KERNELMASTER: AGENTIC GPU KERNEL OPTIMIZATION FROM SEMANTICS TO EXECUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent systems can generate and refine GPU kernels, yet optimizing an unfamiliar operation often requires extensive trial and error. GPU kernel optimization requires coordinated choices about how to reorganize computation, distribute and schedule parallel work, and implement it using GPU instructions. However, runtime and profiler feedback reveal performance bottlenecks but do not directly prescribe effective program changes, making it difficult for an Agent to connect these observations to actionable revisions without repeated trial and error. We present KernelMaster, a system that guides Agent-driven GPU kernel optimization through mathematical dataflow search and hardware-calibrated execution synthesis. We expose mathematical transformations through an Agent-operable MathIR, validate candidates through source coverage and replay, and project selected dataflows into LoopIR. We then synthesize Warp pipelines and CTA programs using target-GPU measurements of service time, concurrency, and HBM bandwidth. The selected execution graph guides Agent-driven CUDA implementation and refinement. On five newly specified attention kernels, KernelMaster achieves 1.08–$3.04\times$ speedup over the strongest Agent baselines and 4.91–$9.18\times$ over PyTorch references. Across established kernels on Hopper and Blackwell, it approaches or exceeds expert implementations, with speedups up to $1.91\times$.

Our code is at: `anonymous.4open.science/r/KernelMaster-4A2B/`.

1 INTRODUCTION

Efficient GPU kernels are essential to the training and inference performance of deep learning models. As models introduce new computations and deployment hardware evolves, developers must implement and optimize specialized kernels for different computations, data layouts, and GPU architectures. FlashAttention (Dao et al., 2022) and FlashInfer (Ye et al., 2025) demonstrate the substantial gains from specialized optimization, but extending these gains to emerging workloads requires sustained effort from developers with algorithmic and hardware expertise. Automating kernel development can therefore reduce the engineering cost of translating model innovations into efficient deployments. Recent advances in LLM code generation and tool use enable Agents to generate CUDA or Triton (Tillet et al., 2019) kernels from task descriptions or reference implementations, such as PyTorch programs. To improve the performance of these initial implementations, Agents iteratively revise the generated code using compiler diagnostics, correctness checks, and profiling results. KernelBench (Ouyang et al., 2025) provides a common set of tasks and an evaluation framework for these capabilities, supporting GPU kernel generation as an application of Agent-based automation.

Toward this goal, recent systems organize the Agent's generation, evaluation, and refinement process in different ways. AKO4X provides benchmarking, profiling, and recorded optimization trajectories (Xie et al., 2026). AdaExplore combines failure-driven adaptation with diverse candidate search, including structural regeneration (Du et al., 2026). AutoKernel orchestrates profiling and iterative Triton/CUDA optimization (Jaber & Jaber, 2026); KernelAgent combines problem decomposition, verified generation, and hardware-guided refinement (PyTorch, 2026). These systems broaden exploration and improve diagnosis, yet a measured implementation still combines several coupled decisions: which mathematical intermediates to materialize, how to partition and pipeline work, and how to realize the plan in instructions. Latency and profiler feedback expose the outcome of that combination. Determining whether to revise the dataflow, execution structure, or concrete code can

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

therefore require many complete generation, compilation, and measurement cycles, especially for unfamiliar computations.

We observe that when optimization branches into many possible directions without a clear basis for choosing among them, the available feedback can leave Agents spending substantial effort on ineffective program revisions, sometimes failing even to match baseline performance (Sun et al., 2026; Wang et al., 2026a). This uncertainty arises from the many possible mathematical transformations, diverse execution schedules, and instruction-level implementations with different performance characteristics (Zhu et al., 2026). While Agents can effectively refine instruction-level implementations, they still struggle to identify promising mathematical transformations and execution schedules. To address this challenge, we introduce KernelMaster, which combines mathematical dataflow search with hardware-calibrated execution synthesis to guide Agent-driven kernel implementation and refinement. Our contributions are:

- **Agent-operable mathematical dataflow discovery.** We introduce MathIR to expose bounded mathematical transformation opportunities that Agents can explore and validate through source coverage, algebraic state contracts, and executable replay. We deterministically project validated candidates into LoopIR, making iteration, state, and dependencies explicit for downstream execution search.
- **Feedback-coupled hierarchical execution synthesis.** We jointly search Warp pipelines and CTA programs, preserving dependencies and value lifetimes across execution levels. We reuse target-GPU measurements of local execution, concurrency, and memory response to rank complete execution graphs and refine Warp candidate selection under different CTA configurations. We then translate the selected graph into an execution contract that guides Agent-driven CUDA implementation and profiler-driven refinement.

We evaluate this combination from two perspectives: optimization from reference semantics on unfamiliar kernels, and adaptation to target hardware on established kernels. On five newly specified attention kernels, KernelMaster exceeds the fastest of four Agent baselines by 1.08–$3.04\times$ and the PyTorch references by 4.91–$9.18\times$. Its first implementation surpasses all baselines' final results on two tasks, and subsequent rounds improve the remaining tasks. Across ten established kernel–GPU comparisons, it achieves lower measured latency in eight and remains close in two, with speedups up to $1.91\times$ on Hopper and Blackwell. Together, these results support the central premise of KernelMaster: discovering mathematical dataflows and synthesizing their hardware-calibrated execution graphs gives Agents an effective initial structure and room for continued implementation refinement.

2 BACKGROUND

Mathematical representations. Mathematical representations describe computations and their dependencies, providing a basis for transformations that preserve program semantics. Halide separates algorithm definitions from execution schedules, enabling transformations of locality, parallelism, and recomputation (Ragan-Kelley et al., 2013). TensorIR extends loop representations with tensor computation blocks to support scheduling and mapping to hardware tensor instructions (Feng et al., 2023). Triton expresses computation through operations on tiles and compiles these operations into GPU code (Tillet et al., 2019). Mirage represents computation across kernel, thread-block, and thread levels to jointly search algebraic and scheduling transformations (Wu et al., 2025). These representations determine which transformations an optimizer can express and how it checks dependencies and semantic preservation.

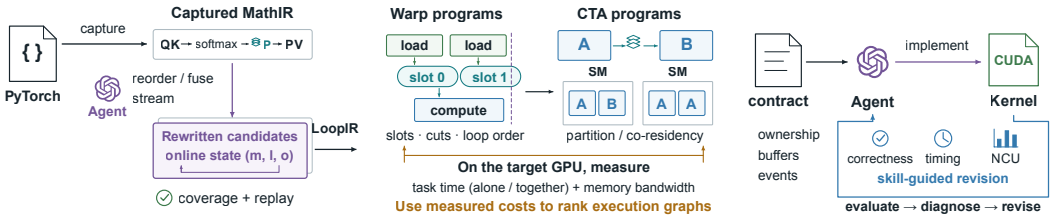
Warp and CTA execution. CUDA groups threads into 32-thread Warps and thread blocks, also called cooperative thread arrays (CTAs). A CTA executes on one streaming multiprocessor (SM), where its Warps cooperate through shared memory and synchronization; SMs can host multiple CTAs subject to resource limits (NVIDIA, 2026b). Hopper's asynchronous tensor transfers support specialized producer and compute Warps (NVIDIA, 2026c). Blackwell retains this programming hierarchy with architecture-specific resource capacities (NVIDIA, 2026a). Pipeline depth and buffering can improve overlap while increasing resource use and reducing residency. Conversely, CTA partitioning changes local work and memory concurrency. ThunderKittens (Spector et al., 2025)

exposes tile operations and asynchronous producer–consumer templates across this hierarchy. Local overlap, SM residency, and device-wide memory throughput are consequently coupled: a faster local pipeline need not yield a faster complete kernel.

GPU kernel measurement and profiling. Developers use CUDA events to measure kernel execution time and hardware profilers to inspect resource utilization. Nsight Compute reports compute and memory throughput, occupancy, and Warp stalls, helping developers locate performance bottlenecks (NVIDIA, 2026d). KPerfIR integrates profiling into compiler passes to expose fine-grained execution behavior within kernels (Guan et al., 2025), while Neutrino uses programmable assembly-level probes to collect instruction-level timing and runtime values across NVIDIA and AMD GPUs (Huang & Wu, 2025). More recently, KEET uses LLM Agents to interpret Nsight Compute profiles and produce explanations and optimization suggestions grounded in measured execution behavior (Davis et al., 2026).

3 System Design

Given an executable PyTorch reference, workload, and target GPU, KernelMaster uses semantic checks and reusable measurements to guide CUDA implementation. Figure 1 shows three stages: MathIR search and LoopIR projection (a), hardware-calibrated execution graph synthesis (b), and contract-guided Agentic CUDA lowering (c).

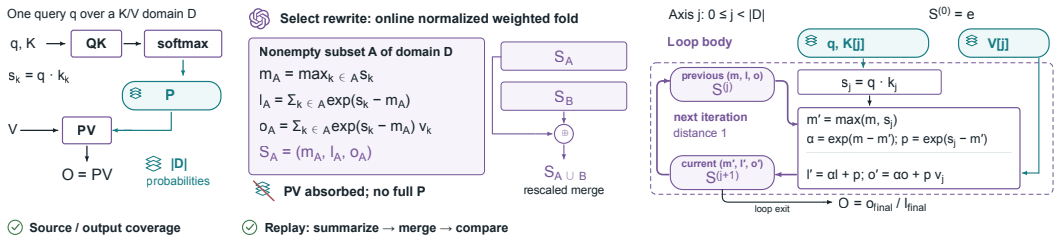


(a) Mathematical dataflow search (b) Execution graph synthesis (c) Agentic CUDA lowering

Figure 1: Overview of KernelMaster.

3.1 Agent-Operable Mathematical Dataflow Synthesis

Figure 2 illustrates attention for query q over a nonempty finite domain $\mathcal{D}$ of keys k_j and values v_j, collected in K and V. The reference (a) materializes softmax probabilities P from scores $s_j = q \cdot k_j$; the MathIR rewrite (b) absorbs PV into an online fold, eliminating P. Its state (m, l, o) holds the running score maximum, rescaled normalizer, and weighted value sum; subscripts denote domain subsets. In (c), primes mark updated state, α rescales the previous state, and p is the current unnormalized weight. The backedge carries state to the next iteration; final output $O = o_{\text{final}}/l_{\text{final}}$ is computed outside the loop.



(a) Reference attention (b) Agent-guided MathIR rewrite (c) Deterministic LoopIR projection

Figure 2: Attention example: MathIR eliminates the intermediate probability tensor; LoopIR exposes the state recurrence.

3.1.1 SOURCE-ACCOUNTED MATHIR SEARCH

Capturing the optimization problem. The frontend captures a PyTorch reference and deterministic workload, recording the tensor/scalar ABI, ordered arguments, use–def edges, shapes, dtypes, index transformations, conditionals, and available effect/alias contracts. Each source node is accounted for exactly once. Layout, cast, and index-map chains folded into edges remain recoverable through a coverage ledger; unknown operations and unresolved effects remain explicit boundaries. This compact representation preserves traceability to source operations and observable outputs.

An Agent-operable search space. Admitted pure regions comprise pointwise maps, indexed substitutions, contractions, selections, normalizations, and reductions over named domains. Deterministic analysis exposes guarded opportunities for deferred indexed reads, shared-value placement, operator-preserving fusion, fold partitioning, and normalized weighted folds. Local opportunity descriptions and overlap constraints avoid expanding their Cartesian product while keeping unresolved semantic gaps explicit.

The MathIR Agent reviews every exposed opportunity and emits a small number of complete candidates. A candidate contains equations for every observable output, typed intermediate states, instantiated algebraic laws and their domain conditions, a finite-precision policy, an executable PyTorch replay, and an expected-performance analysis. In particular, a partitionable reduction is represented by an identity, a local fold, a state merge, and a finalizer:

$$S_{\mathcal{P}} = \mathrm{fold}_{i\in\mathcal{P}}\, g(x,i), \qquad S_{\mathcal{P}\cup\mathcal{Q}} = S_{\mathcal{P}} \otimes S_{\mathcal{Q}}, \qquad y = \mathrm{finalize}(S_{\mathcal{D}}). \tag{1}$$

Here x denotes inputs, i a domain element, g its state contribution, $S_{\mathcal{P}}$ the summary of subset $\mathcal{P} \subseteq \mathcal{D}$, $\otimes$ the declared state merge, and $\mathrm{finalize}$ the map to output y. Disjoint subsets $\mathcal{P}, \mathcal{Q}$ may merge under the declared algebraic and ordering conditions. Folding starts from identity e; empty summaries are handled explicitly. Weighted folds can incorporate downstream linear computation, removing an intermediate tensor or another domain traversal.

The Agent ranks candidates by arithmetic, state size, global materialization, parallelism, and critical path. KernelMaster freezes the top candidate, checks source/output coverage and state contracts, replays declared workload cases, and recaptures it under the same admission rules. The numerical policy governs reassociation and reduced precision.

3.1.2 VERIFIED LOOPIR PROJECTION

MathIR deliberately contains no loop nest, tile, CTA task, or execution order. KernelMaster next identifies the regions whose selected mathematical semantics admit a finite iterative realization. A region is *loop-normalizable* when (i) every produced tensor has a finite iteration domain, (ii) each access has a defined coordinate map, (iii) every reduction or recurrence provides an initial state, update, merge when partitionable, and finalizer, (iv) effects and conditional boundaries are ordered or isolated, and (v) every observable output can be reconstructed from the resulting values and final states. An admitted opaque primitive may remain atomic under an explicit semantic contract; a region lacking these conditions does not enter hierarchical execution search.

Deterministic projection maps domains to LoopIR axes, algebraic states to versioned recurrences, and index relations to operand mappings. For an admissible enumeration $d_0, \ldots, d_{n-1}$ of $\mathcal{D}$, with $n = |\mathcal{D}|$ elements, it constructs

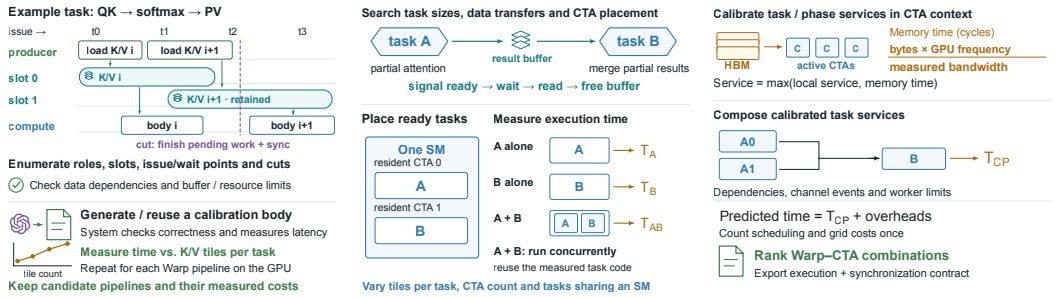
$$S^{(0)} = e, \qquad S^{(i+1)} = S^{(i)} \otimes g(x, d_i), \qquad y = \mathrm{finalize}\Big(S^{(n)}\Big). \tag{2}$$

Here $S^{(i)}$ summarizes the first i elements, $0 \le i < n$ indexes updates, and e is the merge identity. LoopIR records whether an operand refers to an external, previous, current, final, all-iterations, or complete version. These versions mechanically induce dependency edges and iteration distances. Operations are placed in the program prologue, loop body, loop exit, or epilogue without selecting a CUDA schedule.

Projection invariant. Content hashes bind source, MathIR, workload, and LoopIR; each operation records source coverage. Coordinate mappings preserve outputs, and prefix induction identifies $S^{(i)}$ with the MathIR fold over the first i elements. Declared merge laws extend the invariant to partitions; finalization reconstructs outputs under the numerical policy. Tiles, Warp roles, buffers, CTA placement, and instructions remain execution choices.

3.2 Feedback-Coupled Hierarchical Execution Synthesis

KernelMaster factors execution search into intra-CTA Warp programs and inter-task CTA programs, retaining their shared resource and memory context. Figure 3 shows the full workflow: local pipeline search and calibration (a), CTA synthesis and concurrent measurement (b), and hardware-calibrated joint ranking (c).



(a) Warp-role pipeline search (b) Inter-task CTA synthesis (c) Hardware-calibrated ranking

Figure 3: Workflow of feedback-coupled hierarchical execution synthesis.

Let $c \in \mathcal{C}$ be a legal CTA topology and $w_r \in \mathcal{W}_r(c)$ a compatible intra-CTA Warp program for each logical task r defined by c. On target GPU H, the desired solution is

$$(c^*, \{w_r^*\}) = \arg \min_{c \in \mathcal{C},\, w_r \in \mathcal{W}_r(c)} T_H(c, \{w_r\}), \tag{3}$$

where T_H is actual latency and stars denote minimizing choices. KernelMaster ranks legal combinations with its calibrated estimate $\widehat{T}_H$ rather than generating every complete kernel. It composes interface- and resource-compatible programs and refines local candidates in their CTA context. We call the intra-CTA representation *WrapIR*.

3.2.1 Warp-Role Pipeline Synthesis

A legal semantic partition of LoopIR defines typed logical CTA tasks, each with operations, local axes/states, input/output ports, and a domain share. KernelMaster checks that tasks and their dependencies reconstruct LoopIR's operations, values, transitions, and outputs. Within each task, a *Warp role* assigns one or more Warps to production or computation; Wrap search synthesizes their pipeline.

For each logical CTA task, Wrap search enumerates operation grouping, producer and consumer Warp roles, asynchronous issue and wait points, on-chip placement, iteration distance, and logical pipeline slots. A slot names a reusable buffer instance and its lifetime, independently of the Warp that accesses it. Dependencies, implementation capabilities, buffer lifetimes, and hardware resource limits remove illegal candidates. The search also introduces a *Wrap cut*, an ordering boundary at which preceding in-flight work is drained and the Warp organization may change. A cut synchronizes one logical CTA; it does not impose a grid-wide barrier. Values needed after the cut remain live under an explicit handoff contract that records their producer, consumers, first-readable point, final consumer, and release point. Consequently, a K/V tile or recurrence state may survive a phase change while temporary storage is reclaimed after its last use. Figure 4 shows two consecutive K/V tiles in one task. The issue order t_0–t_4 aligns producer and compute Warps with shared-memory slots 0 and 1. Each tile is ready before QK/PV reads and released after its final PV read. The cut drains and synchronizes pending work, retaining prefetched tile $i{+}1$ in slot 1 and carrying (m, l, o) into the next QK–softmax–PV body.

A legal task-local Wrap program w exports the interface

$$\Sigma_r(w) = \big(T_r(w, \cdot), \rho_r(w), \mu_r(w), E_r(w), P_r(w)\big), \tag{4}$$

where $T_r(w, \cdot)$ is the service-response function, ρ_r the resource envelope, μ_r the memory-traffic contract, E_r the event interface, and P_r the typed, versioned ports of program w for task r. At local iteration count L, $T_r(w, L) = T_{\text{Wrap}}(w, L)$ below. This interface enables reuse of verified local programs and calibration across CTA topologies.

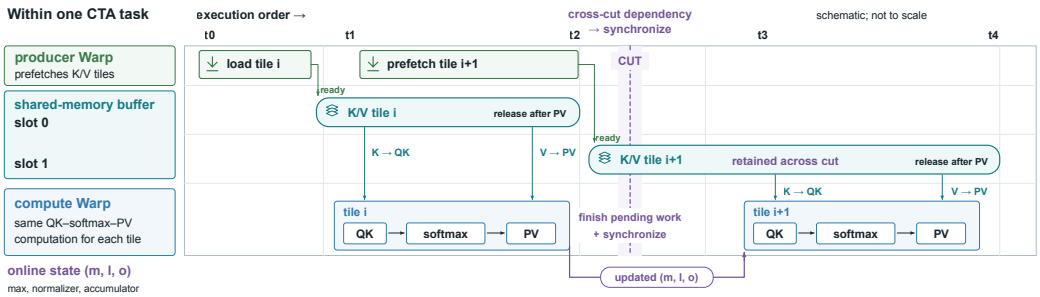


Figure 4: WrapIR example: K/V reads, slot lifetimes, and state handoff across a cut.

3.2.2 Inter-Task CTA Program Synthesis

CTA synthesis maps logical tasks to a bounded pool of physical CTA workers; each task binds a complete Wrap program. Cross-task edges specify producer and consumer ports, iteration mapping, value version, placement, and transport. Each bounded channel enforces

$$\text{produce} \rightarrow \text{publish} \rightarrow \text{acquire} \rightarrow \text{consume} \rightarrow \text{release}. \tag{5}$$

Publication makes a version visible, acquisition transfers ownership or read permission, and final-consumer release returns the channel credit for reuse. The scheduler may assign a worker only to a ready logical task; waiting for data does not authorize an unready task to retain a worker. KernelMaster checks version consistency, producer–consumer iteration mappings, shared-launch resource envelopes, channel capacities, final-consumer release, and progress under finite workers and finite credits.

CTA synthesis searches work partitioning, partial-state production and merge, physical worker count, task co-residency, channel realization, and ready-task scheduling. It can therefore express persistent execution and producer–consumer pipelines without assuming a one-to-one mapping between a logical task and a physical CTA.

Compositional invariant. Each Wrap program refines a typed task, and each CTA edge matches port types, versions, and iteration maps. Release follows the last declared consumer; progress checks reject cyclic waits over finite workers and channel credits. Under these conditions, composition preserves LoopIR dependencies and state semantics. Hardware ranking selects among these legal compositions.

3.2.3 Target-Calibrated Cross-Level Ranking

KernelMaster reevaluates Wrap candidates for CTA-specific work, residency, channels, and traffic, reusing calibration indexed by operations, resources, slots, cuts, and work size. An Agent supplies missing CUDA calibration bodies; the system checks compilation, correctness, and resources, then measures them. For candidate w and $L \geq 1$ local iterations, the calibrated response is

$$T_{\text{Wrap}}(w, L) = T_{\text{first}}(w) + (L-1)T_{\text{step}}(w) + T_{\text{final}}(w). \tag{6}$$

T_{first} spans startup through the first completed iteration, T_{step} is the steady-state increment per iteration, and T_{final} is one-time finalization. Direct measurements handle short or nonlinear regimes. Matched separate/concurrent probes of tasks A and B yield T_A, T_B, T_{AB} (Figure 3b); the joint probe captures co-admission and interference under the chosen launch/residency policy.

For D bytes transferred by a task or phase, the HBM response is

$$T_{\text{HBM}} = \frac{f_{\text{GPU}} D}{B_{\text{eff}}(N_{\text{CTA}}, N_{\text{Warp}}, \text{access size}, \text{co-runners})}. \tag{7}$$

B_{eff} is measured bandwidth (bytes/s), N_{CTA} the CTA count in the modeled residency scope, and N_{Warp} participating Warps per CTA. Access size specifies transfer size; co-runners specify concurrent traffic. Reference frequency f_{GPU} (cycles/s) puts all costs in cycles. Cache-resident and sparse traffic use corresponding measurements.

KernelMaster takes the maximum of local service and HBM time for each task or overlapping phase at matched work/residency coordinates. It composes serial phases, merges, and finalization in order, then builds the execution graph with channel events, calibrated concurrency, and worker-serialization constraints:

$$\widehat{T}_H = T_{\mathrm{CP}} + T_{\mathrm{sched}} + T_{\mathrm{grid}}. \tag{8}$$

Here T_{CP} is the graph's critical-path time, and T_{sched} and T_{grid} are residual scheduling and grid costs. Accounting scopes exclude costs already charged in events. Complete-graph ranking thus feeds CTA-conditioned memory and residency effects back into Wrap selection.

3.3 Contract-Guided Agentic CUDA Lowering

The highest-ranked execution graphs become hash-bound contracts for the Contract-Guided CUDA Realization Agent. Mathematical states, task domains, Wrap phases/cuts, task interactions, channel identities/capacities, publication protocol, work ownership, synchronization, and compile target remain fixed. The Agent chooses instructions, register representations, shared-memory layouts, vectorization, and instruction schedules. Its first implementation therefore starts from a semantically checked, hardware-ranked structure; later rounds refine its machine realization within that contract.

Figure 5 separates the fixed execution contract from the CUDA lowering and validation components, which form an iterative feedback loop. The contract guides both the initial CUDA implementation and later revisions. Each round compiles the body and inspects PTXAS/SASS evidence, checks correctness against the reference, and measures latency with GPU events. Conditional NCU profiling supplies resource and stall evidence when the measured behavior needs diagnosis. An independent, read-only review turns these external observations into mandatory repairs or one bounded performance hypothesis, with an acceptance criterion and a preserve list for the next round. KernelMaster returns the fastest externally validated body over all rounds, together with its realized-lowering manifest and measurement evidence.

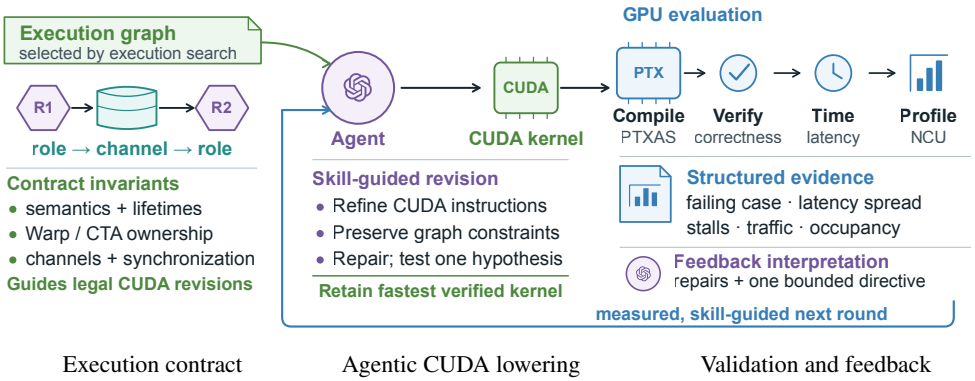


Figure 5: Contract-guided agentic CUDA lowering.

4 Experimental Results

We evaluate five newly specified attention kernels against Agent baselines, then five established kernels on each of two GPU architectures, covering ten hardware–task settings. All reported results pass reference correctness checks. External CUDA-event timings exclude compilation and setup. Speedup is the fastest plotted baseline's latency divided by implementation latency; values above $1\times$ indicate an improvement over that baseline.

4.1 Newly Specified Kernels

We construct five new kernel workloads from recently proposed model mechanisms to evaluate optimization on unfamiliar tasks. This design aims to reduce the influence of optimized implementations potentially encountered during model training. To test whether a system can discover

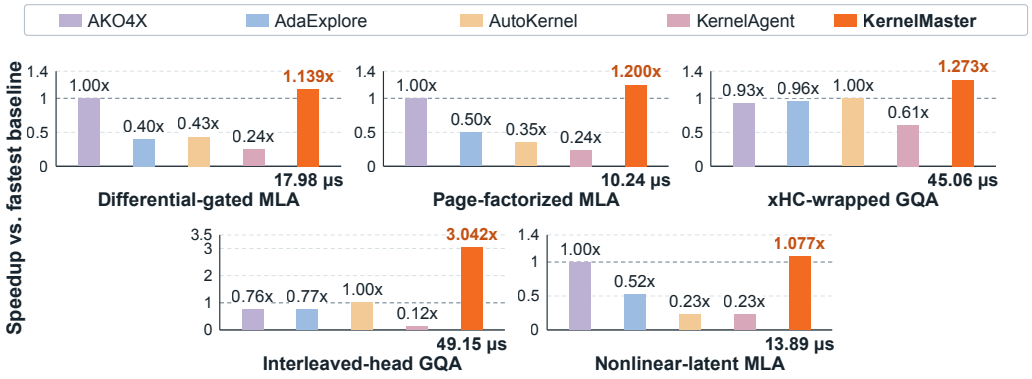


Figure 6: Five new kernel tasks; speedup over the fastest plotted Agent baseline.

efficient implementations for entirely new tasks, we design the following workloads with distinct dataflow and scheduling challenges: differential-gated MLA uses the excitatory/inhibitory branches of M-DGSA (Lygizou et al., 2025); page-factorized MLA combines dense recent pages and low-rank historical pages following PuzzleKV (Wang et al., 2026b); xHC-wrapped GQA mixes residual streams around attention (Zhang et al., 2026); interleaved-head GQA introduces cross-head pseudo Q/K/V (Duvvuri et al., 2026); and nonlinear-latent MLA combines grouped scores, latent-value decoding, and query-dependent gating following GTA (Sun et al., 2025).

All five run on the RTX 5090 D against AKO4X (Xie et al., 2026), AdaExplore (Du et al., 2026), AutoKernel (Jaber & Jaber, 2026), and KernelAgent (PyTorch, 2026). KernelMaster and the iterative baselines receive five optimization steps; KernelAgent contributes its first correct result. Figure 6 compares their best validated implementations. KernelMaster leads on every task by 1.077–$3.042\times$ over the fastest Agent baseline, and 4.911–$9.181\times$ over PyTorch. Interleaved-head GQA shows the largest gap: 49.152 versus $149.504\,\mu$s. Here, cross-head mixing and multiple attention states make the global organization of work consequential.

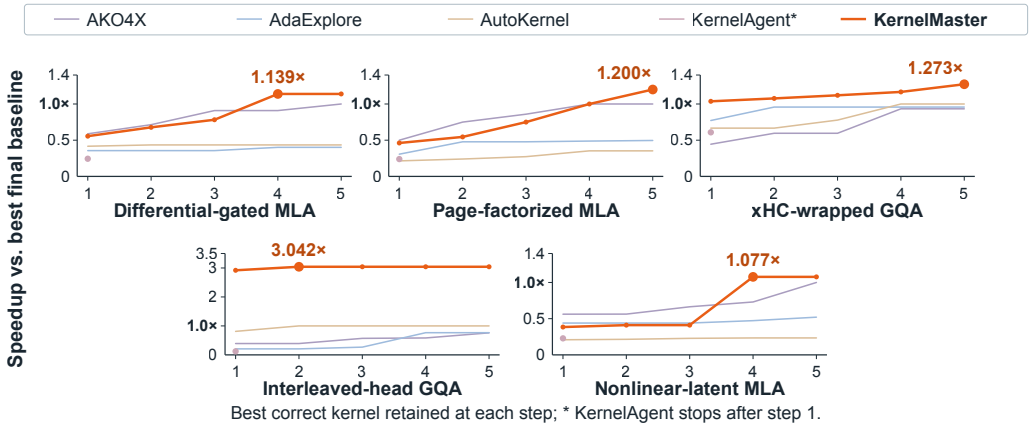


Figure 7: Best validated performance after each optimization step on new kernel tasks.

Figure 7 tracks the best validated implementation after each step, normalized to the same final baseline. KernelMaster's first implementation exceeds every baseline's final result on xHC-wrapped and interleaved-head GQA, and ranks among the two fastest first-step results on four tasks. This supports the searched contract's value as an initial organization. Subsequent revisions pass the strongest final baseline by step four on differential-gated and nonlinear-latent MLA, and by step five on page-factorized MLA. The curves support both a strong starting point and continued improvement under measured feedback. KernelAgent appears once because it stops at its first correct result.

Performance analysis. We use NCU to examine why KernelMaster achieves a $1.273\times$ speedup over AutoKernel, the fastest Agent baseline on xHC-wrapped GQA. KernelMaster partitions the K/V domain across 1,024 CTAs; AutoKernel processes the full domain in each of 128 CTAs. NCU reports achieved occupancy of 49.23% versus 16.66% and an issue-active rate of 36.33% versus 11.75%, showing substantially greater utilization of execution resources. DRAM read volumes remain similar (4.44 versus 4.35 MB), while read throughput increases from 72.5 to 97.7 GB/s. Together, these measurements support greater parallelism and more effective latency hiding as explanations for the speedup. This case illustrates how the searched execution structure helps the Agent expose parallel work and improve hardware utilization.

4.2 Established Kernels Across GPU Architectures

Paged GQA (Ainslie et al., 2023; Ye et al., 2025), 7168-wide RMSNorm (Zhang & Sennrich, 2019), DeepSeek-V4 sparse MLA (DeepSeek-AI, 2026), fused residual addition with 4096-wide RMSNorm (Zhang & Sennrich, 2019; FlashInfer Contributors, 2026), and 16-head dense MLA decode (DeepSeek-AI, 2024) with a paged KV cache (Ye et al., 2025) run on both an RTX 5090 D and an H20 96 GB. Figure 8 compares applicable TensorRT-LLM, FlashInfer, FlashMLA, Triton, cuDNN, PyTorch, and `torch.compile`/Inductor implementations with reference correctness checks and a common memory-state timing protocol.

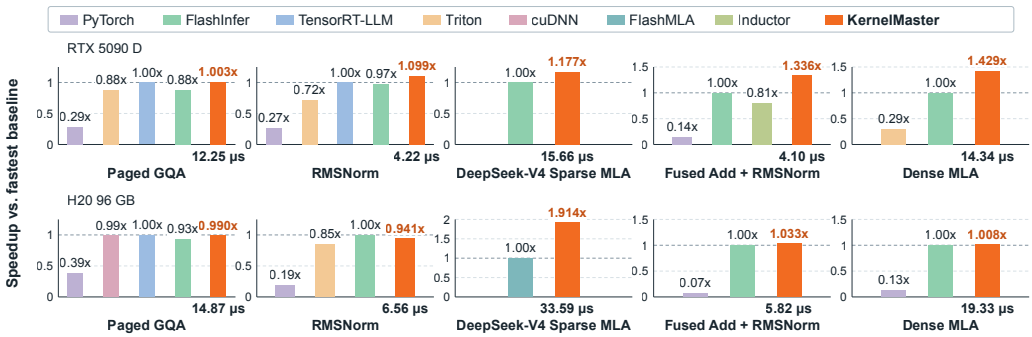


Figure 8: Established kernels on two GPUs; speedup over the fastest plotted baseline.

KernelMaster achieves lower measured latency in eight of ten settings. On the RTX 5090 D, paged GQA, RMSNorm, and sparse MLA achieve $1.003\times$, $1.099\times$, and $1.177\times$. Fused Add + RMSNorm reaches $4.096\ \mu$s versus FlashInfer's 5.472 and Inductor's $6.752\ \mu$s: $1.336\times$ over the fastest baseline and $9.262\times$ over PyTorch. Dense MLA reaches $14.336\ \mu$s versus FlashInfer's $20.480\ \mu$s ($1.429\times$).

On H20, paged GQA is within 1.02% of TensorRT-LLM XQA and RMSNorm within 6.25% of FlashInfer; sparse MLA reaches $1.914\times$ over FlashMLA. KernelMaster directly specializes the 16-head sparse MLA workload, while FlashMLA executes a padded 64-head representation. Fused Add + RMSNorm and dense MLA take 5.824 and $19.328\ \mu$s, achieving $1.033\times$ and $1.008\times$ over FlashInfer, and $14.24\times$ and $7.69\times$ over PyTorch, respectively. Across all five kernels on both GPUs, KernelMaster approaches or surpasses expert implementations, demonstrating hardware adaptation with competitive performance.

5 Conclusion

KernelMaster provides structural guidance for Agent-driven GPU kernel optimization by combining mathematical dataflow discovery with hardware-calibrated execution synthesis. Explicit semantic states and composable Warp–CTA programs turn coupled optimization decisions into checkable candidates; execution contracts connect the selected structure to Agent implementation and measured refinement. New-kernel experiments show strong initial implementations and continued improvement, while paired cross-GPU evaluations demonstrate competitive performance against expert implementations. Together, mathematical dataflow discovery and feedback-coupled execution synthesis connect reference semantics to hardware-specific implementation, providing an effective starting structure and sustained optimization guidance for Agent-driven kernel development.

Ethics Statement

This work studies automated GPU kernel optimization using programmatically generated tensor workloads; it does not involve human participants or personal data. We credit the sources of workload mechanisms and baseline systems. To address reliability risks from generated code, KernelMaster checks execution contracts and validates generated kernels against executable references before reporting performance. The system is intended to reduce kernel-development effort and improve computational efficiency. Its use in downstream applications should retain correctness checks appropriate to those applications.

Reproducibility Statement

Across KernelMaster and all Agent baselines, we use Codex CLI with GPT-5.6-sol and a reasoning effort of `high`. Section 3 describes the representations, semantic checks, execution search, calibrated cost model, and Agent refinement procedure. Section 4 specifies workload sources, GPU platforms, baseline systems, optimization budgets, correctness requirements, and timing and speedup normalization protocols. Figures 6–8 report final performance and optimization trajectories; Section 4.1 provides NCU measurements for the xHC comparison. We retain executable references, generated kernels, per-round validation and timing records, and raw profiler outputs to trace reported results to the evaluated implementations. Our implementation and evaluation code is available through the anonymous repository linked in the abstract.

AI Use Statement

[The authors conceived the research ideas, designed and conducted the experiments. AI agents are studied as part of the system and evaluation. LLMs assisted with language polishing.]

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, 2023. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298/.

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

Joshua H. Davis, Klaudiusz Rydzy, Srinivasan Ramesh, Aadit Nilay, Daniel Nichols, Swapna Raj, Nikhil Jain, and Abhinav Bhatele. KEET: Explaining performance of GPU kernels using LLM agents, 2026. URL https://arxiv.org/abs/2605.04467.

DeepSeek-AI. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre Wang He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, Yiming Yang, and Sean Welleck. Adaexplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation, 2026. URL https://arxiv.org/abs/2604.16625.

Sai Surya Duvvuri, Chanakya Ekbote, Rachit Bansal, Rishabh Tiwari, Devvrit Khatri, David Brandfonbrener, Paul Liang, Inderjit Dhillon, and Manzil Zaheer. Interleaved head attention, 2026. URL https://arxiv.org/abs/2602.21371.

Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 804–817, 2023. doi: 10.1145/3575693.3576933. URL https://arxiv.org/abs/2207.04296.

FlashInfer Contributors. FlashInfer: Fused add RMSNorm. API documentation, 2026. URL https://docs.flashinfer.ai/generated/flashinfer.norm.fused_add_rmsnorm.html.

Yue Guan, Yuanwei Fang, Keren Zhou, Corbin Robeck, Manman Ren, Zhongkai Yu, Yufei Ding, and Adnan Aziz. KPerfIR: Towards a open and compiler-centric ecosystem for GPU kernel performance tooling on modern AI workloads. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 205–220. USENIX Association, 2025. URL https://www.usenix.org/conference/osdi25/presentation/guan.

Songlin Huang and Chenshu Wu. Neutrino: Fine-grained GPU kernel profiling via programmable probing. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 331–355. USENIX Association, 2025. URL https://www.usenix.org/conference/osdi25/presentation/huang-songlin.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search, 2026. URL https://arxiv.org/abs/2603.21331.

Elpiniki Maria Lygizou, Mónika Farsang, and Radu Grosu. Differential gated self-attention, 2025. URL https://arxiv.org/abs/2505.24054.

NVIDIA. NVIDIA Blackwell tuning guide. Documentation, 2026a. URL https://docs.nvidia.com/cuda/blackwell-tuning-guide/index.html.

NVIDIA. Cuda programming guide: Programming model. Documentation, 2026b. URL https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html.

NVIDIA. NVIDIA Hopper tuning guide. Documentation, 2026c. URL https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html.

NVIDIA. Nsight Compute: Profiling guide. Documentation, 2026d. URL https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

PyTorch. Kernelagent: Autonomous gpu kernel generation and optimization via deep agents. Software, 2026. URL https://github.com/meta-pytorch/KernelAgent.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 519–530, 2013. doi: 10.1145/2491956.2462176. URL https://people.csail.mit.edu/jrk/halide-pldi13.pdf.

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Daniel Y. Fu, and Christopher Ré. ThunderKittens: Simple, fast, and adorable kernels. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=0fJfVOSUra.

Luoyang Sun, Cheng Deng, Jiwen Jiang, Xinjian Wu, Haifeng Zhang, Lei Chen, Lionel Ni, and Jun Wang. Gta: Grouped-head latent attention, 2025. URL https://arxiv.org/abs/2506.17286.

Qitong Sun, Jun Han, Tianlin Li, Zhe Tang, Sheng Chen, Fei Yang, Aishan Liu, Xianglong Liu, and Yang Liu. KernelSkill: A multi-agent framework for GPU kernel optimization, 2026. URL https://arxiv.org/abs/2603.10085.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019. doi: 10.1145/3315508.3329973. URL https://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf.

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

Zizhong Wang, Jieying Wang, Zhao Zhang, and Jiajia Li. Puzzlekv: Page-wise low-rank decomposition for kv cache compression, 2026b. URL https://arxiv.org/abs/2608.23843.

Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 21–38. USENIX Association, 2025. URL https://www.usenix.org/conference/osdi25/presentation/wu-mengdi.

Shuxiao Xie, Shuyang Xie, Dezhi Ran, Wei Yang, and Tao Xie. Ako: Agentic kernel optimization. Technical report, 2026. URL https://tongminglaic.github.io/AKO/. AKO4X implementation: https://github.com/TongmingLAIC/AKO4X.

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

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://papers.nips.cc/paper_files/paper/2019/hash/1e8a19426224ca89e83cef47f1e7f53b-Abstract.html.

Xiangdong Zhang, Xiaohan Qin, Sunan Zou, Tuo Dai, Xiaoming Shi, Huaijin Wu, Yebin Yang, Zhuo Xia, Shaofeng Zhang, Lin Yao, Yuliang Liu, Yu Cheng, and Junchi Yan. xhc: Expanded hyper-connections, 2026. URL https://arxiv.org/abs/2607.14530.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. QiMeng-Kernel: Macro-thinking micro-coding paradigm for LLM-based high-performance GPU kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29168–29176, 2026. doi: 10.1609/aaai.v40i34.40155. URL https://ojs.aaai.org/index.php/AAAI/article/view/40155.