Efficient Agentic GPU Kernel Optimization with a Compact Domain-Specific Language and Speed-of-Light Guidance

Siva Kumar Sastry Hari, Vignesh Balaji, Sana Damani, Qijing Huang, Christos Kozyrakis
{shari, vbalaji, sdamani, jennyhuang, ckozyrakis}@nvidia.com
NVIDIA

Abstract

LLM agents can optimize GPU kernels, but each candidate requires generation, compilation, correctness testing, and profiling. Our objective is to reduce the number of expensive attempts needed to find a correct, fast kernel. We study two complementary ways to improve this attempt efficiency by changing the representation that the agent writes in and adding a first-principles performance signal that estimates remaining headroom. We introduce μCUTLASS, a compact in-context learnable DSL that exposes high-impact CUTLASS optimization choices while hiding template plumbing, and pair it with Speed-of-Light (SOL) guidance for search steering, budget allocation, and integrity checking. On 59 KernelBench problems on H100, switching GPT-5-mini from low-level code generation to μCUTLASS changes a $0.40\times$ geomean regression versus PyTorch into a $1.27\times$ speedup. Adding SOL-guided steering, it reaches $1.56\times$. Across model tiers, this combination lets weaker models match or exceed stronger raw-code baselines, and SOL-guided scheduling saves 19–43% of tokens while retaining at least 95% geomean speedup. We also show that integrity filtering is essential because, without SOL-assisted checks, reported speedups can be inflated by up to $1.9\times$ by benchmark-gaming or PyTorch-only solutions.

1 Introduction

GPU kernel performance is central to modern machine learning systems and LLM agents offer a path to automate kernel optimization. They can propose candidate CUDA or CUTLASS implementations, compile them, test correctness, profile runtime, and iterate. The bottleneck is that every attempt is expensive. In one representative run in our system, LLM calls only accounted for roughly 21% of wall-clock time while compile, run, and profile tool actions accounted for the rest. We therefore optimize for *attempt efficiency*, the speedup obtained per generate–compile–test–profile cycle (attempt).

Two issues limit current agents. First, low-level code generation entangles choosing an optimization strategy with producing valid, architecture-specific code. The model must reason about tiling, fusion, scheduling, and data movement while also emitting low-level, template-heavy CUDA/CUTLASS code that satisfies layout, alignment, scheduler, and integration constraints. When a candidate fails for incidental implementation reasons, an expensive tool cycle yields little useful performance information. Second, profiling is local. It explains the current implementation but does not say how much speedup is still physically plausible, whether the agent should change strategy, or whether more attempts should be spent on this problem instead of another one.

In this paper, we show that LLM kernel optimization can be made more efficient by changing the *representation* the agent writes in and the *global performance signal* that guides search. Specifically:

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

- We introduce μCUTLASS, a compact DSL that exposes a typed, validated action space for CUTLASS-backed kernels. This space covers high-impact choices such as operator family, data type, tiling, scheduler, epilogue fusion, and pipeline boundaries, while a compiler handles low-level CUTLASS generation and static validation.
- We use Speed-of-Light (SOL) analysis to estimate first-principles runtime bounds. SOL guidance steers within-problem optimization by prioritizing hypotheses with remaining headroom, and reallocates cross-problem budget away from kernels that are near their bound or have stopped improving.
- We use SOL as an integrity signal. Since LLMs can exploit specification loopholes or fall back to PyTorch-only solutions, SOL-derived expected work helps flag candidates that are implausibly fast or skip the intended computation.

These mechanisms address different parts of the optimization loop. μCUTLASS improves the quality of individual attempts by changing the action space from low-level code to validated kernel specifications. SOL guidance uses estimated headroom to inform within-problem steering and cross-problem budget allocation. The first makes each tool cycle more likely to test a meaningful optimization choice, while the second helps decide which choices and problems deserve more cycles.

We evaluate on 59 KernelBench Ouyang et al. (2025) problems selected to emphasize operators common in modern LLM workloads, on H100, across GPT-5-mini, GPT-5, and GPT-5.2. With the same attempt budget, switching GPT-5-mini from low-level code generation to μCUTLASS changes a $0.40\times$ geomean regression against PyTorch into a $1.27\times$ speedup. Adding SOL-guided steering reaches $1.56\times$. The combination lets GPT-5-mini exceed raw-code GPT-5 and GPT-5 match raw-code GPT-5.2 at lower token cost. Steering benefits depend on model and representation: it helps GPT-5-mini and GPT-5 with μCUTLASS, while GPT-5.2 with μCUTLASS shows a small observed decrease in geomean speedup. SOL-guided scheduling saves 19–43% of tokens while retaining at least 95% of geomean speedup. Finally, integrity filtering matters because, without SOL-assisted gaming and PyTorch-only filters, reported geomean speedups can be inflated by up to $1.9\times$.

2 Related Work

Compiler autotuning systems such as Halide Ragan-Kelley et al. (2013), OpenTuner Ansel et al. (2014), TVM Chen et al. (2018), Ansor Zheng et al. (2020), and MetaSchedule Shao et al. (2022) show that constrained schedule spaces make optimization search more productive. We study a different setting where the search policy is an LLM, the DSL must be learned in context, and every invalid attempt incurs a costly compile/run/profile cycle.

CUTLASS/CuTe NVIDIA (2024b,a) provide high-performance kernels but expose a large configuration surface. Triton-style DSLs Tillet et al. (2019) reduce boilerplate, but many newest-architecture kernels still rely on CUDA/CUTLASS designs Wright and Hoque (2024); PyTorch Team (2024); Dao (2024); Wilkinson (2024). μCUTLASS trades generality for reliability by exposing a compact interface to CUTLASS's performance-critical choices, a principle that could also apply to other libraries or IRs Lattner et al. (2020).

Exo Ikarashi et al. (2022) checks scheduling rewrites for computational equivalence and memory safety; Mojo Modular ([n. d.]) offers direct memory and kernel-code control for systems and GPU programming. μCUTLASS exposes configuration and composition over CUTLASS/CuTe implementations: agents choose operators, layouts, tiles, schedules, and fused epilogues, which the compiler validates and lowers to library-backed kernels. This narrower interface trades expressiveness for choices agents can generate and check within prompt and tool budgets.

The roofline model Williams et al. (2009) introduced bound-and-bottleneck analysis as a visual framework for reasoning about whether a kernel is compute-bound or memory-bound, given peak hardware throughput. This idea has since been applied broadly, including to LLM inference workloads Yuan et al. (2024), and extended with simulation/mapping-exploration-based analyses that derive attainable data movement and compute intensity bounds with some buffer capacity constraints Huang et al. (2024). We use a lightweight roofline-style SOL estimate as an agent-facing signal rather than as a final performance proof. It provides headroom, bottleneck, scheduling, and integrity information that local profiling alone does not expose.

LLM-based code generation and tool-using agents are now common Li et al. (2022, 2023); Rozière et al. (2024); Yang et al. (2024); Wang et al. (2025); Shinn et al. (2023). In GPU kernel optimization, each attempt incurs an expensive compile/run/profile loop and low-level code generation can waste iterations. KernelBench Ouyang et al. (2025) establishes an evaluation framework and finds iterative refinement with execution feedback necessary for competitive performance, but raw LLM-generated CUDA frequently fails to outperform compiled PyTorch. Autocomp Hong et al. (2025) combines planning with beam search for accelerator kernels. Sakana AI's Evolving CUDA Engineer Sakana AI (2024) applies evolutionary search to CUDA optimization. FunSearch Romera-Paredes et al. (2024), AlphaTensor Fawzi et al. (2022), and AlphaDev Mankowitz et al. (2023) explore evolutionary or reinforcement-learning-based program discovery. Deep RL also targets CUDA kernel generation. CUDA-L1 Li et al. (2026) uses contrastive reinforcement learning to improve kernel optimization, CUDA Agent Dai et al. (2026) scales agentic RL for high-performance kernel generation, and K-Search Cao et al. (2026) maintains a co-evolved intrinsic world model to guide search. Recent concurrent work instead targets progress per tool action. GPU Kernel Scientist Andrews and Witteveen (2025) uses profiling signals to generate targeted hypotheses. Astra Wei et al. (2025) studies multi-agent controllers. Kevin Baronio et al. (2025) explores multi-turn reinforcement learning, and minimal executable programs Chu et al. (2025) reduce feedback cost. GEPA and TextGrad Agrawal et al. (2026); Yuksekgonul et al. (2024) optimize agent instructions from textual execution feedback. Most of these methods improve search within existing code-level representations or controller loops. Our work instead changes the action representation via μCUTLASS and uses SOL bounds to reason about remaining headroom, stopping, budget reallocation, and evaluation integrity. These ideas are complementary to stronger local search, better memory, and improved controller design Zhang et al. (2026); Asai et al. (2023); Packer et al. (2024); Madaan et al. (2022).

3 μCUTLASS: A DSL for CUTLASS GPU Kernels

μCUTLASS raises the agent action space from low-level CUDA/CUTLASS code to compact kernel specifications. It is designed around three requirements. The language must fit in the prompt and be learnable in context. It must let the model reason about high-level optimization choices, spending fewer reasoning resources on template plumbing. It must also retain the high-impact controls needed for performance, such as changing data types, tile shapes, scheduling policies, epilogue fusion, and layout or dtype conversion boundaries.

The DSL is intentionally not a general GPU programming language. It is a narrow interface to CUTLASS/CuTe that factors out boilerplate and catches many structural errors before C++ compilation, including architecture gating, alignment/layout compatibility, and operator-specific restrictions. Agents write short specifications, often 10–20 lines, and the compiler emits a PyTorch-compatible generated header. The supported space covers the operators and compositions most relevant to our workload subset, including GEMM variants, convolutions, fused epilogues through a >> composition operator, and explicit multi-stage pipelines. Configurations expose data types, row- or column-major GEMM and channels-last convolution layouts, tile shapes, stages, and schedules, subject to operator and architecture restrictions. The contribution of μCUTLASS is its compact, typed, and validated action space and its lowering to CUTLASS/CuTe implementations. Function calling or JSON could expose the same configurations, checks, and lowering. Our experiments evaluate this interface as a whole and do not isolate an advantage of its syntax. Full grammar and coverage details are in Appendix A.1.

Arbitrary reductions, scans, and general control flow lie outside the DSL. Attention or convolution algorithms requiring unsupported operations need custom code or compiler extensions. Extending operator or architecture coverage may require front-end, validation, and lowering changes. Agents can also edit surrounding CUDA (Section 5), so providing the DSL tool does not require an entire candidate to consist of DSL-generated code. The GPT-5.2 μCUTLASS-agent runs illustrate this distinction. A *DSL-only* candidate for L1-1 calls a generated GEMM with host-side input checks and no agent-written CUDA compute kernel. A *hybrid* candidate for L2-53 combines a generated GEMM with custom CUDA fusing bias, scaling, clamp, and GELU, achieving $7.01\times$ speedup over PyTorch. L2-62 is also hybrid: a generated GEMM followed by custom CUDA fusing bias, GroupNorm, affine transformation, LeakyReLU, and output scaling achieves $7.33\times$. Its GEMM uses BF16 inputs with FP32 accumulation and output. These speedups measure the complete hybrid candidates.

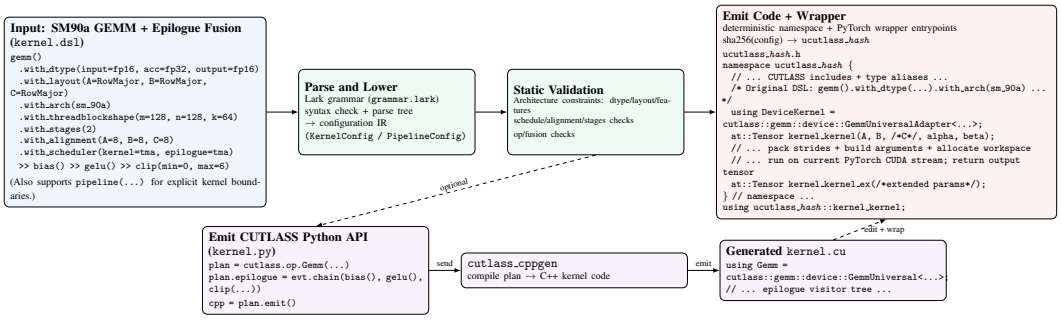


Figure 1: μCUTLASS compilation pipeline. The compiler parses and lowers a `kernel.dsl` program to a typed configuration IR, validates architecture and other constraints, and emits the CUTLASS C++ code in a wrapper header (`ucutlass_hash.h`). Optionally, it emits a CUTLASS Python API plan and invokes `cutlass_cppgen` to produce `kernel.cu`, which is then wrapped into the final header.

For a fixed GEMM implementation with a small enumerable configuration space, systematic autotuning may be preferable if exploration fits the budget. LLM guidance targets broader choices in our agent's action space, including operator formulation, fusion, precision/layout transformations, pipeline boundaries, and multi-kernel decomposition. We have also added localized autotuning to μCUTLASS, with results in Appendix A.3.5. Integrating this tuning capability with the agent's higher-level choices remains future work. Appendix A.3.4 reports our agent search time and logged implementation-agent cost. We have not performed a matched agent-versus-autotuner comparison and do not claim a cost or performance advantage over autotuning.

Compilation failures in raw code can reflect C++ or template mistakes rather than poor optimization ideas. A validated μCUTLASS specification satisfies the DSL's structural constraints, letting the expensive compile/run/profile loop more often assess reasonable performance choices. These checks do not establish semantic equivalence between the complete candidate and the reference computation; agent-written wrappers and custom CUDA remain outside the DSL validator. Prompt instructions, finite numerical tests, and the heuristic integrity filters in Section 5.2 do not prove correctness.

Individual attempts illustrate both the benefit and the remaining cost-modeling challenge. An SM90 cooperative-TMA Stream-K GEMM reaches $6.30\times$ speedup on L1-2, whereas a Conv1d pipeline on L1-67 passes the recorded correctness checks but reaches only $0.46\times$. Its input/output layout and precision conversions consume 2.42 ms of the 5.79 ms profiled total. Appendix A.1.1 gives the compact specifications, runtime integration details, and timing breakdowns.

The first input box in Figure 1 illustrates the intended abstraction boundary. The agent specifies the operator, data types, architecture, tile and scheduler choices, and fused epilogue operations, but it does not emit CUTLASS template instantiations, wrapper headers, or PyTorch binding code. These hidden details are still performance-critical, so the compiler validates them and lowers the specification to the appropriate CUTLASS backend. The LLM therefore spends attempts on choices that change the optimization point, while the compiler handles code-generation validity before the expensive compile/run/profile loop.

For well-formed programs, the compiler routes to an appropriate backend. SM90+ GEMMs use the CUTLASS 3.x CollectiveBuilder API, while other operations can be emitted through the CUTLASS Python API and wrapped into the final header. Generated files use deterministic namespaces derived from configuration hashes, which supports caching and traceability. On SM90+, epilogue chains lower to Epilogue Visitor Trees Chen et al. (2024). Pipeline programs lower to multi-stage drivers that execute transforms, kernels, and optional output transforms. We expose this compiler as an agent tool, so the model can request a generated header from a DSL string and include it from the candidate code (`cuda_model.cu`).

4 Speed-of-Light (SOL) Guidance

For each problem, we compute a Speed-of-Light (SOL) estimate with a roofline-style model of the reference computation. The analysis identifies operators, dimensions, data types, FLOPs, and

best-case DRAM bytes, then combines them with H100 peak compute throughput and memory bandwidth to estimate the fastest plausible runtime. The output is a compact report containing the bound, arithmetic intensity, bottleneck classification, and expected work. We use FP32/TF32 assumptions for search steering and a tighter FP16-based estimate for scheduling and integrity checks, since optimized kernels may use reduced-precision arithmetic while inputs and outputs remain FP32. For scheduling and integrity, we derive the FP16 estimates from the same modeling assumptions and cross-check them against SOLAR-style analytical bounds NVlabs (2025). Appendix A.2 gives the formula, a full report, and discusses assumptions.

Within-problem steering. Let t_{best} be the fastest correct candidate so far. The gap $g = t_{\text{best}}/t_{\text{SOL}}$ estimates remaining headroom. We use this headroom signal in an agentic loop, MANTIS (Measure–Analyze–Nominate–Triage–Implement–Summarize), that emphasizes analysis, triage, and learning-driven iteration. Measure and Analyze combine the current best kernel, profiler output, and the SOL report to identify the dominant gap. Nominate and Triage turn that gap into ranked hypotheses, favoring high-upside changes when far from SOL and lower-risk refinements when close. Implement tests the selected hypotheses, and Summarize records compact lessons for later iterations and problems. We evaluate both an *orchestrated* version with explicit artifacts between phases and an *in-prompt* version where the same methodology is described in the controller prompt.

Cross-problem scheduling. SOL estimates headroom that PyTorch-relative speedup alone cannot reveal, enabling scheduling across problems instead of fixed per-problem budgets. We evaluate a round-robin scheduler by offline replay of completed traces, using two stopping rules. After beating PyTorch, a problem stops receiving work when either $t_{\text{best}} \leq (1+\epsilon)t_{\text{SOL}}$ or it has made no best-speedup progress for w consecutive attempts. The scheduler then assigns work to another eligible problem. For fixed (ϵ, w), the rule uses only information from work completed so far in the replay.

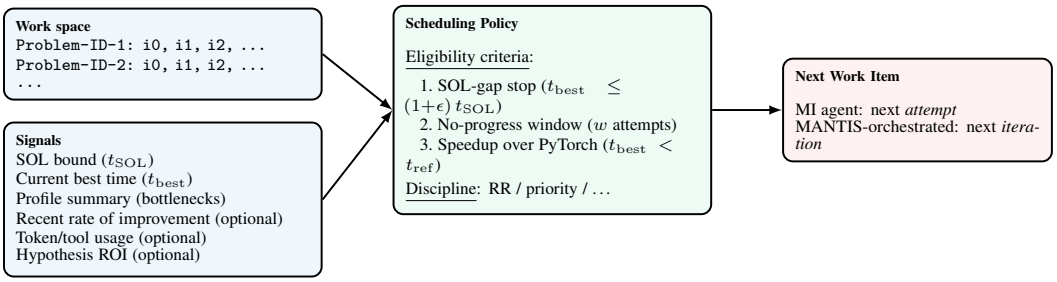


Figure 2: SOL-guided budget scheduling. The scheduler combines SOL gap, progress, and speedup signals to determine problem eligibility, then selects the next work item.

Integrity checking. Correctness tests do not fully specify what counts as a legitimate kernel. Agents may exploit benchmark-specific algebra, skip required work while staying within the test's tolerance, or compose PyTorch library calls instead of writing custom CUDA/CUTLASS code. SOL report helps because it describes the work that a legitimate implementation should perform. We use it in two ways. A runtime ceiling flags candidates more than 10% faster than the FP16 SOL bound (leaving a small margin for measurement error), and an LLM-based game detector reviews candidate code with the reference program and SOL report. A separate static detector rejects PyTorch-only solutions. The complete policy appears in Section 5.2. Examples and details are in Appendix A.5.

5 Evaluation Methodology

We evaluate if μCUTLASS and SOL guidance improves progress under matched tool-action budgets. All primary results use NVIDIA H100 (SM90a) with locked clocks. Nsight Compute measures on-GPU kernel time, excluding launch overhead, host synchronization, and Python framework latency.

Workloads and models. We use a 59-problem subset of KernelBench Ouyang et al. (2025), drawn from Levels 1–3 and selected to emphasize operators common in Transformer, SSM, and modern recurrent workloads. The subset is intended to stress the kinds of dense linear algebra, fusion, normalization, and layout transformations that dominate contemporary text-model inference and training kernels. This selection also keeps the comparison across model tiers, representations, and

steering variants within a practical experimental budget: extending the same experiment matrix to all 250 problems would require roughly four times the compute and token budget. Omission from this subset does not imply that a problem is inexpressible in μCUTLASS. We use the same subset for every variant comparison. We exclude two problems whose evaluation references admit unintended shortcut implementations. The full selection rule and IDs are in Appendix A.3.1. We evaluate GPT-5-mini, GPT-5, and GPT-5.2, priced at \$0.25, \$1.25, and \$1.75 per million input tokens at the time of experimentation. Most experiments ran from December 2025 to February 2026.

Agent setup. Each problem is bootstrapped into a workspace with a C++ test driver, a seed `cuda_model.cu` that delegates to PyTorch, and compile/run/profile targets. The agent edits `cuda_model.cu`. μCUTLASS variants additionally receive a `ucutlass_compile` tool that turns a DSL string into a generated header. The bootstrapper profiles PyTorch to obtain t_{ref} and generates one SOL report per problem, reused across variants.

5.1 Variants and Metrics

Table 1 summarizes the matched-budget variants. MI is a flat Measure–Implement controller that runs a Generate–Compile–Test–Profile attempt. SOL guidance is evaluated either in prompt or as an orchestrated MANTIS controller. For reporting, we use the better SOL-guided implementation for each model tier and discuss implementation effects separately.

Variant	Representation	Steering	Budget
MI	Raw CUDA/CUTLASS	None	40 attempts
MI + μCUTLASS	μCUTLASS	None	40 attempts
SOL-guided	Raw CUDA/CUTLASS	In-prompt or orchestrated	40 attempts
μCUTLASS + SOL-guided	μCUTLASS	In-prompt or orchestrated	40 attempts

Table 1: Primary variants. Orchestrated MANTIS uses 5 iterations, 2 hypotheses per iteration, and 4 attempts per hypothesis, matching the 40-attempt budget.

We report speedup over PyTorch, geomean and median speedups across problems, and Fast-p curves, which show the fraction of problems whose best candidate reaches speedup at least r. Attempt-Fast-$p(2)$ measures how quickly variants reach at least $2\times$ speedup as attempts accumulate. For scheduling, we replay completed traces under stopping policies and report token savings, speedup retention, and efficiency gain.

$$\mathrm{gain} = \frac{g_{\mathrm{policy}}}{g_{\mathrm{fixed}}} \times \frac{\tau_{\mathrm{fixed}}}{\tau_{\mathrm{policy}}},$$

where g is geomean speedup and τ is the total recorded prompt and completion tokens consumed during search. Gain measures the relative improvement in speedup per search token. We report retention and token savings separately because a higher gain can trade lower final kernel speedup for fewer search tokens. It does not estimate deployment savings: the value of a faster kernel depends on its invocation count and execution cost. Appendix A.3.4 distinguishes search tokens, the figure's pricing proxy, logged LLM charges, and search wall time.

5.2 Integrity and Baselines

All reported speedups apply the same offline integrity filter before aggregate metrics are computed. The filter has three parts: an SOL-ceiling check for physically implausible runtimes, an LLM-based game detector that reviews candidate code against the reference and SOL report and labels each candidate as *No Issues*, *Minor Issues*, or *Gaming*, and a static PyTorch-only detector. We accept attempts labeled *No* or *Minor Issues*. We exclude attempts flagged by the SOL-ceiling detector, labeled as *Gaming* by the LLM-based game detector, or classified as PyTorch-only. This policy is applied uniformly across all variants, including external-baseline review. Minor Issues are accepted because the offline review gave the agent no chance to correct them and the issue is not expected to materially affect performance if fixed. Gaming and PyTorch-only attempts are removed because our target is custom kernel development, not benchmark-specific shortcuts or library-call composition. Appendix A.5 details the labels and detector protocol.

For budget scheduling, we replay completed traces using the SOL-headroom threshold ϵ and no-progress window w from Section 4. We sweep both parameters and compare each policy against fixed 40-attempt allocation. Speedup retention is computed retrospectively against the completed

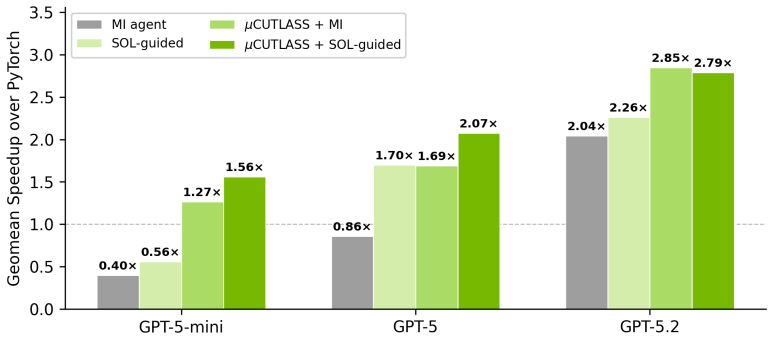


Figure 3: Geomean speedup over PyTorch across model tiers with matched attempt budgets.

fixed-budget baseline; it is an evaluation metric, not an input available to the scheduler. An online deployment would fix (ϵ, w) before execution or calibrate them on separate historical runs, rather than select them using the evaluated run's eventual retention.

As contextual prior-work evidence, we compare against the public Sakana AI CUDA Engineer archive Lange et al. (2025). Because our PyTorch baseline and measurement methodology differ, we use the archive's reported speedup numbers for a fair comparison. Details are in Appendix A.6.

6 Results

All results below use the integrity filtering policy from Section 5.2. We organize the results around four questions: whether representation and SOL guidance improve speedup under a fixed attempt budget, when explicit steering helps, how much scheduling can reduce cost, and how much integrity filtering changes the reported outcome.

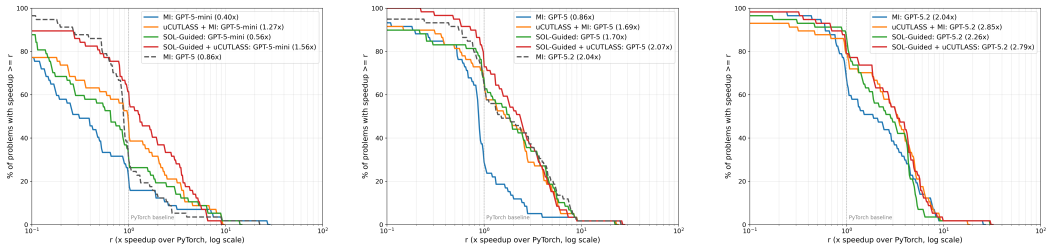
6.1 Representation and Guidance Improve Attempt Efficiency

Figure 3 shows the central result. The use of μCUTLASS yields the largest single improvement for weaker models. GPT-5-mini moves from a $0.40\times$ geomean regression against PyTorch to a $1.27\times$ speedup, a $3.2\times$ improvement over the raw-code MI baseline. GPT-5 improves from $0.86\times$ to $1.69\times$, and GPT-5.2 improves from $2.04\times$ to $2.85\times$.

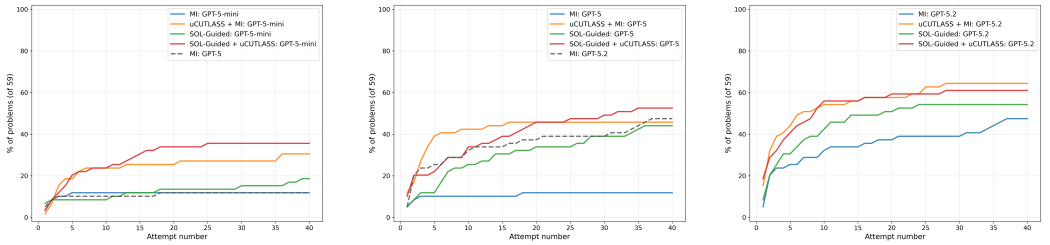
Adding SOL-guided steering further helps GPT-5-mini and GPT-5, reaching $1.56\times$ and $2.07\times$ respectively. This is enough for GPT-5-mini with μCUTLASS+SOL to exceed the raw-code GPT-5 baseline, and for GPT-5 with μCUTLASS+SOL to match the raw-code GPT-5.2 baseline. Besides just absolute speedups, these gains substitute for a model-tier upgrade at lower token cost. Steering is not uniformly beneficial: on GPT-5.2 with μCUTLASS, the observed geomean speedup decreases from $2.85\times$ without steering to $2.79\times$ with steering. One explanation is that, once μCUTLASS removes much of the low-level implementation burden, GPT-5.2 can already plan effectively, leaving less room for additional steering to improve search. The pattern clarifies abstraction's role. The weakest model benefits most from μCUTLASS because many raw-code failures are implementation failures rather than bad optimization ideas. As model capability increases, the raw-code baseline improves, but the DSL still raises the floor by making valid high-impact choices easier to express.

Attempt-level statistics for GPT-5.2 provide evidence for both reliability and candidate quality (Table 7 in Appendix A.3.2). Across the recorded attempts, μCUTLASS reduces C++ compilation failures from 18.7% to 5.1%. Among successfully built, integrity-valid timed candidates, the fraction that beats PyTorch rises from 34.9% to 48.0%. Runtime and correctness failures remain, showing that the DSL reduces compilation burden without eliminating integration errors. These effects cannot be separated additively because failures change the feedback available to subsequent search.

Performance also varies by operator category (Table 8). For the GPT-5.2 μCUTLASS agent, the largest geomean speedups are on fused Level-2 operators ($4.61\times$) and composite attention/model



(a) GPT-5-mini Fast-p (b) GPT-5 Fast-p (c) GPT-5.2 Fast-p



(d) GPT-5-mini Attempt-Fast-$p(2)$ (e) GPT-5 Attempt-Fast-$p(2)$ (f) GPT-5.2 Attempt-Fast-$p(2)$

Figure 4: Fast-p and Attempt-Fast-$p(2)$ curves for the primary variants across model tiers. Top row shows final speedup distributions; bottom row shows how quickly variants reach at least $2\times$ speedup.

blocks ($4.33\times$), while convolution remains a weakness ($0.40\times$). These means cover solved problems; the table also reports solved/total counts to make coverage explicit. The results include hybrid DSL/custom-CUDA implementations.

Fast-p and Attempt-Fast-$p(2)$ analyses in Figure 4 support the same conclusion. GPT-5-mini with μCUTLASS+SOL reaches its $\geq 2\times$ plateau within 10 attempts, while MI alone barely accumulates $\geq 2\times$ solutions across the full budget. GPT-5 shows the same pattern, with μCUTLASS variants reaching more than 40% of problems at $\geq 2\times$ within 5–10 attempts. The bottom row also shows limited additional benefit from SOL steering for GPT-5.2 with μCUTLASS, consistent with the DSL reducing implementation difficulty enough that the strongest model needs less external structure.

6.2 SOL Steering Depends on Model and Representation

SOL steering performance depends on model and representation. Orchestrated MANTIS makes measurement, analysis, hypothesis nomination, and triage explicit. With GPT-5.2 plus μCUTLASS, the in-prompt variant performs better than explicit orchestration. Our interpretation is that μCUTLASS removes much of the implementation burden that orchestration helps weaker models manage. The strongest model can then exploit simple in-prompt guidance effectively, while additional orchestration appears to constrain rather than improve its search. Component ablations are consistent with this interpretation. On GPT-5-mini without μCUTLASS, every removed MANTIS component hurts, with Triage and Summarize most important. With μCUTLASS, Analyze remains useful but other orchestration components are closer to a wash. These results suggest that representation and controller structure can partially substitute for one another. A practical system should therefore adapt the degree of steering to the model and representation. These comparisons concern within-problem steering; the scheduling and integrity analyses below evaluate separate uses of SOL.

6.3 SOL-Guided Scheduling Saves Tokens

We evaluate scheduling through offline replay of completed fixed-budget traces. SOL-guided scheduling converts each fixed-budget run into a cost–speedup frontier, as shown in Figure 5. Each point corresponds to one replayed (ϵ, w) stopping policy for a particular agent variant. The x-axis is a normalized token-cost proxy using the model's input-token rate for all recorded tokens (Appendix A.3.4), and the y-axis is geomean speedup. Gray hexagons mark fixed-budget MI reference points. The most important comparisons are geometric. Points above a gray hexagon show better performance

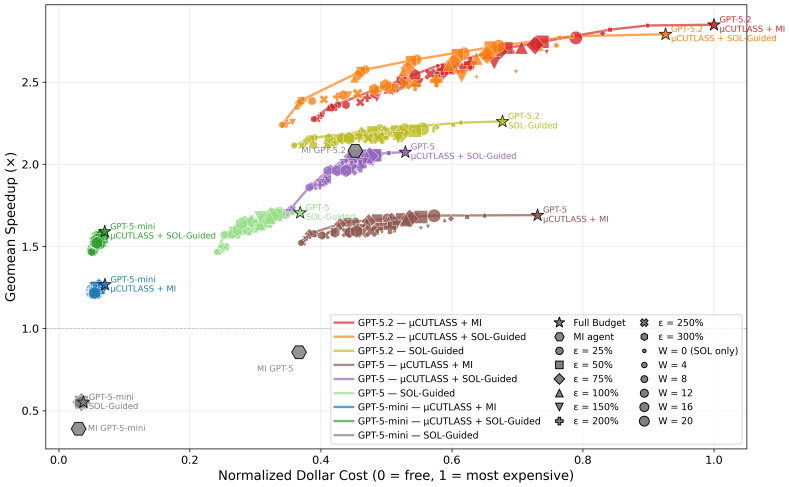


Figure 5: Scheduler policy Pareto frontiers. Each point is one replayed (ϵ, w) stopping policy for one variant, plotted by normalized token-dollar cost and achieved geomean speedup. Gray hexagons mark fixed-budget MI references. Points that are higher at similar cost show better performance for the same budget, points farther right show the gains from spending more resources, and points left of but above a gray hexagon show better performance at lower cost.

at comparable or lower cost. Points farther right on a frontier show the additional performance available when we spend more search tokens. Points that lie left of a gray hexagon but above it are especially useful, since they achieve better speedup for less cost. The envelopes summarize the best achievable trade-off for each variant. This view separates two roles. Agent design determines the vertical position of a frontier, while scheduling selects an operating point along that frontier.

Across evaluated variant–model configurations, the best policy under a $\geq 95\%$ geomean-retention constraint saves 19–43% of tokens. The strongest efficiency result is GPT-5 μCUTLASS+SOL at $1.68\times$ gain, using $\epsilon = 250\%$ and $w = 12$, with 43% token savings and 96% speedup retention. A common policy of $\epsilon = 50\%$ and $w = 16$ saves 23.6% of tokens on average while retaining 97.8% of fixed-budget geomean speedup across twelve flat-agent configurations spanning three model tiers. The scheduler levers behave differently across model tiers. Weaker models often do not reach tight FP16 SOL thresholds, so no-progress windows are more useful. Stronger models produce more near-SOL kernels, so both SOL-headroom and no-progress criteria matter.

6.4 Integrity Filtering Changes the Reported Result

Integrity checking is essential because correctness tests do not fully encode the intended computation. Figure 6 shows that filtering is not a small bookkeeping detail. PyTorch-only filters matter most for μCUTLASS variants. GPT-5-mini μCUTLASS+MI would rise from $1.27\times$ to $2.28\times$ if PyTorch-only library compositions were accepted, an 80% inflation. LLM-detected gaming is even more consequential for stronger models. GPT-5.2 μCUTLASS+MI would rise from $2.85\times$ to $5.34\times$ without the game detector, a $1.9\times$ inflation. SOL-guided variants are less affected but not immune. GPT-5.2 μCUTLASS+SOL would rise from $2.79\times$ to $3.57\times$ without integrity filtering. Prompt-level guardrails are insufficient. Explicit anti-PyTorch-only instructions helped reduce PyTorch-only codes, but did not eliminate them, and gaming did not consistently decrease with anti-gaming instructions in a repeat experiment with GPT-5-mini. The evaluation therefore needs a detector, not just instructions.

This changes how we interpret LLM-kernel benchmarks. Correctness tests certify agreement with a finite harness, not necessarily that the candidate performed the intended computation. As models become stronger, integrity checking becomes a growing concern because models can become better at finding both optimizations and loopholes. We present one approach to improve integrity by using augmented SOL analysis as additional specification for what work the kernel is expected to perform, but more research is needed on robust evaluation protocols for LLM-generated kernels.

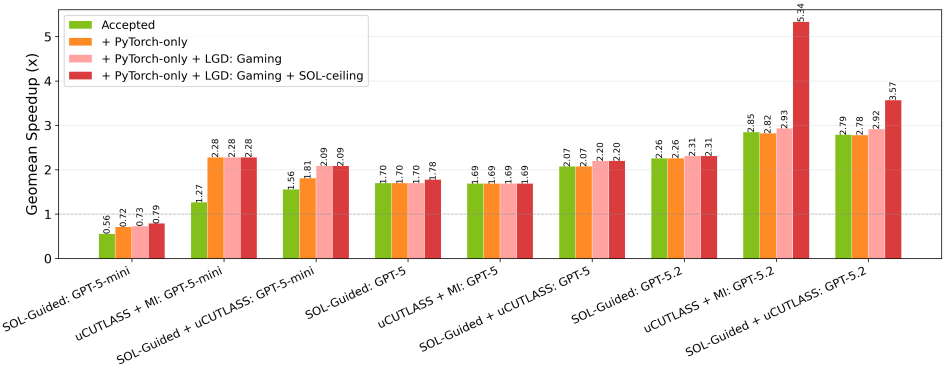


Figure 6: Speedup inflation under different integrity policies. The filtered result is compared with variants that admit PyTorch-only, LLM-detected gaming, or SOL-ceiling flagged candidates.

6.5 Stability and External Context

Since LLM requests can be inherently non-deterministic, we study performance stability. We find the main conclusions persist across nearby repeats and ablations. Every GPT-5-mini configuration with μCUTLASS in our variation study exceeds the GPT-5 MI baseline of $0.86\times$, including the lowest ablation variant and an independent repeat with stronger anti-gaming instructions. Variation is larger for GPT-5-mini (13–15% coefficient of variation) than for GPT-5.2 (5–7%), consistent with weaker models being more sensitive to prompt and controller changes. Appendix A.6 and Figure 13 give the full variation summary. Our comparison against the public Sakana AI CUDA Engineer archive Lange et al. (2025) is contextual because software stacks and timing methodology differ. After applying a similar integrity review to the archive, the accepted Sakana kernels achieve $1.13\times$ geomean over their own PyTorch baseline on our 59-problem subset. All three μCUTLASS+SOL variants exceed this reference, with GPT-5-mini already reaching $1.56\times$. Figure 12 gives the full Fast-p comparison. Finally, selecting the fastest integrity-filtered kernel found across all our variants yields $3.91\times$ geomean speedup, while the FP16 SOL curve reaches $7.46\times$, indicating remaining headroom for adaptive variant selection and stronger search.

7 Conclusion

This paper shows that LLM kernel optimization benefits from designing the agent's action space and feedback signals around iteration efficiency. μCUTLASS removes low-level generation burden while preserving high-impact optimization choices, and SOL guidance adds a first-principles headroom signal for steering, scheduling, and integrity checking. Across 59 KernelBench problems on H100, μCUTLASS with SOL steering lets weaker models match or exceed stronger raw-code baselines. The added benefit of steering depends on model and representation: it improves GPT-5-mini and GPT-5 with μCUTLASS, while GPT-5.2 with μCUTLASS shows a small observed decrease. The separate scheduling and integrity analyses show token savings and the removal of misleading speedups from benchmark gaming, respectively.

The broader lesson is that agent systems should expose the right optimization decisions, not merely more code-generation freedom. In this setting, a compact DSL raises the floor by turning fragile implementation details into compiler responsibilities, while SOL analysis makes optimization progress measurable enough to guide search, stop unpromising work, and audit suspicious wins. These lessons are scoped to a 59-problem KernelBench subset, H100/SM90a, three closed GPT model tiers, and SOL models that depend on assumptions about fusion, caching, precision, and hardware peaks. A limited GB300 study additionally demonstrates successful code generation and execution with μCUTLASS (Appendix A.3.5). Future work should broaden the operator and hardware coverage, learn compact DSLs automatically, choose agent strategies per problem, and move integrity feedback into the online optimization loop so agents can repair minor issues during search. Evaluation across other model families remains future work.

Acknowledgments and Disclosure of Funding

We thank Sahil Modi, Edward Lin, Mark Stephenson, and Humphrey Shi for their valuable feedback at different stages of the project.

All authors are employed by NVIDIA. This work received no external funding. The authors declare no competing interests beyond their employment at NVIDIA.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. arXiv:2507.19457 [cs.CL] https://arxiv.org/abs/2507.19457

Martin Andrews and Sam Witteveen. 2025. GPU Kernel Scientist: An LLM-Driven Framework for Iterative Kernel Optimization. arXiv:2506.20807 [cs.LG] https://arxiv.org/abs/2506.20807

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. arXiv:2310.11511 [cs.CL] https://arxiv.org/abs/2310.11511

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-Turn RL for Generating CUDA Kernels. arXiv:2507.11948 [cs.LG] https://arxiv.org/abs/2507.11948

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. 2026. K-Search: LLM Kernel Generation via Co-Evolving Intrinsic World Model. arXiv:2602.19128 [cs.AI] https://arxiv.org/abs/2602.19128

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

Zhaodong Chen, Andrew Kerr, Richard Cai, Jack Kosaian, Haicheng Wu, Yufei Ding, and Yuan Xie. 2024. EVT: Accelerating Deep Learning Training with Epilogue Visitor Tree. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 301–316. doi:10.1145/3620666.3651369

Ruifan Chu, Anbang Wang, Xiuxiu Bai, Shuai Liu, and Xiaoshe Dong. 2025. GPU Kernel Optimization Beyond Full Builds: An LLM Framework with Minimal Executable Programs. arXiv:2512.22147 [cs.DC] https://arxiv.org/abs/2512.22147

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. 2026. CUDA Agent: Large-Scale Agentic RL for High-Performance CUDA Kernel Generation. arXiv:2602.24286 [cs.LG] https://arxiv.org/abs/2602.24286

Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. PyTorch Blog. https://pytorch.org/blog/flashattention-3/

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610, 7930 (2022), 47–53. doi:10.1038/s41586-022-05172-4

Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. 2025. Autocomp: A Powerful and Portable Code Optimizer for Tensor Accelerators. arXiv:2505.18574 [cs.PL] https://arxiv.org/abs/2505.18574

Qijing Huang, Po-An Tsai, Joel S. Emer, and Angshuman Parashar. 2024. Mind the Gap: Attainable Data Movement and Operational Intensity Bounds for Tensor Algorithms. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 150–166. doi:10.1109/ISCA59077.2024.00021

Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. ACM, 703–718. doi:10.1145/3519939.3523446

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization. arXiv:2509.14279 [cs.SE] https://arxiv.org/abs/2509.14279

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL] https://arxiv.org/abs/2002.11054

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2026. CUDA-L1: Improving CUDA Optimization via Contrastive Reinforcement Learning. arXiv:2507.14111 [cs.AI] https://arxiv.org/abs/2507.14111

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158

Aman Madaan, Niket Tandon, Peter Clark, and Yiming Yang. 2022. Memory-assisted prompt editing to improve GPT-3 after deployment. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 2833–2861. doi:10.18653/V1/2022.EMNLP-MAIN.183

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, A. Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin A. Riedmiller, Oriol Vinyals, and David Silver. 2023. Faster sorting algorithms discovered using deep reinforcement learning. *Nature* 618, 7964 (2023), 257–263. doi:10.1038/s41586-023-06004-9

Modular. [n. d.]. Mojo Vision. Mojo documentation, version 1.2.0. https://mojolang.org/docs/vision/ Accessed November 5, 2026.

NVIDIA. 2024a. CuTe. NVIDIA CUTLASS Documentation. https://docs.nvidia.com/cutlass/latest/media/docs/cpp/cute/index.html

NVIDIA. 2024b. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass GitHub repository.

NVlabs. 2025. SOLAR: Speed of Light Analysis for ML Model Runtime. https://github.com/NVlabs/SOLAR. Accessed: 2026-03-20.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? arXiv:2502.10517 [cs.LG] https://arxiv.org/abs/2502.10517

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560 [cs.AI] https://arxiv.org/abs/2310.08560

PyTorch Team. 2024. Deep Dive on the Hopper TMA Unit for FP8 GEMMs. PyTorch Blog. https://pytorch.org/blog/hopper-tma-unit/

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. doi:10.1145/2491956.2462176

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature* 625, 7995 (2024), 468–475. doi:10.1038/s41586-023-06924-6

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

Sakana AI. 2024. Evolving AI CUDA Engineer. Sakana AI Blog. https://sakana.ai/evolving-ai-cuda-engineer/

Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. arXiv:2205.13603 [cs.LG] https://arxiv.org/abs/2205.13603

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html

Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. 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, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. doi:10.1145/3315508.3329973

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] https://arxiv.org/abs/2407.16741

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A Multi-Agent System for GPU Kernel Performance Optimization. arXiv:2509.07506 [cs.DC] https://arxiv.org/abs/2509.07506

Lucas Wilkinson. 2024. Introducing Machete, a Mixed-Input GEMM Kernel Optimized for NVIDIA Hopper GPUs. Red Hat Developer. https://developers.redhat.com/articles/2024/10/14/introducing-machete-mixed-input-gemm-kernel

Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. doi:10.1145/1498765.1498785

Less Wright and Adnan Hoque. 2024. Deep Dive on CUTLASS Ping-Pong GEMM Kernel. PyTorch Blog. https://pytorch.org/blog/cutlass-ping-pong-gemm-kernel/

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793

Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. arXiv:2402.16363 [cs.CL] https://arxiv.org/abs/2402.16363

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. TextGrad: Automatic "Differentiation" via Text. arXiv:2406.07496 [cs.CL] https://arxiv.org/abs/2406.07496

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2026. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv:2510.04618 [cs.LG] https://arxiv.org/abs/2510.04618

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

A Appendix

The appendix follows the main paper order. It first gives μCUTLASS details, then SOL and MANTIS details, then experimental setup and more results, followed by integrity review and external-baseline details.

A.1 μCUTLASS Specification and Coverage

μCUTLASS is implemented as a Python CLI and agent tool that accepts DSL text, performs grammar parsing and static validation, and emits a generated `ucutlass_*.h` header for inclusion from `cuda_model.cu`. We developed the grammar and compiler with AI-assisted programming and human review, then tested them using standalone examples that compile DSL programs, build C++/CUDA drivers, and compare against PyTorch references.

Category	Support	Notes
GEMM	SM70+	Standard, batched, grouped
Convolution	SM70+	1D/2D/3D variants, NHWC/NDHWC-oriented
Depthwise/grouped conv	SM70+ / SM80+	Backend and architecture restrictions apply
Types	FP64/FP32/FP16/BF16/FP8/INT8	FP8 requires SM90+
Configuration	tiles, stages, cluster, scheduler	Architecture-gated where needed
Epilogues	activations, bias, scaling, custom exprs	Composed with >>, EVT on SM90+
Pipelines	transforms plus kernels	Explicit boundaries via `pipeline(...)`

Table 2: μCUTLASS coverage summary. The DSL is narrower than CUTLASS/CuTe and exposes the controls used by the agent experiments.

A.1.1 Successful and Unsuccessful Optimization Attempts

We examine two individual GPT-5.2 μCUTLASS-agent attempts on H100: L1-2 attempt 4 and L1-67 attempt 1. Both passed the recorded harness checks (`rtol=atol=0.1`). The listings reproduce their saved DSL requests, with line breaks added for readability; timings include all CUDA kernels in each saved profile, rather than only the central CUTLASS kernel.

Successful GEMM. For $M = 2048$, $N = 4096$, and $K = 8192$, the specification selects FP16 inputs with FP32 accumulation/output, a $256 \times 192 \times 64$ tile, a $2 \times 1 \times 1$ cluster, two stages, and cooperative TMA with Stream-K scheduling. The source wrapper checks shapes and casts the FP32 inputs to FP16 before calling the generated GEMM; it adds no custom CUDA compute kernel. The full profiled candidate takes 0.51750 ms versus 3.26000 ms for PyTorch, a $6.30\times$ speedup. This illustrates how the DSL expresses several coupled CUTLASS choices without requiring the agent to write their template instantiations.

Listing 1: Saved GEMM DSL request for L1-2, attempt 4.

```
gemm()
  .with_dtype(input=fp16, acc=fp32, output=fp32)
  .with_layout(A=RowMajor, B=RowMajor, C=RowMajor)
  .with_arch(sm_90a)
  .with_threadblockshape(m=256, n=192, k=64)
  .with_cluster(m=2, n=1, k=1)
  .with_alignment(A=8, B=8, C=4)
  .with_stages(2)
  .with_scheduler(tile=stream_k, kernel=tma_cooperative,
                  epilogue=tma_cooperative, splits=8)
```

Unsuccessful Conv1d. For batch size 32, 64 input channels, 128 output channels, input length 131072, and kernel width 3, the pipeline converts input/output layouts and precision around a generated convolution. The saved candidate also includes runtime-wrapper changes for weight caching, synchronization removal, and current-stream execution; these are outside the DSL request. It takes 5.79471 ms versus 2.68744 ms for PyTorch ($0.46\times$). Input/output transpose-and-cast kernels consume 2.42083 ms (41.8%); the convolution itself takes 3.04000 ms, already exceeding the PyTorch reference. Weight conversion/packing adds only 0.00790 ms. Thus, layout/precision conversion is a substantial additional cost, while the convolution implementation also remains slower. Structural validation alone cannot predict whether a complete pipeline is profitable.

Listing 2: Saved Conv1d pipeline request for L1-67, attempt 1.

```
pipeline(
  transpose(input, NCL, NLC, fp32, fp16),
  transpose(weight, NCL, NLC, fp32, fp16),
```

```
  conv1d_fprop(kernel_w=3)
    .with_dtype(input=fp16, acc=fp32, output=fp16)
    .with_arch(sm_90a)
    .with_threadblockshape(m=256, n=128, k=64)
    .with_cluster(m=1, n=1, k=1)
    .with_alignment(A=8, B=8, C=8)
    .with_scheduler(kernel=tma_cooperative,
                    epilogue=tma_cooperative)
    .with_stages(2),
  transpose(output, NLC, NCL, fp16, fp32)
)
```

The saved headers also expose limits in propagating requested settings: the GEMM request's `splits=8` is not explicitly forwarded to the scheduler arguments, and the convolution builder uses automatic stage and schedule selection despite the explicit requests above. The measurements describe the saved implementations, including these runtime integration details.

Profile component	GEMM (ms)	Conv1d (ms)
CUTLASS compute	0.41664	3.04000
Input/output layout or precision conversion	0.09344	2.42083
Other CUDA kernels	0.00742	0.33388
Candidate total	0.51750	5.79471
PyTorch reference	3.26000	2.68744
Speedup over PyTorch	6.30×	0.46×

Table 3: Saved profile breakdowns for the two individual attempts. Other CUDA kernels comprise zero-fill for GEMM and zero-fill plus weight conversion/packing for Conv1d. Totals are sums of profiled GPU-kernel durations, not host wall-clock measurements.

A.1.2 Full EBNF Grammar

We include the full EBNF grammar used as the reference specification here.

Listing 3: μCUTLASS DSL grammar (EBNF).

```
(* μCUTLASS DSL Grammar (EBNF) *)
(* Clean, unquoted syntax - no string quotes needed except for custom expressions *)

(* TOP-LEVEL *)
start = kernel | pipeline ;
kernel = operation , { configuration } , { epilogue } ;

(* PIPELINES *)
pipeline = "pipeline(" , stage , { "," , stage } , ")" ;
stage = transform_stage | kernel_stage ;
kernel_stage = operation , { configuration } , { epilogue } ;
transform_stage = transpose_op ;

(* Transpose with optional dtype conversion:
 *   transpose(input, NCL, NLC)                - same dtype
 *   transpose(input, NCL, NLC, fp32, fp16)    - fp32 → fp16 conversion
 *   transpose(output, NLC, NCL, fp16, fp32)   - fp16 → fp32 conversion
 *   transpose(input, NCHW, NHWC)              - 4D layout swap (implemented by flattening HW)
 * Dtype conversion is fused with transpose (essentially free)
 *)
transpose_op = "transpose(" , TENSOR_REF , "," , LAYOUT_3D , "," , LAYOUT_3D ,
               [ "," , DTYPE , "," , DTYPE ] , ")" ;
TENSOR_REF = "input" | "output" | "weight" ;
(* NOTE: LAYOUT_3D also includes 4D logical layouts (NCHW/NHWC). *)
LAYOUT_3D = "NCL" | "NLC" | "NCHW" | "NHWC" ;

(* OPERATIONS *)
operation = gemm_op | batched_gemm_op | grouped_gemm_op
          | conv2d_fprop | conv2d_dgrad | conv2d_wgrad
          | conv1d_fprop | depthwise_conv1d | group_conv1d
          | conv3d_fprop | conv3d_dgrad | conv3d_wgrad
          | depthwise_conv2d | group_conv2d | group_conv3d ;

gemm_op = "gemm()" ;
batched_gemm_op = "batched_gemm()" ;
```

```
grouped_gemm_op = "grouped_gemm(" , "expert_count=" , INTEGER , ")" ;

conv2d_fprop = "conv2d_fprop(" , "kernel_h=" , INTEGER , "," , "kernel_w=" , INTEGER , ")" ;
conv2d_dgrad = "conv2d_dgrad(" , "kernel_h=" , INTEGER , "," , "kernel_w=" , INTEGER , ")" ;
conv2d_wgrad = "conv2d_wgrad(" , "kernel_h=" , INTEGER , "," , "kernel_w=" , INTEGER , ")" ;

conv1d_fprop = "conv1d_fprop(" , "kernel_w=" , INTEGER , ")" ;
depthwise_conv1d = "depthwise_conv1d(" , "kernel_w=" , INTEGER , ")" ;
group_conv1d = "group_conv1d(" , "kernel_w=" , INTEGER , "," , "groups=" , INTEGER , ")" ;

conv3d_fprop = "conv3d_fprop(" , "kernel_d=" , INTEGER , "," , "kernel_h=" , INTEGER , "," , "kernel_w=" ,
      INTEGER , ")" ;
conv3d_dgrad = "conv3d_dgrad(" , "kernel_d=" , INTEGER , "," , "kernel_h=" , INTEGER , "," , "kernel_w=" ,
      INTEGER , ")" ;
conv3d_wgrad = "conv3d_wgrad(" , "kernel_d=" , INTEGER , "," , "kernel_h=" , INTEGER , "," , "kernel_w=" ,
      INTEGER , ")" ;

depthwise_conv2d = "depthwise_conv2d(" , "kernel_h=" , INTEGER , "," , "kernel_w=" , INTEGER , ")" ;
group_conv2d = "group_conv2d(" , "kernel_h=" , INTEGER , "," , "kernel_w=" , INTEGER , "," , "groups=" ,
     INTEGER , ")" ;
group_conv3d = "group_conv3d(" , "kernel_d=" , INTEGER , "," , "kernel_h=" , INTEGER , "," , "kernel_w=" ,
      INTEGER , "," , "groups=" , INTEGER , ")" ;

(* CONFIGURATION - all use unquoted identifiers *)
configuration = dtype_config | layout_config | arch_config | tile_config | stages_config
              | alignment_config | cluster_config | swizzle_config | scheduler_config | scaling_config
              | iterator_config | split_k_config | operand_swap_config ;

dtype_config = ".with_dtype(" , "input=" , DTYPE , "," , "acc=" , DTYPE , "," , "output=" , DTYPE , ")" ;
layout_config = gemm_layout_config | conv_layout_config ;
gemm_layout_config = ".with_layout(" , "A=" , GEMM_LAYOUT , "," , "B=" , GEMM_LAYOUT , "," , "C=" ,
     GEMM_LAYOUT , ")" ;
conv_layout_config = ".with_layout(" , "input=" , CONV_LAYOUT , "," , "filter=" , CONV_LAYOUT , "," , "
     output=" , CONV_LAYOUT , ")" ;

arch_config = ".with_arch(" , ARCH , ")" ;
tile_config = (".with_tile(" | ".with_threadblockshape(") , "m=" , INTEGER , "," , "n=" , INTEGER , "," , "
     k=" , INTEGER , ")" ;
stages_config = ".with_stages(" , INTEGER , ")" ;
alignment_config = ".with_alignment(" , "A=" , INTEGER , "," , "B=" , INTEGER , "," , "C=" , INTEGER , ")"
      ;
cluster_config = ".with_cluster(" , "m=" , INTEGER , "," , "n=" , INTEGER , "," , "k=" , INTEGER , ")" ;
swizzle_config = ".with_swizzle(" , "pattern=" , SWIZZLE , ")" ;
scheduler_config = ".with_scheduler(" , [ "tile=" , TILE_SCHEDULER ] ,
                   [ "," , "kernel=" , KERNEL_SCHEDULE ] ,
                   [ "," , "epilogue=" , EPILOGUE_SCHEDULE ] ,
                   [ "," , "splits=" , INTEGER ] , ")" ;
scaling_config = ".with_scaling(" , "alpha=" , FLOAT , "," , "beta=" , FLOAT , ")" ;

iterator_config = ".with_iterator(" , ITERATOR , ")" ;
split_k_config = ".with_split_k(" , "mode=" , SPLIT_K , "," , "slices=" , INTEGER , ")" ;
operand_swap_config = ".with_operand_swap(" , BOOL , ")" ;

(* EPILOGUE *)
epilogue = ">>" , epilogue_op ;
epilogue_op = simple_activation | parametric_activation | broadcast_op | fusion_op | custom_op ;

simple_activation = "relu()" | "gelu()" | "silu()" | "mish()" | "sigmoid()" | "tanh()" | "hardswish()" ;

parametric_activation = "leaky_relu(" , "alpha=" , FLOAT , ")"
                      | "elu(" , [ "alpha=" , FLOAT ] , ")"
            | "clip(" , "min=" , FLOAT , "," , "max=" , FLOAT , ")"
                      | "clamp(" , "min=" , FLOAT , "," , "max=" , FLOAT , ")" ;

broadcast_op = "bias()" | "per_channel_scale()" | "per_row_scale()" | "per_col_scale()"
             | "row_broadcast(" , IDENTIFIER , ")" | "column_broadcast(" , IDENTIFIER , ")" ;

fusion_op = "scale(" , "alpha=" , FLOAT , ")"
          | "aux_store(" , IDENTIFIER , ")" | "aux_load(" , IDENTIFIER , ")" ;

(* Custom requires quotes for free-form expressions *)
custom_op = "custom(" , STRING , [ "," , "inputs=" , input_dict ] , ")" ;
input_dict = "{" , input_pair , { "," , input_pair } , "}" ;
input_pair = STRING , ":" , STRING ;

(* TERMINALS - unquoted identifiers *)
DTYPE = "fp64" | "float64" | "fp32" | "float32" | "fp16" | "float16"
      | "bf16" | "bfloat16" | "tf32"
      | "fp8_e4m3" | "e4m3" | "fp8_e5m2" | "e5m2"
      | "int8" | "s8" | "int16" | "s16" | "int32" | "s32"
      | "uint8" | "u8" | "uint16" | "u16" | "uint32" | "u32" ;
```

```
ARCH = "sm_100" | "sm100" | "sm_90a" | "sm90a" | "sm_90" | "sm90"
     | "sm_89" | "sm89" | "sm_86" | "sm86" | "sm_80" | "sm80" | "sm_70" | "sm70" ;

GEMM_LAYOUT = "RowMajor" | "ColumnMajor" ;
CONV_LAYOUT = "TensorNHWC" | "TensorNDHWC" ;
SWIZZLE = "Identity1" | "Identity2" | "Identity4" | "Identity8" | "StreamK" ;
TILE_SCHEDULER = "default" | "persistent" | "stream_k" | "streamk" ;
KERNEL_SCHEDULE = "auto" | "cp_async" | "cp_async_cooperative" | "tma" | "tma_cooperative" | "tma_pingpong"
    ;
EPILOGUE_SCHEDULE = "auto" | "tma" | "tma_cooperative" | "no_smem" ;
ITERATOR = "analytic" | "optimized" | "fixed_channels" | "few_channels" | "fixed_stride_dilation" ;
SPLIT_K = "none" | "serial" | "parallel" ;
BOOL = "true" | "false" ;

INTEGER = DIGIT , { DIGIT } ;
FLOAT = [ "-" ] , INTEGER , [ "." , INTEGER ] ;
STRING = "'" , { ANY_CHAR - "'" } , "'" ;
IDENTIFIER = LETTER , { LETTER | DIGIT | "_" } ;
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
LETTER = "a" .. "z" | "A" .. "Z" ;

(* CONSTRAINTS (compiler-enforced):
 *
 * REQUIRED: .with_dtype(), .with_arch(), .with_layout() (GEMM only)
 *
 * ARCH-GATED:
 *   SM70-89: .with_tile(), .with_swizzle(), .with_iterator()
 *   SM90+:   .with_threadblockshape(), .with_cluster(), .with_scheduler(), .with_operand_swap()
 *   SM80+: bf16 | SM90+: fp8_e4m3, fp8_e5m2
 *
 * SM90+ CONSTRAINTS (compiler-enforced):
 *   1. ALWAYS use sm_90a (not sm_90) - the "a" enables wgmma/warp-specialized features
 *      - Applies to ALL schedules: tma, tma_cooperative, cp_async, etc.
 *   2. Use .with_threadblockshape() NOT .with_tile() - compiler rejects .with_tile() on SM90+
 *   3. TMA alignment: (alignment * element_size) % 16 == 0
 *      - fp16/bf16: A/B/C >= 8 (8 * 2 = 16 bytes)
 *      - fp32: A/B/C >= 4 (4 * 4 = 16 bytes)
 *   4. kernel=tma_cooperative requires epilogue=tma_cooperative (or auto)
 *      - Mismatch causes "MMA_TILE_M must divide EPI_TILE_M" error
 *   5. Cooperative kernels: tile_m / cluster_m >= 128
 *      - Example FAIL: m=128 + cluster_m=2 → per_cta_m=64 < 128
 *   6. TMA cooperative REQUIRES explicit .with_stages(2) - compiler rejects missing stages
 *      - stages = (228KB - epilogue_smem - 8KB) / per_stage_smem
 *      - Large fp32 tiles exhaust smem (e.g., 256×128×64 fp32 → only 1 stage)
 *      - Fix: add .with_stages(2), use smaller tile, fp16/bf16, or kernel='tma'
 *   7. .with_operand_swap(true) REQUIRES square output (M == N) - runtime check enforced
 *      - Uses (A@B)^T = B^T @ A^T identity for zero-overhead layout reinterpretation
 *      - Enables RS GMMA variant (lower SMEM) and larger clusters for FP32 GEMM
 *      - Column-major [N,M] == Row-major [M,N] only when M == N
 *      - Note: FP16/BF16 already use RS GMMA with RowMajor B; operand_swap is FP32-specific benefit
 *
 * SM90+ GEMM template:
 *   gemm().with_dtype(input=fp16, acc=fp32, output=fp16)
 *     .with_layout(A=RowMajor, B=ColumnMajor, C=RowMajor).with_arch(sm_90a)
 *     .with_threadblockshape(m=256, n=128, k=64).with_alignment(A=8, B=8, C=8)
 *     .with_scheduler(kernel=tma_cooperative, epilogue=tma_cooperative)
 *
 * SM90+ FP32 GEMM with operand swap (square matrices only):
 *   gemm().with_dtype(input=fp32, acc=fp32, output=fp32)
 *     .with_layout(A=RowMajor, B=RowMajor, C=RowMajor).with_arch(sm_90a)
 *     .with_threadblockshape(m=128, n=128, k=32).with_cluster(m=4, n=2, k=1)
 *     .with_alignment(A=4, B=4, C=4).with_stages(3).with_operand_swap(true)
 *)
```

A.2 SOL Modeling Details

The SOL estimate is a roofline-style lower bound. It proceeds in four steps. First, problem characterization identifies the operators, dimensions, data types, total FLOPs, and best-case DRAM bytes. Second, hardware limits are obtained from H100 peak compute throughput and memory bandwidth, adjusted by observed or default clocks. Third, the roofline bound computes compute-bound time, memory-bound time, and the maximum of the two. Fourth, bottleneck classification compares arithmetic intensity to the ridge point.

The bound assumes best-case data movement, full fusion of feasible intermediate tensors, and hardware peaks. These assumptions can make the bound too tight when on-chip storage or layout

constraints force additional movement, or too loose when a legitimate optimized kernel uses lower precision than the reference formulation. We use FP32/TF32 assumptions for optimization steering and FP16 assumptions for scheduling and integrity filtering. The latter is conservative for detecting kernels that are faster than plausible reduced-precision execution.

A.2.1 MANTIS Workflow Details

MANTIS expands SOL-guided steering into six phases.

1. *Measure* profiles the current best kernel with Nsight Compute.
2. *Analyze* combines the SOL report, profiler summary, and source code to estimate the gap $g = t_{\text{best}}/t_{\text{SOL}}$ and attribute bottlenecks.
3. *Nominate* generates optimization hypotheses with causal links to those bottlenecks.
4. *Triage* ranks hypotheses by expected return and implementation or performance risk.
5. *Implement* executes the selected generate–compile–test–profile attempts.
6. *Summarize* records concise lessons that can be reused as cross-problem/iteration memory.

In the orchestrated controller, these phases are separate steps with structured artifacts passed between them. In the in-prompt controller, the same method is described in the agent prompt. The triage phase uses a gap-aware return-on-investment score,

$$\text{ROI}(h) = \frac{(\widehat{S}(h))^{1+\max(0,\log_{10}(g/5))}}{\widehat{R}_{\text{impl}}(h)\widehat{R}_{\text{perf}}(h)},$$

where $\widehat{S}(h)$ is estimated speedup and the two $\widehat{R}$ terms are implementation and performance risk estimates. The exponent favors more ambitious hypotheses when the current kernel is far from SOL.

We include a sample SOL report for KernelBench Problem 001 (4096×4096 FP32 GEMM). The main report uses FP32/TF32 assumptions matching the problem's declared precision. The FP16 augmentation shows the tighter bound used for scheduling and integrity checks.

Listing 4: SOL report for Problem 001 (4096×4096 FP32 GEMM).

```
# Speed-of-Light (SOL) Analysis

## 1. Problem Characterization
We analyze the reference op: square GEMM
C = A x B, where A,B,C in R^{NxN}, with N = 4096 and FP32 inputs.

### FLOP count (lower bound)
For dense GEMM, each output element is a dot product of length N:
- per C_{ij}: N multiplies + (N-1) adds ~ 2N FLOPs (2 FLOPs per MAC).
- total elements: N^2

Total FLOPs = 2N^3

Plug in N=4096:
- 4096^3 = 68,719,476,736
- 2 x 4096^3 = 137,438,953,472

=> Total FLOPs = 1.374e+11 FLOPs

### Global memory bytes (best-case lower bound)
Best-case DRAM traffic assumes:
- A is read once (each element brought from DRAM at least once)
- B is read once
- C is written once
- No extra reads/writes for intermediates (none exist beyond GEMM)

Each matrix has N^2 elements, FP32 = 4 bytes/element.

N^2 = 4096^2 = 16,777,216 elements
Bytes per matrix: 16,777,216 x 4 = 67,108,864 bytes
Total bytes (2 reads + 1 write): 3 x 67,108,864 = 201,326,592 bytes

=> Total bytes = 2.013e+08 bytes (~192 MiB)

### Arithmetic intensity
AI = FLOPs / Bytes = 1.374e+11 / 2.013e+08 ~ 682.6 FLOPs/byte

### TF32 note for FP32 GEMM
Inputs are FP32 and this is GEMM. Given the runtime context:
- torch.backends.cuda.matmul.allow_tf32: True
- torch.get_float32_matmul_precision(): highest
```

```
We assume PyTorch uses TF32 Tensor Cores on H100 (SM90) for matmul.
SOL therefore uses TF32 Tensor Core peak throughput.

---

## 2. Hardware Limits (Clock-aware)
GPU: NVIDIA H100 80GB HBM3 (Hopper, SM90)

Locked application clock:
- SM clock: 1500 MHz (locked application clock for benchmarking)
- Max SM clock: 1980 MHz
- Memory clock: 2619 MHz (max)

Peak specs (H100 SXM 80GB, at max clocks, dense, no sparsity):
- TF32 Tensor Core peak: 494.7 TFLOP/s (dense; the 989 TFLOP/s figure
  is with structured sparsity)
- FP16 Tensor Core peak: 989.4 TFLOP/s (dense; 1979 TFLOP/s is sparse)
- HBM3 peak bandwidth: 3.35 TB/s = 3350 GB/s

Clock-scaling (linear with clock ratio):

SM clock ratio:      1500 / 1980 = 0.7576
Effective TF32 peak: 494.7 TFLOP/s x 0.7576 = 374.77 TFLOP/s
Effective FP16 peak: 989.4 TFLOP/s x 0.7576 = 749.55 TFLOP/s

Memory clock ratio:  2619 / 2619 = 1.0
Effective bandwidth: 3.35 TB/s x 1.0 = 3.35 TB/s

=> Effective peak TF32:     ~374.77 TFLOP/s
=> Effective peak FP16:     ~749.55 TFLOP/s
=> Effective peak bandwidth: ~3.35 TB/s

---

## 3. Theoretical Minimum Time (TF32)

Compute-bound time:
  T_compute = FLOPs / Peak FLOP/s
            = 1.374e+11 / 374.77e+12
            = 0.367 ms

Memory-bound time:
  T_mem = Bytes / Peak BW
        = 2.013e+08 / 3.35e+12
        = 0.060 ms

SOL = max(T_compute, T_mem) = max(0.367, 0.060) = 0.367 ms
Primary bottleneck: Compute-bound.

---

## 4. Roofline Analysis

Ridge point = Peak FLOP/s / Peak BW
            = 374.77e+12 / 3.35e+12
            = 111.9 FLOPs/byte

Kernel AI ~ 682.6 FLOPs/byte >> Ridge ~ 111.9 FLOPs/byte
=> Compute-bound region on the roofline plot.

---

## 5. Summary
- Operation: FP32 GEMM (assumed TF32 Tensor Cores)
- Total work: 2N^3 = 1.374e+11 FLOPs for N=4096
- Best-case DRAM traffic: 2.013e+08 bytes (~192 MiB)
- Arithmetic intensity: ~682.6 FLOPs/byte
- Clock-aware effective peaks (at 1500 MHz locked clock):
  - TF32 compute: ~374.77 TFLOP/s (dense, scaled by 1500/1980)
  - HBM bandwidth: ~3.35 TB/s
- Lower-bound times:
  - Compute time: ~0.367 ms
  - Memory time:  ~0.060 ms

=> Theoretical minimum execution time (SOL): ~0.367 ms
=> Primary bottleneck: Compute throughput

================================================================
# FP16 Augmentation
================================================================

Precision Assumption:
  Kernel casts FP32 data to FP16 on-chip and uses FP16 Tensor Cores
  (2x throughput). Inputs, outputs, and weights remain FP32 in DRAM
  - memory traffic is unchanged.

- Compute peak: FP16 TC dense (749.55 TFLOP/s at 1500 MHz,
                scaled from 989.4 TFLOP/s at 1980 MHz)
- Memory traffic: FP32 data (4 bytes/element) - NOT halved,
                  because inputs/outputs enter and leave DRAM as FP32
- HBM bandwidth: 3350.0 GB/s
```

```
Work Estimates:
- Total FLOPs:        1.3744e+11 (unchanged)
- Total bytes (FP32): 2.0133e+08 (unchanged from TF32 SOL)

Theoretical Minimum Times:
              TF32 (dense)  FP16 (dense)
  Peak        374.77        749.55        TFLOP/s
  Compute     0.3667 ms     0.1834 ms
  Memory      0.0601 ms     0.0601 ms
  SOL         0.3670 ms     0.1834 ms
  Bottleneck  compute       compute

FP16/TF32 ratio: 0.500x (pure 2x compute speedup; memory-bound
unchanged because DRAM I/O remains FP32)

================================================================
# Structured JSON Output
================================================================

{
  "problem_id": "001",
  "timestamp": "2026-01-27 19:22:15",
  "total_flops": 137438953472,
  "total_bytes": 201326592,
  "theoretical_runtime_ms": 0.367,
  "sm_clock_mhz": 1500.0,
  "peak_type": "TF32 TC (dense)",
  "peak_tflops_effective": 374.77,
  "theoretical_runtime_ms_fp16": 0.1834,
  "fp16_peak_tflops_effective": 749.55,
  "fp16_note": "FP16 compute (2x TC throughput), FP32 memory (inputs/outputs/weights stay FP32 at DRAM boundary)",
  "details": {
    "full_analysis": "<see TF32 markdown report above>"
  }
}
```

A.3 Experimental Setup

Each KernelBench problem is converted into a workspace containing a driver, initial PyTorch fallback, build targets, correctness checks, and Nsight Compute profiling. The same bootstrap profile supplies the PyTorch reference time for all variants. We use OpenHands as the multi-turn agent runtime and give μCUTLASS variants an additional compile tool. Runs are matched at 40 attempts per problem. Orchestrated MANTIS uses 5 outer iterations, 2 hypotheses per iteration, and 4 attempts per hypothesis.

Variant	μCUTLASS	SOL guidance	Total attempts
MI	–	–	40
MI + μCUTLASS	✓	–	40
In-prompt steering	–	In prompt	40
In-prompt steering + μCUTLASS	✓	In prompt	40
Orchestrated steering	–	MANTIS	40
Orchestrated steering + μCUTLASS	✓	MANTIS	40

Table 4: Full variant grid used to isolate representation and SOL-guidance effects.

Variant	M	A	N	T	I	S	Cross-problem memory
MANTIS	✓	✓	✓	✓	✓	✓	✓
MNTIS	✓	–	✓	✓	✓	✓	✓
MANIS	✓	✓	✓	–	✓	✓	✓
MANTI	✓	✓	✓	✓	✓	–	–
MANTIS-noXmem	✓	✓	✓	✓	✓	✓	–

Table 5: MANTIS ablations used to study which steering components matter.

A.3.1 KernelBench LLM-Relevant Problem Subset

Shortcut exclusions. We exclude L2-28 (BMM–InstanceNorm–Sum–ResidualAdd–Multiply) and L2-80 (GEMM–Max–Subtract–GELU). For L2-28, the bootstrapped C++ reference uses a singleton spatial domain for instance normalization, making the normalized result zero and allowing the output $y \odot y$ without executing the linear layer. This shortcut arises in the generated evaluation reference. For L2-80, the maximum reduction leaves a singleton dimension; subtracting the mean over that

Table 6: KernelBench problems in our LLM-relevant subset (Levels 1–3).

Level	ID	Rationale for inclusion
L1	1	Basic GEMM baseline.
L1	2	LLM-like GEMM shapes (e.g., M=2048, K=8192, N=4096).
L1	3	Batched matmul (BMM) used in attention score/value products.
L1	4	Matrix–vector multiply representative of single-token decode.
L1	6	Matmul with large K (common in MLP projections).
L1	7	Matmul with small K (e.g., attention head dimension).
L1	8	Irregular shapes (non power-of-2) that occur in practice.
L1	9	Tall-skinny matmul (prefill with long sequences).
L1	16	Transposed-A layout variant common in GEMM calls.
L1	17	Transposed-B layout variant common for weight matrices.
L1	18	Both operands transposed (layout coverage).
L1	21	Sigmoid for gating patterns (e.g., GLU-style gates).
L1	22	Tanh used in some gating/activation variants.
L1	23	Softmax (core attention primitive).
L1	24	LogSoftmax (normalization used in token prediction and loss computation).
L1	25	SiLU/Swish (dominant MLP activation in many modern LLMs).
L1	26	GELU (used in GPT-2/BERT and some contemporary models).
L1	36	RMSNorm (dominant normalization in modern decoder LLMs).
L1	40	LayerNorm (still used in many transformer variants).
L1	47	Sum reduction used inside normalization and statistics.
L1	48	Mean reduction used inside LayerNorm and statistics.
L1	67	1D convolution used in SSM/long-conv text models.
L1	76	Dilated/strided 1D conv variant for hierarchical SSM designs.
L1	86	Depthwise-separable conv (efficient channel-wise processing).
L1	87	Pointwise conv (channel mixing / fusion proxy).
L1	88	Fast GELU approximation (common fused activation variant).
L1	89	Cumsum (prefix-scan) used in SSM/linear-attention recurrences.
L1	90	Cumprod used in some state-space dynamics.
L1	91	Exclusive cumsum variant (scan coverage).
L1	92	Reverse cumsum variant (reverse-time scan coverage).
L1	95	Cross-entropy loss (standard LLM training objective).
L1	97	Scaled dot-product attention (maps to FlashAttention in practice).
L2	9	Fused matmul + elementwise (proxy for epilogue and MLP fusions).
L2	29	Fused linear + activation (MLP fusion pattern).
L2	37	Fused linear + normalization (proxy for norm-adjacent fusions).
L2	40	Fused linear + residual add (transformer block core pattern).
L2	41	GEMM + multi-activation fusion (MLP epilogue diversity).
L2	53	GEMM + activation fusion (covers activation/scaling variants).
L2	56	Matmul + gating + reduction (proxy for gated aggregation patterns).
L2	59	Matmul + SiLU/Swish + scaling (common MLP fusion).
L2	62	Matmul + normalization + activation (fused post-linear processing).
L2	63	GEMM + ReLU + divide (activation + scaling fusion).
L2	66	Attention-like fusion with dropout (training attention pattern).
L2	70	GEMM + sigmoid gate + residual add (SwiGLU-like gating proxy).
L2	76	GEMM + bias add + ReLU (classic epilogue fusion).
L2	81	Complex epilogue fusion with Swish (stress fused elementwise).
L2	86	Matmul + divide + GELU (MLP fusion with scaling).
L2	88	SwiGLU-like gated fusion (common LLM MLP pattern proxy).
L2	94	Expert MLP proxy: GEMM + bias/activation + normalization.
L2	97	Matmul + bias + norm + Swish (fused post-linear processing).
L2	99	Attention-like fusion (matmul + GELU + softmax).
L3	1	MLP (basic feedforward block).
L3	2	Shallow wide MLP (LLM FFN-like width).
L3	3	Deep narrow MLP (depth/width trade-off).
L3	43	Causal attention block (core decoder attention).
L3	44	Full GPT block (attention + FFN).
L3	48	Mamba SSM block (emerging text SSM architecture).
L3	49	Mamba SSM with state output (streaming/stateful variant).
L3	50	ReLU self-attention variant (alternative attention formulation).

dimension and applying GELU gives zero, allowing the GEMM to be skipped. Appendix A.5 describes the broader integrity review.

Included KernelBench IDs (Levels 1–3). **L1:** 1, 2, 3, 4, 6, 7, 8, 9, 16, 17, 18, 21, 22, 23, 24, 25, 26, 36, 40, 47, 48, 67, 76, 86, 87, 88, 89, 90, 91, 92, 95, 97.
L2: 9, 29, 37, 40, 41, 53, 56, 59, 62, 63, 66, 70, 76, 81, 86, 88, 94, 97, 99.
L3: 1, 2, 3, 43, 44, 48, 49, 50.

A.3.2 Attempt Outcomes and Candidate Quality

We analyze 2,360 raw-code MI attempts and 2,340 μCUTLASS-agent attempts for GPT-5.2 across the 59 problems. The available μCUTLASS records for problem 203 contain 20 attempts; all other problem–variant pairs contribute 40 attempts.

Outcome or metric	Raw-code MI	μCUTLASS
Recorded attempts	2,360	2,340
C++ compilation failures	441 (18.7%)	120 (5.1%)
Linking failures	5 (0.2%)	9 (0.4%)
Runtime failures after successful build	65 (2.8%)	197 (8.4%)
Correctness failures after successful build	154 (6.5%)	241 (10.3%)
Integrity-valid, timed candidates	1,382	1,444
Of these, candidates beating PyTorch	482 (34.9%)	693 (48.0%)
New incumbents exceeding PyTorch	111 (4.7%)	150 (6.4%)

Table 7: GPT-5.2 attempt outcomes. Failure and new-incumbent percentages use all recorded attempts as their denominator. The fraction beating PyTorch uses integrity-valid, timed candidates as its denominator.

Failure categories are mutually exclusive: an attempt is assigned to its first failed stage. A timed candidate must pass compilation, linking, and correctness checks and have a positive measured runtime. For this attempt-level analysis, we retain the recorded gaming and static-integrity exclusions, including insufficient executed-work coverage and PyTorch-only flags. A new incumbent must strictly improve on all previous accepted candidates for that problem and exceed the PyTorch baseline.

For the GPT-5.2 μCUTLASS agent, six of 1,560 recorded μCUTLASS tool calls (0.38%) failed grammar parsing, while 483 (31.0%) returned validation or compiler errors, commonly involving shared-memory limits, tile configurations, alignment, or scheduling constraints. Agents could revise the DSL and retry within the same attempt. We therefore report these tool-call failures separately from attempt-level compilation, runtime, and correctness failures.

A.3.3 Performance by Operator Category

Table 8 groups the GPT-5.2 μCUTLASS-agent results using the same build, correctness, timing, and recorded-integrity checks as Appendix A.3.2. A problem is *solved* if at least one candidate passes these checks, even if it is slower than PyTorch. We select each solved problem's best accepted speedup and compute the geometric mean over solved problems within each category. Unsolved problems are excluded from that mean but included in the category totals. Results include hybrid implementations that combine generated kernels with custom CUDA (Section 3).

Operator category	Solved / total	Geomean speedup
Dense linear algebra	11 / 11	$2.85\times$
Elementwise / normalization / reduction / scan	14 / 16	$2.23\times$
Convolution	4 / 4	$0.40\times$
Fused Level-2	19 / 19	$4.61\times$
Composite attention / model blocks	5 / 9	$4.33\times$
Overall	53 / 59	$2.85\times$

Table 8: GPT-5.2 μCUTLASS-agent performance by operator category. Speedups are relative to PyTorch and use the best accepted candidate for each solved problem.

Dense linear algebra comprises the selected L1 IDs 1–4, 6–9, and 16–18. Convolution comprises L1-67, L1-76, L1-86, and L1-87. Composite attention/model blocks comprise L1-97 and the selected Level-3 problems. All selected Level-2 problems form the fused category; the remaining selected Level-1 problems form the elementwise/normalization/reduction/scan category. The complete workload list appears in Table 6.

A.3.4 Search Cost and Deployment Savings

Kernel speedup measures execution performance after optimization. Search tokens, LLM charges, and optimization wall time describe the expenditure incurred to produce the kernel. The scheduling efficiency gain is a ratio of speedup per search token; it gives proportional weight to speedup retention and inverse token consumption. It can therefore increase even when the final kernel is slower than

the fixed-budget result. We report both factors separately and do not infer deployment break-even or lifetime savings without an invocation count and execution-cost model.

For the 58 completed GPT-5.2 μCUTLASS-agent searches, each with 40 attempts, the mean recorded implementation-agent usage was 11.72 million prompt-plus-completion tokens and $14.4 per problem. Recorded optimization wall time ranged from 1.89 to 4.96 hours per problem (mean 3.00 hours). Token and dollar totals sum the available counters from 2,318 of 2,320 archived attempts; two attempts lack usage/cost counters. The dollar figures are logged implementation-agent cost estimates, excluding separate conversation-summary calls, bootstrapping, SOL analysis, offline integrity review, and hardware charges. Wall time includes the optimization loop's agent/tool execution and summaries, so its scope differs from the logged LLM charges.

For Figure 5, the plotted cost is

$$C_{\text{proxy}} = r_m \left(N_{\text{prompt}} + N_{\text{completion}}\right)/10^6,$$

normalized by the largest cost among the swept policies and fixed-budget points used to construct the plot. The model-specific weights r_m are $0.25, $1.25, and $1.75 for GPT-5-mini, GPT-5, and GPT-5.2, respectively. This proxy applies the input-token rate to all recorded tokens, including failed or integrity-rejected attempts. Token savings also do not imply proportional wall-time savings because generation, compilation, testing, and profiling have different costs across attempts.

A.3.5 Blackwell Support and Localized Tuning

We include a case study focused on an NVIDIA Blackwell GPU to demonstrate that μCUTLASS can target kernels on a newer architecture and that its implementation choices can be explored through localized tuning.

Operators. μCUTLASS supports Blackwell workloads spanning FP16, BF16, FP8, and block-scaled FP4, including native NVFP4, MXFP4, and the distinct benchmark-format FP4 path. Supported operator families include dense, grouped, and batched GEMM; convolution; and attention. Operator extensions include fused bias, scaling, and activation epilogues, SwiGLU, and FP4 GEMM with maximum-absolute-value output. Table 9 summarizes the data types, operator families, and variants exercised by the measured GB300 examples, which include 256 variant measurements across 40 example drivers.

Category	Measured GB300 examples
Dense GEMM	FP16; direct, tensor-scaled, and blockwise FP8; native NVFP4, MXFP4, and benchmark-format FP4; FP4 with maximum-absolute-value output; NN/NT/TN/TT variants.
Grouped and batched GEMM	Grouped FP16, BF16, blockwise FP8, and benchmark-format FP4; batched GEMM.
Fused operations	Bias with SiLU, alpha/beta scaling with bias, activation epilogues, and SwiGLU.
Convolution	Conv2d forward, input-gradient, and weight-gradient variants, including BF16; Conv3d forward and input-gradient; depthwise Conv2d; causal and cached causal Conv1d.
Attention	Decode, paged decode, and ragged causal prefill.

Table 9: Categories exercised by our GB300 microbenchmark catalog.

Blackwell lowering supports one- and two-SM matrix-multiply schedules. Additional execution-control experiments exercise tile-dependent tensor-memory (TMEM) single-buffer and ping-pong policies. These experiments vary multiple configuration choices together and therefore do not isolate TMEM's individual performance effect.

Localized tuning. For a supported workload, μCUTLASS deterministically emits a kernel from a DSL specification by default. Localized autotuning explores implementation choices left unspecified, including supported tile shapes, cluster dimensions, pipeline stages, and execution schedules, through a bounded search. It holds the operation, data types, mathematical epilogue, storage and quantization contracts, and architecture fixed. Candidates are materialized as ordinary DSL programs, validated by the compiler, batch-compiled, and checked against the workload's numerical reference before

selection. Large workloads use a smaller screening shape to rank candidates, then measure multiple finalists on the requested target shape.

Table 10 summarizes recorded runs using CUDA 13.4.1, NGC PyTorch 26.08, and CUTLASS 4.8.0dev. The μCUTLASS reference is a fully specified starting configuration, using compiler defaults where available. Both latency columns measure the same operation, precision, and target shape; speedup is reference latency divided by latency with localized tuning. Measurements use CUDA events, separately from the Nsight Compute methodology of the primary H100 results.

Workload	μCUTLASS reference (μs)	μCUTLASS + localized tuning (μs)	Speedup	Search samples	Cold search (s)
FP16 GEMM	99.6	92.6	1.08×	16	184.2
FP8 GEMM	125.4	54.9	2.28×	16	178.3
Native NVFP4 GEMM	26.8	26.8	1.00×	12	311.9
Benchmark-format FP4 GEMM	28.5	28.5	1.00×	12	300.6
FP16 Conv2d	13.8	11.0	1.26×	8	139.5

Table 10: Recorded GB300 kernel latencies and localized tuning costs. GEMMs use $M = N = K = 4096$; FP16 and FP8 use AB^T. Conv2d uses $N = 1$, $H = W = 56$, $C = K = 64$, and a 3×3 filter. Search samples include screening candidates, only a subset of which are promoted to the target. Cold search is total tuning wall time, including compilation.

Localized tuning improves the measured FP16, FP8, and Conv2d configurations, while the compiler defaults remain best among the evaluated candidates for both FP4 modes. Compilation accounts for approximately 75–95% of cold search time in these runs.

A.3.6 Scheduling Replay Protocol

Scheduling experiments are offline replays of completed traces. A replay scheduler assigns work items in breadth-first round-robin order, where a work item is an attempt for MI agents or an iteration for orchestrated MANTIS. A problem remains eligible while it is behind PyTorch. After it beats PyTorch, it can become ineligible through either of two rules. The SOL-headroom rule stops a problem when $t_{\text{best}} \leq (1 + \epsilon)t_{\text{SOL}}$. The no-progress rule stops a problem after w consecutive attempts without best-speedup improvement. We sweep $\epsilon \in \{25\%, 50\%, \ldots, 300\%\}$ and $w \in \{0, 4, 8, 12, 16, 20\}$, then report token savings, attempt savings, speedup retention, and efficiency gain. Each fixed policy uses only the completed prefix of a trace for its stopping decisions. The completed fixed-budget traces are used afterward to evaluate retention and select policies from the sweep; this retrospective selection is separate from policy execution.

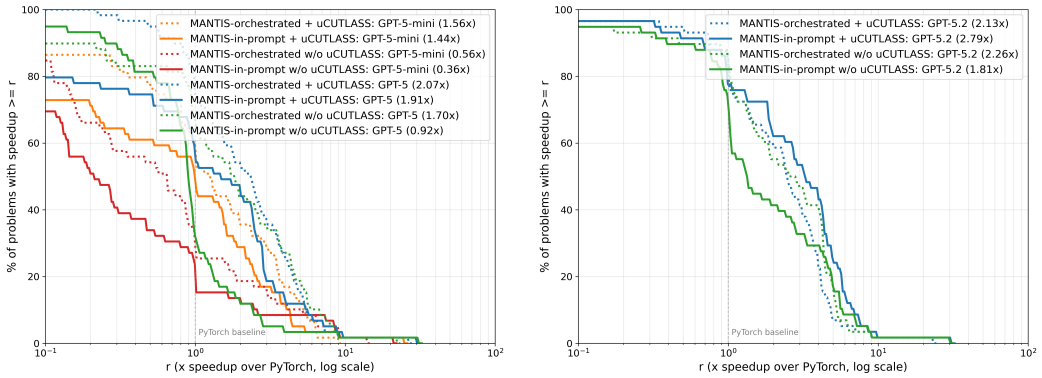
A.4 Detailed Result Breakdown

The main text reports the primary geomean speedups. The detailed Fast-p and Attempt-Fast-$p(2)$ curves support three qualitative findings. μCUTLASS improves both final speedup and early-attempt convergence. SOL steering is most valuable for weaker and mid-tier models. With μCUTLASS, the strongest model can sometimes do better with in-prompt guidance than with rigid orchestration.

Component ablations show that all MANTIS components matter for GPT-5-mini without μCUTLASS, while Analyze is the most consistently useful component after adding μCUTLASS. The observed run-to-run and nearby-variant variation is larger for GPT-5-mini (13–15% coefficient of variation) than for GPT-5.2 (5–7%), but all GPT-5-mini configurations with μCUTLASS remain above the GPT-5 raw-code MI baseline.

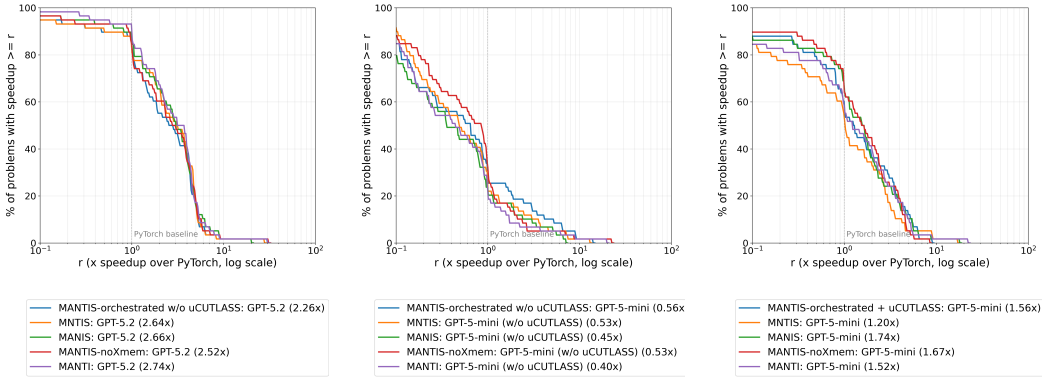
The detailed results also show model-dependent orchestration effects. Without μCUTLASS, orchestrated steering dominates in-prompt steering for GPT-5-mini and GPT-5. With μCUTLASS, the gap narrows, consistent with partial substitution between representation and controller structure. GPT-5.2 reverses the pattern under μCUTLASS, where in-prompt guidance performs better than rigid orchestration.

These plots explain two supporting results behind Section 6.2. First, they separate the effect of implementing SOL guidance as explicit orchestration from describing it in the prompt. Second, they show which MANTIS phases matter most once the code-generation representation changes.



(a) GPT-5-mini and GPT-5 (b) GPT-5.2

Figure 7: Orchestrated versus in-prompt SOL guidance. Dotted curves show orchestrated steering, solid curves show in-prompt steering, and colors identify representation choices.



(a) GPT-5.2, no μCUTLASS (b) GPT-5-mini, no μCUTLASS (c) GPT-5-mini, with μCUTLASS

Figure 8: MANTIS component ablations on configurations where SOL guidance is most informative.

A.4.1 Offline Replay of SOL-Guided Budget Scheduling

The main paper uses the empirical Pareto frontier to summarize offline scheduling replay. The additional replay sweeps below show how each stopping signal contributes. The independent sweep isolates the SOL-headroom threshold and no-progress window, while the best-policy summary retrospectively selects the highest efficiency-gain policy that preserves at least 95% of fixed-budget geomean speedup for each configuration.

A.5 Integrity Review Protocol and Outcomes

The integrity pipeline assigns each attempt to mutually exclusive categories. *No Issues* and *Minor Issues* are accepted. Minor issues include small math approximations, cached parameters, contiguity assumptions, or default-stream usage when the issue is not expected to materially affect performance if fixed. Rejected categories include SOL-ceiling violations, PyTorch-only attempts, Original Gaming, and Inherited Gaming. Gaming categories include benchmark-input exploitation, constant or hardcoded outputs, skipped computation steps, fake transpose/view reinterpretation when a real transform is required, and incomplete computation.

In the attempt-level audit of Table 7, 1,759 GPT-5.2 μCUTLASS+MI attempts passed compilation, linking, and numerical checks and had positive timings. The recorded gaming and static-integrity flags excluded 315 of these attempts (17.9%), leaving 1,444 accepted candidates. This uses the

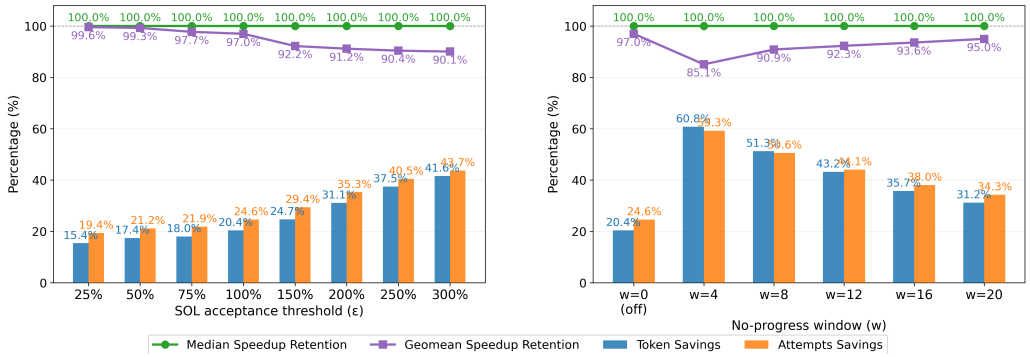


Figure 9: Independent scheduler parameter sweeps in offline replay. Bars show token and attempt savings, while lines show retrospectively measured median and geomean speedup retention.

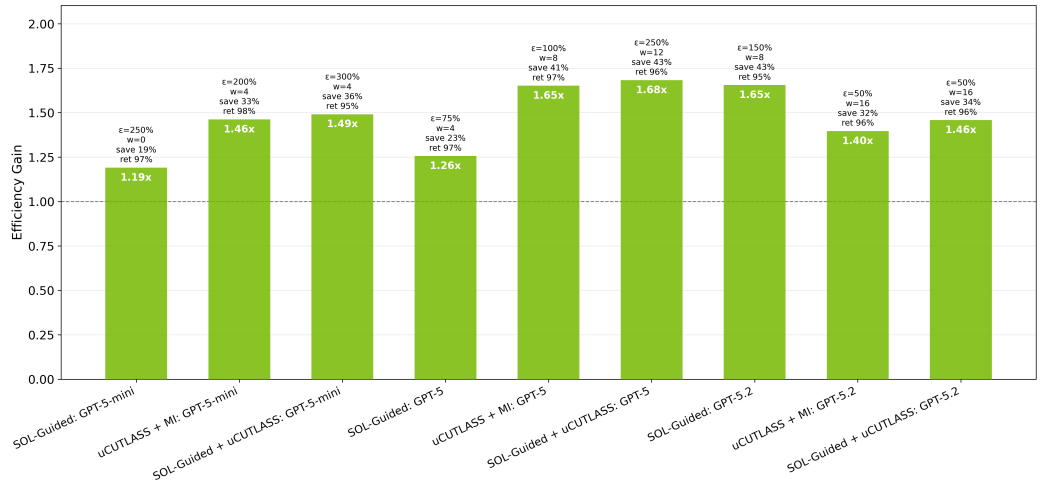


Figure 10: Retrospectively selected best replay policy per variant under a $\geq 95\%$ fixed-budget geomean-retention constraint.

same recorded-exclusion policy as the table, including insufficient executed-work coverage flags, and illustrates why passing numerical tests alone is insufficient.

A possible extension is to require candidates to remain within the DSL, check their specifications against the reference computation, and verify that lowering preserves declared operations, tensor shapes, data types, and input-to-output dependencies. This would also require preventing wrapper code from bypassing the verified computation; such semantic verification remains future work.

One motivating loophole comes from KernelBench L2-80, whose reference computes a GEMM, reduces with `max` to a singleton dimension, subtracts the mean over that singleton dimension, and applies GELU. The subtraction produces zero, so an agent can emit a trivial zero-producing kernel that passes the harness while skipping the intended GEMM. We exclude such known loophole problems from the evaluation subset, but similar shortcuts can arise in less obvious forms. This motivates using SOL-derived expected work as a detector input rather than relying only on a list of known exploit patterns.

Prompt guardrails reduce but do not eliminate problematic behavior. In a GPT-5-mini repeat, explicit anti-PyTorch-only instructions reduced PyTorch-only flags for μCUTLASS+MI from 345 to 51 attempts, but gaming for the same variant increased from 50 to 95 attempts. This motivates detector-backed reporting rather than instruction-only mitigation.

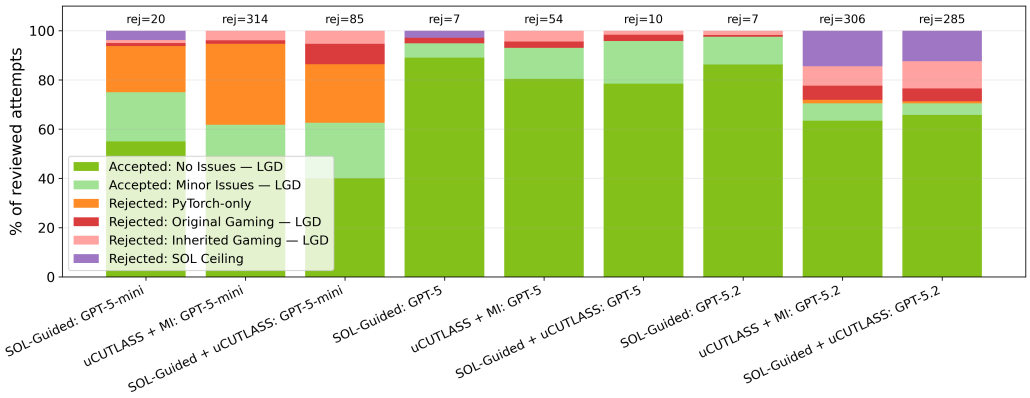


Figure 11: Review outcome composition for each variant. Bars show accepted attempts, minor issues, SOL-ceiling rejections, PyTorch-only rejections, original gaming, and inherited gaming.

Table 11: Prompt-level integrity guardrails on GPT-5-mini. Run 1 used the original prompt. Run 2 added anti-PyTorch-only and anti-gaming instructions.

	PyTorch-only		Gaming	
Variant	Run 1	Run 2	Run 1	Run 2
MI	113	8	9	1
μCUTLASS + MI	345	51	50	95
SOL-guided	15	10	3	11
μCUTLASS + SOL-guided	63	26	35	19

A.6 External Baseline and Run Variation

We compare against the public Sakana AI CUDA Engineer archive as contextual evidence. Because our PyTorch baseline and measurement methodology differ, we use the archive's reported speedup numbers for a fair comparison. For our 59-problem subset, 2 problems have no correct kernel in the archive. Among the remaining problems, our review accepts 52 kernels and rejects 5 after a fallback process over candidate kernels. The accepted kernels achieve $1.13\times$ geomean speedup over Sakana's own PyTorch baseline. We also include a variation summary to show that the main conclusions are not driven by one unusually favorable run.

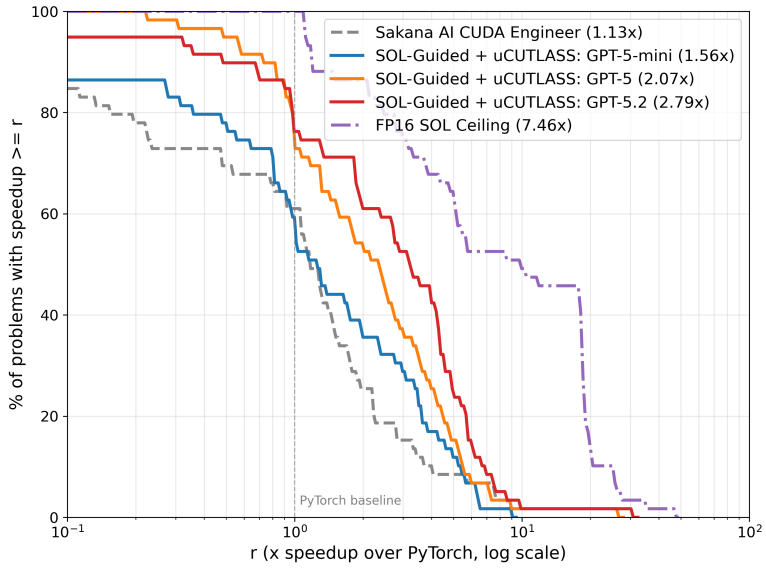


Figure 12: Fast-p curves comparing μCUTLASS+SOL-guided variants against the Sakana AI CUDA Engineer archive. The FP16 SOL curve shows the theoretical speedup limit under the reduced-precision bound.

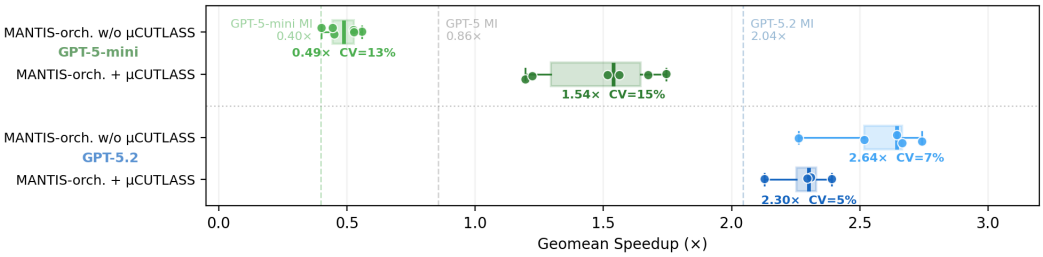


Figure 13: Run-to-run and nearby-variant variation across model tiers. CV denotes coefficient of variation.

NeurIPS Paper Checklist

1. **Claims.** Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   [Yes]. The claims are scoped to LLM-guided GPU kernel optimization on a 59-problem KernelBench subset on H100, and the main results in Sections 1 and 6 support those claims.

2. **Limitations.** Does the paper discuss the limitations of the work performed by the authors?

   [Yes]. The paper discusses workload, model, and hardware scope in Section 5, DSL coverage in Section 3, SOL modeling assumptions in Section 4, and integrity-checking, variability, and external-comparison limits in Sections 5.2, 6.4, and 6.5.

3. **Theory, Assumptions, and Proofs.** If the paper includes theoretical results, are the assumptions and proofs stated?

   [N/A]. The paper uses roofline-style analytical SOL estimates as engineering signals. Modeling assumptions are stated in Section 4 and Appendix A.2.

4. **Experimental Result Reproducibility.** Are the steps needed to reproduce or verify the main results described?

   [Yes]. Sections 5 and 5.2 describe the workload, models, budgets, metrics, timing, and integrity policy. Appendices A.1, A.2, A.3, A.3.1, A.3.6, and A.5 give the DSL, SOL

assumptions, setup details, problem subset, scheduling replay protocol, and integrity-review categories needed to verify the analyses.

5. **Open Access to Data and Code.** Does the paper include code, data, and instructions needed to reproduce the main experimental results?

   [No]. We do not include an anonymized code artifact with this submission. The paper provides the KernelBench subset, experimental protocol, metrics, integrity policy, and full μCUTLASS EBNF grammar, but full code, prompts, per-run SOL reports, and analysis scripts are not included with the submission.

6. **Experimental Setting and Details.** Are all important experimental details specified?

   [Yes]. Section 5 gives the hardware, locked-clock timing setup, workload, model tiers, budgets, runtime measurement, metrics, and filtering policy. Appendix A.3 gives additional setup and ablation details.

7. **Experiment Statistical Significance.** Does the paper report error bars or other information about statistical significance or variability?

   [Yes]. Section 6.5 and Figure 13 report repeat/ablation-based variability and coefficient-of-variation ranges. The paper does not claim classical statistical significance from independent random trials.

8. **Experiments Compute Resources.** Does the paper provide sufficient information about compute resources?

   [Yes]. Section 5 identifies the H100/SM90a hardware, locked-clock timing setup, model tiers, token prices, and matched attempt budgets; Appendix A.3 gives the full variant setup, and Section 6.3 reports token-cost trade-offs.

9. **Code of Ethics.** Have the authors read and ensured conformance with the NeurIPS Code of Ethics?

   [Yes]. The work is a systems/optimization study using public benchmarks and model APIs. The integrity pipeline is included specifically to reduce misleading benchmark claims.

10. **Broader Impacts.** If appropriate, does the paper discuss potential negative societal impacts?

    [Yes]. Sections 4, 5.2, and 6.4 discuss benchmark gaming and misleading automated optimization outputs. The main mitigation is detector-backed integrity filtering.

11. **Safeguards.** Are safeguards in place for responsible release of models or assets with high misuse risk?

    [N/A]. The paper does not release a model or other high-risk asset.

12. **Licenses.** Were existing assets cited and used according to license and terms?

    [Yes]. The paper cites the existing assets used in the study, including KernelBench, CUTLASS/CuTe, OpenHands, Nsight/NVIDIA tools where relevant, and the Sakana archive.

13. **New Assets.** If new assets are released, are they documented?

    [N/A]. We do not release new assets with this submission.

14. **Crowdsourcing and Human Subjects.** If crowdsourcing or human-subject research was used, are details provided?

    [N/A]. The work does not use crowdsourcing or human-subject data collection.

15. **IRB Approvals.** If applicable, were participant risks described and IRB approvals obtained?

    [N/A]. The work does not involve human subjects.

16. **Declaration of LLM Usage.** Does the paper describe LLM usage when it is an important or non-standard component of the research?

    [Yes]. LLMs are central to the evaluated agents, SOL report generation, and LLM-based game detector. The evaluated model tiers are described in Section 5, and the SOL and integrity uses are described in Sections 4 and 5.2.