA-only MS view. Their speedups range from 0.18× to 0.63× relative to different local references and therefore are not direct cross-platform hardware-performance comparisons. Beyond this controlled experiment, the same infrastructure has supported multiple vendor-collaborative kernel-development challenges and cross-platform leaderboards, providing operational evidence that the adapter and verification pipeline can be used in real kernel-development workflows.

Hardware-Aware Prompt Templates. The framework maintains a separate prompt template set per device type, automatically selecting and assembling the corresponding template at runtime based

Table 12: Cross-vendor production-runtime and library-reference mappings in KernelGenBench. The completed adapter layer supports source–platform evaluation and vendor-collaborative kernel development; Platform A and Platform B retain de-identified mappings.

Platform	**Execution stack**	**Production-runtime path**	**Vendor-library path**
NVIDIA	CUDA	Upstream vLLM	cuBLAS (native interface)
Moore Threads	MUSA / MTGPU	vLLM-MUSA	muBLAS (cuBLAS-style mapping)
Hygon	DTK / HIP–ROCm	vLLM on DTK	hipBLAS (cuBLAS-style mapping)
Iluvatar CoreX	IXUCA	CoreX vLLM distribution	CoreX cuBLAS-compatible library
Platform A	De-identified	De-identified vLLM/runtime adapter	De-identified vendor-library adapter
Platform B	De-identified	De-identified vLLM/runtime adapter	De-identified vendor-library adapter

Table 13: Cross-axis validation on the complete 50-task cuBLAS-derived subset. All results use Claude Code (Opus-4.6) under the same evaluation protocol. Each platform uses its corresponding vendor-optimized library implementation as the local reference. **Acc.** denotes clean pass rate, and **Spd.** denotes the successful-set geometric-mean speedup of generated Triton kernels relative to the local reference.

Platform	**Local Reference**	**Passed / 50**	**Acc. (%) ↑**	**Spd. (×) ↑**
NVIDIA	cuBLAS	47 / 50	94	0.51
Moore Threads	muBLAS	47 / 50	94	0.56
Hygon	hipBLAS (DTK)	48 / 50	96	0.63
Iluvatar CoreX	CoreX cuBLAS-compatible library	36 / 50	72	0.58
Platform A	De-identified vendor-library reference	46 / 50	92	0.18
Platform B	De-identified vendor-library reference	47 / 50	94	0.61

Note. Speedup is computed independently on each platform relative to its local vendor-optimized library and should not be interpreted as a direct cross-platform hardware-performance comparison.

on detected hardware (e.g., injecting platform-specific import statements and runtime initialization code).

E Distributed Sandbox Infrastructure

This section details the production-grade dispatch registration and the process isolation and distributed scheduling infrastructure underlying KernelGenBench's full-node parallel evaluation.

Production-Grade Dispatch Registration. For ATen operators, the system performs an AST scan to capture `@register` decorators and force-mounts the generated Triton kernels into PyTorch's `torch.library` dispatch tree. All ATen operators are therefore tested under PyTorch's full low-level dispatch mechanism, not isolated Python function calls.

Full-Node Parallel Evaluation with Process Isolation. To enable full-node utilization, KernelGenBench dispatches all operators in parallel across all available accelerator devices on the machine. Each operator runs in a fully isolated subprocess (`multiprocessing spawn`), with its own workspace directory and exclusive device assignment, ensuring that a crash, timeout, or device memory corruption in one operator cannot affect any other. The distributed resource scheduler acquires hardware control via kernel-level file locks, recording the holder's PID and automatically reclaiming locks from crashed processes to prevent permanent device resource leaks. When a subprocess triggers a timeout (300s for correctness, 600s for performance), the parent process sends `SIGKILL` to destroy the entire process tree.

Adaptive Numerical Tolerance. Numerical comparison of low-level operators is sensitive to floating-point truncation errors, so every comparison is made under dtype-specific relative and absolute tolerances; the exact rule is implemented in the released test utilities. Complex-valued outputs are compared directly under the complex-dtype rules rather than being cast to a real dtype. For MC, L1 static scanning and L2 dual-execution comparison form the common enforcement protocol on every platform. NVIDIA additionally applies the L3 GPU-profiler fingerprint after L1 and L2; equivalent profiler interfaces are unavailable across the other hardware stacks, so the cross-platform comparison rests on the shared L1/L2 protocol while NVIDIA adds one further check. The representative shortcut behaviors in Table 16 are rejected by the shared L1/L2 checks.

Platform-local reference policy. KernelGenBench-MC evaluates the fixed 110-task ATen suite on every platform. Each task is checked against the native ATen reference available in that platform's local PyTorch backend. If this reference API is unavailable, the task remains in the 110-task denominator and is recorded as unsuccessful. MC accuracy therefore measures end-to-end platform success across generation, compilation, local-reference verification, and execution, rather than generated-kernel correctness in isolation.

F Evaluation Pipeline Algorithm Details

This section provides the full pseudocode for both the agent dispatch pipeline and the LLM Sampling Methods generation pipeline described in Section 3.3.

Common External Budget and Method-Specific Resources. All evaluated methods receive the same target-operator specification and test suite, and every final submission is judged by the same correctness and applicable anti-hack checks under a common 30-minute wall-clock budget per operator task. We control this task-level budget rather than the number of internal iterations: agent frameworks define an iteration differently and autonomously allocate their time among generation, debugging, verification, and optimization. Pass@k and vanilla agentic frameworks do not receive profiling access from the benchmark. Some kernel-specialized agents include profiling, skills, or hardware tools as intrinsic parts of their published workflows; these capabilities are retained because the evaluated object is the complete kernel-generation system. Thus, the comparison standardizes external task conditions and final evaluation, while intentionally preserving method-specific mechanisms. We report token use and cumulative solving-time cost to capture the computational consequences of these method-specific search strategies.

Formal definition of the LLM Sampling Methods pipeline: For a given operator, the system draws up to k independent samples. Every round reconstructs the prompt from the original operator

Algorithm 1 Vanilla Agentic Framework Evaluation Dispatch

Require: Operator set $\mathcal{O}$, agent backend M, device pool $\mathcal{G}$, timeout $T_{\max}$
$\mathcal{C} \leftarrow \{\}$ {Empty operator-to-candidate map}
for each $op \in \mathcal{O}$ **in parallel** (bounded by $|\mathcal{G}|$) **do**
  $dev \leftarrow$ AcquireDevice($\mathcal{G}$) {Works identically on all platforms}
  $ws \leftarrow$ CreateWorkspace(op)
  $prompt \leftarrow$ BuildPrompt($op, dev,$ VERIFYCOMMAND)
  $proc \leftarrow M$.launch($prompt, ws, dev$) {Spawn agent process}
  {Agent autonomously writes, verifies, debugs, and optimizes the kernel}
  wait $proc$ until completion or $T_{\max}$
  $code \leftarrow M$.finish($proc, ws$) {Extract final kernel}
  $\mathcal{C}[op] \leftarrow code$
  ReleaseDevice(dev)
end for
return BatchAntiHackCheck($\mathcal{C}$)

specification, without including previous candidates, execution errors, or tracebacks. Sampling stops at the first candidate that is both numerically correct and free of shortcut behaviour, so the reported clean pass rate is the standard Pass@k quantity: the fraction of operators for which at least one of the k samples is numerically correct and passes the anti-hack checks. Reported Pass@k speedup is measured from that retained candidate rather than selected as the maximum speedup among k samples.

Algorithm 2 LLM Sampling Methods: Pass@K Sequential Early-Stopping Evaluation Pipeline

Require: Operator set $\mathcal{O}$, max sampling rounds K, model policy π, evaluator $\mathcal{E}$
$\mathcal{O}_{remaining} \leftarrow \mathcal{O}, \mathcal{C}_{passed} \leftarrow \{\}$ {Empty operator-to-candidate map}
for $round = 1$ **to** K **do**
  for each $op \in \mathcal{O}_{remaining}$ **do**
    $p \leftarrow$ ConstructPrompt(op) {Fresh prompt; no feedback from earlier rounds}
    $c \leftarrow \pi(p)$ {Generate code}
    $is_correct, speedup \leftarrow \mathcal{E}$.verify($c$)
    $is_clean \leftarrow \mathcal{E}$.antiHack($c$)
    if $is_correct$ **and** is_clean **then**
      $\mathcal{C}_{passed}[op] \leftarrow (c, speedup)$
    end if
  end for
  $\mathcal{O}_{remaining} \leftarrow \mathcal{O}_{remaining} \setminus$ keys($\mathcal{C}_{passed}$) {Early stopping on a clean success}
  if $\mathcal{O}_{remaining} == \emptyset$ **then**
    break
  end if
end for
return $\mathcal{C}_{passed}$

G STRESS TEST SHAPE AND DISPATCH CONFIGURATIONS

This section details the combinatorial test case construction and shape/stride configurations used in KernelGenBench's parametrized test cases. The constants are defined by the accompanying test utilities.

Extreme Tensor Distributions and Boundary Stress Testing. Test cases deliberately inject many non-aligned long-tail sizes (odd boundaries constructed as 2^d+17), mixed with full-dimensional coverage from scalar `()` to 5D tensors `(16, 7, 57, 32, 29)`. Minimal-size tests (e.g., `shape=(1, 2)`) expose kernel launch overhead degradation.

Prompt information across sources. For every source, prompts are constructed from public API-level documentation and exclude the underlying reference implementation. Every task is a single API:

the signature, argument order, in-place behaviour, and the call that the generated kernel must serve. cuBLAS entries are one routine per precision and batching mode, which is how the library itself factors its API; each such routine is one task, as for the other sources. Prompts are assembled from the public interface documentation of each source, and none of them contains the implementation behind the reference. Per-source results describe these task sets.

Example: `softmax` Test Case Construction. For `softmax`, the semantic parameter is `dim` (reduction axis). The $k_i = 18$ test cases are the Cartesian product of three axes: shape $\in \{(1, 256), (4096, 256), (200, 2560, 3)\}$, dtype $\in \{\text{float16}, \text{float32}, \text{bfloat16}\}$, and dim $\in \{0, 1\}$. A kernel is counted as passing only if all 18 cases satisfy the numerical tolerance. The shape and stride grids used across all operator categories are listed in Tables 14 and 15.

Table 14: Representative shape configurations by operator category; the complete grids ship with the released test utilities.

Category	Shapes
Pointwise	`()`, `(1,)`, `(1024, 1024)`, `(20, 320, 15)`, `(16, 128, 64, 60)`, `(16, 7, 57, 32, 29)`
Reduction	`(1, 2)`, `(4096, 256)`
Reduction (small)	`(1, 2)`, `(4096, 256)`, `(200, 2560, 3)`
BLAS (M, N)	`(1, 32)`, `(160, 1024)`
BLAS (M, N, K)	`(1, 1, 32)`, `(15, 160, 1024)`
1D (power-of-2)	`(16,)`, `(64,)`, `(256,)`, `(1024,)`
1D (non-aligned)	`(33,)`, `(81,)`, `(273,)`, `(1041,)`
Upsample	`(32,16,128,128)`, `(15,37,256,256)`, `(3,5,127,127)`, `(128,192,42,51)`, `(3,7,1023,1025)`

Table 15: Representative stride configurations for memory layout stress testing; the complete grid ships with the released test utilities.

Layout Type	Shape	Stride
1D Contiguous		
	`(1,)`	`(1,)`
	`(1024,)`	`(1,)`
	`(1000000,)`	`(1,)`
1D Dilated		
	`(1,)`	`(2,)`
	`(1024,)`	`(2,)`
	`(1000000,)`	`(2,)`
2D Contiguous		
	`(1, 1024)`	`(1024, 1)`
	`(10000, 128)`	`(128, 1)`
2D Transposed		
	`(1024, 1)`	`(1, 1024)`
	`(128, 10000)`	`(1, 128)`
3D Contiguous		
	`(20, 320, 15)`	`(4800, 15, 1)`
3D Transposed		
	`(320, 20, 15)`	`(15, 4800, 1)`

H Additional Analysis on Generation Behaviors

This section provides supplementary observations regarding model-specific generation bottlenecks and anti-hack interception statistics.

Generation truncation in GLM-5.0. Examining GLM-5.0's generation logs under Pass@5, we find that a non-trivial fraction of generated kernels are incomplete: the model frequently produces excessive intermediate variables and verbose comments, causing core code logic to be truncated before completion within the uniform 16K max token limit. This generation inefficiency, which reflects low effective information per token, contributes to incomplete and functionally incorrect submissions under Pass@5.

Anti-hack interception patterns. Table 16 illustrates how the evaluator rejects both direct reuse and hybrid fallback. In the first observed case, a `margin_ranking_loss` submission defined a decorative Triton kernel but obtained its result by invoking the existing ATen operator. L1 detects the explicit high-level call, while L2 ghost replay independently tests whether replacing the declared Triton function with a no-op changes the output. In the second case, a Huber-loss submission implemented only the elementwise portion in Triton and delegated the required reduction to PyTorch. This partial implementation is rejected as a high-level framework fallback.

Table 16: Representative shortcut behaviors observed during benchmark evaluation.

Pattern	Observed example	Detection	Reason for rejection
Direct target-API call	`torch.ops.aten.margin_ranking_loss(...)` with a decorative Triton kernel	L1 static scan; L2 causal-use check	The output is obtained from an existing implementation rather than the generated Triton kernel.
Hybrid framework fallback	Triton elementwise computation followed by `output.mean()` or `output.sum()`	L1 high-level API checks	The target reduction is delegated to PyTorch, so the submission is not a complete Triton replacement.

Table 17 shows the per-model anti-hack interception counts under Pass@1 and Pass@5, over all evaluated configurations including the legacy model baselines of Appendix L. Interceptions are heavily concentrated in ATen (29/30 under Pass@5), with vLLM accounting for one case. These cases are excluded from the clean pass rate. The concentration is consistent with ATen exposing many easily callable high-level operators, whereas vLLM's non-standard APIs make simple shortcut calls less available.

Table 17: Snapshot of anti-hack interception details (hack count and subset distribution).

Model	Pass@1		Pass@5	
	Hacks	Subset	Hacks	Subset
Opus-4.6	3	3 vLLM	7	7 ATen
Opus-4.5	3	2 ATen + 1 vLLM	2	1 ATen + 1 vLLM
GPT-5.4	3	3 ATen	8	8 ATen
GPT-5.2	3	3 ATen	5	5 ATen
GLM-5.0	1	1 ATen	1	1 ATen
GLM-4.7	0	—	7	7 ATen
Total	**13**	9 ATen + 4 vLLM	**30**	29 ATen + 1 vLLM

I SPEEDUP STATISTICAL METRICS

The analysis pipeline provides three built-in statistical metrics. Given M operators that pass correctness verification in an experiment, where the i-th operator has k_i test cases, the operator-level speedup is defined as:

$$S_i = \left(\prod_{j=1}^{k_i} \frac{T_{\text{base}}^{(i,j)}}{T_{\text{triton}}^{(i,j)}} \right)^{1/k_i}$$

The overall speedup is aggregated via a two-level geometric mean:

$$S_{\text{overall}} = \left(\prod_{i=1}^{M} S_i \right)^{1/M}$$

Why geometric rather than arithmetic averaging? Speedup is a multiplicative ratio, so equal-factor improvements and regressions should be treated symmetrically. Consider two test cases

with speedups $0.5\times$ and $2.0\times$: one is twice as slow and the other twice as fast. Their arithmetic mean is $1.25\times$, which incorrectly suggests a net improvement, whereas their geometric mean is $\sqrt{0.5 \times 2.0} = 1.0\times$, correctly indicating that the reciprocal changes cancel. Equivalently, speedups are naturally compared in log space, where $\log 2 = -\log 0.5$; the geometric mean is the exponentiated arithmetic mean of these log-speedups. The two-level construction applies this principle first within each operator and then across operators, preventing operators with more test cases from receiving disproportionate weight.

In addition to GeoMean, the framework includes two robust auxiliary metrics:

1. Median

Sort $S_1, \ldots, S_M$ and take the middle value:

$$\text{Median} = \begin{cases} S_{(M+1)/2} & M \text{ is odd} \\ \frac{1}{2}\left(S_{M/2} + S_{M/2+1}\right) & M \text{ is even} \end{cases}$$

The median is immune to extreme outliers and provides a robust central tendency estimate when inter-operator speedup variance is large.

2. Interquartile Mean (IQM)

Let $F_M^{-1}(u)$ denote the empirical quantile function of the M operator-level speedups. The IQM is the mean over the central half of this empirical distribution:

$$\text{IQM} = \frac{1}{0.5} \int_{0.25}^{0.75} F_M^{-1}(u)\, du.$$

This definition trims 25% from each tail and handles fractional boundary observations without an indexing ambiguity. IQM balances the statistical efficiency of the mean with the robustness of the median, and has been widely adopted in reinforcement learning benchmarks such as Atari 100k (Agarwal et al., 2021; Kaiser et al., 2020), particularly for scenarios with limited samples and skewed distributions.

The three metrics serve different purposes: GeoMean is appropriate for aggregating multiplicative ratios and consistent with system evaluation standards such as SPEC CPU; Median provides the most conservative central estimate; IQM provides a more stable mean after trimming tail noise. All main tables in this paper report GeoMean.

J Repeated-Run Robustness Analysis

We repeat two configurations on the full 210-operator KernelGenBench-MS suite on NVIDIA: five runs of DeepSeek-V4-Pro with one sample per task at `temperature`=0.8, and the Pass@5 Opus-4.6 configuration of Table 1 together with four independent replications of it. The replications use the same code and protocol as the main-table experiment and were executed on a separate machine. Table 18 reports the accuracy of each run.

Table 18: Repeated runs of two sampling configurations on the 210-operator MS suite. Accuracy is the fraction of operators solved in each run; in the Opus-4.6 row the first column is the main-table run and the remaining four are independent replications.

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Mean**	**SD**
DeepSeek-V4-Pro, one sample per task							
Solved operators	20	24	26	21	20	22.2	—
Accuracy (%)	10	11	12	10	10	11	1.3 pp
Opus-4.6, Pass@5, `temperature`=0.8							
Accuracy (%)	57	53	54	53	55	54.4	1.7 pp

Across these repeated configurations the sample SD is 1.3–1.7 percentage points: the DeepSeek-V4-Pro repeats span 10–12% (mean 11%) and the Opus-4.6 configuration spans 53–57% across the main-table run and its four replications (mean 54.4%). Variation of this size is why closely spaced single-trajectory results, such as 87% versus 83%, are descriptive rather than established orderings.

K Additional Reproducibility Details and Evaluation Boundaries

Repeated-run coverage. Most agent results use one generation trajectory per task, aggregating 110 or 210 task outcomes without a configuration-level variance estimate. The full study consumed over 15 billion tokens (approximately $20,000 in API expenditure). Appendix J reports repeated runs of two sampling configurations. Kernel timing uses warm-up and repeated measurement: standard Triton tests use a 100-ms `do_bench` measurement window, and custom cuBLAS tests average 100 explicit executions. A representative task subset would make repeated agent evaluation more affordable.

Prompt protocol. Prompt templates are applied systematically without per-operator tuning. The reported results characterize this fixed, released protocol, enabling future work to study prompt sensitivity under the same tasks and verification pipeline.

Verification coverage. L1 static scanning and L2 ghost replay are the shared MC enforcement protocol. On NVIDIA the pipeline applies L1, L2 and then L3 profiling; the other stacks apply L1 and L2, because a portable profiler interface is currently unavailable there. The representative shortcut behaviors in Table 16 are already rejected by the shared L1/L2 checks. Platform-native tracing can be incorporated as vendor interfaces become available.

Metric interpretation. Global speedup is computed over each method's successful operator set. ATen, vLLM, and cuBLAS also use references at different abstraction and optimization levels. We therefore emphasize per-source accuracy and speedup; matched-intersection analysis is the appropriate protocol when comparing performance over an identical set of solved operators.

System-level evaluation. The benchmark standardizes task specifications, tests, final verification, and wall-clock budgets while preserving each framework's prompts, skills, tools, and native profiling access. Accordingly, its intended unit of comparison is the complete LLM–agent system.

Benchmark scope and disclosure. KernelGenBench implements multi-source, multi-chip evaluation paths through a common Triton target and platform-specific runtime and vendor-library adapters; direct CUDA/CUTLASS generation remains an NVIDIA-only complementary direction. The main paper reports two controlled analytical views rather than treating all source–platform pairs as one leaderboard. Across heterogeneous accelerators, production-runtime ports and vendor-library counterparts can differ in semantics, plugin coverage, implementation path, and maturity. Pooling these references would confound source-conditioned kernel generation with vendor-stack maturity. Establishing broader semantically matched and vendor-authorized references is a focus of future co-development. Platforms A and B remain de-identified under commercial requirements, and the available platform configurations are reported in Appendix D. The 210 operators cover the three sources studied here; the wider production-kernel surface is larger.

L Legacy Model Baselines on NVIDIA A100

Tables 19 and 20 report the static generation performance of legacy and reference foundation models (Opus-4.5, GPT-5.4, GPT-5.2, GLM-4.7) on NVIDIA A100, provided for historical reference and cross-generation comparison.

M Fast_p Evaluation and Cost Results

This section reports the full fast_p distribution for all methods on NVIDIA A100 (Table 21), across six heterogeneous platforms (Table 22), and the complete cross-platform cost breakdown (Table 23).

N Raw Solution Consistency and Matched-Set Analysis

We measure raw R0 solution consistency from pre-anti-hack verification outcomes across four general-agent configurations: Claude Code and OpenCode, each paired with Opus-4.6 and GLM-5.0. For operator i, we define $d_i = 1 - \frac{1}{4}\sum_c \mathbb{1}[c \text{ solves } i]$. Each configuration is evaluated over the common 210-operator task universe, and an unsuccessful operator counts as not solved. Because this diagnostic precedes anti-hack filtering, it characterizes agreement in initial functional verification

Table 19: Legacy and reference foundation-model evaluation on NVIDIA A100 (Pass@1 and Pass@5).

Method & Setup	Overall (210)		ATen (110)		vLLM (50)		cuBLAS (50)	
	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)	Acc (%)	Spd (×)
LLM Sampling Methods								
Pass@1 (Opus-4.5)	35	0.78	33	0.94	22	0.83	54	0.59
Pass@1 (GPT-5.4)	21	0.69	21	0.76	30	0.54	12	0.65
Pass@1 (GPT-5.2)	20	0.55	24	0.53	22	0.64	12	0.57
Pass@1 (GLM-4.7)	12	0.56	15	0.49	10	1.30	10	0.35
Pass@5 (Opus-4.5)	50	0.73	53	0.91	28	0.88	64	0.46
Pass@5 (GPT-5.4)	35	0.72	41	0.87	38	0.44	20	0.60
Pass@5 (GPT-5.2)	38	0.52	47	0.49	36	0.58	18	0.56
Pass@5 (GLM-4.7)	24	0.67	26	0.55	22	1.10	20	0.70

Table 20: Appendix: fast_p evaluation of legacy and reference foundation models on NVIDIA A100. All entries are subset-normalized percentages.

Method & Setup	Overall (210)			ATen (110)			vLLM (50)			cuBLAS (50)		
	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$
LLM Sampling Methods												
Pass@1 (Opus-4.5)	20	16	2	25	22	2	16	14	6	10	4	0
Pass@1 (GPT-5.4)	12	9	0	17	12	1	8	6	0	6	4	0
Pass@1 (GPT-5.2)	9	7	1	15	10	2	6	6	2	0	0	0
Pass@1 (GLM-4.7)	7	5	1	8	5	1	10	10	4	0	0	0
Pass@5 (Opus-4.5)	30	21	4	43	31	3	20	14	8	12	6	2
Pass@5 (GPT-5.4)	21	13	1	33	21	1	10	6	2	8	4	0
Pass@5 (GPT-5.2)	18	11	1	28	18	1	8	6	4	4	0	0
Pass@5 (GLM-4.7)	12	8	3	15	7	1	14	12	8	4	4	2

Table 21: fast_p evaluation on NVIDIA A100 across 210 operators. Entries are subset-normalized percentages satisfying speedup $> 0.80\times$, $1.00\times$, or $1.50\times$.

Method & Setup	Overall (210)			ATen (110)			vLLM (50)			cuBLAS (50)		
	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$	$\mathrm{fast}_{0.8}$	$\mathrm{fast}_{1.0}$	$\mathrm{fast}_{1.5}$
LLM Sampling Methods												
Pass@1 (Opus-4.6)	22	17	2	32	24	1	12	12	4	12	6	2
Pass@1 (GLM-5.0)	7	6	2	6	5	0	10	10	6	4	4	4
Pass@1 (Qwen3.5-27b)	5	3	0	7	5	0	2	2	2	4	0	0
Pass@1 (MiniMax M-2.5)	1	1	0	3	3	0	0	0	0	0	0	0
Pass@5 (Opus-4.6)	32	21	3	47	31	3	18	14	4	12	6	4
Pass@5 (GLM-5.0)	17	11	4	20	11	2	16	16	8	10	8	4
Pass@5 (Qwen3.5-27b)	9	7	2	12	10	1	8	6	6	4	0	0
Pass@5 (MiniMax M-2.5)	10	8	1	16	12	1	4	4	2	2	2	2
Vanilla Agentic Frameworks												
Claude Code (Opus-4.6)	51	35	6	69	44	4	48	40	14	14	10	4
Claude Code (GLM-5.0)	40	27	6	54	32	4	36	32	12	16	12	4
Claude Code (Qwen3.5-27b)	35	23	5	53	33	4	26	20	10	6	6	4
Claude Code (MiniMax M-2.5)	29	17	1	45	25	1	16	14	2	4	2	2
OpenCode (Opus-4.6)	53	36	8	71	46	5	46	40	14	20	10	6
OpenCode (GLM-5.0)	44	32	5	64	45	4	38	32	10	8	6	2
OpenCode (Qwen3.5-27b)	32	26	4	40	29	1	38	36	14	8	8	2
OpenCode (MiniMax M-2.5)	22	12	2	35	17	0	12	10	6	4	4	2
Kernel-Specialized Agents												
AKO4all (Opus-4.6)	65	47	10	85	59	6	56	50	20	30	16	10
CUDA Opt. Skill (MiniMax M-2.5)	27	15	3	42	25	4	16	10	4	6	0	0
AutoKernel (GLM-5.0)	59	38	4	87	53	0	34	32	12	22	10	6
AutoKernel (Qwen3.5-27b)	42	28	2	69	44	0	16	16	6	8	6	4
AutoKernel (MiniMax M-2.5)	40	24	2	66	37	0	18	18	8	2	2	2

Table 22: fast_p cross-platform evaluation on 110 ATen operators. All entries are subset-normalized percentages satisfying the indicated speedup threshold.

Method & Setup	NVIDIA			Moore Threads			Hygon			Iluvatar CoreX			Platform A			Platform B		
	$\text{fast}_{0.8}$	$\text{fast}_{1.0}$	$\text{fast}_{1.5}$	$\text{fast}_{0.8}$	$\text{fast}_{1.0}$	$\text{fast}_{1.5}$	$\text{fast}_{0.8}$	$\text{fast}_{1.0}$	$\text{fast}_{1.5}$	$\text{fast}_{0.8}$	$\text{fast}_{1.0}$	$\text{fast}_{1.5}$	$\text{fast}_{0.8}$	$\text{fast}_{1.0}$	$\text{fast}_{1.5}$	$\text{fast}_{0.8}$	$\text{fast}_{1.0}$	$\text{fast}_{1.5}$
LLM Sampling Methods																		
Pass@1 (Opus-4.6)	32	24	1	30	21	8	31	16	4	28	22	3	9	6	5	26	18	1
Pass@1 (Qwen3.5-27b)	7	5	0	3	1	0	6	4	0	5	3	0	1	1	0	6	3	0
Pass@1 (MiniMax M-2.5)	3	3	0	6	3	3	4	2	0	4	4	0	1	1	0	5	4	0
Pass@5 (Opus-4.6)	47	31	3	42	33	14	43	27	5	41	33	5	11	8	6	51	33	3
Pass@5 (Qwen3.5-27b)	12	10	1	10	7	3	13	11	0	14	7	0	4	2	0	10	2	0
Pass@5 (MiniMax M-2.5)	16	12	1	8	3	2	10	8	2	8	7	0	4	3	0	6	3	0
Vanilla Agentic Frameworks																		
Claude Code (Opus-4.6)	69	44	4	61	45	23	65	46	6	60	39	6	12	9	6	67	42	3
Claude Code (GLM-5.0)	54	32	4	47	35	22	41	24	2	34	30	4	9	6	6	40	26	0
Claude Code (Qwen3.5-27b)	53	33	4	43	31	12	58	47	11	19	11	2	16	9	7	61	39	5
Claude Code (MiniMax M-2.5)	45	25	1	46	36	16	44	31	3	46	31	5	8	7	6	51	25	2
Kernel-Specialized Agents																		
AKO4all (Opus-4.6)	85	59	6	81	55	8	79	56	10	75	48	7	16	12	7	82	60	10
CUDA Opt. Skill (MiniMax M-2.5)	42	25	4	45	29	13	41	27	4	38	24	3	9	7	5	43	23	1
AutoKernel (GLM-5.0)	87	53	0	56	28	1	57	29	2	25	21	0	47	25	0	59	28	0
AutoKernel (Qwen3.5-27b)	69	44	0	75	38	3	64	36	2	19	16	0	18	12	3	72	41	1
AutoKernel (MiniMax M-2.5)	66	37	0	70	53	15	64	35	0	48	37	0	44	26	2	70	35	1

Table 23: Cross-platform computational cost evaluation on 110 ATen operators (agentic methods only). Reports Total Tokens (M), Tokens per Success (M), and cumulative per-operator solving time (h) across six heterogeneous hardware platforms.

Method & Setup	NVIDIA			Moore Threads			Hygon			Iluvatar CoreX			Platform A			Platform B		
	Tok (M)	Tok/Suc (M)	Time (h)	Tok (M)	Tok/Suc (M)	Time (h)	Tok (M)	Tok/Suc (M)	Time (h)	Tok (M)	Tok/Suc (M)	Time (h)	Tok (M)	Tok/Suc (M)	Time (h)	Tok (M)	Tok/Suc (M)	Time (h)
Vanilla Agentic Frameworks																		
Claude Code (Opus-4.6)	84	0.83	9	128	1.25	16	107	1.10	15	123	1.35	16	173	1.77	18	111	1.05	15
Claude Code (GLM-5.0)	181	2.29	18	174	2.42	44	186	2.58	35	234	5.71	47	197	2.74	43	233	3.58	40
Claude Code (Qwen3.5-27b)	175	1.99	19	243	2.93	37	202	2.43	52	257	10.28	55	199	2.31	34	211	2.34	55
Claude Code (MiniMax M-2.5)	191	2.51	22	255	3.15	44	172	2.15	51	247	3.25	42	282	3.71	32	175	1.92	41
Kernel-Specialized Agents																		
AKO4all (Opus-4.6)	422	4.22	44	396	4.08	46	381	3.93	35	423	4.81	48	397	4.32	42	412	4.34	39
CUDA Opt. Skill (MiniMax M-2.5)	294	4.26	30	289	4.13	32	261	3.62	31	336	5.25	47	187	3.22	22	304	4.11	34
AutoKernel (GLM-5.0)	113	1.18	47	200	3.23	54	174	2.49	55	151	5.39	47	231	3.98	55	167	2.57	54
AutoKernel (Qwen3.5-27b)	133	1.75	36	178	2.14	51	131	1.82	54	166	7.22	53	136	3.09	55	141	1.74	50
AutoKernel (MiniMax M-2.5)	162	2.22	37	226	2.90	45	211	2.89	53	248	4.51	54	245	3.66	55	209	2.68	54

rather than clean benchmark correctness. The subsequent matched-set analysis uses clean successes after anti-hack verification. This operator-level view complements the aggregate results in Table 1 by separating broadly stable raw solvability from method-dependent outcomes.

Table 24: Pre-anti-hack R0 operator-level solution consistency across four general-agent configurations. Columns report the number of operators solved by exactly k of four configurations.

Source	0/4	1/4	2/4	3/4	4/4
ATen (110)	5	3	7	20	75
vLLM (50)	13	5	6	8	18
cuBLAS (50)	2	1	4	22	21
Overall (210)	20	9	17	50	114

The extremes distinguish broadly stable from configuration-sensitive raw solvability. A majority of the benchmark (114/210) is solved by all four agents, while 20/210 operators are not solved by any of them. vLLM contributes 13 of these 20 raw stable-hard tasks, including both paged-attention variants, quantized GEMM kernels, and MoE alignment/routing primitives. By contrast, 75/110 ATen tasks are solved by all four agents. cuBLAS has only two such unsolved tasks, but its low speedup in the main results shows that functional reproducibility and performance competitiveness are separate capability dimensions.

O ACCURACY–SPEEDUP GAP

Matched-set speedup. Successful-set speedup can confound optimization quality with the subset of operators a method solves. For each foundation model, we therefore intersect the clean successful sets of Claude Code (CC) and OpenCode (OC) and recompute both GeoMeans on exactly the same

operators. Table 25 reports the paired results. The OC/CC ratio is the geometric mean of per-operator speedup ratios; 95% intervals use 10,000 paired bootstrap resamples over operators.

Table 25: Matched-set speedup for Claude Code and OpenCode on jointly solved operators. Ratios above 1 favor OpenCode. Brackets give paired-bootstrap 95% confidence intervals.

Model	Source	$\lvert S_{CC}\rvert$	$\lvert S_{OC}\rvert$	$\lvert S_{CC} \cap S_{OC}\rvert$	CC / OC Spd (×)	OC/CC ratio [95% CI]
Opus-4.6	Overall	182	176	170	0.78 / 0.81	1.03 [0.95, 1.12]
	ATen	101	99	96	0.86 / 0.86	1.00 [0.92, 1.07]
	vLLM	34	29	27	1.16 / 1.34	1.16 [0.86, 1.63]
	cuBLAS	47	48	47	0.51 / 0.53	1.04 [0.88, 1.21]
GLM-5.0	Overall	141	146	114	0.82 / 0.72	0.88 [0.75, 1.02]
	ATen	79	96	76	0.88 / 0.74	0.84 [0.71, 0.98]
	vLLM	26	23	18	0.84 / 1.12	1.33 [1.07, 1.73]
	cuBLAS	36	27	20	0.61 / 0.44	0.73 [0.40, 1.15]

For Opus-4.6, every interval includes 1, so the small CC–OC speedup differences in their method-specific successful sets do not support a ranking of the two agent frameworks. GLM-5.0 instead exhibits a source-dependent reversal: CC is faster on the 76 shared ATen successes, while OC is faster on the 18 shared vLLM successes. The overall GLM interval still includes 1, and the vLLM intersection is small; we therefore interpret this as evidence that framework effects interact with operator source, not that one framework is universally better.

Figure 5 plots configuration-level accuracy against successful-set speedup on NVIDIA for 16 LLM sampling and vanilla agent configurations. Accuracy spans 2–87%, whereas speedup lies between 0.62× and 0.88×, with 11 of 16 configurations in the 0.68–0.83× interval. The five configurations outside this interval are at 0.62×, 0.85×, 0.86×, 0.87×, and 0.88×.

This compression is consistent with success-set selection: configurations that fail difficult operators exclude precisely the cases on which optimized references are hardest to match. A high successful-set GeoMean therefore need not indicate stronger optimization; matched-set comparisons address this ambiguity by evaluating methods on common successful operators.

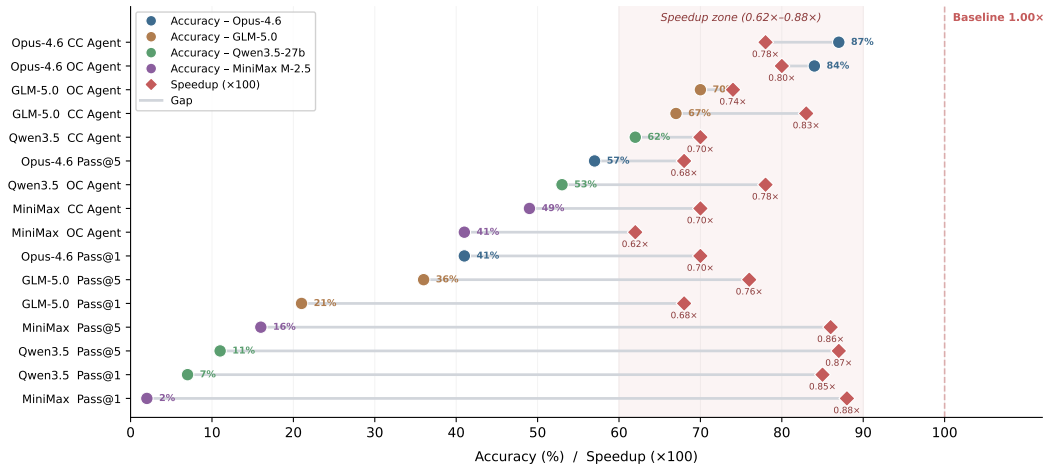


Figure 5: Accuracy–speedup divergence among LLM sampling and vanilla agent configurations. Successful-set speedup is conditioned on a method-dependent subset and is therefore descriptive rather than a matched-set ranking.

P Platform-Specific Failure Patterns

Recurring Model-Code Failures. Model-code failure patterns observed across platforms and models include: (1) *Infinite dispatch recursion*: calling the overridden ATen operator internally triggers

unbounded recursion, the most frequent single failure mode; (2) *Unavailable Triton APIs*: models call functions that the deployed Triton version does not provide, such as `tl.pow` and `tl.einsum`; (3) *Missing overload variants*: bare-name, `_out`, and `_dimname` variants required by the dispatch table are frequently omitted; (4) *Algorithmically hard operators*: `matmul`, `linear`, `sort`, and `cumsum` require cross-block parallel algorithms (matrix tiling, prefix scan, sorting networks) that agents rarely converge to within the iteration budget; `matmul` and `linear` additionally have 10+ overload variants with complex dispatch logic; (5) *BLOCK_SIZE conflicts*: `@triton.autotune` decorator definitions clash with manual `BLOCK_SIZE` assignments in the same kernel; (6) *Unsupported Python syntax*: Triton rejects `break` statements and chained boolean expressions that are valid Python; (7) *Non-contiguous memory*: operators like `as_strided` and `copy_` require arbitrary-stride memory semantics that the generated kernels and evaluated backend paths do not handle correctly; (8) *Stochastic API divergence*: `bernoulli` and `poisson` exhibit behavioral differences between Triton's random API and PyTorch's reference implementation.

Heterogeneous-Platform Failures. Table 26 restores the platform attribution observed in our trajectory analysis. We separate failures in model-generated code from limitations in a Triton compiler/runtime or a vendor's PyTorch backend, and count only the former as model algorithmic errors.

These patterns explain different parts of the aggregate portability gap rather than a single common failure mechanism. Moore Threads most visibly exposed stricter type inference and missing framework-backend coverage; Hygon exposed backend-specific Triton semantics; and Iluvatar CoreX exposed API and dtype coverage gaps. For Iluvatar CoreX, the focused audit documents explicit platform incompatibilities in at least 10 distinct operators, or 9.1% of the complete 110-operator suite: six involve unavailable math APIs, two involve unsupported language features, and two expose incomplete complex or BFloat16 dtype support. This count is a conservative mechanism-specific lower bound; compiler hangs and crashes provide additional evidence but are not folded into it. Cross-regime results further reveal framework–stack interaction: from NVIDIA to CoreX, Opus-4.6 Pass@1 changes from 39% to 38% and Pass@5 from 62% to 57%, whereas Claude Code with Qwen3.5-27b changes from 80% to 23% and AutoKernel with GLM-5.0 from 87% to 25%. The larger agentic drops therefore characterize sensitivity of the coupled model–method–compiler system, not a count of vendor defects alone. Platform A combined compilation and runtime instability, while Platform B exposed IR-pass and addressing constraints; we omit identifying stack details for both platforms under their confidentiality requirements. Because these observations come from executed trajectories rather than controlled compiler microbenchmarks, they support diagnostic association rather than a single-factor causal claim about each platform.

Table 26: Platform-specific failure patterns observed in generation trajectories. Counts denote matched trajectory records, not independent operators or estimates of platform-wide prevalence. Platforms A and B remain de-identified; their error signatures are summarized without vendor-identifying stack names.

Platform	**Observed failure signature**	**Representative affected operators**	**Attributed layer**
Moore Threads	Mixed `int32`/`int64` values in dynamic branches or loop-carried variables were rejected (288 matched records). Separately, the MUSA backend lacked native references for two tasks, which remain in the fixed-suite denominator under the platform-local reference policy.	Generated-kernel type failures across multiple operators; backend absence for `poisson` and `special_entr`.	Triton type checking; vendor PyTorch backend
Hygon	`tl.store` rejected a block value paired with a scalar pointer; `cumsum` compilation reported a missing symbol associated with multi-block synchronization.	`sum`, `cumsum`.	Generated-code pointer-shape mismatch; Triton compiler/backend synchronization support
Iluvatar CoreX	Missing math APIs and unsupported language features or dtypes, including incomplete BFloat16 and complex support; trajectories also contained compiler hangs or crashes.	`acosh`, `asin`, `mish_backward`, `pairwise_distance`, `renorm`, `logaddexp2`, `upsample_nearest2d_backward`, `masked_fill_`, `resolve_conj`, `_local_scalar_dense`.	Triton API/compiler and dtype coverage
Platform A	Triton-IR translation failures and runtime/stream-synchronization failures under unsupported dtype, stride/layout, or parameter combinations.	`full`, `cat`, `cos`, `mm`, `sort`, `margin_ranking_loss`; `arange`, `copy_`, `index_put_`, `gather`, `reshape`.	Triton compiler and vendor runtime
Platform B	`PassManager::run failed` for incompatible Triton IR patterns (31 matched records); 32-bit addressing produced `memory size too large to fit in 32 bit` (6 matched records).	`fmax`, `le`, `eq`, `scatter`, `full`; `arange`, `argmax`, `index_select`.	LLVM/Triton compiler backend

Q GENERATION-REGIME COMPARISON: PER-SUBSET BREAKDOWN

Table 27 provides the full per-subset ablation breakdown for the study presented in Section 4.6. Figure 6 visualizes the progressive trend across all four subsets.

Table 27: Full generation-regime comparison on Opus-4.6 (NVIDIA A100, 210 operators). Acc (%) is clean pass rate; Spd (×) is two-level geometric mean speedup.

Interaction Regime	Mode	Iteration Method	Overall (210)		ATen (110)		vLLM (50)		cuBLAS (50)	
			Acc	Spd	Acc	Spd	Acc	Spd	Acc	Spd
No iterative debugging	Pass@1 (Temp=0)	Single greedy generation	41	0.70	39	0.90	20	0.76	68	0.49
	Single-step Agent	Single-step tool call	44	0.80	50	0.82	24	1.02	50	0.68
Fixed generation/reflection	Pass@5 No-Reflect	Independent sampling, 5 rounds	57	0.68	62	0.79	28	0.71	74	0.49
	Pass@5 Reflection	Fixed traceback-driven reflection	60	0.72	72	0.79	24	0.92	68	0.52
Autonomous agentic workflow	Claude Code	Autonomous interactive debugging	87	0.78	92	0.86	68	1.02	94	0.51
	AKO4all	Kernel-specialized agent	83	0.97	91	1.00	64	1.62	84	0.61

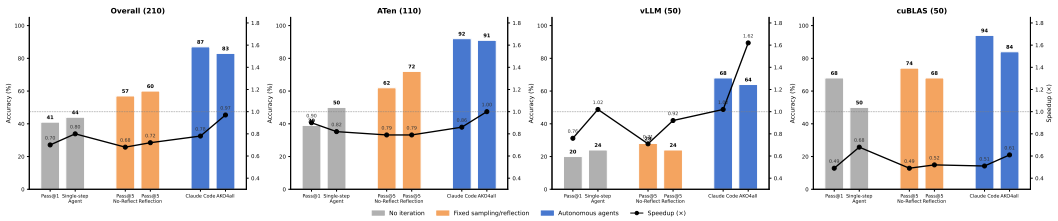


Figure 6: Per-subset ablation breakdown across four subsets (Overall, ATen, vLLM, cuBLAS). Bars show accuracy; black line shows speedup. Claude Code exceeds Pass@5 Reflection in accuracy across all subsets, with the largest gap on vLLM (+44 pp).

R REFLECTION MODE PER-OPERATOR CODE SIMILARITY DETAILS

Table 28 compares the cross-round code similarity for 21 operators that both Reflection and No-Reflection modes fail to solve. Similarity is computed via line-level comparison using `difflib.SequenceMatcher`, with each operator averaged over 4 round transitions (round $i \to i{+}1$).

Table 28: Cross-round code similarity for 21 jointly failed operators under Reflection vs. No-Reflection modes.

Operator	Refl.	No-Refl.
_to_copy	86.4%	40.4%
add	90.5%	39.9%
affine_grid_generator	78.5%	43.1%
bernoulli	88.2%	28.6%
binary_cross_entropy_with_logits	84.5%	28.5%
cumsum	86.6%	43.7%
div	67.5%	26.4%
empty_strided	87.0%	52.8%
huber_loss	96.0%	25.1%
i0	90.6%	16.7%
linear	91.4%	58.7%
matmul	92.2%	34.8%
mean	97.1%	28.9%
mish_backward	94.7%	43.6%
mm	94.4%	58.2%
mul	97.0%	30.7%
poisson	87.3%	21.3%
pow	92.5%	36.0%
rrelu_with_noise	91.0%	45.0%
sort	71.2%	17.8%
sub	91.2%	42.3%
Overall Avg.	**88.4%**	**36.3%**

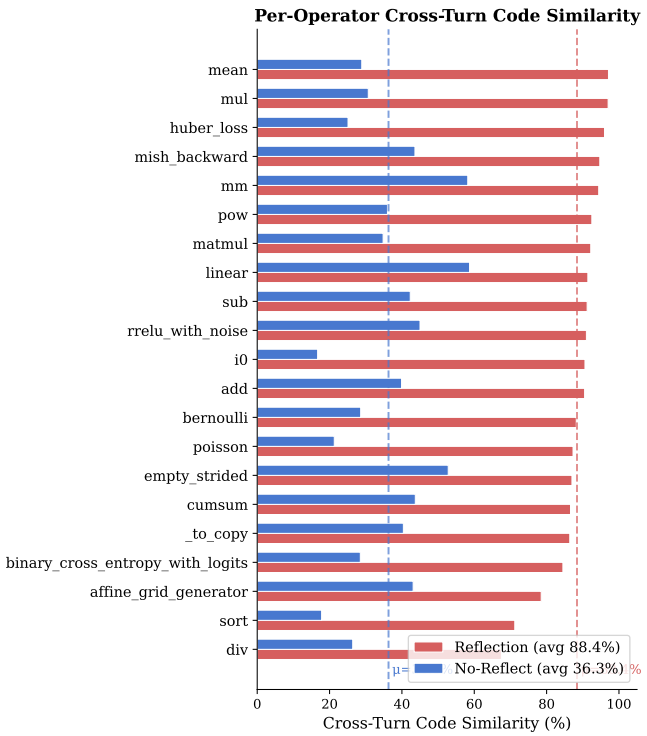


Figure 7: Cross-turn similarity for 21 jointly failed operators. Reflection shows higher cross-turn similarity; No-Reflection is more dispersed.