PTXBENCH: INTERPRETABLE EVALUATION OF LLMS FOR ARCHITECTURE-SPECIFIC GPU PROGRAMMING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reaching peak performance on modern GPUs requires specialized hardware-accelerated mechanisms. Writing kernels that use them is therefore a key capability for LLMs. Yet correctness and speedup alone cannot show whether a generated kernel uses the intended hardware mechanisms. We propose instruction-qualified speedup, which counts a kernel's speedup only if the kernel is correct and executes selected tensor-compute or data-movement instructions. Our framework, PTXBench, measures this metric while controlling four conditions: supplied architectural knowledge, execution feedback, harness capabilities, and generation budgets. Ablations show that supplied architectural knowledge and execution feedback improve instruction-qualified speedup. Using a shared multi-turn refinement harness, we evaluate five recent LLMs on one GEMM and four attention workloads across NVIDIA H100 and B200. Correctness is not enough. Even with full knowledge and execution feedback, 25.9% of trajectories that produce a correct kernel never produce an instruction-qualified one. In all ten GPU-workload settings, the fastest CUDA–PTX kernels execute both accelerated compute and data movement. Still, no model's median instruction-qualified speedup reaches the vendor-library reference. Under the same budget, evaluated LLMs reach higher median instruction-qualified speedups with Triton than with CUDA–PTX, even without Triton's autotuning.

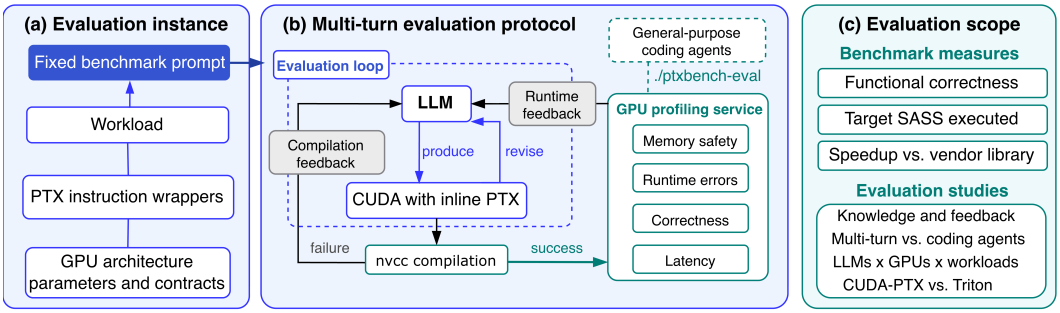

Figure 1: PTXBench workflow. (a) The model receives the workload and architectural knowledge. (b) It generates kernels and receives execution feedback. (c) We measure correctness, speed, and instruction execution separately.

1 INTRODUCTION

Recent NVIDIA architectures expose hardware-accelerated mechanisms for tensor computation and data movement (NVIDIA Corporation, 2023; 2025a). Efficient kernels require coordinating these mechanisms with appropriate data layouts and execution schedules (Zhao et al., 2025). High-level kernel languages such as Triton aim for performance portability (Tillet et al., 2019), but their compilers require continued engineering to support new hardware mechanisms (Zhou et al., 2026; Chen et al., 2026) and validation across a broad range of programs and configurations (Driscoll et al., 2025). Large language models (LLMs) offer a complementary route: they could optimize and validate a kernel directly for a given workload and GPU.

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.

In this paper, we study architecture-specific GPU programming: the ability to explicitly select and coordinate architecture-specific mechanisms to implement correct and efficient GPU kernels. We test this capability through generating CUDA kernels that contain inline PTX instructions, which we call CUDA–PTX. PTX (Parallel Thread Execution), NVIDIA's virtual instruction set, provides explicit control over the compute and data-movement operations studied here (NVIDIA Corporation, 2026c). Using LLMs for kernel optimization in CUDA–PTX is gaining interest (INT21, 2026; Intent Lab, 2026). Recent work such as Hawkeye (Tschand et al., 2026) uses expert-authored unit tests as guidance to help coding agents implement architecture-specific optimizations. We focus on making assessments of this capability interpretable. We verify that kernels execute the target hardware mechanisms, and we fix the conditions under which kernels are generated.

We call a kernel *instruction-qualified* if it is correct and executes targeted hardware-accelerated instructions at runtime. This separates kernels that use the targeted mechanisms from kernels that compute the same output without them. An LLM's architecture-specific GPU programming capability can thus be measured by its *instruction-qualified speedup*. However, it is sensitive to the prompt, environment, and harness. Therefore, PTXBench measures instruction-qualified speedup while controlling four conditions: supplied architectural knowledge, execution feedback, generation budgets, and harness capabilities.

These controls define what an evaluation tests. Supplying PTX instruction wrappers and GPU architecture contracts tests whether a model can apply documented interfaces. Withholding them also tests the model's prior knowledge. Execution feedback is often treated as essential for kernel refinement (Zhang et al., 2025). Yet providing this feedback takes substantial engineering and slows evaluation, motivating us to measure how much it actually helps. The harness determines which tools a model can use and how much work it can do between evaluation requests. Fixing these conditions makes model comparisons interpretable.

We apply PTXBench to evaluate five recent LLMs on NVIDIA H100 and B200 GPUs with GEMM and four attention workloads: forward and backward, each with and without a causal mask. These workloads are central to modern LLMs, so their underlying mathematics should be familiar to the evaluated models. This choice helps isolate hardware scheduling from algorithmic rewrites. For each GPU architecture, we define a separate set of target instructions for hardware-accelerated tensor computation and data movement. Each selected workload requires the use of these instructions to achieve vendor-library performance (Thakkar et al., 2023; Shah et al., 2024; Dao, 2026). We compare against well-optimized cuBLAS and cuDNN implementations. We also compare CUDA–PTX with Triton, to see how outcomes change when the compiler handles more of the hardware-specific work. This paper makes three contributions:

- **Instruction-qualified speedup.** To evaluate LLMs' capability of architecture-specific GPU programming, we count a kernel's speedup only if the kernel is correct and executes targeted tensor-compute or data-movement instructions at runtime. This shows whether a fast kernel actually uses the specialized hardware, which speedup alone cannot show.
- **Generation conditions shape measured capability.** More architectural knowledge raises instruction-qualified speedup. Execution feedback raises the qualification rate and usually improves speed. Those gains vary by workload. More complex harnesses do not always help: multi-turn refinement has higher median instruction-qualified speedups on all attention workloads, while coding agents do on GEMM. Evaluations should therefore report and fix these conditions.
- **Using the right instructions is not enough to match experts.** In every setting, the fastest correct CUDA–PTX kernel executes both accelerated compute and data movement. Yet no model's median instruction-qualified speedup reaches the vendor library. Evaluated LLMs also reach higher median speedups with Triton than with CUDA–PTX, even without autotuning. Current LLMs still benefit substantially from compiler support.

2 Hardware Background

PTX and SASS. PTX is the lowest level a CUDA programmer can explicitly control. CUDA–PTX lets the model specify hardware-accelerated compute and data-movement operations directly through inline PTX. The compiler lowers CUDA–PTX to the native assembly instruction set called Streaming Assembly (SASS) (NVIDIA Corporation, 2026a). Programmers cannot write SASS di-

Table 1: PTX interfaces and SASS execution criteria for the selected BF16 mechanisms.

Mechanism	Indicator	PTX interface	SASS execution criterion
Tensor compute	C_{Gen4}	`wgmma.mma_async`	$N_{\text{HGMMA}} > 0$
	C_{Gen5}	`tcgen05.mma`, `tcgen05.ld`	$(N_{\text{UTCHMMA}} > 0) \wedge (N_{\text{LDTM}} > 0 \vee N_{\text{LDT}} > 0)$
Data movement	D_{TMA}	`cp.async.bulk.tensor`	$N_{\text{UTMALDG}} > 0 \vee N_{\text{UTMASTG}} > 0$
	D_{TMEM}	`tcgen05.cp`, `tcgen05.st`	$N_{\text{UTCCP}} > 0 \vee N_{\text{STTM}} > 0 \vee N_{\text{STT}} > 0$

rectly, but NVIDIA's binary utilities expose it in parsable form, and Nsight Compute records per-instruction execution counts. Since SASS is what actually executes on the hardware, it is the right level at which to verify instruction qualification. SASS also provides a common basis for verification across kernel languages that lower to PTX, such as Triton.

Hardware-accelerated Mechanisms. Hopper exposes warpgroup matrix multiplication through `wgmma.mma_async` and tensor transfers through `cp.async.bulk.tensor`. Blackwell adds fifth-generation tensor-compute interfaces (`tcgen05`) and tensor memory (TMEM). These need their own compute, transfer, and readback rules. Figure 2 illustrates the selected paths. The two generations of GPUs use different hardware-accelerated mechanisms, so they test whether our evaluation protocol carries across generations.

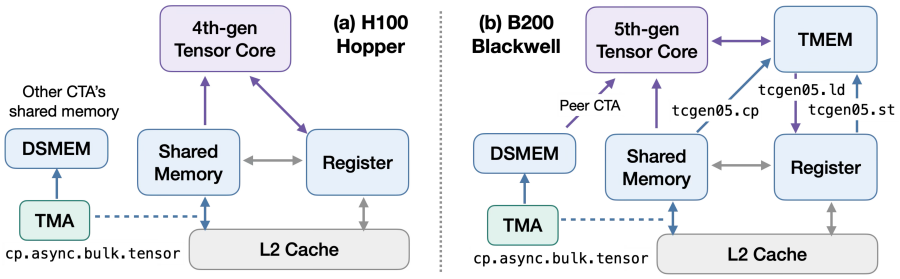

Figure 2: Tensor-compute and data-movement paths on (a) H100 (Hopper) and (b) B200 (Blackwell). Purple arrows indicate tensor-compute paths, including TMEM readback; blue arrows indicate specialized data movement; gray arrows indicate conventional memory transfers. TMEM denotes tensor memory; DSMEM denotes distributed shared memory.

3 PTXBench: Measurement and Controlled Protocol

Evaluating mechanism use needs two things: verifying that target instructions run, and fixing the conditions under which kernels are generated. PTXBench does both (Figure 1). We first define how target-instruction execution is measured and used to qualify kernels (Section 3.1). We then describe the knowledge supplied to models (Section 3.2), the refinement protocol (Section 3.3) and correctness and timing (Section 3.4).

3.1 Measuring Target Instruction Execution

Runtime evidence. Table 1 maps PTX interfaces to SASS instructions that must execute. We replay each correct kernel on the original workload under Nsight Compute (NCU), with an NVTX range around one kernel invocation. Warmup runs happen outside this range. We sum instruction-level counters over all kernels launched inside the range. N_s is the thread-level, predicate-enabled execution count for instruction family s (`thread_inst_executed_true`). $N_s > 0$ shows that s executed. It does not show that s contributed to the output; Sections 4.4 and A.3 check contribution.

Qualification rule. Let C denote accelerated tensor compute and D denote selected data movement. The architecture-level predicates are

$$\text{H100:} \quad C = C_{\text{Gen4}}, \quad D = D_{\text{TMA}},$$
$$\text{B200:} \quad C = C_{\text{Gen5}}, \quad D = D_{\text{TMA}} \vee D_{\text{TMEM}}.$$

On B200, tensor compute requires both a tensor-core MMA and a readback from TMEM (`LDTM` or `LDT`). We count readback as compute because it is the only way to access tensor-core results instead of an optional data-movement optimization. We exclude cache-control and prefetch instructions from D because they do not move data into on-chip buffers. A correct kernel is instruction-qualified (*Inst.*) if it satisfies $C \vee D$ and *non-Inst.* if it satisfies $\neg C \wedge \neg D$. A trajectory's instruction-qualified speedup is the best speedup among its Inst. kernels, or zero if it has none.

3.2 Supplied Architectural Knowledge

Knowledge in the prompt decides which capability an evaluation measures. PTXBench uses three knowledge components. Architecture parameters give the GPU specification, such as the size and bandwidth of each memory level. PTX instruction wrappers are CUDA functions that wrap hardware-accelerated PTX instructions and include debugging hints. Architecture contracts document the semantics that the wrappers do not show, such as data layouts and memory consistency.

We combine these components into three nested conditions. Each condition adds one component, so it relies less on the model's prior knowledge. With parameters alone, the model must recall both the interfaces and the semantics of the target instructions. Adding wrappers supplies the interfaces, but the model must still know their semantics. Adding contracts supplies the semantics as well. The evaluation then tests whether the model can apply and coordinate documented interfaces. Because each condition measures a different capability, PTXBench fixes and reports the condition used.

Scheduling guidance can also change results. We therefore prepare three scheduling-guidance variants per workload: one with knowledge only and two with workload-specific guidance (Table 12). We extract this guidance from existing materials, and a human expert validated it (Section A.6). As Figure 6 will show, no variant is best for every workload or model.

3.3 Multi-turn Refinement Protocol

The refinement loop performs execution-guided refinement by retaining previously generated kernels together with their execution feedback. This loop forms the core of many kernel-generation agents (Cao et al., 2026; Zhang et al., 2026; Hong et al., 2025). At each iteration, a GPU profiling service compiles the candidate kernel, verifies its correctness and memory safety, and measures its latency. The service returns compilation and sanitizer errors, numerical correctness results, and latency and speedup measurements. If a kernel exceeds the time limit, for example because of a naive loop structure or incorrect synchronization, the service returns a timeout error. If the source lacks some target PTX instructions, the service also warns and names them, without describing their use.

A generation budget limits the number of attempts to produce a candidate kernel; an evaluation budget limits candidate submissions to the GPU profiling service. Each turn of multi-turn refinement is one generation attempt, and a trajectory records the sequence. We count model calls and observed evaluations separately because some model calls may not return extractable kernels.

3.4 Correctness and Performance

A kernel is correct if it compiles, passes Compute Sanitizer memcheck, and passes numerical validation. Kernels that include cuBLAS or cuDNN headers are marked incorrect regardless of execution results. We follow FlashInfer-Bench for numerical checks and kernel timing (Xing et al., 2026). Speedup is reference latency divided by kernel latency. Section A.2 gives the checking and timing configuration.

4 Evaluation: Protocol Controls and Applications

We first test PTXBench's controls on H100: supplied knowledge (Section 4.1), execution feedback (Section 4.2), and multi-turn refinement vs. coding agents (Section 4.3). We then apply PTXBench

Table 2: Gemini 3.1 Pro H100 best speedups under three knowledge conditions. Inst. and non-Inst. results are selected independently within each trajectory.

Knowledge condition	Metric	GEMM	MHA-Fwd	MHA-Fwd-Causal	MHA-Bwd	MHA-Bwd-Causal
Architecture parameters	non-Inst.	0.26 (0.00–0.51)	0.06 (0.00–0.11)	0.05 (0.00–0.29)	0.11 (0.00–0.27)	0.07 (0.00–0.15)
	Inst.	– (0.00–0.09)	–	–	–	–
+ PTX instruction wrappers	non-Inst.	–	–	–	–	– (0.00–0.06)
	Inst.	0.62 (0.44–0.90)	0.41 (0.00–0.56)	0.24 (0.00–0.50)	0.26 (0.00–0.54)	0.29 (0.00–0.50)
++ Architecture contracts	non-Inst.	–	–	–	–	– (0.00–0.06)
	Inst.	**0.76** (0.44–0.96)	**0.54** (0.37–0.73)	**0.47** (0.00–0.77)	**0.27** (0.00–0.51)	**0.30** (0.00–0.64)

on H100 and B200 to compare five models (Section 4.4) and to compare kernel languages with different levels of compiler support (Section 4.5).

Experiment settings. We instantiate PTXBench on one BF16 GEMM and four BF16 attention variants including forward and backward. Reaching vendor-library performance on these workloads requires coordination of tensor operations and asynchronous data movement; H100 and B200 expose distinct interfaces for these mechanisms. We use cuBLAS for GEMM and cuDNN for attention (NVIDIA Corporation, 2026d; Chetlur et al., 2014). Section A.2 describes the evaluation procedure and software versions. Fwd denotes MHA forward with log-sum-exp (LSE) output, Bwd denotes backward attention, and Causal denotes a lower-triangular mask. Unless stated otherwise, tables report the median and observed minimum–maximum of per-trajectory best speedups in each setting, using groups for the feedback study. "–" before a range denotes zero median; a bare "–" denotes zero maximum. Figures 3, 5, 7 and 8 report median best instruction-qualified speedup with pointwise 95% bootstrap intervals. Section A.1 gives specifications and study settings, and defines the common aggregation rule.

4.1 Supplied Architectural Knowledge

For Gemini 3.1 Pro on H100, with eight generation attempts per trajectory, the fraction of trajectories producing at least one instruction-qualified candidate rises from 1.7% with architecture parameters alone to 80.0% with PTX instruction wrappers and 91.7% with architecture contracts. Parameters alone yield positive median non-instruction-qualified speedups on all five workloads but zero instruction-qualified ones. Without wrappers, the model falls back to conventional CUDA or the synchronous WMMA API (`wmma::mma_sync`) that does not use the Hopper-specific asynchronous `wgmma` path, so its prior knowledge of these PTX interfaces is not enough to use them. This holds even though the evaluator's warning names some missing target instructions. Adding architecture contracts to wrappers further increases median instruction-qualified speedup on each workload (Table 2). Once wrappers are supplied, median non-Inst. speedup drops to zero on every workload, suggesting that the model switches to the target instructions. These results show that **supplied knowledge changes what the evaluation measures, so it should be reported and fixed when comparing models.**

4.2 Execution Feedback

Execution feedback is costly to build and slows evaluation. We therefore measure what it adds. We compare refinement with execution feedback, refinement without execution feedback, and independent sampling for Gemini 3.1 Pro on H100, each with a generation budget of eight attempts. Both refinement conditions retain prior candidates; without execution feedback, evaluator observations are replaced with "`Revise the previous kernels to improve correctness and performance.`" Independent sampling selects the best of eight separate generations.

Using refinement with execution feedback, 91.7% of groups produce an Inst. candidate, compared with 70.0% for blind refinement and 63.3% for independent sampling. Refinement with execution feedback also achieves higher median best instruction-qualified speedups than both alternatives on four of the five workloads (Table 3). On MHA-Bwd, blind refinement without execution feedback is slightly higher, although both refinement conditions round to $0.27\times$. Blind refinement still helps where efficient instruction-qualified kernels are hard (both backward workloads), but it trails independent sampling on GEMM and forward attention. Blind revision can therefore hurt performance.

Table 3: Gemini 3.1 Pro H100 best instruction-qualified speedups under three feedback conditions. For the refinement conditions, each trajectory forms one group. For independent sampling, we group eight single-turn trajectories in their original order and select the best result.

Feedback condition	GEMM	MHA-Fwd	MHA-Fwd-Causal	MHA-Bwd	MHA-Bwd-Causal
Refinement with execution feedback	**0.76** (0.44–0.96)	**0.54** (0.37–0.73)	**0.47** (0.00–0.77)	**0.27** (0.00–0.51)	**0.30** (0.00–0.64)
Refinement without execution feedback	0.50 (0.00–0.82)	0.38 (0.00–0.60)	0.32 (0.00–0.64)	**0.27** (0.00–0.54)	0.16 (0.00–0.63)
Independent sampling	0.62 (0.34–0.78)	0.39 (0.00–0.66)	0.34 (0.00–0.63)	– (0.00–0.48)	– (0.00–0.44)

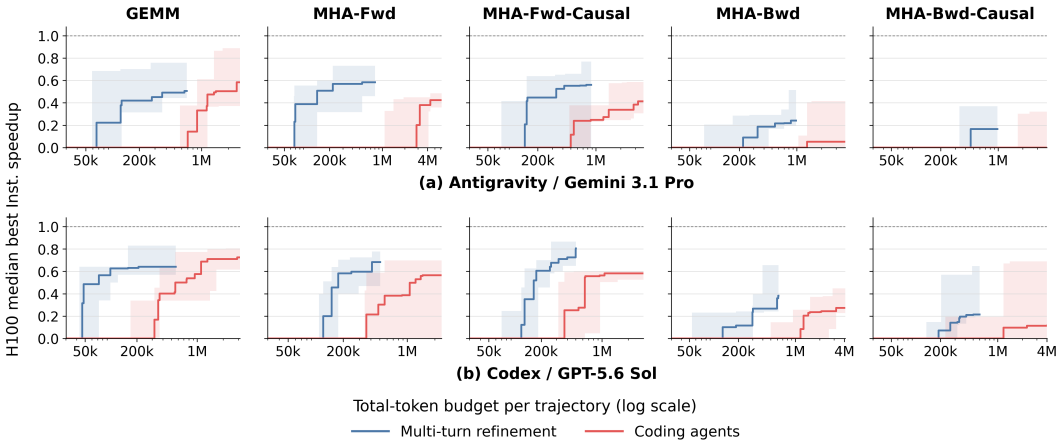

Figure 3: Coding agents versus multi-turn refinement on H100. Lines show median best instruction-qualified speedup across all trajectories per method at shared total-token budgets.

Execution feedback increases the instruction-qualification rate and generally improves performance, so it should be provided and held fixed when comparing models.

4.3 Multi-turn Refinement vs. Coding Agents

We examine whether multi-turn refinement is competitive with coding agents at the same evaluation budget, and compare token use. This study fixes evaluation requests instead of generation attempts, because coding agents have no clear turn boundaries.

We compare Antigravity with Gemini 3.1 Pro and Codex with GPT-5.6 Sol against multi-turn refinement using the same respective models and model settings on H100. Each method has four trajectories per workload, each completing eight evaluation requests. All harnesses receive the same task specification and full knowledge, without additional scheduling guidance, and use the same feedback format. Coding agents submit candidates through a command-line client `./ptxbench-eval` which connects to the same GPU profiling service as multi-turn refinement. The comparison therefore holds the GPU evaluation budget fixed while allowing model calls and tool usage to differ across harnesses (Section A.4).

Figure 3 shows that coding agents achieve higher median best instruction-qualified speedups on GEMM for both models, while multi-turn refinement achieves higher medians on all four attention workloads at the final budget. Coding agents use 3.32–5.78× as many total tokens as multi-turn refinement (per-workload medians). Under these evaluated conditions, **multi-turn refinement achieves competitive instruction-qualified outcomes with lower token usage, supporting its use as a common harness for the subsequent model comparison.**

4.4 Measuring Architecture-Specific GPU Programming Capability

As a first application, we compare five models on H100 and B200. All use full architectural knowledge, execution feedback, and multi-turn refinement with up to eight attempts.

Table 4: Correct and instruction-qualified outcomes. Inst. requires correct execution and runtime evidence for $C \vee D$; Δ counts correct non-Inst. trajectories.

	H100 ($n = 300$)			B200 ($n = 300$)			All ($n = 600$)		
Speedup threshold	Correct	Inst.	Δ	Correct	Inst.	Δ	Correct	Inst.	Δ
Any positive speedup	239	205	34	209	127	82	448	332	116
$\geq 0.25\times$	186	178	8	68	68	0	254	246	8
$\geq 0.50\times$	121	121	0	38	38	0	159	159	0
$\geq 1.00\times$	0	0	0	2	2	0	2	2	0
Correct but no Inst. / correct	34/239 (14.2%)			82/209 (39.2%)			116/448 (25.9%)		

Table 4 shows that 25.9% of trajectories with correct kernels produce no instruction-qualified kernel. The share is higher on B200 (39.2%) than H100 (14.2%). These fallbacks are correct but slow: the number of correct non-Inst. trajectories drops from 116 to 8 at $0.25\times$ and to 0 at $0.50\times$.

Figure 4 breaks this down by setting and mechanism. The gap is uneven across workloads: on B200 attention backward, most correct candidates execute neither mechanism (the darkest cells). At the top, the picture is consistent. In all ten settings, the fastest kernel executes both mechanisms ($C \wedge D$) and is strictly faster than the best kernel in every other category, and every correct kernel at or above $0.50\times$ executes at least one target family.

Target-instruction execution, even with high speed, does not prove that the instructions contribute to the output. We check output contribution with Codex source inspection and targeted human review (details in Section A.3). A positive judgment requires an explicit path from target-produced data to a required output. After automated and human audit, every best Inst. candidate (332/332) has at least one contributing target family, and all 300 $C \wedge D$ candidates, including the ten performance leaders, have both.

Table 5 compares models using per-trajectory best instruction-qualified speedup, pooled across scheduling-guidance variants. Claude Opus 4.8 has the highest median on GEMM for both GPUs and both H100 forward-attention workloads; GPT-5.6 Sol leads the remaining settings. No model leads across workloads. Claude reaches $0.92\times$ on H100 GEMM but only $0.03\times$ on H100 MHA-Bwd-Causal, where GPT-5.6 Sol reaches $0.41\times$, so a single-workload evaluation would misrank models. Rankings are a secondary output: allowing non-Inst. kernels leaves the leader unchanged in all ten settings, because the fastest kernels already use the target instructions (Figure 4). What qualification adds is the diagnosis above, which correctness and speedup scoring cannot produce.

Knowledge cutoff may matter. Gemini 3.1 Pro's cutoff is about the same month as the Blackwell PTX release (Table 10). Its GEMM median drops from $0.76\times$ on H100 to $0.51\times$ on B200. Claude Opus 4.8 and GPT-5.6 Sol have cutoffs 12–13 months after that release, and they stay near their H100 levels ($0.92\times \rightarrow 0.94\times$ and $0.68\times \rightarrow 0.64\times$). SWE-bench Pro scores do not predict these results. Gemini has the lowest SWE-bench Pro score of the five models, yet it beats GLM-5.2 and Qwen3.8-27B on every H100 workload. We treat these as observations, not trends.

No model's median instruction-qualified speedup reaches its vendor-library reference, exposing a remaining gap to efficient architecture-specific GPU programming. Despite demonstrated mechanism use and supporting evidence of output contribution, the highest median is $0.94\times$ (Claude, B200 GEMM), and the best median on every B200 attention workload is at most $0.28\times$.

4.5 Comparing Kernel Languages

As a second application, we use PTXBench to compare instruction-qualified outcomes across kernel languages with different levels of compiler assistance. CUDA–PTX requires explicit selection and coordination of hardware instructions, while Triton delegates lower-level implementation decisions to its compiler. We also adapt supplied knowledge and scheduling guidance for Triton.

We evaluate CUDA–PTX, Triton, and Triton without autotuning for Gemini 3.1 Pro and GPT-5.6 Sol on H100 and B200. Triton's autotuner can try many configurations within one program, and a

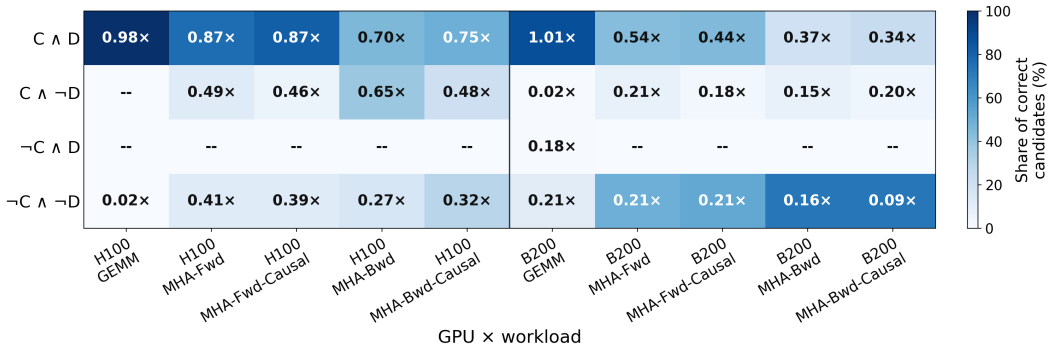

Figure 4: Runtime compute/data categories among correct CUDA–PTX candidates pooled across all five models. Color gives the share in each architecture–workload setting; annotations give the best recorded speedup in each category. Dashes indicate no candidates. Failed attempts are excluded.

Table 5: Five-model CUDA–PTX instruction-qualified speedup evaluation. Note that most Qwen trajectories ended early at the context limit.

GPU	Model	GEMM	MHA-Fwd	MHA-Fwd-Causal	MHA-Bwd	MHA-Bwd-Causal
H100	Gemini 3.1 Pro	0.76 (0.44–0.96)	0.54 (0.37–0.73)	0.47 (0.00–0.77)	0.27 (0.00–0.51)	0.30 (0.00–0.64)
	Claude Opus 4.8	**0.92** (0.89–0.98)	**0.75** (0.37–0.84)	**0.71** (0.38–0.81)	0.25 (0.09–0.49)	0.03 (0.00–0.50)
	GPT-5.6 Sol	0.68 (0.47–0.87)	0.71 (0.33–0.87)	0.66 (0.42–0.87)	**0.59** (0.24–0.70)	**0.41** (0.15–0.75)
	GLM-5.2	0.51 (0.37–0.69)	– (0.00–0.61)	– (0.00–0.53)	– (0.00–0.44)	–
	Qwen3.8-27B	0.60 (0.00–0.96)	– (0.00–0.53)	– (0.00–0.66)	– (0.00–0.43)	– (0.00–0.21)
B200	Gemini 3.1 Pro	0.51 (0.00–0.89)	0.09 (0.00–0.28)	– (0.00–0.25)	– (0.00–0.20)	–
	Claude Opus 4.8	**0.94** (0.85–1.01)	0.10 (0.00–0.30)	0.12 (0.00–0.27)	– (0.00–0.15)	–
	GPT-5.6 Sol	0.64 (0.49–0.82)	**0.25** (0.06–0.54)	**0.28** (0.12–0.44)	**0.16** (0.00–0.37)	**0.17** (0.00–0.34)
	GLM-5.2	0.27 (0.00–0.63)	– (0.00–0.03)	– (0.00–0.10)	–	–
	Qwen3.8-27B	0.16 (0.00–0.77)	–	– (0.00–0.32)	–	–

generation budget does not cap this search. We therefore add a condition that forbids autotuning, enforced by source checks. We qualify Triton kernels with the same SASS criteria as CUDA–PTX.

Both Triton conditions have higher median instruction-qualified speedup than CUDA–PTX in all 20 model-GPU-workload settings as shown in Table 6, where "Triton (w/o tune)" is short for Triton without autotuning. CUDA–PTX's best kernel sometimes beats one Triton condition on H100, but never both. Figures 5 and 8 plot medians against reasoning tokens, which separates reasoning from program length. At every budget, both Triton conditions are at least as high as CUDA–PTX. Triton without autotuning often matches or beats Triton with autotuning, especially for Gemini. **Evaluated LLMs achieve higher median instruction-qualified speedups with Triton than with CUDA–PTX. The advantage persists without autotuning, so it does not depend on the autotuner.**

5 Related Work

GPU kernel generation benchmarks. Various kernel benchmarks (Ouyang et al., 2025; Saroufim et al., 2025; Li et al., 2025; Wang et al., 2026; Zhu et al., 2026; Guan et al., 2026; Wang et al., 2025; Zang et al., 2026; Oliaro et al., 2026; Xing et al., 2026) score generated kernels by correctness and speedup across different task collections, kernel languages, and deployment settings. Other work strengthens the correctness (Lange et al., 2025; Driscoll et al., 2025). PTXBench addresses a different gap. A kernel can be correct and fast without using the hardware mechanisms a workload is meant to exercise. PTXBench therefore adds runtime SASS evidence of target-instruction execution to the score, and it controls the conditions under which kernels are generated.

LLM systems for GPU kernel generation. A parallel line of work builds methods that produce better GPU kernels (Wang et al., 2025; Li et al., 2026; Du et al., 2026a;b). Closest to our setting,

Table 6: Best instruction-qualified speedups for CUDA–PTX and two Triton conditions.

GPU	Model	Language and autotuning	GEMM	MHA-Fwd	MHA-Fwd-Causal	MHA-Bwd	MHA-Bwd-Causal
H100	Gemini 3.1 Pro	CUDA–PTX	0.76 (0.44–0.96)	0.54 (0.37–0.73)	0.47 (0.00–0.77)	0.27 (0.00–0.51)	0.30 (0.00–0.64)
		Triton	**0.98** (0.96–0.99)	0.82 (0.79–0.86)	0.75 (0.73–0.79)	0.60 (0.49–0.74)	0.47 (0.33–0.73)
		Triton (w/o tune)	**0.98** (0.94–0.99)	**0.83** (0.77–0.85)	**0.78** (0.69–0.84)	**0.67** (0.44–0.78)	**0.55** (0.29–0.76)
	GPT-5.6 Sol	CUDA–PTX	0.68 (0.47–0.87)	0.71 (0.33–0.87)	0.66 (0.42–0.87)	0.59 (0.24–0.70)	0.41 (0.15–0.75)
		Triton	0.98 (0.97–0.99)	0.81 (0.75–0.97)	**0.81** (0.67–0.85)	**0.71** (0.46–0.84)	0.53 (0.07–0.73)
		Triton (w/o tune)	**0.99** (0.98–1.00)	**0.84** (0.76–0.94)	0.78 (0.71–0.92)	**0.71** (0.53–0.83)	**0.57** (0.00–0.81)
B200	Gemini 3.1 Pro	CUDA–PTX	0.51 (0.00–0.89)	0.09 (0.00–0.28)	– (0.00–0.25)	– (0.00–0.20)	–
		Triton	0.85 (0.76–0.94)	0.41 (0.38–0.52)	0.38 (0.29–0.51)	**0.44** (0.27–0.51)	0.29 (0.18–0.44)
		Triton (w/o tune)	**0.88** (0.80–0.97)	**0.42** (0.40–0.54)	**0.41** (0.32–0.55)	**0.44** (0.26–0.52)	**0.37** (0.18–0.43)
	GPT-5.6 Sol	CUDA–PTX	0.64 (0.49–0.82)	0.25 (0.06–0.54)	0.28 (0.12–0.44)	0.16 (0.00–0.37)	0.17 (0.00–0.34)
		Triton	**0.96** (0.84–0.98)	**0.58** (0.42–0.72)	**0.45** (0.35–0.61)	**0.43** (0.32–0.63)	**0.40** (0.00–0.49)
		Triton (w/o tune)	0.89 (0.80–0.98)	0.41 (0.39–0.74)	0.42 (0.37–0.68)	0.40 (0.29–0.53)	0.38 (0.00–0.53)

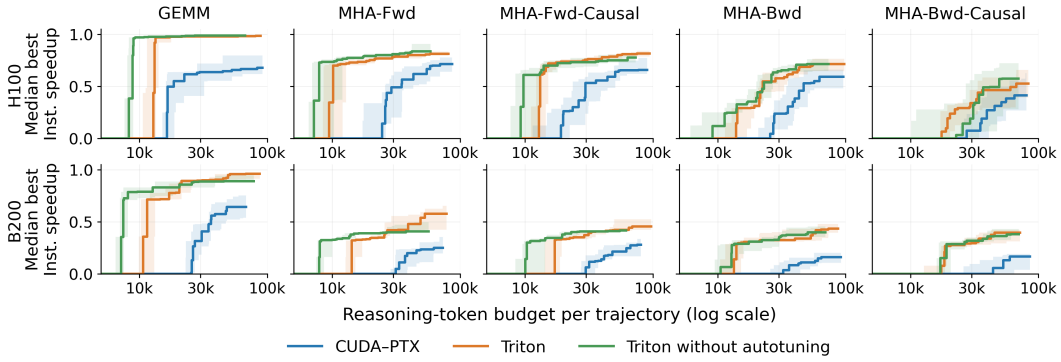

Figure 5: GPT-5.6 Sol median best instruction-qualified speedup at shared reasoning-token budgets.

Hawkeye (Tschand et al., 2026) guides coding agents toward architecture-specific optimizations with expert-authored unit tests that demonstrate hardware features, similar in role to our PTX wrappers. These systems treat supplied knowledge, feedback, and harness design as choices to optimize, and they report correctness and speedup. PTXBench takes the evaluation view instead. It fixes these choices as controlled conditions, measures how each one shifts outcomes (Sections 4.1 to 4.3), and counts speedup only when the targeted mechanisms execute. This lets such methods be compared on whether their gains come from the intended hardware.

6 Conclusion and Limitations

Conclusion. We introduced instruction-qualified speedup to evaluate LLMs' ability to generate efficient kernels that correctly execute the targeted hardware-accelerated instructions. PTXBench measures this capability under explicit controls on supplied architectural knowledge, execution feedback, generation budgets, and harness capabilities. Our ablations show that supplied knowledge and execution feedback improve instruction-qualified speedup. Multi-turn refinement is already competitive with more complex coding agents. Evaluations should therefore report and fix these conditions. Across five LLMs, two GPUs, and five workloads, the fastest correct CUDA–PTX kernels use both accelerated compute and data movement. Yet no model's median reaches the vendor library. With the same budget, Triton gives higher medians than CUDA–PTX, even without autotuning, consistent with the continued benefits of compiler support. Together, PTXBench provides a basis for interpretable measurement of LLMs' architecture-specific GPU programming capability.

Limitations. Our study covers only BF16 GEMM and attention on two NVIDIA GPUs, so conclusions may not transfer to other precisions, operators, or vendors' accelerated instructions, and observations such as knowledge-cutoff effects remain suggestive rather than conclusive. Runtime counts show that target instructions execute, but not how much each contributes to speedup.

AI DISCLOSURE

Generative AI tools assisted software implementation and debugging, experimental analysis, literature review, reference formatting, and manuscript drafting and revision. The authors are responsible for reviewing the final text, code, analysis, and artifacts.

REPRODUCIBILITY STATEMENT

The anonymized repository provides the prompts, run configurations, workload definitions, evaluator and profiler code, archived candidate kernels, and recorded outcomes. Its result map links experimental figures and tables to their runs; the included scripts verify kernel source hashes and regenerate the data-bearing figures, tables, and calculated claims from retained results. Repeating model generation and GPU profiling requires access to the evaluated models and H100 and B200 GPUs. https://anonymous.4open.science/r/ptxbench-submission-D66C.

REFERENCES

Anthropic. Claude Opus 4.8. https://www.anthropic.com/news/claude-opus-4-8, May 2026. Released May 28, 2026; knowledge cutoff January 2026.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-Search: LLM kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026. URL https://arxiv.org/abs/2602.19128.

Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, et al. Tawa: Automatic warp specialization for modern GPUs with asynchronous references. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 255–267. IEEE, 2026.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pp. 184–197, 2015.

Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. https://tridao.me/blog/2026/flash4/, March 2026.

Benjamin Driscoll, Kshitij Dubey, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. Equivalence checking of ML GPU kernels. *arXiv preprint arXiv:2511.12638*, 2025. URL https://arxiv.org/abs/2511.12638.

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, Yining Li, et al. Kernel-Smith: A unified recipe for evolutionary kernel optimization. *arXiv preprint arXiv:2603.28342*, 2026a. URL https://arxiv.org/abs/2603.28342.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre Wang He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, et al. AdaExplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation. *arXiv preprint arXiv:2604.16625*, 2026b.

Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 228–239, 2003.

Google DeepMind. Gemini 3.1 Pro. https://deepmind.google/models/model-cards/gemini-3-1-pro/, February 2026. Published February 19, 2026. The January 2025 cutoff for the Gemini 3 family is documented at https://ai.google.dev/gemini-api/docs/gemini-3.

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 *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, Seoul, South Korea, 2026. PMLR. URL https://icml.cc/virtual/2026/poster/65003.

Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. Autocomp: A powerful and portable code optimizer for tensor accelerators. *arXiv preprint arXiv:2505.18574*, 2025. URL https://arxiv.org/abs/2505.18574.

INT21. Inference Engine Factory, 2026. URL https://int21.ai/products/inference-engine-factory/. Accessed: 2026-09-20.

Intent Lab. Turn your intent into production systems, 2026. URL https://intentlab.ai/blog/turn-your-intent-into-production-systems. Accessed: 2026-09-20.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025. URL https://arxiv.org/abs/2509.14279.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1183. URL https://aclanthology.org/2025.findings-acl.1183/.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=igZItUbY6n.

NVIDIA Corporation. *Parallel Thread Execution ISA, Version 8.0*. NVIDIA Corporation, December 2022. URL https://docs.nvidia.com/cuda/archive/12.0.0/pdf/ptx_isa_8.0.pdf. Released with CUDA Toolkit 12.0 in December 2022.

NVIDIA Corporation. *NVIDIA Hopper Tuning Guide*. NVIDIA Corporation, 2023. URL https://docs.nvidia.com/cuda/hopper-tuning-guide/. Accessed: 2026-07-29.

NVIDIA Corporation. *NVIDIA Blackwell Tuning Guide*. NVIDIA Corporation, 2025a. URL https://docs.nvidia.com/cuda/blackwell-tuning-guide/. Accessed: 2026-07-29.

NVIDIA Corporation. *Parallel Thread Execution ISA, Version 8.7*. NVIDIA Corporation, January 2025b. URL https://docs.nvidia.com/cuda/archive/12.8.0/pdf/ptx_isa_8.7.pdf. Released with CUDA Toolkit 12.8.

NVIDIA Corporation. *CUDA Binary Utilities*. NVIDIA Corporation, 2026a. URL https://docs.nvidia.com/cuda/cuda-binary-utilities/. Accessed: 2026-08-15.

NVIDIA Corporation. *CUDA Profiling Tools Interface (CUPTI)*. NVIDIA Corporation, 2026b. URL https://docs.nvidia.com/cupti/. Accessed: 2026-08-10.

NVIDIA Corporation. *Parallel Thread Execution ISA*. NVIDIA Corporation, 2026c. URL https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2026-09-19; historical build targets are documented separately.

NVIDIA Corporation. *cuBLAS Library*. NVIDIA Corporation, 2026d. URL https://docs.nvidia.com/cuda/cublas/. CUDA Toolkit Documentation.

Gabriele Oliaro, Yichao Fu, May Jiang, Owen Lu, Junli Wang, Zhihao Jia, Hao Zhang, and Samyam Rajbhandari. FastKernels: Benchmarking GPU kernel generation in production. *arXiv preprint arXiv:2605.23215*, 2026. URL https://arxiv.org/abs/2605.23215.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, July 2026. Released July 9, 2026. The February 16, 2026 knowledge cutoff is documented at https://developers.openai.com/api/docs/models/gpt-5.6-sol.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Qwen Team. Qwen3.8-27B model card. https://huggingface.co/Qwen/Qwen3.8-27B, August 2026.

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. GitHub repository, 2025. URL https://github.com/meta-pytorch/BackendBench.

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

Stuart Sul and Christopher Ré. ThunderKittens 2.0: Even faster kernels for your GPUs, February 2026. URL https://hazyresearch.stanford.edu/blog/2026-02-19-tk-2. Hazy Research blog, accessed 2026-07-29.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Aditya Atluri, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS. https://github.com/NVIDIA/cutlass, January 2023. CUDA C++ template abstractions for high-performance matrix multiplication and related computations.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Arya Tschand, Kesavan Ramakrishnan, Alexander Ingare, Simon Guo, Jeffrey Jian Ma, Zishen Wan, Simran Arora, Azalia Mirhoseini, and Vijay Janapa Reddi. Hawkeye: Hardware-aware GPU kernel optimization with minimal supervision. In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026. URL https://openreview.net/forum?id=e3pxJbBRBk.

Simon Veitner. SBO and LBO explained visually, April 2026. URL https://veitner.bearblog.dev/sbo-and-lbo-explained-visually/. Accessed: 2026-07-29.

Han Wang, Jintao Zhang, Kai Jiang, Haoxu Wang, Jianfei Chen, and Jun Zhu. KernelBenchX: A comprehensive benchmark for evaluating LLM-generated GPU kernels. *arXiv preprint arXiv:2605.04956*, 2026. URL https://arxiv.org/abs/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 and evaluation benchmarks. *arXiv preprint arXiv:2507.23194*, 2025. URL https://arxiv.org/abs/2507.23194.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie F. 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. In A. Chowdhery and Z. Jia (eds.), *Proceedings of Machine Learning and Systems*, volume 8, pp. 2016–2064. MLSys, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/37e44c4b5321605735be9761f9b758fc-Paper-Conference.pdf.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *Proceedings of Machine Learning and Systems*, 8:912–926, 2026.

Z.ai. GLM-5.2. https://z.ai/blog/glm-5.2, June 2026. Released June 16, 2026; knowledge cutoff not publicly disclosed.

Peiyu Zang, Jian Tao, Jialing Zhang, Yichen Yuan, Wentao Zhang, Guang Liu, and Yonghua Lin. KernelGenBench: A multi-source and multi-chip benchmark for LLM-based kernel generation. *arXiv preprint arXiv:2607.27231*, 2026. URL https://arxiv.org/abs/2607.27231.

Genghan Zhang, Weixin Liang, Olivia Hsu, and Kunle Olukotun. Adaptive self-improvement LLM agentic system for ML library development. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 75427–75452. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/zhang25at.html.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In A. Chowdhery and Z. Jia (eds.), *Proceedings of Machine Learning and Systems*, volume 8, pp. 541–568. MLSys, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/0f8426558905746fc38da5e335700aec-Paper-Conference.pdf.

Chenggang Zhao, Zhean Xu, Liang Zhao, Jiashi Li, Chenhao Xu, Anyi Xu, Shengyu Liu, Kexing Zhou, and Kuai Yu. DeepGEMM: clean and efficient BLAS kernel library on GPU. https://github.com/deepseek-ai/DeepGEMM, 2025.

Size Zheng, Xuegui Zheng, Hanshi Sun, Qi Hou, Wenlei Bao, Shiyu Li, Haojie Duanmu, Jin Fang, Chenli Xue, Chenhui Huang, et al. Ditron: Distributed multi-level tiling compiler for parallel tensor programs. In *Forty-third International Conference on Machine Learning*, 2026.

Keren Zhou, Mario Lezcano-Casado, Adam P Goucher, Akhmed Rakhmati, Jeff Niu, Justin Lebar, Pawel Szczerbuk, Peter Bell, Phil Tillet, Thomas Raoux, et al. Linear layouts: robust code generation of efficient tensor computation using $\mathbb{F}_2$. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 132–146, 2026.

Jiace Zhu, Wentao Chen, Qi Fan, Zhixing Ren, Junying Wu, Xing Zhe Chai, Chotiwit Rungrueangwutthinon, Yehan Ma, and An Zou. CUDABench: Benchmarking LLMs for text-to-CUDA generation. *arXiv preprint arXiv:2603.02236*, 2026. URL https://arxiv.org/abs/2603.02236.

A APPENDIX

A.1 EXPERIMENT SETUP

Populations and shared baselines. Table 7 summarizes the study populations and generation and evaluation budgets. The instruction analysis and five-model comparison in Section 4.4, together with the supplementary refinement curves, use the same primary CUDA–PTX corpus of 600 trajectories. The full knowledge condition in Section 4.1 and the execution-feedback condition in Section 4.2 share the same Gemini/H100 trajectories from this corpus. The CUDA–PTX baselines in Section 4.5 reuse the corresponding Gemini 3.1 Pro and GPT-5.6 Sol trajectories on H100 and B200.

Table 7: Study populations and generation and evaluation budgets. Shared baselines are included in each study's counts.

Study	Study design	Population	Budget
Knowledge	Gemini/H100; 3 knowledge conditions × 5 workloads × 3 scheduling-guidance variants × 4 experimental repeats	180 trajectories; 60 per condition	Generation: 8 attempts per trajectory
Feedback	Gemini/H100; 3 feedback conditions × 5 workloads × 3 scheduling-guidance variants × 4 experimental repeats	180 groups; 60 per condition	Generation: 8 attempts per group
Coding agents	2 models × 2 methods × 5 workloads × 4 experimental repeats	40 agent and 40 multi-turn refinement trajectories	Evaluation: 8 requests per trajectory
Primary CUDA–PTX	5 models × 2 GPUs × 5 workloads × 3 scheduling-guidance variants × 4 experimental repeats	600 trajectories	Generation: 8 attempts per trajectory
Language	2 models × 2 GPUs × 5 workloads × 3 language and autotuning conditions × 3 scheduling-guidance variants × 4 experimental repeats	720 trajectories	Generation: 8 attempts per trajectory

Trajectory completion. The 600 trajectories in primary CUDA–PTX experiments allow up to eight generation attempts each and contain 4,423 saved responses: 4,368 extracted candidates with evaluation records and 55 extraction failures, including formatting failures. 102 of 120 Qwen trajectories reach the context limit early so 377 turns do not have responses. All other trajectories complete eight responses.

Workloads. Table 8 specifies the five workloads. All attention workloads use batch size $b = 4$, head count $h = 48$, sequence length $n = 4096$, and head dimension $d = 128$. In the table, $s = (b, h, n, d)$ and $r = (b, h, n)$; L is the FP32 log-sum-exp output. Forward and backward attention each have non-causal and causal variants; the latter apply a lower-triangular mask. Section A.2 describes input sampling, retained state, output initialization, and timing boundaries.

Table 8: Workload inputs and outputs. Attention shapes s and r are defined in the text.

Workload	Inputs	Outputs
GEMM, $C = AB^\top$	BF16 A: 8192×5120; BF16 B: 7168×5120	BF16 C: 8192×7168
MHA-Fwd / MHA-Fwd-Causal	BF16 Q, K, V: shape s	BF16 O: shape s; FP32 L: shape r
MHA-Bwd / MHA-Bwd-Causal	BF16 Q, K, V, O, dO: shape s; FP32 L: shape r	BF16 dQ, dK, dV: shape s

LLM inference. Inference configurations are model-specific (Table 9). We do not override temperature, top-p, or top-k except for Qwen3.8-27B which we serve locally with SGLang. For serving

Qwen3.8-27B, we use the recommended configuration which sets `preserve_thinking=True` by default.

Table 9: Model-specific inference settings. Gemini uses the same settings across all studies.

Model	Sampling, thinking, and provider settings	Output token budget per call
Qwen3.8-27B	Temperature 1.0; top-p 0.95; top-k 20; presence penalty 1.5; context limit 262,144 tokens	81,920
GLM-5.2	OpenRouter, restricted to `z-ai/fp8` or `fireworks`	131,072
Gemini 3.1 Pro	Default thinking setting	65,536
Claude Opus 4.8	Adaptive thinking, `xhigh` reasoning effort setting	128,000
GPT-5.6 Sol	Adaptive thinking, `xhigh` reasoning effort setting	128,000

Token-budget aggregation. For Figures 3, 5, 7 and 8, methods share the union of recorded cumulative token checkpoints within each panel. Checkpoints correspond to completed responses or, in the harness comparison, evaluation requests. At each budget, each trajectory contributes its best instruction-qualified speedup available at an affordable checkpoint, or zero if none qualifies. Token costs include failed responses and instruction-unqualified candidates. The median includes all trajectories in the setting, and shorter trajectories continue to contribute their final best result. Bands are pointwise 95% percentile intervals from 10,000 whole-trajectory bootstrap resamples, reusing the sampled trajectories across budgets and stratifying by scheduling-guidance variant where applicable. Each curve and band stop at that method's largest recorded token consumption. Each application specifies its token unit, sample size, and stratification.

Model metadata. Table 10 summarizes the evaluated models' reported release dates, knowledge cutoffs, and SWE-bench Pro scores (Google DeepMind, 2026; Anthropic, 2026; OpenAI, 2026; Z.ai, 2026; Qwen Team, 2026). Monthly lags are measured from the Hopper PTX ISA 8.0 and Blackwell PTX ISA 8.7 releases (NVIDIA Corporation, 2022; 2025b); release dates provide only an upper bound when a cutoff is undisclosed.

Table 10: Reported metadata for the evaluated models.

Model	**SWE-bench Pro (%)**	**Model release**	**Knowledge cutoff**	**Knowledge-cutoff lag after PTX release (months)**	
				Hopper	**Blackwell**
Gemini 3.1 Pro	54.2	Feb. 2026	Jan. 2025	25	≈ 0
Claude Opus 4.8	69.2	May 2026	Jan. 2026	37	12
GPT-5.6 Sol	64.6	July 2026	Feb. 2026	38	13
GLM-5.2	62.1	June 2026	Not disclosed	≤ 42	≤ 17
Qwen3.8-27B	61.7	Aug. 2026	Not disclosed	≤ 44	≤ 19

A.2 Kernel Evaluation Details

Numerical acceptance. The checker verifies output shapes and data types and rejects non-finite outputs. Kernel and reference output values are converted to `float32`. For kernel output value x and reference value y, an element fails only if both $|x-y| > 10^{-2}$ and $|x-y|/(|y|+10^{-8}) > 10^{-2}$. Every element of every output must pass in all three trials.

Inputs and reference results. The evaluator prepares three workload trials. GEMM and forward-attention inputs are sampled from a normal distribution. Backward attention uses stored Q, K, V, O, L and generates a new upstream gradient dO for each trial. Reference outputs, reference latencies, and trial inputs are cached on the GPU for reuse across generated kernels. Each kernel receives copies of the inputs and reuses them for correctness and timing measurements.

Output initialization and timing. For kernels that write into supplied output buffers, the evaluator allocates uninitialized storage before timing. Any initialization required by the kernel must occur inside its invocation. Kernels that return tensors manage their own output storage. Kernel latency includes all device operations launched during the invocation, including auxiliary kernels and output initialization. Input generation, reference preparation, compilation, and NCU instruction profiling are performed separately.

The timing procedure flushes L2 before each timing iteration and excludes the flush from the measured interval. CUPTI (NVIDIA Corporation, 2026b) measures from the first device activity's start to the last activity's end, including gaps between operations. CUDA graphs are disabled. Each workload trial includes setup calls, ten warmup iterations, and 50 timing iterations. The three trial medians are averaged separately for the kernel and reference.

Execution checks and software versions. The evaluator checks compilation, source restrictions, memory safety, and numerical correctness, and provides debugging feedback when available. It runs `memcheck` up to twice, stopping at the first detected issue or timeout. The profiling service can recover after a kernel crashes or hangs. Only kernels that can compile and run are included in instruction classification. Experiments use H100 80 GB and B200 GPUs; the reported library versions are cuBLAS 13.1.0 and cuDNN 9.20.0.

A.3 Instruction-Dependency Audit

Population and method. We inspected the best instruction-qualified candidate from each of the 332 successful primary CUDA–PTX trajectories: 300 executing both target families and 32 executing compute only. The 300 include all ten architecture–workload performance leaders. Codex (GPT-6 Astra, `xhigh`) traced producer-to-output paths in the complete source using workload contracts, retained runtime evidence, and supplied wrapper context. Generating-model identities, speedups, and source comments were withheld. A fixed rubric records separate compute (C) and data-movement (D) judgments with source-line citations.

Contribution criterion. In the original source audit, a positive judgment requires an explicit path from target-produced data to a required output; execution, shared barriers, or address dependence alone are insufficient. Each family is labeled *yes*, *no*, *unresolved*, or *not observed*, keeping ambiguous mappings separate from established non-contribution. A positive label requires at least one contributing site.

Review and findings. The original judgments support at least one contributing family in 330/332 selected kernels and both families in 297/300 kernels executing both (two with no contributing family, one with a single contributing family). We also select 69 kernels for a second review using the same Codex configuration, blinded to the first judgments. The 69 kernels are composed of 60 kernels randomly sampled from 332, the 8 performance leaders not already in the random sample, and the only correct D-only kernel in the corpus, which is outside of 332 kernels because it is not the best in its trajectory. Both runs agreed on contribution labels in all 69 selected cases. Targeted human review adjudicated contribution in the three remaining B200 GEMM cases using source paths, matching correctness records, retained runtime execution, and the absence of an alternative output producer. With this review, the totals are 332/332 and 300/300. This review does not quantify how much each instruction contributes to speedup. Such effects are hard to isolate: the costs of overlapping compute and memory events interact rather than add (Fields et al., 2003), and profiles that show where time is spent do not predict the gain from optimizing that code (Curtsinger & Berger, 2015). Attributing speedup to specific mechanisms therefore requires controlled ablations like those in FlashAttention-3 (Shah et al., 2024), which we leave to future work.

A.4 Coding-Agent Settings and Token Usage

The coding agents use the same models and model settings as their corresponding multi-turn refinement baselines: Gemini 3.1 Pro for Antigravity and GPT-5.6 Sol for Codex. They also share the task specification and supplied architectural knowledge. Architecture parameters and contracts are given as a markdown file. PTX wrappers are given as header files. No scheduling guidance is

included. The task prompt contains references to these materials. Each evaluation request submits one candidate kernel. All trajectories complete the eight requests included in Figure 3.

Antigravity has file-editing tools and access to the evaluator, with web and external-network tools disabled. Codex has shell and file-editing tools, with web search and network access disabled. These tools let agents inspect and revise files between evaluation requests.

For both coding agents and their multi-turn refinement baselines, total token usage includes all reported input and output tokens up to each evaluation request. Cached input is counted once, and output already includes reasoning.

Figure 3 applies the common aggregation rule as in Section A.1 to total-token budgets over all four trajectories per method and model–workload setting, without scheduling-guidance variants. Each panel ends at the larger method endpoint. For these four-trajectory samples, the bootstrap interval limits coincide with the observed minimum and maximum at every checkpoint.

A.5 Triton Source Checks and PyTorch Use

For the Triton without autotuning condition, source checks enforce the restriction on runtime configuration search while permitting explicit launch parameters and deterministic choices based on tensor metadata. No candidate among the 1,920 failed these checks.

PyTorch initialization is allowed and included in kernel timing. Triton host wrappers use PyTorch for device management, tensor views, and descriptor-buffer allocation. Nine instruction-qualified candidates call `Tensor.zero_()` for output initialization. Two are the best qualifying candidates in their trajectories: Gemini's B200 implementations without autotuning for MHA-Bwd and MHA-Bwd-Causal, at $0.402\times$ and $0.256\times$, respectively. Their attention arithmetic remains implemented in Triton. Source inspection found no direct PyTorch matrix-multiplication, softmax, or attention calls.

A.6 Supplied Architectural Knowledge and Scheduling Guidance

Supplied architectural knowledge components. Table 11 reports Qwen3.8-27B token counts for the architecture parameters, architecture contracts, and PTX instruction wrappers included in the H100 and B200 prompts. B200 prompts are longer because they include additional contracts such as descriptors for shared memory, peer CTAs, and tensor memory. Hardware-accelerated PTX instructions are unevenly documented and sometimes underspecified (Veitner, 2026) or even wrong (Sul & Ré, 2026). The full knowledge condition supplies architecture parameters, PTX instruction wrappers, and architecture contracts. Some wrappers are adapted from Ditron (Zheng et al., 2026); comments retain polymorphic instruction definitions while the wrapper demonstrates one representative instance.

Table 11: Token counts for supplied architectural knowledge components.

GPU	Component	Tokens
H100	Architecture parameters	259
	PTX instruction wrappers	18,596
	Architecture contracts	3,454
	Total	22,309
B200	Architecture parameters	413
	PTX instruction wrappers	18,871
	Architecture contracts	9,531
	Total	28,815

Scheduling-guidance variants. Each scheduling-guidance variant in Table 12 has four experimental repeats. The Triton variants apply to both autotuning conditions. We extract scheduling guidance from Zheng et al. (2026); Shah et al. (2024); Zadouri et al. (2026) and Triton tutorials.

One expert with 5 years of GPU kernel optimization experience reviewed each variant for correctness and fidelity to its source.

Table 12: Supplied scheduling guidance. Each row also includes a variant with no additional scheduling guidance. Attention rows cover both causal and non-causal workloads. FA3 and FA4 denote FlashAttention-3 and FlashAttention-4.

GPU	Workload	Guided variant 1	Guided variant 2
CUDA–PTX			
H100	GEMM	Cluster multicast loads	Warp specialization
	MHA-Fwd	FA3 warp specialization and ping-pong scheduling	Same guidance plus intra-warpgroup GEMM–softmax overlap
	MHA-Bwd	FA3 separate dQ and dK/dV kernels	Same backward guidance plus forward scheduling guidance
B200	GEMM	Cluster multicast loads	TMA stores through shared memory
	MHA-Fwd	FA4 ping-pong pipeline with software exponential approximation	Same pipeline without software exponential approximation
	MHA-Bwd	FA4 guidance covering one- and two-CTA schedules	FA4 one-CTA schedule
Triton			
H100	GEMM	L2-aware grouped tile ordering	Persistent tile scheduling
	MHA-Fwd	FA3 warp specialization and ping-pong scheduling	Same guidance plus intra-warpgroup GEMM–softmax overlap
	MHA-Bwd	FA3 separate dQ and dK/dV kernels	Same backward guidance plus forward scheduling guidance
B200	GEMM	L2-aware grouped tile ordering	Persistent GEMM with tensor descriptors and compiler warp specialization
	MHA-Fwd	Streaming online softmax with additional non-GEMM refinements	Streaming online softmax
	MHA-Bwd	Separate ownership of dQ and dK/dV tiles	One program per query tile, with local dQ and accumulated dK/dV

One H100 forward variant with intra-warpgroup overlap includes corrected FlashAttention-3 pseudocode. Section A.7 presents the published and corrected versions and explains the correction.

A.7 Published and corrected FlashAttention-3 pseudocode

The published FlashAttention-3 Algorithm 2 (Shah et al., 2024) has a short loop bound, omits promotion of the next probability state, and leaves the final probability and rescaling undefined. It is therefore undefined as a literal program. The corrected pseudocode iterates through the last score tile, promotes both pipeline states, saves the old row maximum before rescaling the accumulated numerator, and divides by the denominator in the epilogue. The listings below restate the published and corrected algorithms.

```
Require: Q_i ∈ R^{B_r × d} and K, V ∈ R^{N × d} in HBM; key block size B_c and T_c = ⌈N/B_c⌉.
Reallocate registers as a function of the number of consumer warps.
On chip, initialize O_i = 0 and ℓ_i, m_i = 0, −∞.
Wait for Q_i and K_0 in shared memory.
Compute S_cur = Q_i K_0^⊤ using WGMMA. Commit and wait.
Release stage 0 of the buffer for K.
Compute m_i, P̃_cur, and ℓ_i from S_cur, and rescale O_i.
for 1 ≤ j < T_c − 1 do
  Wait for K_j in shared memory.
  Compute S_next = Q_i K_j^⊤ using WGMMA. Commit; do not wait.
  Wait for V_{j−1} in shared memory.
  Compute O_i = O_i + P̃_cur V_{j−1} using WGMMA. Commit; do not wait.
  Wait for the WGMMA Q_i K_j^⊤.
  Compute m_i, P̃_next, and ℓ_i from S_next.
  Wait for the WGMMA P̃_cur V_{j−1}; then rescale O_i.
```

```
  Release stages (j mod s) and ((j − 1) mod s) for K and V, respectively.
  Copy S_next to S_cur.
end for
Wait for V_{T_c−1} in shared memory.
Compute O_i = O_i + P̃_last V_{T_c−1} using WGMMA. Commit and wait.
Epilogue: Rescale O_i based on m_i. Compute L_i from m_i and ℓ_i; write O_i, L_i to HBM.
```

Listing 1: FlashAttention-3 Algorithm 2 as published. The loop bound and missing probability-state update make the pseudocode internally inconsistent.

```
Require: Q_i ∈ ℝ^{B_r×d} and K, V ∈ ℝ^{N×d} in HBM; key block size B_c and T_c = ⌈N/B_c⌉.
Reallocate registers as a function of the number of consumer warps.
On chip, initialize O_i = 0 and ℓ_i, m_i = 0, −∞.
Wait for Q_i and K_0 in shared memory.
Compute S_cur = Q_i K_0^⊤ using WGMMA. Commit and wait.
Release stage 0 of the buffer for K.
Compute m_i, P̃_cur, and ℓ_i from S_cur.
for 1 ≤ j < T_c do
  Wait for K_j in shared memory.
  Compute S_next = Q_i K_j^⊤ using WGMMA. Commit; do not wait.
  Wait for V_{j−1} in shared memory.
  Compute O_i = O_i + P̃_cur V_{j−1} using WGMMA. Commit; do not wait.
  Wait for the WGMMA Q_i K_j^⊤.
  Save m_i^old ← m_i.
  Update m_i and ℓ_i online; compute P̃_next = exp(S_next − m_i).
  Wait for the WGMMA P̃_cur V_{j−1}; then set O_i = diag(exp(m_i^old − m_i)) O_i.
  Release stages (j mod s) and ((j − 1) mod s) for K and V, respectively.
  Copy S_next to S_cur and P̃_next to P̃_cur.
end for
Wait for V_{T_c−1} in shared memory.
Compute O_i = O_i + P̃_cur V_{T_c−1} using WGMMA. Commit and wait.
Epilogue: Set O_i = diag(ℓ_i)^{−1} O_i and L_i = m_i + log(ℓ_i); write O_i, L_i to HBM.
```

Listing 2: Corrected FlashAttention-3 pseudocode supplied in the prompt.

A.8 Supplementary Refinement Progress

In Table 5, the pooled leader leads under every scheduling-guidance variant except in the two H100 forward-attention settings, where Claude's margin over GPT-5.6 Sol is smallest (0.04× and 0.05×; Figure 6).

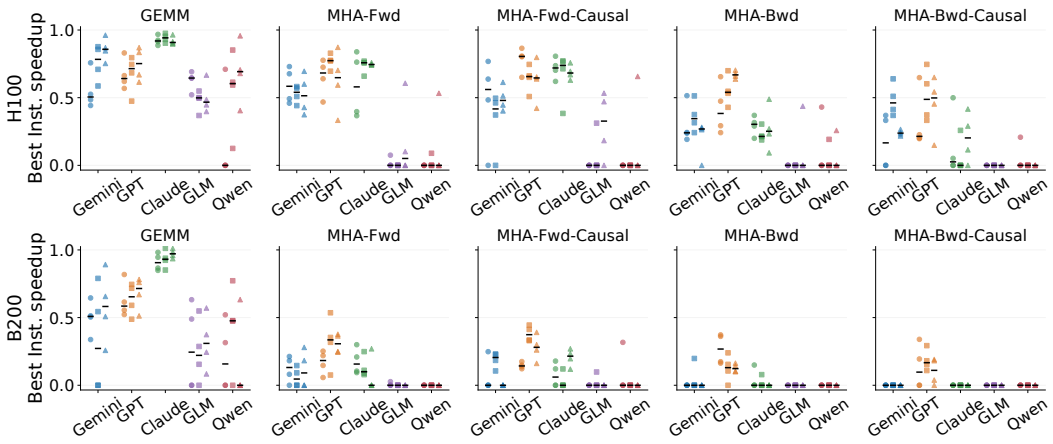

Figure 6: Individual primary CUDA–PTX trajectories by scheduling-guidance variant. Each point shows one trajectory's best instruction-qualified speedup relative to the vendor-library reference; zero denotes no qualifying candidate. At each model, circles, squares, and triangles denote p0 (no additional scheduling guidance), p1, and p2, respectively, with four repeats per variant. Black marks show the median within each variant.

Models differ in the gains from additional refinement. Figure 7 compares the instruction-qualified performance that models attain at shared output-token budgets, including reasoning. All five models improve median GEMM speedup on both GPUs over the observed runs, whereas several attention settings remain at zero median despite additional output tokens. Since best-so-far curves cannot decrease, the informative differences are the size and timing of gains.

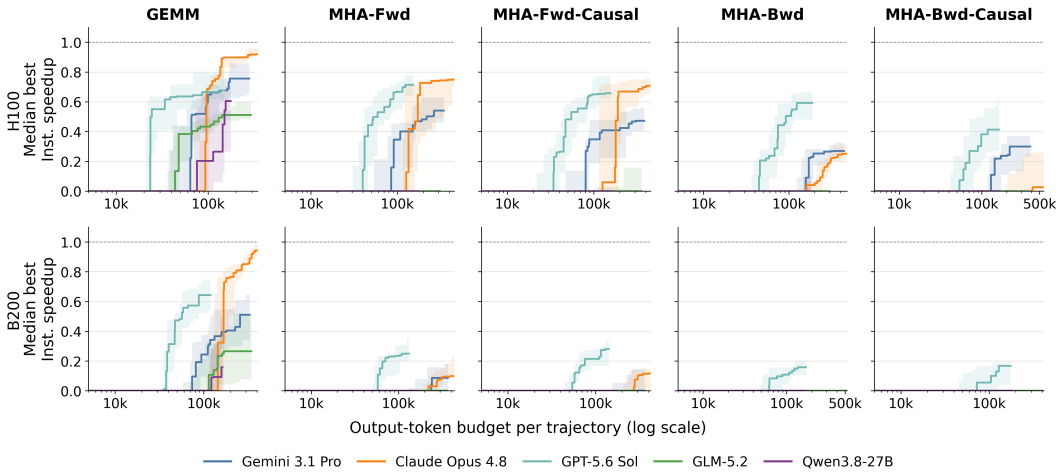

Figure 7: Median best instruction-qualified speedup at shared output-token budgets on H100 and B200, over all twelve trajectories per model and workload. Bands show 95% bootstrap intervals.

Figure 7 applies the common aggregation rule (Section A.1) to output-token budgets, including both reasoning and other output content. Each model–workload setting includes all twelve trajectories, with bootstrap resampling within the three scheduling-guidance variants. Each panel ends at the largest model endpoint.

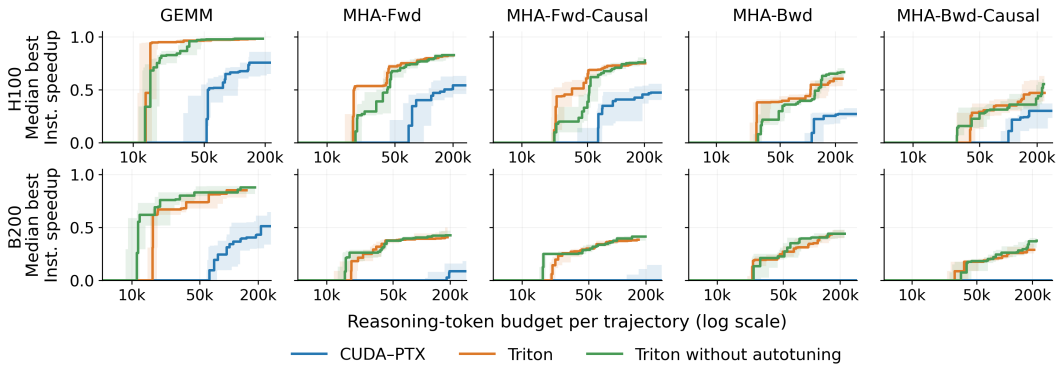

Figure 8: Gemini 3.1 Pro counterpart to Figure 5: median best instruction-qualified speedup over all twelve trajectories at shared reasoning-token budgets. Bands show 95% bootstrap intervals.