Hawkeye: Hardware-Aware GPU Kernel Optimization with Minimal Supervision

Anonymous Author(s)
Affiliation
Address
email

Abstract

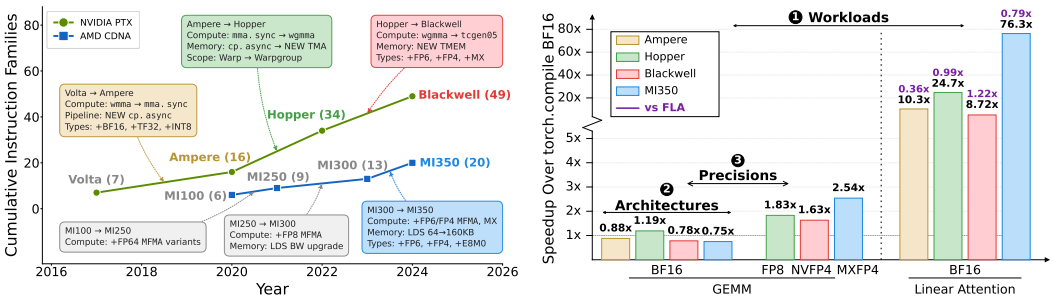
Achieving peak GPU kernel performance increasingly relies on architecture-specific optimizations targeting new hardware features. While AI coding agents show promise in generating performant kernels, they lack the necessary context to effectively implement and stack hardware-specific optimizations, especially on newer GPU architectures. We propose HAWKEYE (**H**ardware-**Aw**are **Ke**rnel Optimization), an open-source framework that grounds autonomous kernel generation in a minimal, extensible taxonomy with only one unit test per optimization strategy per target architecture. Supporting a new accelerator therefore requires only 10 expert-written unit tests per architecture (one per recurring optimization strategy) that generalize across downstream workloads, rather than hand writing a new kernel for each workload and precision. HAWKEYE effectively scales the test-time compute of coding agents with this minimal expert supervision to enable kernel generation that consistently leverages hardware-specific features, approaching and even surpassing expert-written PyTorch or Triton in BF16 and emerging low precision (FP8, NVFP4, MXFP4) across Ampere, Hopper, Blackwell, and MI350 GPUs. HAWKEYE demonstrates that minimally supervised coding agents can exploit architecture-specific hardware features and reduce the overhead of supporting emerging hardware accelerators.

1 Introduction

Modern machine learning systems depend on GPU kernels carefully co-designed with the underlying architecture [Dao et al., 2022]. As GPUs evolve, the number of architecture-specific instruction families grows substantially (Figure 1a) [Yadav et al., 2025]. The underlying programming models have also become increasingly hardware-specific, with new tensor core instructions, functional units, and warp specialization strategies introduced with each generation. Tensor core paths provide roughly 94% of Hopper FLOPs [NVIDIA, 2022] and over 99% of Blackwell FLOPs [NVIDIA, 2024a], making hardware co-design essential for peak kernel performance.

The cost and complexity of supporting new hardware can be seen in real open-source projects. Architecture-specific code in NVIDIA's CUTLASS library has grown $29\times$ across recent versions and is now the majority of the codebase (per-version breakdown in Appendix A.6). Manually porting projects like FlashAttention [Dao et al., 2022], FlashInfer [Ye et al., 2025] and ThunderKittens [Spector et al., 2024] across GPU generations or vendors takes months to years of expert effort (porting timelines from Git history in Appendix A.6). The alternative of using high-level DSLs like Triton [Tillet et al., 2019] takes months to support new hardware (Appendix A.6) and still struggles with new hardware features [jeromeku, 2025]. Hardware-specific kernels that achieve optimal performance on one hardware generation do not automatically achieve optimal performance on the next. Closing this gap, or providing *day-zero porting support*, means retargeting high-performance kernels to

Submitted to 40th Conference on Neural Information Processing Systems (NeurIPS 2026). Do not distribute.



(a) The number of unique assembly instruction families grows with each new GPU generation across NVIDIA and AMD product lines, as architects expose more hardware-specific optimizations to the programmer.

(b) HAWKEYE shows *hardware awareness* from unit tests enables agents to port kernels ❶ across workloads, ❷ architectures and vendors, and ❸ 16, 8, and 4-bit precision (vs. `torch.compile` and Triton FLA BF16).

Figure 1: Each GPU generation introduces new hardware-specific instruction families that widen the gap between peak hardware capability and what software actually delivers. HAWKEYE brings *hardware awareness* to coding agents through a minimal taxonomy of expert-authored unit tests, enabling autonomous kernel porting across workloads, GPU architectures and vendors, and precisions.

a newly released chip in hours rather than months while maintaining peak utilization on the new hardware.

Coding agents accelerate these manual porting timelines with test-time feedback loops [Ouyang et al., 2025, Lange et al., 2025], reinforcement learning from runtime rewards [Baronio et al., 2025, Dai et al., 2026], and retrieval over CUDA documentation or hand-authored skills [Dong et al., 2026, Hugging Face, 2026]. However, these approaches lack the context structure to enable architecture-specific kernel optimizations. In practice, kernel engineers add raw hardware context like programming model documentation [NVIDIA, 2024b, AMD, 2024] or kernels from older generations, which can pollute context without preventing common hardware-awareness failures (Section 4.4). For example, frontier coding agents can rarely implement Blackwell-specific optimizations such as `tcgen05` tensor-core instructions or TMEM accumulators. Instead, they often fall back to Hopper-era `wgmma` code paths, which cannot fully utilize Blackwell's new hardware and leave much of the chip idle (Appendix B).

We define this missing capability as *hardware awareness*, the ability to identify hardware-specific optimizations and correctly implement kernels that utilize them. Without it, agents fall into predictable failure modes by generating hardware-unaware kernels that reuse optimization patterns from the wrong GPU generation, misconfigure architecture-specific data layouts, and abandon specialized low-precision paths for naive code that leaves tensor cores idle (Appendix D.2). These failures compound on newer and non-NVIDIA architectures (Appendix B), exactly where day-zero porting support is most valuable. This motivates our central question: *How can we make coding agents hardware-aware with minimal expert intervention?*

HAWKEYE introduces a generalizable (Section 4.1), minimal (Section 4.4), and composable (Appendix E.3) taxonomy of unit tests that enables coding agents to scale test-time compute more effectively (Section 4.3) and generate hardware-aware kernels. Each unit test pairs an optimization strategy with a profiling metric, and the agent composes them into high-performance kernels for downstream workloads. In contrast with baselines of hardware-aware agent context, HAWKEYE's taxonomy has better context efficiency (Section 4.4) by organizing rows of recurring optimization strategies (MMA, async pipelines, shared-memory swizzling, and so on) and columns of target architectures, with one expert-authored kernel per cell. Adding a new accelerator therefore costs only 10 unit tests, regardless of the number of downstream workloads the agent later optimizes.

We evaluate HAWKEYE on porting PyTorch workloads to high-performance kernels across NVIDIA Ampere, Hopper, Blackwell, and AMD MI350, and across BF16, FP8, NVFP4, and MXFP4 precisions (Section 4). On established workloads, where `torch.compile` dispatches to expert-tuned vendor libraries like cuBLAS, cuDNN, and FlashAttention, HAWKEYE matches or exceeds it in both BF16 and low precision, including in formats PyTorch cannot natively run. On emerging attention variants where `torch.compile` cannot fuse non-standard scans and gates, HAWKEYE reaches an $18.9\times$ geomean speedup. Against expert-authored Triton kernels from the Flash Linear Attention library [Yang and Zhang, 2024], HAWKEYE approaches or exceeds FLA on Linear Attention across

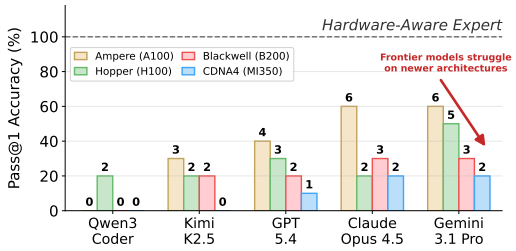
every architecture, including $1.22\times$ on Blackwell and $1.00\times$ on MI350. HAWKEYE also enables native HIP kernel generation from a single set of MI350 unit tests for cross-vendor porting.

In summary, our contributions are as follows.

1. **An extensible taxonomy of unit tests for hardware-aware kernel optimization.** The taxonomy is a minimal and generalizable expert input needed to create hardware-aware kernels for a target accelerator, requiring only 10 unit tests per architecture (one per recurring optimization strategy) that generalize to downstream workloads.

2. **Scaling test-time compute to generate hardware-aware kernels from minimal supervision.** HAWKEYE is an open-source framework that scales coding agents to compose the per-architecture taxonomy unit tests into performant GPU kernels with better context efficiency than baselines. We release the taxonomy, kernels, and experimental harness for all four GPU architectures.

3. **Enabling kernel porting across architectures, vendors, and precisions.** Across NVIDIA Ampere, Hopper, Blackwell, and AMD MI350 in BF16, FP8, NVFP4, and MXFP4, HAWKEYE reaches $1.13\times$ BF16 and $1.28\times$ low-precision geomean speedup against `torch.compile` on established workloads, beats it by $18.9\times$ geomean on emerging attention variants, and approaches or exceeds expert-written Triton FLA kernels.



(a) Pass@1 of frontier LLMs on the taxonomy unit tests. Accuracy drops on newer GPU architectures, indicating a lack of architecture-specific domain knowledge.

Context structure	Description	Turn-0	Workspace
Triton FLA	DSL kernel	3.4k	13.6k
Raw docs	ISA reference	2.1k	1.22M
Augmented docs	Curated ISA index	5k	1.22M
Hw-aware kernel	Native example	26k	104k
Vendor FMHA	Production kernel	2.3k	287k
PTX abstractions	PTX functions	0.9k	17k
HAWKEYE	**Unit tests + guides**	**11k**	**43k**

(b) Hardware-aware context structures against HAWKEYE. Turn-0 tokens are in the agent prompt. Workspace tokens are on disk and accessed through read tools.

Figure 2: Frontier models struggle to implement architecture-specific kernel optimizations, especially on newer GPU architectures. HAWKEYE addresses this shortcoming, and we evaluate against other structures of coding agent context to write performant and hardware-aware kernels.

2 HAWKEYE

HAWKEYE is a framework that bootstraps coding agents with a taxonomy of architecture-specific unit tests and enables agents to generate hardware-aware kernels for a diverse set of workloads. The methodology walks through the steps of a kernel engineer automating day-zero kernel optimization Lin [2025], Liao et al. [2025], namely the alternative frameworks that lead to coding agents (Section 2.1) and what to put in an agent's context window (Section 2.2). The progression of context structures motivates the HAWKEYE unit test taxonomy (Section 2.3) and kernel generation (Section 2.4).

2.1 Frameworks for Kernel Optimization with Context-Engineering

Frontier coding agents lack hardware awareness on day-zero GPU architectures, and the gap widens with each new generation (Figure 2a, Appendix B.1). Recent work has explored several context-engineering frameworks for kernel optimization (evaluated in Table 2). Iterative refinement [Ouyang et al., 2025] uses hardware specifications and runtime feedback over candidate kernels. GEPA-style prompt evolution [Agrawal et al., 2026] self-maintains a markdown reflection across attempts. Agent skills [Hugging Face, 2026] curate static programming model guides for specific optimizations. None inject architecture-specific knowledge that the agent can reliably retrieve and compose, since iterative feedback adds no external knowledge between trajectories, self-rewriting reflections cannot contribute new knowledge, and skills lack extensibility and detailed implementation guidance (Appendix D.4). The open design question is what hardware-aware content the agent context should hold.

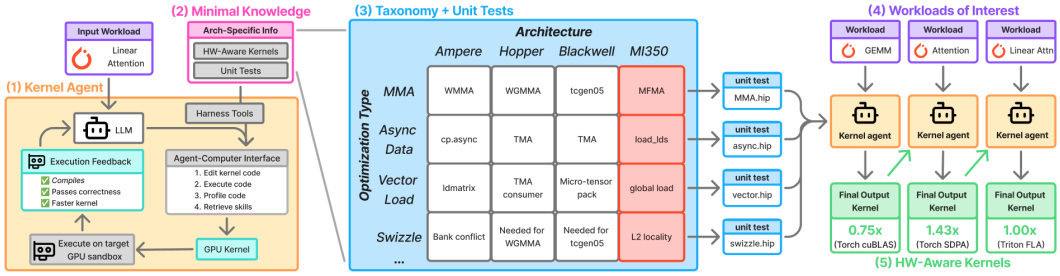


Figure 3: HAWKEYE framework. A kernel agent writes an optimized GPU kernel for a PyTorch workload, iteratively editing, executing, and profiling in a GPU sandbox. The agent is bootstrapped with the minimal knowledge needed for hardware awareness, consisting of 10 unit tests grounded in the taxonomy and previously generated kernels on the same architecture. Generalizing to different workloads of interest produces hardware-aware kernels evaluated against expert baselines.

2.2 A Progression of Hardware-Aware Context Structures

On day-zero of a new architecture, kernel engineers choose hardware-aware content to put in the agent's context window. We walk through a progression of prior approaches to context structures, ordered from portable algorithmic representations to curated architecture-specific instruction wrappers. Each exposes a structural deficit that the HAWKEYE taxonomy addresses (Section 4.4).

Algorithm without hardware syntax. High-level GPU DSLs like Triton are designed for portability by expressing kernels through abstractions that the compiler lowers for each target. Using DSL kernels as context, for example the FLA library [Yang and Zhang, 2024] at 3.4k tokens (Table 2b, token counts via [tiktoken, 2022]), gives an agent the algorithm but abstracts away the native CUDA or HIP architecture-specific instructions (Appendix D.3). *The agent has the algorithm but no architecture-specific syntax to implement it, and falls back to older generation tensor-core paths.*

Hardware documentation. An alternative provides the chip's full programming model documentation [NVIDIA, 2024b, AMD, 2024], with a 2k-token index and complete reference (1.22M tokens on NVIDIA, 256k on AMD) on disk. Every primitive is present, but the corpus pollutes the context window and string-match retrieval anchors in single chapters (Appendix D.4). Curating the documentation into a 5–7k-token index of structured excerpts removes the navigation cost, but the depth required still floods the context with instruction semantics rather than usable syntax (Appendix D.5). *The agent finds the right primitives but cannot compose them, because there is no feedback on optimization effectiveness and the cross-primitive recipe is split across separate sections.*

Few-shot hardware-aware kernels. A more involved approach replaces verbose documentation with hand-written kernel examples. A 26–35k-token GEMM kernel in context demonstrates MMA and asynchronous data movement primitives, but entangles algorithmic shape with descriptor strides and scheduler choices. Porting to new algorithms requires changing these optimizations in coordination (Appendix D.6). Production kernels like CUTLASS [NVIDIA, 2025] and Composable Kernel [Liu et al.] fused multi-head attention stack every architecture-specific optimization at once across 287–854k tokens, but bury the primitive optimization recipes under many layers of abstractions (Appendix D.7). *The agent has a working kernel composition but cannot decompose it because it has no granular abstraction boundary to get usable insights into individual optimizations.*

Curated abstractions for important optimizations. An engineer can extract architecture-specific, PTX-level abstractions from these production kernels and put these minimal API wrappers in context. At under 1k turn-zero prompt tokens the agent receives correct syntax for every primitive, but composing primitives into a coherent kernel requires reasoning about their interactions, which the wrappers alone do not encode (Appendix D.8). *The agent implements each optimization correctly but cannot stack them, since the wrappers encode syntax in isolation without a multi-instruction pattern.*

2.3 The Hawkeye Taxonomy of Unit Tests

We organize hardware-aware optimizations into a taxonomy, with recurring optimization strategies as rows and target GPU architectures (NVIDIA Ampere, Hopper, Blackwell, and AMD MI350) as columns (Table 1). Each cell implements the corresponding strategy using the architecture-specific instructions, data movement primitives, and execution patterns of that chip (Appendix A).

Table 1: The HAWKEYE taxonomy is a minimal and generalizable context structure for autonomous kernel optimization. Each row is a general optimization strategy and each unit test cell shows its architecture-specific details. Three representative rows are shown here, and the full taxonomy of 10 rows and profiling breakdown can be found in Appendix C.1.

	Ampere (A100)	Hopper (H100)	Blackwell (B200)	MI350
MMA Unit	`mma.sync` + loading `ldmatrix/.trans`	`wgmma` + TMA 128B-swizzle loads	`tcgen05` + TMA bulk loads + TMEM accum.	MFMA `32x32x16`, AGPR accumulators
Shared Memory Layout	XOR swizzle (`s32/64/256`) for conflict-free	128B column-group XOR swizzle matching `wgmma` descriptors	Parameterized 64B `offset_swizzle` for `tcgen05` descriptors	64-bank LDS XOR swizzle for conflict-free
Async Pipeline	`cp.async` pipeline, `commit_group`	TMA `bulk.tensor` + mbarrier pipeline	TMA 2D + mbarrier pipeline, phase-parity	Triple-buffered LDS pipeline

Deriving the rows. We derive the rows by decomposing hand-written kernels from FlashAttention-2, 3, and 4 [Dao, 2024, Shah et al., 2024, Zadouri et al., 2026], ThunderKittens [Spector et al., 2024], HipKittens [Hu et al., 2025], and CUTLASS [NVIDIA, 2025] into optimization regions (Appendices A.4 and A.5). A region becomes a row when it (1) recurs across workloads, (2) requires different implementations across GPU architectures, and (3) has a profiling counter that confirms the intended hardware behavior. Regions using the same instruction family and validated by the same profiling counter are grouped into a single row. Applying these criteria yields ten rows spanning compute, data movement, and scheduling. These rows are minimal under the grouping criteria and are not intended to cover the full kernel design space. Workload-specific optimizations, such as QK lookahead and reverse causal-chunk ordering developed by the agent for attention variants (Appendix E.3.4), compose existing rows. New hardware features introduce additional rows when they satisfy the three inclusion criteria and fall outside the instruction families represented by existing rows.

Cost of a column. Adding a new accelerator to the taxonomy costs only one column of unit tests, bounding the per-architecture human effort to $O(\text{optimization strategies})$ rather than $O(\text{workloads} \times \text{precisions})$. In practice, two of the authors, graduate students with prior industry experience in kernel optimization, authored all four columns in roughly ten full days each, about 200 expert-hours in total: two days identifying rows and eight days building and validating cells (Appendix C.2). This is a one-time cost per architecture that amortizes across downstream workloads and precisions. Each unit test is the minimal abstraction that pairs a human-authored solution kernel with the profiling metric that verifies the optimization. The solution kernel is wrapped as a callable function with a short usage guide so the agent can read it as a syntax example, invoke it directly, or compose fragments into a larger kernel. This matters for instructions that postdate the pretraining corpus of frontier models, where the unit test is the only way the agent learns the feature exists at all (Appendix C.2).

2.4 Hardware-Aware Kernel Generation

Figure 3 shows the kernel generation loop where the agent receives a PyTorch reference for the target workload, the column of unit tests for the target architecture (Section 2.3), and previously composed hardware-aware kernels for other workloads on the same architecture. The agent iterates through a closed loop of editing the kernel, compiling for the target, executing in the sandbox, validating against the reference, and measuring performance and per-row profiling metrics. Over the course of many turns, the agent identifies the bottleneck the next optimization should address, consults the unit test that targets it, and stacks hardware-aware optimizations into one complete kernel. The per-row counter lets the agent verify that the targeted hardware feature fired before moving on, so the trajectory accumulates composed optimizations (Appendix E.1). Correct kernels that improve over the previous best are added to the workspace and condition the agent on subsequent workloads.

3 Experimental Setup

Hardware and precisions. We evaluate on NVIDIA A100-40GB (Ampere), H100 (Hopper), B200 (Blackwell), and AMD MI350 (CDNA4) using CUDA with inline PTX and HIP. Generated kernels are disallowed from using any external libraries. For each architecture we run BF16 and the lowest natively-supported low-precision format, namely FP8 (E4M3) on Hopper, NVFP4 on Blackwell, and MXFP4 [Rouhani et al., 2023] on MI350. Ampere lacks sub-BF16 floating-point support (INT8 only). Inputs are generated directly in the target precision to avoid dequantization overhead.

Established workloads. GEMM, Conv2D, and standard attention cover the compute-intensive subset of KernelBench [Ouyang et al., 2025] and have years of investment in vendor-tuned implementations. We compare against `torch.compile` with *max-autotune*, which dispatches to cuBLAS [NVIDIA, 2026], cuDNN [Chetlur et al., 2014], and FlashAttention. GEMM uses $M{=}N{=}K{=}4096$ (KernelBench standard). Conv2D uses the first AlexNet [Krizhevsky et al., 2012] layer with $N{=}256$, $3{\to}96$ channels, $224{\times}224$ input, $11{\times}11$ kernel, $S{=}4$, and $P{=}2$. Attention uses the canonical FlashAttention [Dao et al., 2022] configuration $(B, H, S, D){=}(4, 8, 4096, 64)$.

Emerging workloads. ReBased Linear Attention [Aksenov et al., 2024], DeltaNet [Yang et al., 2024], and Forgetting Attention [Lin et al., 2025] offer architectural advantages over the standard transformer but lack vendor kernel support. We additionally compare against hand-authored Triton kernels in the Flash Linear Attention (FLA) library [Yang and Zhang, 2024], autotuned per problem size, as a human upper bound. All attention variants are causal at $(B, H, S, D){=}(4, 8, 4096, 64)$.

Agent baselines. All agents have a budget of 100 *productive turns* of kernel edits, evaluations, and taxonomy or deduplicated kernel-pool reads. We compare HAWKEYE against (1) $10{\times}10$ iterative refinement loop over profiler feedback [Ouyang et al., 2025], (2) a coding agent with the HuggingFace CUDA-kernels skill [Hugging Face, 2026] and programming guides, (3) GEPA [Agrawal et al., 2026] with a self-rewriting markdown reflection, and (4) a coding agent given programming-model docs or other kernels as context. Full prompt templates and workspace layouts are in Appendix G.

Agent harness. We use OpenHands [Wang et al., 2025] to operate a shell-capable coding agent with sandboxed GPU access. The agent invokes `nvcc` or `hipcc` to compile, `python` to evaluate, and `ncu` or `rocprof` to profile. Each generated kernel is a standalone CUDA or HIP file exposing setup and forward functions via pybind11, JIT-loaded with PyTorch, and validated against a reference. We selected Gemini 3.1 Pro from a sweep over five frontier models (per-model pass@1 in Appendix B).

Metrics. Correctness uses an oracle in-precision naive kernel (Appendix G.3). Performance is in TFLOPS (workload's theoretical FLOP count divided by latency) using $2MNK$ for GEMM, im2col count for Conv2D, $2BHS^2D$ for the causal attentions, and $4BHSD^2$ for DeltaNet. We use Triton `do_bench` with a 25 ms warmup and 100 ms timed-rep budget to obtain median latency. Speedups are calculated as TFLOPS ratios against `torch.compile` in Table 2 and expert Triton FLA in Figure 4.

4 Results

We evaluate HAWKEYE on the established and emerging workloads in Table 2. Section 4.1 covers the porting axes across architectures, vendors, and precisions. Section 4.2 compares against expert-authored Triton kernels on emerging attention variants. Section 4.3 measures how kernel quality scales with the agent's turn budget and digs into the structural reason HAWKEYE trajectories continue to improve while alternatives plateau. Section 4.4 compares against six alternative hardware-aware context structures and walks through why HAWKEYE overcomes each baseline's shortcomings.

4.1 Porting Across Architectures, Vendors, and Precisions

Across GPU architectures (Ampere, Hopper, Blackwell, and MI350). HAWKEYE uses the per-architecture unit test column as context to port the PyTorch reference across architectures. On Blackwell, the agent directly composes `tcgen05.mma` and TMEM accumulators, while translating a Hopper `wgmma`-based GEMM with static conversion tools bypasses these primitives entirely (Appendix A.1). On established workloads this gives $0.88\times$ BF16 GEMM, $1.69\times$ Conv2D, and $0.98\times$ standard attention geomean against `torch.compile` across four chips. On emerging variants where `torch.compile` cannot fuse non-standard scans and gates, HAWKEYE reaches $19.70\times$, $26.47\times$, and $12.89\times$ geomean speedup on causal Linear Attention, DeltaNet, and Forgetting Attention.

Across vendors (NVIDIA and AMD). HAWKEYE enables cross-vendor porting that requires translation across programming models and core hardware primitives [Davis et al., 2025]. The only autonomous NVIDIA to AMD tool is source-level translation with Hipify [AMD ROCm Team, 2024] that can only compile kernels without Blackwell-specific syntax. On GEMM, Conv2D, and attention, Hipify MI350 kernels have a geomean $0.05\times$ performance against `torch.compile` (Appendix A.2), while HAWKEYE uses the MI350 column of unit tests to reach $1.26\times$ speedup over `torch.compile`.

Across precisions (BF16, FP8, NVFP4, and MXFP4). Roofline analysis (Appendix F.6) shows the low-precision HAWKEYE kernels increasing both the arithmetic intensity and achievable ceiling,

Table 2: HAWKEYE reaches speedups against `torch.compile` on established and emerging workloads results across four GPU architectures and four precisions, with the largest gains on newer generations and lower precisions. X denotes incorrect or runtime errors (penalized in geomean).

		Ampere (A100)		Hopper (H100)				Blackwell (B200)				MI350				Geomean	
		BF16		BF16		FP8		BF16		NVFP4		BF16		MXFP4		BF16	Low Prec.
Method	**Workload**	TFLOPS	Spdup	TFLOPS	Spdup	TFLOPS	Spdup	TFLOPS	Spdup	TFLOPS	Spdup	TFLOPS	Spdup	TFLOPS	Spdup	Spdup	Spdup
10×10 Iterative Refinement	GEMM	96.55	0.49x	X	X	X	X	X	X	10.53	0.01x	359.71	0.50x	26.87	0.01x	0.07x	0.01x
	Conv2d	9.57	0.57x	13.85	0.58x	13.95	0.58x	3.64	0.11x	3.68	0.12x	28.24	0.61x	7.20	0.15x	0.39x	0.22x
	Attention	2.89	0.04x	0.33	0.00x	X	X	3.33	0.01x	X	X	0.74	0.00x	2.13	0.01x	0.01x	0.01x
	Lin. Attn.	X	X	X	X	0.12	0.02x	X	X	X	X	X	X	X	X	0.01x	0.01x
	Delta Net	X	X	X	X	0.66	11.63x	X	X	0.50	13.06x	X	X	X	X	0.01x	1.15x
	FoX	6.18	2.02x	15.33	2.37x	X	X	2.45	0.13x	X	X	2.02	0.37x	11.01	2.00x	0.69x	0.06x
Kernel Optimization Agent Skill	GEMM	147.40	0.75x	701.00	**1.26x**	X	X	X	X	16.95	0.02x	201.11	0.28x	236.82	0.11x	0.23x	0.03x
	Conv2d	17.20	1.02x	3.23	0.13x	X	X	60.10	1.82x	5.03	0.16x	81.92	1.76x	73.89	1.49x	0.81x	0.13x
	Attention	X	X	X	X	0.06	0.00x	0.13	0.00x	X	X	X	X	X	X	0.01x	0.01x
	Lin. Attn.	22.51	6.15x	12.40	1.79x	X	X	3.21	0.10x	X	X	X	X	X	X	0.32x	0.01x
	Delta Net	X	X	X	X	X	X	X	X	0.01	0.26x	X	X	X	X	0.01x	0.03x
	FoX	27.20	8.91x	0.38	0.06x	X	X	X	X	X	X	0.01	0.00x	0.01	0.00x	0.09x	0.01x
GEPA Reflective Prompt Evolution	GEMM	161.40	0.82x	698.10	1.26x	4.70	0.00x	X	X	4.93	0.01x	291.52	0.40x	187.65	0.09x	0.25x	0.02x
	Conv2d	16.50	0.98x	3.50	0.15x	X	X	X	X	3.63	0.12x	19.85	0.43x	67.53	1.37x	0.16x	0.12x
	Attention	22.20	0.34x	X	X	222.40	**1.50x**	X	X	X	X	X	X	X	X	0.02x	0.05x
	Lin. Attn.	15.90	4.34x	X	X	X	X	X	X	X	X	X	X	X	X	0.05x	0.01x
	Delta Net	X	X	X	X	X	X	X	X	0.30	7.84x	X	X	X	X	0.01x	0.09x
	FoX	39.90	13.07x	57.75	8.95x	90.20	13.88x	0.01	0.00x	2.39	0.13x	37.75	6.85x	X	X	1.68x	0.26x
HAWKEYE	GEMM	173.79	**0.88x**	660.40	1.19x	1018.96	**0.78x**	760.66	**0.78x**	1577.64	**1.61x**	544.96	**0.75x**	1844.80	**0.88x**	**0.88x**	**1.03x**
	Conv2d	20.82	**1.24x**	38.57	**1.61x**	47.44	**1.97x**	71.78	**2.17x**	62.66	**2.04x**	87.53	**1.88x**	125.60	**2.54x**	**1.69x**	**2.17x**
	Attention	116.38	**0.88x**	337.34	**1.13x**	387.46	1.31x	455.44	**0.66x**	253.15	**0.54x**	372.47	**1.43x**	307.90	**1.17x**	**0.98x**	**0.94x**
	Lin. Attn.	75.79	**10.35x**	342.30	**24.70x**	346.07	**25.20x**	548.54	**8.72x**	378.26	**3.03x**	326.46	**67.57x**	368.59	**76.29x**	**19.70x**	**17.99x**
	Delta Net	1.91	**33.50x**	2.51	**25.50x**	1.77	**31.20x**	2.33	**36.20x**	1.24	**32.40x**	3.90	**15.88x**	3.60	**15.43x**	**26.47x**	**24.99x**
	FoX	104.13	**17.05x**	245.49	**19.02x**	246.17	**18.94x**	440.20	**11.89x**	287.47	**7.75x**	39.47	**7.16x**	49.80	**9.05x**	**12.89x**	**10.99x**

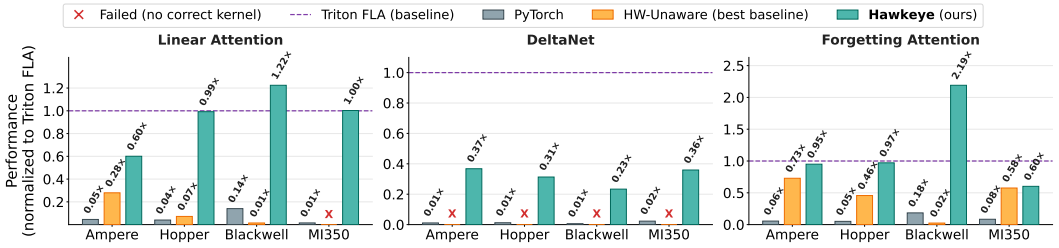


Figure 4: On attention variants, HAWKEYE approaches or exceeds performance against PyTorch, the best baseline from Table 2, and expert-written Triton FLA kernels [Yang and Zhang, 2024] (dotted line). HAWKEYE achieves 0.92× geomean against FLA on causal Linear Attention and 2.19× speedup on Blackwell Forgetting Attention. There is still headroom on the expert DeltaNet kernel.

lifting GEMM throughput by 1.5× on H100, 2.1× on B200, and 3.4× on MI350. Because the unit tests are specific to precision, translations are clean trajectories rather than retrofits of BF16 kernels. Across the established workloads in low precision, HAWKEYE reaches a 1.28× geomean against `torch.compile` in formats PyTorch cannot run natively. Some low precision kernels show a slowdown versus BF16 because they introduce intermediate quantization overhead to pass correctness.

Beyond tensor-core workloads. The same taxonomy, with no rows added, also produces fast kernels for workloads with no matrix-multiply component. On three bandwidth-bound operators in BF16 on H100 and B200 (RMSNorm and softmax at $M{=}N{=}8192$, and a row reduction at $M{=}8192$, $N{=}16384$), HAWKEYE is 1.20–4.62× faster than the stronger of `torch.compile` and PyTorch's fused ATen kernel, composing the Vectorized Memory and Warp/Wave Reduction rows, plus Persistent Scheduling for RMSNorm and the reduction. On the FP32 FDTD-2D electromagnetic stencil from SwizzlePerf [Tschand et al., 2025], a scientific workload outside the taxonomy's source kernels, HAWKEYE reaches 2.50 ms on H100 and 1.12 ms on B200, 3.19× and 4.81× faster than the strongest baseline in each cell. These are single runs at one problem size on two architectures (Appendix F.5).

4.2 Comparing Against Expert-Written Triton Kernels

HAWKEYE approaches or exceeds Triton kernels in the Flash Linear Attention library [Yang and Zhang, 2024], which is the closest proxy to an expert upper bound on the emerging attention

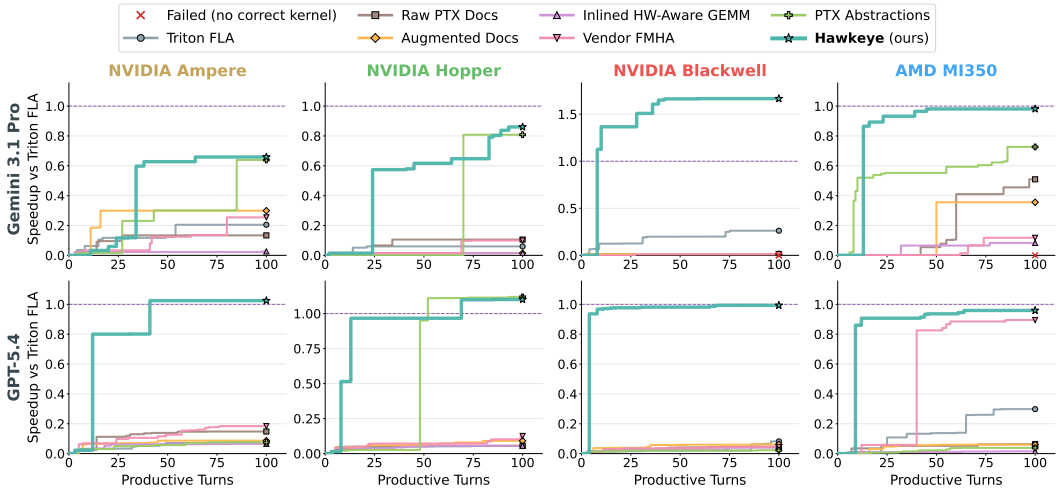


Figure 5: Scaling agent test-time compute with HAWKEYE yields strong speedups on causal Linear Attention with Gemini 3.1 Pro and GPT-5.4 against baseline context structures, all measured against expert-written Triton FLA (dashed line). HAWKEYE is the only method that keeps producing faster kernels as productive turns grow (kernel edits, evaluations, and abstraction reads, see Section 3), reaching $0.78\times$–$1.65\times$ FLA across architectures while baselines plateau at lower speedups.

variants. HAWKEYE reaches geomeans of $0.92\times$, $0.31\times$, and $1.05\times$ FLA on causal Linear Attention, DeltaNet, and Forgetting Attention (Figure 4), including $1.22\times$ and $1.00\times$ on Linear Attention on Blackwell and MI350 and $2.19\times$ on Forgetting Attention on Blackwell where the Triton backend is still maturing [Ren et al., 2026]. Approaching this performance autonomously, in dollars and minutes per kernel rather than days of expert effort, is a substantial productivity win over hand-written kernels. On DeltaNet, trajectory analysis shows that HAWKEYE attempts hardware-aware optimizations but struggles to compose the corresponding unit tests into a complete kernel, ultimately falling back to simpler implementations to satisfy correctness (Appendix E.2). At the evaluated shape, DeltaNet is memory-bound, with an arithmetic intensity of approximately 32 FLOP per byte of minimum required HBM data movement, 5–$12\times$ below the BF16 ridge points of the evaluated GPUs (Appendix F.6). The generated kernels achieve only 1–3% of peak HBM bandwidth, indicating that they do not saturate the available memory bandwidth. The taxonomy already contains the required optimization strategies; the observed difficulty lies in composing them into a correct, optimized kernel.

4.3 Scaling Kernel Performance with Test-Time Compute

Figure 5 shows that scaling test-time compute with HAWKEYE generates the most performant kernels across architectures. In observed trajectories, early turns retrieve unit tests, plan stacking strategies, and analyze weaker kernels via per-cell profiling metrics (Appendix E.2). The jump arrives when optimizations can be composed into a coherent kernel. The depth of the required stack explains the curve shapes across architectures. On MI350 the agent reaches $0.90\times$ expert in 23 productive turns because MFMA, LDS, and wave-occupancy compose into a thin stack. On Blackwell the agent needs roughly 50 productive turns to reach $1.65\times$ expert because TMEM accumulators, ring-buffered TMA, cluster-multicast loads, and a phase-bit protocol must all converge. Past the transition, additional turns invent cross-optimization patterns no unit test describes, like ping-pong TMEM and a reversed causal chunk schedule that close the gap on Blackwell (Appendix E.3).

Variance across trajectories. Agent trajectories are stochastic, so we repeat causal Linear Attention in BF16 with five independent trajectories on MI350 and B200, holding the taxonomy, harness, backbone, and 100-turn budget fixed. On MI350 the five runs reach a mean of $68.8\times$ against `torch.compile` (standard deviation $8.2\times$, range 61.3–$82.2\times$), consistent with the single trajectory in Table 2. On B200, measured against the naive quadratic-form PyTorch reference rather than `torch.compile`, the mean is $56.3\times$ (standard deviation $12.8\times$, range 42.9–$71.2\times$). The spread is 12% and 23% of the mean, small relative to the gains.

The unit test profiling metrics enable the MI350 trajectory to debug tightly coupled individual optimizations and avoid regression from partial composition (Figure 6). On Hopper, a producer-

consumer split without the ring buffer regresses throughput by roughly half, and the per-optimization signal lets the agent revert (Appendix E.1). Baseline context structures plateau because each fails to implement or compose hardware-aware optimizations in distinct ways (Appendix D.2).

4.4 Evaluating Context Efficiency Against Other Hardware-Aware Context Structures

A context structure is efficient for GPU kernel optimization when it is minimal enough to avoid context pollution and detailed enough to enable optimization stacking. We evaluate six baselines along this spectrum (Figure 2b), and HAWKEYE addresses each baseline's unique failure modes.

Comparing to algorithm without hardware syntax. DSL-level conditioning shows the agent an effective algorithm but no path to the architecture's tensor-core primitives. On Blackwell, the trajectory plateaus at 220.8 TFLOPS ($0.26\times$ expert) by falling back to Ampere-era MMA syntax without ever reaching `tcgen05` or its TMEM accumulator. Ampere correctly uses `wmma` syntax, but gets to $0.27\times$ because it cannot orchestrate asynchronous data movement to saturate bandwidth (Appendix D.3). HAWKEYE *closes this gap because each unit test shows the architecture-specific syntax for optimizations, so the agent never has to fall back on syntax from older generations.*

```
// Q: reload from gmem each iter
load_to_LDS(Q[q_start..+64]);
// GEMM1: S = Q @ K^T
// 3 of 4 waves idle in MFMA
if (warp_id == 0)
  for (k,r,c)
    s[r*4+c] = mfma_16x16x16(
        scalar_gather(LDS_Q),
        scalar_gather(LDS_K),
        s[r*4+c]);
// causal: per-elem branch
// fires on every tile
for_each_elem (r,c,e):
  if (col > row) v = 0;
```

(a) `Vendor FMHA` (39 TFLOPS)

```
// Q: load once, re-read from LDS
load_to_LDS(Q);
a = LDS_Q[row*16 + lane];
// GEMM1: S = Q @ K^T
// all 4 waves cooperate, but
// small tile -> 4 outer steps
for (k = 0; k < 4; k++)
  for (r,c in 2x2)
    C[r][c] = mfma_16x16x16(
        a, b, C[r][c]);
// causal: per-elem branch
// fires on every tile
for_each_elem:
  if (col > row) v = 0;
```

(b) `Raw PTX Docs` (170 TFLOPS)

```
// Q: load once, kept in VGPRs
q_regs = load_Q_to_REGS();
// GEMM1: S = Q @ K^T
// 4 waves, 4x larger MFMA tile
for (ki = 0; ki < D; ki += 16)
  s = mfma_32x32x16(
      q_regs[ki/16],
      LDS_K[cur][ki], s);
// causal: tile-level fast path,
// only diagonal tile pays
if (kv_start + BN <= q_start)
  write_W();
else
  write_W_causal();
```

(c) **HAWKEYE (326 TFLOPS)**

Figure 6: HAWKEYE wins by composing hardware-native primitives end-to-end, while baselines stop part way through the stack. Inner KV-tile loop of three causal Linear Attention kernels (Gemini 3.1 Pro, MI350, BF16). The `Vendor FMHA` baseline only utilizes a quarter of the matrix cores, and the `Raw PTX Docs` baseline stays on the smaller MFMA shape and reloads operands through shared memory. HAWKEYE stacks all three optimizations and reaches 98% of the hand-tuned expert.

Comparing to hardware documentation. Raw and augmented documentation show that the right primitives are present in the corpus but cannot be composed. The Blackwell raw-docs trajectory stops at 5.6 TFLOPS ($0.013\times$ expert) because string-match retrieval anchors in the legacy `wgmma` chapter and never reaches `tcgen05`. Curated documentation removes the navigation cost, but Hopper never beats the naive baseline because information across primitives like `wgmma`, TMA, and mbarrier is split across excerpts the agent cannot compose (Appendix D.5). HAWKEYE *closes this gap because each unit test shows a minimal but complete set of optimizations, so the agent easily retrieves syntax and composition rather than reassembling them across documentation excerpts.*

Comparing to few-shot hardware-aware kernels. Inlined kernel examples and vendor production references show that finished compositions cannot be decomposed into the primitives they are built up from. An inlined hardware-aware GEMM exposes the right tensor-core and asynchronous-copy primitives but entangles them with the source workload's tile geometry and scheduling choices (Appendix D.6). Vendor production kernels like CUTLASS and Composable Kernel are written for human reuse and their abstractions hide the primitive recipes the agent needs to extract (Appendix D.7). HAWKEYE *closes this gap because the taxonomy shows composable optimization boundaries, so the agent reuses an abstraction the engineer authored rather than inferring it from a finished kernel.*

Comparing to curated abstractions for important optimizations. The PTX-abstractions show that primitives alone do not enable multi-optimization composition. Without the HAWKEYE unit tests and corresponding profiling metrics, the Hopper kernel only reaches 240.9 TFLOPS with annotations where the agent recognizes the producer-consumer split as the next move but cannot implement it. The MI350 control case shows a 35% gap to HAWKEYE with Gemini and a $28\times$ gap with GPT-5.4

(Appendix D.8). HAWKEYE *closes this gap because the unit tests guide optimization stacking that primitives alone cannot encode to turn correct syntax into composed kernels.*

5 Related Work

Training-based methods. KevinRL [Baronio et al., 2025] and CudaAgent [Dai et al., 2026] both fine-tune coding agents on runtime rewards, but rely on the target architectures and precisions being in the training distribution to hillclimb on rewards. HAWKEYE does not rely on the programming model to be in distribution and instead exposes the architecture-specific optimizations as a structured taxonomy of expert-authored unit tests, inspectable and extensible to new accelerators.

Search-based methods. K-Search [Cao et al., 2026] and AI CUDA Engineer [Lange et al., 2025] explore the kernel space with a world model and evolutionary discovery, while CudaForge [Zhang et al., 2025] and KernelEvolve [Liao et al., 2025] augment search with runtime feedback or retrieval over a kernel corpus. KernelFoundry [Wiedemann et al., 2026] combines MAP-Elites quality-diversity search, prompts co-evolved with kernels, and template-based parameter tuning for SYCL and CUDA kernels; its prompt evolution over a fixed context is closest to our GEPA baseline. These methods search without per-architecture grounding, so they can struggle to use the required primitives. HAWKEYE operates at test time but grounds the search in architecture-specific unit tests so the agent reaches the right primitives directly.

Context engineering methods. KernelBlaster [Dong et al., 2026] retrieves cross-task examples into a continual memory, while SwizzlePerf [Tschand et al., 2025] and HuggingFace CUDA skills [Hugging Face, 2026] ship hand-written context for specific optimizations. KernelAgent [Cheng et al., 2026] orchestrates profiling, diagnosis, and optimization agents that read Nsight Compute metrics and retrieve optimization patterns from a curated database to optimize Triton kernels. Its agents must infer the bottleneck from profiler output, whereas each HAWKEYE cell binds a specific counter to the optimization it verifies, and it generates Triton rather than the native instructions of each architecture. These methods inject hardware-aware content directly but are limited to a static set of optimizations or architectures. HAWKEYE is a structured and executable form of agent skills rather than an alternative to them: each cell pairs a skill-style guide with a unit test, an expert solution kernel, and the counter that verifies the optimization, and the strategy-by-architecture structure extends to a new accelerator with one new column of unit tests (full comparison in Table 5). The contribution is this context structure, not the CUDA and HIP artifacts, which another backend would replace.

6 Discussion

Limitations. HAWKEYE does not eliminate the need for expert input, since each new accelerator still requires an expert to author the corresponding column of unit tests. Our evaluation covers single-GPU kernels for tensor-algebra workloads, bandwidth-bound operators, and one scientific stencil, and excludes communication-bound multi-GPU kernels. Irregular workloads such as sparse operators and graph analytics are not evaluated; their difficulty comes from irregularity in the workload itself, which may be layered above the existing rows or require new rows derived from sparse and graph frameworks. Low-precision kernels depend on quantization tolerance [Dettmers et al., 2022, Xiao et al., 2023], and attention variants often force mixed precision due to compounding loss. On workloads with heavy vendor tuning, the agent does not always reach hand-written performance because it struggles to compose the isolated unit tests into complete kernels. We view this as an opportunity for future work in autonomous kernel *composition* now that agents are capable of recognizing and implementing hardware-specific optimizations.

Conclusion. HAWKEYE is a taxonomy of expert-authored domain knowledge that grounds coding agents in the architecture's programming model with only 10 unit tests per accelerator. By scaling test-time compute against this minimal supervision, HAWKEYE generates hardware-aware kernels that match or exceed expert-written PyTorch and Triton in BF16 and low precision across NVIDIA Ampere, Hopper, Blackwell, and AMD MI350. We release the taxonomy, hardware-aware kernels, and evaluation harness for all four architectures, and hope this points toward a research direction in which the role of the human shifts from writing performant kernels to architecting the structured knowledge that enables coding agents to effectively scale autonomous kernel optimization.

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. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Yaroslav Aksenov, Nikita Balagansky, Sofia Maria Lo Cicero Vaina, Boris Shaposhnikov, Alexey Gorbatovski, and Daniil Gavrilov. Linear transformers with learnable kernel functions are better in-context models, 2024. URL https://arxiv.org/abs/2402.10644.

AMD. *AMD Instinct MI-Series ISA Reference Guides (CDNA1–CDNA4)*, 2024. URL https://developer.amd.com/. Instruction families counted from MFMA and related entries in each CDNA architecture ISA document.

AMD ROCm Team. Hipify: Convert cuda to portable hip c++ code. https://github.com/ROCm/HIPIFY, 2024. Accessed: 2026-03-30.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

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

Kaiming Cheng, Laura Wang, Jack Khuu, Mark Saroufim, Wenyuan Chi, Jiannan Wang, and Joe Isaacson. KernelAgent: Hardware-guided GPU kernel optimization via multi-agent orchestration. PyTorch Blog, https://pytorch.org/blog/kernelagent-hardware-guided-gpu-kernel-optimization-via-multi-agent-orchestration/, 2026. March 6, 2026.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014. URL https://arxiv.org/abs/1410.0759.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.

Joshua Hoke Davis, Pranav Sivaraman, Joy Kitson, Konstantinos Parasyris, Harshitha Menon, Isaac Minn, Giorgis Georgakoudis, and Abhinav Bhatele. Taking gpu programming models to task for performance portability. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pages 776–791, 2025.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022. URL https://arxiv.org/abs/2208.07339.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. Kernelblaster: Continual cross-task cuda optimization via memory-augmented in-context reinforcement learning, 2026. URL https://arxiv.org/abs/2602.14293.

William Hu, Drew Wadsworth, Sean Siddens, Stanley Winata, Daniel Y Fu, Ryann Swann, Muhammad Osama, Christopher Ré, and Simran Arora. Hipkittens: Fast and furious amd kernels. *arXiv preprint arXiv:2511.08083*, 2025.

Hugging Face. Custom kernels for all from codex and claude. https://huggingface.co/blog/custom-cuda-kernels-agent-skills, 2026. Blog post.

jeromeku. Gluon? https://github.com/triton-lang/triton/issues/7392, 2025. GitHub issue #7392, triton-lang/triton, opened July 4, 2025.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

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

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

Tao Lin. Measuring automated kernel engineering. https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/, 02 2025.

Zhixuan Lin, Evgenii Nikishin, Xu Owen He, and Aaron Courville. Forgetting transformer: Softmax attention with a forget gate. *arXiv preprint arXiv:2503.02130*, 2025.

Chao Liu, Jing Zhang, Letao Qin, Qianfeng Zhang, Liang Huang, Shaojie Wang, Anthony Chang, Chunyu Lai, Illia Silin, Adam Osewski, Poyen Chen, Rosty Geyyer, Hanwen Chen, Tejash Shah, Xiaoyan Zhou, and Jianfeng Yan. Composable kernel. URL https://github.com/ROCm/composable_kernel.

NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. Whitepaper V1.01, 2022. URL https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

NVIDIA. NVIDIA Blackwell Architecture Technical Brief. Technical brief, 2024a. URL https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.

NVIDIA. *Parallel Thread Execution ISA Reference Manual*, 2024b. URL https://docs.nvidia.com/cuda/parallel-thread-execution/. Versions 6.0–8.6. Instruction family counts tallied from per-version changelogs.

NVIDIA. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass, 2025. Accessed: 2025-03-29.

NVIDIA. cuBLAS library documentation. NVIDIA Developer Documentation, 2026. URL https://docs.nvidia.com/cuda/cublas/.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

PyTorch Contributors. Pytorch source code: test_fp8.py (line 442). https://github.com/pytorch/pytorch/blob/696b6fa86f8f6443724c7d03756238a032264892/test/inductor/test_fp8.py#L442, 2024. Commit 696b6fa.

Manman Ren, Nick Riasanovsky, Neil Dhar, Hongtao Yu, Jie Liu, Partha Kanuparthy, and Shane Nay. Warp specialization in triton: Design and roadmap. https://pytorch.org/blog/warp-specialization-in-triton-design-and-roadmap/, January 2026. PyTorch Blog.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, et al. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.

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

Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

tiktoken. tiktoken: A fast bpe tokeniser for use with openai's models. `https://github.com/openai/tiktoken`, 2022.

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

Arya Tschand, Muhammad Awad, Ryan Swann, Kesavan Ramakrishnan, Jeffrey Ma, Keith Lowery, Ganesh Dasika, and Vijay Janapa Reddi. Swizzleperf: Hardware-aware llms for gpu kernel performance optimization, 2025. URL `https://arxiv.org/abs/2508.20258`.

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. Openhands: An open platform for ai software developers as generalist agents, 2025. URL `https://arxiv.org/abs/2407.16741`.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. Kernelfoundry: Hardware-aware evolutionary gpu kernel optimization, 2026. URL `https://arxiv.org/abs/2603.12440`.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 Jul 2023. URL `https://proceedings.mlr.press/v202/xiao23c.html`.

Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. Task-based tensor computations on modern gpus. *Proceedings of the ACM on Programming Languages*, 9(PLDI):396–420, 2025.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL `https://github.com/fla-org/flash-linear-attention`.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *Advances in neural information processing systems*, 37: 115491–115522, 2024.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026.

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

Appendix

Table of Contents

A Hardware-specific kernel code anatomy 15
- **A.1 MMA across architectures 15**
- **A.2 Async data movement across architectures 15**
- **A.3 Kernel across precisions 16**
- **A.4 Taxonomy derivation from Flash Attention-2 ..... 18**
- **A.5 Taxonomy derivation from ThunderKittens 19**
- **A.6 Growth of hardware-specific code 20**

B Frontier LLMs on the taxonomy 20
- **B.1 Pass@1 across architectures 20**
- **B.2 Failure-mode buckets 22**

C Structure of the Hawkeye taxonomy and agent workspace 23
- **C.1 Full taxonomy 23**
- **C.2 Anatomy of a taxonomy cell 23**
- **C.3 Workspace layout and retrieval 25**

D Comparative analysis of kernel optimization systems and context structures 25
- **D.1 Related autonomous kernel optimization systems 25**
- **D.2 Three-cluster failure-mode framework 26**
- **D.3 Triton FLA 27**
- **D.4 Raw PTX and ROCm docs 28**
- **D.5 Augmented documentation 30**
- **D.6 Inlined GEMM exemplar 31**
- **D.7 Vendor FMHA 32**
- **D.8 PTX abstractions ablation 33**

E Optimization stacking and composition 35
- **E.1 Closed-loop pattern 35**
- **E.2 Trajectories across workloads 37**
- **E.3 Composition across kernels 38**
  - **E.3.1 Cross-workload reuse 39**
  - **E.3.2 Cross-architecture composition 39**
  - **E.3.3 Cross-precision cascade 41**
  - **E.3.4 Invented patterns 41**

F Generalization across porting axes 42
- **F.1 Cross-workload 42**
- **F.2 Cross-architecture 42**
- **F.3 Cross-precision cascade 44**
- **F.4 Problem-size sweep 44**
- **F.5 Workloads beyond tensor algebra 44**
- **F.6 Roofline analysis 45**

G Experimental infrastructure 46
- **G.1 Prompt templates 46**
- **G.2 Evaluation pipeline 46**
- **G.3 Precision and correctness tolerance 48**
- **G.4 Compilation and profiling 48**

A Hardware-specific kernel code anatomy

The same algorithm requires materially different code on each GPU architecture and each precision, and that hardware-specific code dominates production kernel libraries. Sections A.1–A.3 show side-by-side excerpts from HAWKEYE API wrappers, Section A.4 derives the taxonomy rows from FlashAttention-2, and Section A.6 quantifies the cost of supporting new hardware.

A.1 MMA across architectures

Four HAWKEYE API wrappers realize the same logical $16\times16\times16$ BF16 matmul, with the issue scope widening from a warp on Ampere (`mma.sync`) to a warpgroup on Hopper (Figure 7, 128 threads issuing asynchronously into 64 accumulator registers via 64-bit SMEM descriptors and `wgmma.commit_group` / `wgmma.wait_group`) to a full CTA on Blackwell (`tcgen05.mma` with TMEM accumulators), while CDNA4 holds at the wavefront (Figure 8, 64 threads producing a 32×32 tile in 16 AGPR accumulators per lane with `cbsz`, `abid`, and `blgp` immediates and `s_waitcnt` completion). Each step moves operand staging deeper into the memory hierarchy and pushes tile-shape selection out of the mnemonic and into a runtime descriptor (full Ampere and Blackwell wrappers in the code release).

```
template<int ScaleD, int ScaleA, int ScaleB, int TransA, int TransB>
__device__ __forceinline__ void mma_async_m64n128k16(
        float d[8][8], uint64_t desc_a, uint64_t desc_b) {
    asm volatile(
        "{\n"
        "wgmma.mma_async.sync.aligned.m64n128k16.f32.bf16.bf16 "
        "{%0, %1, ..., %63},"   // 64 f32 accumulator registers
        " %64,"                  // desc_a (SMEM descriptor)
        " %65,"                  // desc_b (SMEM descriptor)
        " %66, %67, %68, %69, %70;\n"  // ScaleD, ScaleA, ScaleB, TransA, TransB
        "}\n"
        : /* d[0][0] ... d[7][7] as "+f" outputs */
        : "l"(desc_a), "l"(desc_b),
          "n"(int32_t(ScaleD)), "n"(int32_t(ScaleA)),
          "n"(int32_t(ScaleB)), "n"(int32_t(TransA)), "n"(int32_t(TransB)));
}
```

Figure 7: Hopper WGMMA `wgmma.mma_async.sync.aligned.m64n128k16` BF16 wrapper.

```
__device__ __forceinline__
fp32x16 bf16_32x32x16(bf16x8 a, bf16x8 b, fp32x16 c) {
    return __builtin_amdgcn_mfma_f32_32x32x16_bf16(
        a, b, c,
        0, 0, 0   // cbsz, abid, blgp
    );
}
```

Figure 8: CDNA4 / MI350 `v_mfma_f32_32x32x16_bf16` wrapper.

A.2 Async data movement across architectures

Each generation introduces a new load mechanism, shifting both the unit of transfer and the addressing entity. Ampere uses per-thread `cp.async` with software-pipelined commit/wait. Hopper offloads the tile to a hardware DMA engine via a host-side `CUtensorMap`, with completion tracked as an mbarrier byte count (Figure 9), and adds a `multicast::cluster` variant that broadcasts one global read into every CTA in a thread-block cluster. Blackwell keeps the same TMA unit and adds 2-CTA cooperative loads where a single tile feeds a `tcgen05.mma.cta_group::2` spanning the cluster pair. CDNA4 uses a descriptor-driven DMA that flows from HBM into LDS without touching a VGPR, with completion tracked by scalar wait counts (Figure 10). The agent learns each from one

API wrapper paired with one unit test exercising the matching `config.json` counter (full Ampere and Blackwell wrappers in the code release).

```
template<int col_group_elems = 64>
__device__ __forceinline__ void load(
        void* dst,
        void const* const src_tma_map,
        uint64_t* bar,
        int row_offset,
        int col_offset) {
    uint64_t tma_ptr  = reinterpret_cast<uint64_t>(src_tma_map);
    uint32_t mbar_ptr = static_cast<uint32_t>(__cvta_generic_to_shared(bar));
    uint32_t dst_ptr  = static_cast<uint32_t>(__cvta_generic_to_shared(dst));
    asm volatile(
        "cp.async.bulk.tensor.5d.shared::cluster.global.tile."
        "mbarrier::complete_tx::bytes"
        " [%0], [%1, {%3, %4, %5, 0, 0}], [%2];"
        : : "r"(dst_ptr), "l"(tma_ptr), "r"(mbar_ptr),
            "n"(0), "r"(row_offset), "r"(col_offset / col_group_elems)
        : "memory");
}
```

Figure 9: Hopper TMA tile load. One thread instructs the DMA engine to copy a full 2D tile, with mbarrier byte-count completion.

```
__device__ __forceinline__
i32x4 make_srsrc(const void* ptr, uint32_t range_bytes,
                 uint32_t row_stride_bytes = 0) {
    auto as_u64 = (uint64_t)reinterpret_cast<uintptr_t>(ptr);
    // config = 0x110000: NUM_FORMAT=float, DATA_FORMAT=32bit, raw buffer
    buffer_resource rsrc = {as_u64, range_bytes, 0x110000};
    /* optional swizzle-enable bits in row_stride_bytes ... */
    return *reinterpret_cast<const i32x4*>(&rsrc);
}

__device__ __forceinline__
void buffer_load_lds_dwordx4(i32x4 srd, uint32_t lds_base,
                             int voffset, int soffset) {
    uint32_t lds_addr = __builtin_amdgcn_readfirstlane(lds_base);
    asm volatile(
        "s_mov_b32 m0, %0\n"
        "buffer_load_dwordx4 %1, %2, %3 offen lds"
        : : "s"(lds_addr), "v"(voffset), "s"(srd), "s"(soffset)
        : "memory", "m0");
}
```

Figure 10: CDNA4 SRD-based global-to-LDS DMA. Each lane provides only an offset, the base lives in SGPRs, and data flows directly from HBM to LDS without touching VGPRs.

A.3 Kernel across precisions

Moving from BF16 to FP8 to NVFP4 (Blackwell) or MXFP4 (CDNA4) is a cascade through the kernel rather than a datatype substitution. Element size halves, so MMA K per instruction doubles, the SMEM column group (`128B / element_size`) shrinks, the swizzle pattern and TMA tensor map geometry change to match, the minimum BK shifts to fill one swizzle group, and FP4 grows an entire new operand class for block scales (TMEM on Blackwell, LDS on CDNA4) with its own staging code. Precision-invariant steps such as applying β and the causal mask, writing W, and the second matmul in BF16 on the dequantised intermediate are unchanged, so low-precision kernels are actually mixed-precision in practice. The taxonomy carries a dedicated `quantized_mma` cell per architecture so the agent learns the cascade end to end before composing it into a workload.

```
// Step 1: S = Q_consumer @ K^T (64x128) using FP8 WGMMA
// FP8 WGMMA K=32, HEAD_DIM/32 = 4 iterations
float s_acc[BN / 16][8];
memset(s_acc, 0, sizeof(s_acc));

warpgroup::fence();
#pragma unroll
for (int k_it = 0; k_it < HEAD_DIM / WGMMA_K_FP8; ++k_it) {   // WGMMA_K_FP8 = 32
    uint64_t dq = wgmma::make_descriptor(
        &sQ_base[k_it * WGMMA_K_FP8], 16, 512, wgmma::Swizzle::B64);
    uint64_t dk = wgmma::make_descriptor(
        &sK[k_it * WGMMA_K_FP8], 16, 512, wgmma::Swizzle::B64);
    wgmma::mma_async_e4m3<BN, 1, 1, 1>(s_acc, dq, dk);    // 3 immediates, not 5
}
warpgroup::commit();
warpgroup::wait<0>();
```

Figure 11: Hopper FP8 DeltaNet inner loop. Three immediates (`ScaleD, ScaleA, ScaleB`) since FP8 `wgmma` does not expose transpose control, $K{=}32$ per instruction collapses the inner k_{it} loop by $2\times$, and the descriptor stride for the swizzle group is `B64` instead of `B128`.

Figures 11 and 12 show the inner loop of the same DeltaNet kernel under this cascade, contrasting NVIDIA's descriptor side channel with AMD's instruction-level scaling. On Hopper the WGMMA template collapses from five immediates to three because FP8 exposes no transpose control, and the descriptor stride moves to a `B64` swizzle group. CDNA4 reuses the `32x32x64` MFMA family and selects FP4 via `cbsz:4` and `blgp:4`, with MXFP4 block scaling exposed as a separate `v_mfma_scale_f32_32x32x64_f8f6f4` instruction that reaches roughly 87% of unscaled BF16 throughput on MI350. The Hopper BF16 baseline and the Blackwell NVFP4 path appear in the code release.

```
// ---- FP4 MFMA: QK = Q @ K^T with MXFP4 block scaling ----
// One scaled call. Q and K are raw FP4 with E8M0 per-block scales.
fp32x16 qk;
{
    uint8_t* q_lds = (uint8_t*)(smem + MK3_Qfp4);
    uint8_t* k_lds = (uint8_t*)(smem + MK3_Kfp4);
    constexpr int HF             = D_C / 2;
    constexpr int scales_per_row = D_C / 32;       // 1 E8M0 per 32 elements

    intx4_t a_op = *(const intx4_t*)(q_lds + (lm + my_row)*HF + my_kh*16);
    intx4_t b_op = *(const intx4_t*)(k_lds + (ln + my_row)*HF + my_kh*16);

    // Load E8M0 block scales for Q (A operand) and K (B operand)
    int sq = (int)Q_sc[(cs + lm + my_row) * scales_per_row + my_kh];
    int sk = (int)K_sc[(cs + ln + my_row) * scales_per_row + my_kh];

    qk = {};
    qk = mfma_fp4::mfma_scale_32x32x64(a_op, b_op, qk, sq, sk);

    #pragma unroll
    for (int i = 0; i < 16; i++) qk[i] *= scale;       // post-multiply by 1/sqrt(D)
}
```

Figure 12: CDNA4 MXFP4 DeltaNet inner loop. Same MFMA mnemonic family as BF16 (`v_mfma_..._f8f6f4`) with FP4 mode selected via `cbsz:4 blgp:4` immediates and a separate scaled variant `v_mfma_scale_f32_32x32x64_f8f6f4` that takes E8M0 scale registers. $K{=}64$.

A.4 Taxonomy derivation from Flash Attention-2

We derive the taxonomy rows by decomposing hand-written kernels from FlashAttention-2, 3, and 4, ThunderKittens, HipKittens, and CUTLASS into contiguous code regions, each implementing an optimization associated with a specific hardware instruction family. We compare corresponding regions across architecture-specific implementations and workloads to distinguish hardware-dependent optimizations from algorithm-specific logic. A region becomes a taxonomy row only if it satisfies the three conditions in Section 2.3. Table 3 maps each resulting row to the source construct from which it is derived.

Table 3: Source construct for each taxonomy row. Row numbers follow Table 4.

Row	Source construct
1. MMA Unit	FlashAttention-2 `HMMA16816` (`mma.sync.m16n8k16`) calls for QK^T and PV (Figure 13)
2. Quantized Precision	ThunderKittens FP8 GEMM and its Hopper inner loop
3. Shared Memory Layout	FlashAttention-2 128B column-group XOR swizzle, matched to the CUTLASS descriptor layout
4. Vectorized Memory	FlashAttention-2 vectorized scale-and-add and global-load path
5. Async Pipeline	FlashAttention-2 `cp.async.cg` with commit/wait groups on Ampere, generalizing to TMA and mbarrier on Hopper and Blackwell (Section A.2)
6. Producer/Consumer	ThunderKittens Hopper GEMM producer and consumer warpgroups with register reallocation (Figure 14)
7. Epilogue Pipeline	ThunderKittens and CUTLASS collective epilogue
8. Warp/Wave Reduction	FlashAttention-2 row-max and row-sum reductions for online softmax
9. Multi-Unit Coordination	ThunderKittens Blackwell 2-CTA cooperative MMA with cluster multicast (Figure 15)
10. Persistent Scheduling	ThunderKittens and CUTLASS GEMM tile scheduler with L2-swizzled block remapping

Figure 13 shows the FlashAttention-2 MMA Unit region (Row 1). Shared Memory Layout (Row 3, via `ldmatrix`), Vectorized Memory (Row 4, via `float4`-equivalent loads), Async Pipeline (Row 5, via `cp.async.cg` commit/wait), and Warp Reduction (Row 8, for online softmax) follow the same decomposition and appear in the code release.

```
#pragma unroll
for (int j = 0; j < kWarpTileSeqLenK; ++j) {
    HMMA16816(R_S[0][j][0], R_S[0][j][1],
              R_Q[0][0][0], R_Q[0][0][1],
              R_Q[0][0][2], R_Q[0][0][3],
              R_K[j][0], R_K[j][1],
              R_S[0][j][0], R_S[0][j][1]);
}

#pragma unroll
for (int j = 0; j < kWarpTileHeadDimV; ++j) {
    HMMA16816(R_O[0][j][0], R_O[0][j][1],
              R_S[0][w][0], R_S[0][w][1],
              R_S[0][w + 1][0], R_S[0][w + 1][1],
              R_V[j][0], R_V[j][1],
              R_O[0][j][0], R_O[0][j][1]);
}
```

Figure 13: Row 1, MMA Unit. FlashAttention-2 HMMA16816 calls for QKˆT and PV.

The MMA region pins the kernel's compute-side primitive. Both attention matmuls compile to Ampere `mma.sync.aligned.m16n8k16`, and that one instruction choice propagates outward to dictate thread count, output tile size, and operand format across the rest of the kernel. The Async Pipeline region pins the data-side primitive the same way (`cp.async.cg` bypassing L1, commit/wait forming a software pipeline that overlaps the next K/V load with the current matmul, illustrated for each architecture in Section A.2). Each row in the taxonomy isolates one such region so the agent can vary it without disturbing the rest.

Every row pairs to a profiling counter that confirms the optimization fired. The MMA row reads `sm__inst_executed_pipe_tensor`, Shared Memory Layout reads shared-memory efficiency and bank-conflict counters, Async Pipeline reads DRAM bandwidth and the barrier-stall percentage, Vectorized Memory reads `l1tex__t_sectors_pipe_lsu_mem_global_op_ld`, and Warp Reduction reads warp execution efficiency. The mapping appears alongside the row in Table 4, and Section C.2 shows how each counter is wired into the unit-test artifact.

The remaining rows (Quantized MMA, Producer-Consumer, Epilogue Pipeline, Multi-unit Coordination, Persistent Scheduling) cover kernel regions that FA-2 does not reach. We derive these in Appendix A.5 from the Hopper and Blackwell GEMMs in ThunderKittens and the CUTLASS epilogue and tile scheduler, with HipKittens informing the CDNA4 cells.

A.5 Taxonomy derivation from ThunderKittens

The remaining rows draw deep inspiration from ThunderKittens [Spector et al., 2024] on NVIDIA and its AMD counterpart HipKittens [Hu et al., 2025] on CDNA4. We analyze the BF16, FP8, and NVFP4 production GEMMs in ThunderKittens for Hopper and Blackwell to identify the optimizations they rely on for peak throughput, and the operators they are built from. Figures 14 and 15 show the central regions of each.

```
struct producer {
    __device__ static void setup(producer_setup_args<layout> args) {
        warpgroup::decrease_registers<40>();
    }
    __device__ static void load(producer_load_args<layout> args) {
        if (warpgroup::laneid() == 0) {
            tma::expect(args.inputs_arrived, args.input);
            for (int i = 0; i < M_BLOCK; i++)
                tma::load_async(args.input.a[i], args.globals.A,
                                {args.common.coord.x+i, args.iter}, args.
    inputs_arrived);
            for (int i = 0; i < N_BLOCK; i++)
                tma::load_async(args.input.b[i], args.globals.B,
                                {args.iter, args.common.coord.y+i}, args.
    inputs_arrived);
        }
    }
};
struct consumer {
    __device__ static void setup(consumer_setup_args<layout> args) {
        warpgroup::increase_registers<232>();
        kittens::warp::zero(args.state.accum);
    }
    __device__ static void compute(consumer_compute_args<layout> args) {
        warpgroup::mma_AB(args.state.accum,
                          args.input.a[warpgroup::groupid()],
                          reinterpret_cast<wide_tile&>(args.input.b));
        warpgroup::mma_async_wait();
        if (warp::laneid() == 0) arrive(args.inputs_finished);
    }
};
```

Figure 14: ThunderKittens Hopper BF16 GEMM. Producer warpgroup streams A/B tiles in via TMA, consumer warpgroup issues WGMMA, with explicit register reallocation between the two roles.

The recurring regions we extract this way inform both the row inventory and the reference solution kernel in each Hopper, Blackwell, or CDNA4 cell. FA-2 grounds rows 1–5, ThunderKittens and HipKittens ground rows 6–10, and the taxonomy keeps that lineage explicit so adding a new architecture column reduces to porting the same set of regions rather than rediscovering them.

```
d_tt_t d_tt[C::MMA_PIPE_DEPTH];
#pragma unroll
for (int i = 0; i < C::MMA_PIPE_DEPTH; i++)
    d_tt[i] = tm_alloc.template allocate<d_tt_t>((i+warpgroup::warpid())*C::Nb);

for (int task_iter = 0; ; task_iter++) {
    for (int idx = 0; idx < iters_per_task; idx++) {
        tma::cluster::load_async(a_smem[input_ring][i], g.a,
                                 {(tile_coord.x*2+cta_rank)*C::NUM_CONSUMERS+i, idx
    },
                                 inputs_arrived[input_ring], (uint16_t)(1<<cta_rank)
    , 0);
        wait(inputs_arrived[input_ring], get_phasebit<0>(bitfield, input_ring));
        if (idx == 0) mm2_ABt (d_tt[task_iter%C::MMA_PIPE_DEPTH],
                               a_smem[input_ring][warpgroup::warpid()],
                               b_smem[input_ring], inputs_finished[input_ring]);
        else          mma2_ABt(d_tt[task_iter%C::MMA_PIPE_DEPTH],
                               a_smem[input_ring][warpgroup::warpid()],
                               b_smem[input_ring], inputs_finished[input_ring]);
        input_ring = ring_advance<C::LOAD_PIPE_DEPTH>(input_ring);
    }
    detail::tcgen05::commit<C::CLUSTER_SIZE>(outputs_arrived[warpgroup::warpid()]);
    if (!schedule.success) break;
}
```

Figure 15: ThunderKittens Blackwell BF16 GEMM. A 2-CTA cooperative MMA (`mm2_ABt`/`mma2_ABt`) accumulates into tensor memory (`d_tt`); TMA loads carry a cluster multicast mask.

A.6 Growth of hardware-specific code

Supporting a new GPU is dominated by architecture-specific code rather than algorithm. Figure 16 shows the per-architecture cost growing within one project, where NVIDIA's CUTLASS library has expanded $29\times$ across recent versions and architecture-specific instruction wrappers, descriptor builders, and pipeline templates now make up the majority of the codebase. The growth is not in cross-architecture algorithm code, since each new generation introduces a new tier of hardware concept (warpgroup, cluster, TMEM, AGPR, SRD) that requires its own wrapper layer and does not amortize against earlier generations.

Figure 17 shows the same cost across the open-source ecosystem. Manually porting projects like FlashAttention, FlashInfer, Triton, ThunderKittens, and PyTorch between GPU generations or between vendors has historically taken motivated expert teams months to years, and the trend does not flatten with experience because each new tier of hardware concept transfers only partially from prior porting work. Together the two figures bound the manual cost any day-zero porting approach must compete against.

B Frontier LLMs on the taxonomy

The unit tests double as a measurement instrument for hardware awareness in frontier LLMs, motivating the gap (Section 1), pinning model selection (Section 3), and setting up the per-baseline failure clusters (Section D.2).

B.1 Pass@1 across architectures

Per-architecture Pass@1 across the five frontier models is reported in Figure 2a. A cell counts as passed when the first sampled completion compiles, produces an output that matches the expert solution kernel for that cell, and reaches at least 99% of the expert's value on the cell's targeted profiling counter (thresholds from `config.json`, Section C.2, with the 1% slack absorbing measurement noise). The threshold means a passing kernel matches or exceeds expert hardware utilization rather than merely returning a correct numerical result. All five models trace the same Ampere $>$ Hopper

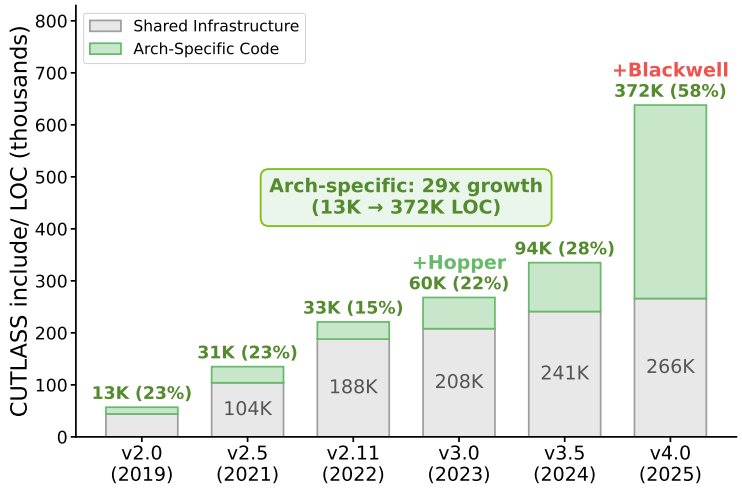


Figure 16: Architecture-specific code in NVIDIA's CUTLASS library [NVIDIA, 2025] has grown 29× across recent versions and now constitutes the majority of the codebase. The growth is in per-architecture instruction wrappers, descriptor builders, and pipeline templates rather than in cross-architecture algorithm code.

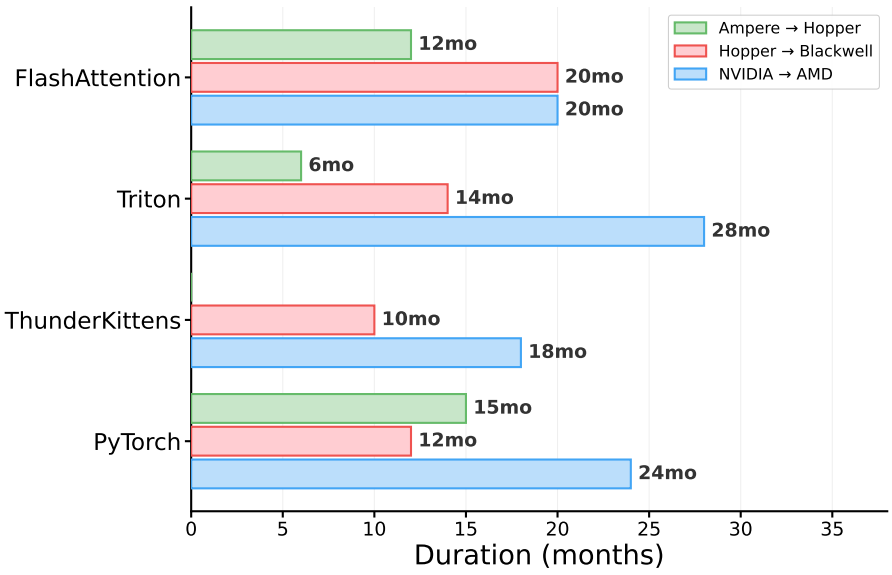


Figure 17: Open-source porting timelines computed from Git history. Manually porting projects like FlashAttention [Dao et al., 2022], FlashInfer [Ye et al., 2025], Triton [Tillet et al., 2019], ThunderKittens [Spector et al., 2024, Hu et al., 2025], and PyTorch [Paszke et al., 2019] between GPU generations or between vendors has historically taken months to years of expert effort. The trend does not flatten with experience because each new generation introduces a new tier of hardware concepts for which prior porting work transfers only partially.

$>$ Blackwell $\geq$ MI350 ordering, losing roughly a third of the Ampere score moving to Hopper and roughly half of the Hopper score moving to Blackwell or MI350, and restricting Pass@1 to cells satisfiable by an Ampere-style implementation collapses the gradient to within 5 percentage points across architectures.

B.2 Failure-mode buckets

Across failed unit tests, three buckets dominate, each tagged with the corresponding agent-loop cluster from Section D.2.

Bucket 1, invalid syntax for the right strategy (missing API ground truth, cluster 1). The model identifies the correct strategy but produces invalid syntax, with malformed `wgmma` descriptor immediates, wrong operand types where a `tmem` register handle is required, and missing `.kind` qualifiers on Blackwell `tcgen05` variants. This is the most common failure on Hopper and Blackwell, illustrated in Figure 18. The code looks plausible until a real GPU run.

```
// Strategy is right (warp-group MMA over a shared-memory descriptor),
// but the descriptor encoding is from an obsolete preview spec.
uint64_t desc = make_smem_desc(
    smem_ptr,
    /*lbo*/  64,    // <-- should be encoded as 64 >> 4 = 4
    /*sbo*/ 128);   //     model emits raw byte counts
// nvcc accepts the literal but the GPU traps at issue.
wgmma::mma_async<128, ...>(&desc, &acc[0]);
```

Figure 18: Bucket 1 example. Hopper `wgmma` descriptor with wrong stride field width.

Bucket 2, naive kernel with no targeted-metric improvement (missing algorithm structure, cluster 2). The model falls back to a functionally correct scalar kernel with hand-rolled FMA loops on cells that exist precisely to exercise tensor cores, so `sm__inst_executed_pipe_tensor_op_hmma` stays at zero and the cell fails. Bucket 2 dominates on Ampere and CDNA4 where the API surface is small enough that syntax is right but the algorithmic mapping is wrong.

Bucket 3, optimization intended for the wrong architecture (missing optimization pattern, cluster 3). The model reaches for an optimization pattern from its corpus that does not match the cell's architecture, emitting Hopper `wgmma` on a Blackwell test, Ampere `cp.async` on a Hopper test, or Ampere-style swizzle on an MI350 LDS layout. Bucket 3 is what the architecture-recency gradient predicts most strongly (Figure 19). The more recent the SM tier, the more often the model substitutes a same-name pattern from the previous tier.

```
// Blackwell cell. tcgen05.mma + tmem is required.
// Model outputs Hopper warp-group MMA instead.
asm volatile("wgmma.mma_async.sync.aligned.m64n128k16.f32.bf16.bf16 "
             "{%0,%1,...}, %16, %17, p, 1, 1, 1, 0;\n" ...);
// On sm_100a this either fails to encode or runs on the legacy
// tensor-core path. tmem stays empty and the cell metric stays at 0.
```

Figure 19: Bucket 3 example. Hopper-style `wgmma` emitted on a Blackwell `tensorcore_mma` cell that targets `tcgen05`.

The three buckets map cleanly to agent-loop clusters. Bucket 1 maps to Cluster 1 (missing API ground truth), patched by an API library that pins the encoding. Bucket 2 maps to Cluster 2 (missing algorithm structure), patched by a per-problem algorithmic recipe. Bucket 3 maps to Cluster 3 (missing optimization pattern), patched by per-arch optimization-pattern modules.

Model selection and GPT replication. A 30-turn agent loop on Hopper BF16 GEMM picks Gemini 3.1 Pro as the backbone for primary trajectories given its highest unguided ceiling and largest hardware-aware delta. We replicate the Linear-Attention-scaling sweep with GPT 5.4 across six baseline contexts and four architectures at 100 productive turns per run, and both models show the

same plateau ordering. On GPT, Hawkeye's full configuration reaches ≥ 0.66 of expert TFLOPS on Ampere, ≥ 0.86 on Hopper, and ≥ 0.91 on MI350. On Gemini, the same architectures reach ≥ 0.94, ≥ 0.99, and ≥ 0.91.

C Structure of the Hawkeye taxonomy and agent workspace

This appendix specifies what the agent reads when it consumes the HAWKEYE taxonomy, backing Section 2 via Sections C.2 and C.3.

C.1 Full taxonomy

The complete HAWKEYE taxonomy is reproduced in Table 4. Rows are recurring optimization strategies and columns are target GPU architectures (Ampere, Hopper, Blackwell, MI350). Each cell records the architecture-specific syntax, the bottleneck utilization counter the unit test verifies, and the end-to-end speedup the optimization buys over a naive baseline. Section C.2 expands what ships inside a single cell, and Section C.3 describes how the agent retrieves them at inference time.

Table 4: Hardware-aware optimization taxonomy. Each row is a general optimization strategy and each cell shows its architecture-specific details, paired with a unit test and expert-authored solution kernel. Every cell reports a bottleneck utilization metric (**M** = MMA, **D** = DRAM, **G** = global memory, **S** = shared memory, **W** = warp/wave efficiency; $0 \rightarrow \infty$ marks features absent in the naive kernel) and the resulting end-to-end speedup over a naive baseline.

	Ampere (A100)		Hopper (H100)		Blackwell (B200)		MI350	
MMA Unit	`mma.sync.m16n8k8` BF16→FP32, `ldmatrix.x2/.trans` loads		WGMMA `m64n64k16` BF16 + TMA 128B-swizzle loads + mbarrier		`tcgen05.mma` BF16→FP32, TMA bulk loads, TMEM accumulators		MFMA 32x32x16 BF16, AGPR accumulators	
	M ↑**8**×	**3.48**×	**M** ↑**2**×	**157.48**×	**M** ∞	**7.17**×	**M** ∞	**108**×
Quantized Precision	No floating-point quantization support		FP8 E4M3 WGMMA `m64n64k32` + TMA, per-row/col scaling		NVFP4 `tcgen05.mma mxf4nvf4` block-scale + 2-CTA cluster		On-the-fly MXFP4 quantization, MFMA matrix core	
	M ↑**2**×	**19.53**×			**M** ∞	**6.73**×	**M** ∞	**6.7**×
Shared Mem Layout	XOR swizzle family (`s32/s64/s256`) for conflict-free `ldmatrix`		128B column-group XOR swizzle matching WGMMA descriptors		Parameterized 64B `offset_swizzle` for `tcgen05` descriptors		64-bank LDS XOR swizzle for bank-conflict-free transpose	
	S ∞	**1.05**×	**S** ∞	**2.59**×	**S** ↑**1238.6**×	**1.62**×	**S** ↑**1.3**×	**1.5**×
Vectorized Memory	`uint4` 128-bit vectorized BF16 scale-and-add, grid-stride loop		`uint4` 128-bit vectorized BF16 scale-and-add, grid-stride loop		`uint4` 128-bit vectorized BF16 scale-and-add, grid-stride loop		128-bit global loads/stores via `buffer_load_dwordx4`	
	G ↑**8**×	**1.26**×	**G** ↑**8**×	**1.50**×	**G** ↑**4**×	**3.26**×	**G** ↑**16**×	**3.7**×
Async Pipeline	3-stage `cp.async` pipeline, `commit_group/wait_group`		TMA `cp.async.bulk.tensor` + mbarrier 3-stage pipeline		TMA 2D + mbarrier 3-stage pipeline, phase-parity tracking		Triple-buffered LDS pipeline overlapping loads with MFMA	
	D ↑**5.2**×	**5.22**×	**D** ↑**2.27**×	**2.21**×	**D** ↑**4.09**×	**3.89**×	**D** ↑**1.2**×	**5.1**×
Producer/Consumer	Warp-specialized WMMA, warp 0 loads A/B, warps 1–7 compute, double-buffered SMEM		WGMMA + TMA with warpgroup register reallocation		`tcgen05.mma` + TMA, producer/consumer warp specialization		Double-buffered 4-wave MFMA, load/compute overlap	
	D ↑**1.2**×	**5.22**×	**D** ∞	**823.28**×	**D** ∞	**834.81**×	**D** ↑**11.4**×	**3.4**×
Epilogue Pipeline	`mma.sync` BF16 GEMM, ReLU fused on FP32 accumulator registers		WGMMA tiled GEMM, fused GeLU epilogue in FP32 accumulators		`tcgen05` GEMM, fused GeLU epilogue from TMEM		Fused bias + GeLU + E8M0 scale + FP8 single pass	
	D ↑**7.7**×	**7.70**×	**D** ↑**46.2**×	**14.62**×	**D** ↑**7.5**×	**9.71**×	**M** ↑**7.9**×	**4.0**×
Warp/Wave Reduction	Hierarchical `block_reduce_sum` via `__shfl_down_sync` shuffles		`block_reduce_sum` via `__shfl_down_sync` + atomic finalize		`block_reduce_sum` via `__shfl_down_sync` + atomic finalize		64-lane `__shfl_xor` butterfly replacing global atomics	
	W ↑**64**×	**1.11**×	**W** ↑**64**×	**1.00**×	**W** ↑**64**×	**1.17**×	**W** ↑**4.0**×	**7.3**×
Multi-Unit Coordination	Cooperative `grid_sync()` two-phase reduction, L2 partial-sum reuse		2-CTA cluster, DSMEM (`mapa.shared::cluster` + `ld.shared`) exchange		2-CTA cluster, DSMEM (`mapa.shared::cluster`) cross-CTA exchange		XCD-aware `blockIdx` remapping with L2 swizzle	
	D ↑**8**×	**14.54**×	**D** ↑**2**×	**1.05**×	**D** ↑**2**×	**1.12**×	**D** ↑**2.8**×	**1.9**×
Persistent Scheduling	`mma.sync` PTX, `SWIZZLE_GROUP=4` L2-friendly `blockIdx` remap		Persistent WGMMA + TMA, `atomicAdd` tile dispatch + L2 swizzle		Persistent `tcgen05` + TMA, `atomicAdd` tile dispatch + L2 swizzle		Persistent MFMA wavefronts with `atomicAdd` tile dispatch	
	M ↑**2**×	**2.14**×	**M** ↑**2**×	**22.15**×	**M** ∞	**3.83**×	**M** ∞	**6.9**×

C.2 Anatomy of a taxonomy cell

A taxonomy cell sits at the intersection of one row (an optimization strategy) and one column (an architecture), and it ships four artifacts. A `naive_kernel.cu` or `naive_kernel.hip` establishes the baseline the optimization has to beat by intentionally avoiding the targeted hardware feature so its profiling counter rides at the floor. An `optimized_kernel.cu` is the expert-authored reference solution that the agent can read as a syntax example, invoke directly as a callable, or lift fragments from when composing into a larger kernel. A `config.json` pairs the cell to a profiling counter and expected gain, and a guide encodes the protocol the agent must follow. The Blackwell async-pipeline cell illustrates the shape (Figures 20 and 21). The optimized kernel is a 3-stage TMA ring with mbarrier multi-phase coordination paired to `dram__throughput.avg.pct_of_peak_sustained_elapsed`, and the guide encodes the phase-bit protocol with initial value `0xFFFF0000` (bits 31–16 producer phase, bits 15–0 consumer phase). Without per-module guides preloaded, the Blackwell agent caps performance at ∼240 TFLOPS and at one point hallucinates the non-existent intrinsic `__pack_half2` at turn 32 (Section D.8).

```
constexpr int NUM_STAGES = 3;

__device__ void mbarrier_arrive_expect_tx(int mbar, int tx_bytes) {
    asm volatile("mbarrier.arrive.expect_tx.release.cta.shared::cta.b64 "
                 "_, [%0], %1;" :: "r"(mbar), "r"(tx_bytes) : "memory");
}
__device__ void tma_load_2d(int dst, const void* tmap,
                            int x, int y, int mbar) {
    asm volatile("cp.async.bulk.tensor.2d.shared::cta.global"
                 ".mbarrier::complete_tx::bytes [%0], [%1, {%2, %3}], [%4];"
                 :: "r"(dst), "l"(tmap), "r"(x), "r"(y), "r"(mbar) : "memory");
}

// Prologue: prefetch first NUM_STAGES-1 tiles
for (int s = 0; s < min(NUM_STAGES - 1, num_blocks_k); s++) {
    if (threadIdx.x == 0) {
        tma_load_2d(smem_addr_s, tensorMapA, s*BK, m_block*BM, mbar_s);
        mbarrier_arrive_expect_tx(mbar_s, tile_bytes);
    }
}
```

Figure 20: Blackwell async-pipeline cell, `optimized_kernel.cu`. 3-stage TMA prologue with mbarrier-based completion.

```
{
  "test_name": "async_data",
  "task_description": "Asynchronous data transfer with TMA and 3-stage
                       pipelining on Blackwell (SM100)",
  "compilation": { "flags": "-arch=sm_100a -O3 -lcuda" },
  "profiling": {
    "metrics": [{
      "name": "dram__throughput.avg.pct_of_peak_sustained_elapsed",
      "description": "DRAM bandwidth utilization percentage",
      "direction": "higher_is_better"
    }]
  }
}
```

Figure 21: Blackwell async-pipeline `config.json`. The `dram__throughput` counter is what the harness reports back to the agent so it can verify the TMA ring actually saturated DRAM rather than some unrelated change accounting for the runtime improvement.

The CDNA4 column shares the four-artifact shape with extensions `.hip` and flags `-offload-arch=gfx950`, and it exposes the AGPR/VGPR split where accumulators live in a separate 256×32-bit register file requiring explicit `v_accvgpr_read_b32` and `v_accvgpr_write_b32` transfers. Its `config.json` pairs the cell to three counters at once, with `SQ_INSTS_VALU_MFMA_BF16` expected to rise and `SQ_INSTS_VALU` and `SQ_WAVES` expected to fall, so the agent can detect MFMA adoption versus a scalar-FMA fallback, and the guide commits to typed wrappers like `mfma::bf16_32x32x16` over raw builtins to avoid silent miscasts. All four artifacts are load bearing. The naive baseline is what makes the targeted counter readable, the optimized kernel is the only executable syntax example for instructions that postdate the model's training corpus, the config is what tells the agent which counter signals success, and the guide is what encodes the protocol the syntax alone does not. The harness always reports the counter alongside TFLOPS so a runtime improvement is verified to have come from exercising the targeted feature (full per-row mapping in Section A.4 and Table 4).

Constructing a cell. Building the four columns took about 200 expert-hours (Section 2.3), most of it in cell construction, which follows a fixed sequence. We first find a profiling counter that cleanly signals the targeted optimization fired, such as tensor-core instruction counts for the MMA row or

DRAM throughput for the async pipeline. We then size a unit test to isolate that one optimization, write a naive kernel that avoids the targeted feature so the counter sits at its floor, and hand-write the solution kernel. A cell is accepted only once its solution kernel passes correctness, improves the targeted counter, and beats the naive kernel end to end. Debugging consists of tightening the kernel or the test size until all three signals agree, and the end-to-end requirement ensures that every row improves performance on its own unit test (Table 4).

C.3 Workspace layout and retrieval

Each method runs in a clean workspace visible to the agent through the shell, with HAWKEYE exposing the architecture column of the taxonomy and the accumulated kernel pool from prior outer-loop runs (Figure 22). The generic and KernelBench workspaces drop `taxonomy/` and `kernel_pool/`, GEPA replaces them with a single `skill.md`, and weaker hardware-aware workspaces swap `taxonomy/` for `docs/` or a single `prior_arch_kernel.cu`.

```
workspace/
  task_prompt.md, workload_implementation_guide.md, spec.json
  eval.py, bf16_torch_baseline.py, bf16_expert_baseline.py
  naive_kernel.cu, optimized_kernel.cu      # agent edits this
  blackwell-apis/                           # local API library
    blackwell.cuh, general.md               # umbrella + arch overview
    mma/, tmem/, tma/, barrier/, cluster/   # each has <name>.cuh + guide.md
    epilogue/, pipeline/, clc/, tile_scheduler/, attention_ops/, nvfp4/
  taxonomy/                                 # column of unit tests
    01_tensorcore_mma/
      naive_kernel.cu, optimized_kernel.cu, config.json
    02_quantized_mma/  ...  10_persistent_scheduling/
  kernel_pool/                              # prior outer-loop kernels
    gemm_bf16.cu, attention_bf16.cu, ...
```

Figure 22: Workspace tree for the HAWKEYE method on Blackwell. The `taxonomy/` column ships only the architecture-specific cells the agent should consult, and the `kernel_pool/` accumulates correct kernels from prior outer-loop runs.

Across instrumented HAWKEYE trajectories, retrievals concentrate on `taxonomy/` and `apis/`, and no substitute carries comparable retrieval-frequency mass through to peak performance (Section D.2). Three Blackwell baselines show what fills the void instead. `ptx_abstractions` strips the per-module `guide.md` files, so the agent `cat`s the headers and re-derives missing protocols inside the kernel (Section D.8). `raw_ptx_docs` replaces `<arch>-apis/` with the full 1.22 M-token PTX ISA reference, and the agent issues 17 distinct `wgmma`-pattern searches across 221 turns. `vendor_fmha` replaces both directories with the CUTLASS or Composable Kernel source tree, retrievals collapse into filesystem navigation, and the kernel body remains an empty stub for 207 turns.

D Comparative analysis of kernel optimization systems and context structures

This appendix positions HAWKEYE comparatively at two levels: a landscape view of recent autonomous GPU kernel optimization systems (Section D.1), and a controlled comparison against six alternative context structures evaluated under a common agent harness and turn budget (Sections D.2–D.8). For the controlled comparison, we build a master baseline matrix across two backbone LLMs and four architectures, frame a three-cluster failure-mode taxonomy, walk each baseline through its plateau pathology, and ablate the optimization-pattern modules to isolate their contribution.

D.1 Related autonomous kernel optimization systems

Table 5 compares HAWKEYE to recent autonomous GPU kernel optimization systems across implementation language, optimization method, hardware-specific support, required domain expertise, target GPU architectures, and supported precisions.

System	Lang.	Method	HW-Specific Opt.	Domain Expertise	GPU Archs.	Precisions
KernelBench	CUDA	Beam search	✗	Basic HW specs	A	FP32
KevinRL	CUDA	RL self-feedback	✗	✗	H	FP32
AI CUDA Eng.	CUDA	Feedback loop	✗	✗	A	BF16, FP32
KernelEvolve	CUDA	Tree search + RAG	✗	RAG on docs	A H M	FP32
KernelFoundry [Wiedemann et al., 2026]	SYCL, CUDA	Quality-diversity search	Some	Templated kernels	A	FP32
KernelBlaster	CUDA	In-context RL	✗	✗	H	FP16, FP32
CudaAgent	CUDA	Planning + RL	✗	✗	H	FP32
CudaForge	CUDA	Feedback loop	✗	✗	A	FP32
SwizzlePerf	Triton	Feedback loop	1 opt type	Hand-written docs	M	BF16, FP32
K-Search	Triton, CUDA	World model search	Some	Past examples	H B	BF16, FP8
HF CUDA Skills	CUDA	Coding agent	Hand-written	Hand-written skills	A H	BF16, FP16, FP32
HAWKEYE	**CUDA, HIP**	**Coding agent with HW-awareness**	✓	**Taxonomy + unit tests**	**A H B M**	**BF16, FP8, NVFP4, MXFP4**

Table 5: Comparison of autonomous GPU kernel optimization systems. Architecture support: **A** = Ampere, **H** = Hopper, **B** = Blackwell, **M** = MI350. Prior systems either lack hardware-specific optimizations entirely (✗), achieve them partially through hand-written or outdated approaches, or require high expert effort that does not scale. Most prior systems also stay within standard precisions (FP32, BF16, FP16). HAWKEYE is the first framework to evaluate hardware-aware kernels across four frontier GPU architectures and four precisions including FP8, NVFP4, and MXFP4.

D.2 Three-cluster failure-mode framework

The baselines span the practitioner spectrum from converged skill-evolution memory to the full vendor ISA reference. We evaluate all six on causal Parallel ReBased Linear Attention (henceforth *Linear Attention* for brevity) across four architectures (Ampere A100, Hopper H100, Blackwell B200, CDNA4 MI350) with two backbones (Gemini 3.1 Pro and GPT-5.4) for $6 \times 4 \times 2 = 48$ runs at a fixed 100-productive-turn budget, and Table 6 reports the best TFLOPS for every (baseline, backbone, architecture) cell together with the total trajectory token cost. Per-baseline failure-cluster assignments are summarised in the prose below.

Table 6: Master baseline matrix. "Best TFLOPS" is the highest correct kernel observed in the trajectory at the 100-productive-turn budget. "–" marks runs where no correct kernel was produced (status `NO_CORRECT_KERNEL` or `best_tflops = null`). Tokens are cumulative trajectory tokens (millions). Expert (peak) reference for Parallel ReBased Linear Attention BF16: Ampere ≈132, Hopper ≈298–300, Blackwell ≈424, CDNA4 ≈333.

	Best TFLOPS (Gemini / GPT)				Tokens (M)	
Baseline	**Ampere**	**Hopper**	**Blackwell**	**CDNA4**	**Gem**	**GPT**
Triton FLA	27.2 / 4.4	17.9 / 16.2	110.4 / 34.6	– / 0.3	8.0	7.0
`ptx_abstractions`	84.7 / 10.1	240.9 / 17.5	– / 9.7	241.8 / 11.2	13.6	6.0
Raw PTX docs	17.8 / 19.6	31.6 / 9.2	5.6 / 19.8	169.7 / 16.0	10.4	8.2
Augmented PTX docs	39.7 / 10.0	4.4 / 27.2	5.6 / 26.2	118.4 / 20.0	9.2	7.0
`inlined_gemm`	3.6 / 8.7	4.4 / 16.6	– / 15.0	27.7 / 2.1	37.4	13.9
Vendor FMHA	33.8 / 18.0	29.7 / 37.6	5.7 / 18.9	39.2 / 0.8	16.6	7.3
HAWKEYE full	**87.3 / 135.8**	**256.4 / 327.9**	**702.0 / 418.9**	**326.5 / 319.2**	**25.0**	**12.1**

Every non-HAWKEYE baseline collapses on at least one architecture under at least one backbone (collapse defined as best TFLOPS at most $0.1\times$ expert), and each collapse is tied to a specific deficit in the context. The patterns partition into three structural clusters matching the missing-context types in Section 2.

Cluster 1. Missing algorithm structure. The agent lacks the high-level loop, pipeline, or composition pattern the architecture demands, so the kernel compiles and is correct but has no architecture-aware structure (a scalar $O(S^2)$ loop on a tensor-core GPU, a single-warpgroup where the chip needs a producer-consumer split, or an empty stub). Triton FLA and `inlined_gemm` land here, with Triton FLA Blackwell collapsing to 110.4 / 34.6 TFLOPS against the ∼424 expert and `inlined_gemm` Hopper to 4.4 / 16.6 against ∼298.

Cluster 2. Missing API ground truth. The agent knows the optimization exists but cannot construct a correct invocation, so descriptor immediates, barrier byte counts, encodings, and swizzle modes are subtly wrong and the kernel compiles to silent garbage or fails on a hallucinated intrinsic. Raw PTX docs is the canonical case. The Hopper turn-201 "Figure 149" hallucination (Figure 24) is the cleanest example, where the agent infers a WGMMA accumulator layout from text alone and produces 1.92% correctness at 77.9 TFLOPS. `ptx_abstractions` on Blackwell exhibits the brittle compile-floor variant where the agent invents `__pack_half2` at turn 32 (Figure 23), retries variants for 167 turns because the `epilogue/ guide.md` with the correct `stmatrix::store_b16_x4` signature is not preloaded, and terminates at `best_tflops = NULL`.

```
uint32_t bf16_vals = __pack_half2(__float2bfloat16(val_x),
                                  __float2bfloat16(val_y));
asm volatile(
    "stmatrix.sync.aligned.m8n8.x4.shared.b16 [%0], %1;\n"
    :: "r"(smem_addr), "r"(bf16_vals));
// error: identifier "__pack_half2" is undefined
```

Figure 23: `ptx_abstractions` on Blackwell at turn 32. The agent hallucinates a non-existent CUDA intrinsic name and never recovers across 199 turns.

```
T201 decision: "The text mentions Figure 149 for the D accumulator
layout, but I can't see the image. However, I know from standard
WGMMA layouts that for m64n64k16 with f32, the D accumulator is
32 registers per thread, laid out as [row_tile, col_tile][i]."

T202 result: 77.9 TFLOPS reported, correctness = 1.92%
            (98% of output elements violate tolerance)
```

Figure 24: Raw PTX docs on Hopper at turn 201 (the Figure 149 hallucination). The agent infers the WGMMA accumulator layout from text alone, producing 1.92% correctness.

Cluster 3. Missing optimization patterns. The agent has correct primitive APIs and the right algorithm but does not know which combinations compose into the throughput-critical pattern, so the kernel is correct but capped and either contains an explicit comment acknowledging the missing optimization or a hand-rolled re-derivation. `ptx_abstractions` on Hopper and CDNA4, augmented PTX docs, and vendor FMHA all land here. The Hopper smoking gun is the literal "`// Single warpgroup for now`" in `ptx_abstractions` at turn 150 (Section D.8), and the CDNA4 smoking gun is the LDS XOR swizzle re-derived at six call sites.

Cross-model robustness. The two backbones disagree quantitatively but agree structurally, with cluster assignments stable across both models. Every baseline that collapses under Gemini also collapses under GPT in the same mode, whether compile floor, hallucinated intrinsic, or plateaued correct kernel, and HAWKEYE is the only context where both LLMs scale to expert-relative parity on every architecture.

Token-cost framing. Across the Linear-Attention sweep, baselines spend 120k–2216k trajectory tokens per TFLOPS of best result while HAWKEYE spends 24–50k, with the token band structurally bounded on both sides. Below ∼5k turn-0 tokens (skill evolution, `ptx_abstractions`, raw PTX docs) the agent has primitives but no organising principle and clusters 2 and 3 dominate. Above ∼25k turn-0 tokens (`inlined_gemm` on Hopper at 32k and Blackwell at 35k) the exemplar's algorithmic shape is hard-coded into the syntax and cluster 1 dominates. Above ∼100k workspace tokens (raw PTX docs at 1.22M), doc-grep dominates and cluster 2 returns. HAWKEYE's 11.8k–20.1k turn-0 band sits in the trough.

D.3 Triton FLA

The `triton` baseline gives the agent a Triton expert reference for causal Parallel ReBased Linear Attention plus the standard Triton DSL, and the agent must emit CUDA or HIP. This is the cleanest

missing-algorithm-structure probe, since the algorithm is fully described but every architecture-specific primitive is hidden behind `tl.dot` and the lowering tables.

The Blackwell run lands on `nvcuda::wmma` at turn 16 and iterates that same kernel for 220 turns, peaking at 110.4 TFLOPS at turn 149 ($0.26\times$ the 424.7 TFLOPS expert) on the WMMA fragment API that the Triton lowering compiles to on every NVIDIA arch (Figure 25). `tl.dot` would expand on Blackwell to `tcgen05.mma.cta_group::1.kind::f16` with TMEM, CLC, and multicast TMA, but Triton hides those and the workspace has no syntactic anchor, so across 237 turns and 9.04 M tokens the kernel never advances past WMMA.

```
wmma::fragment<wmma::matrix_a, 16, 16, 16, bf16, wmma::row_major> q_frag;
wmma::fragment<wmma::matrix_b, 16, 16, 16, bf16, wmma::col_major> k_frag;
wmma::fragment<wmma::accumulator, 16, 16, 16, float> s_frag;

wmma::load_matrix_sync(q_frag, &sq[...], HEAD_DIM);
wmma::load_matrix_sync(k_frag, &sk[...], HEAD_DIM);
wmma::mma_sync(s_frag, q_frag, k_frag, s_frag);
```

Figure 25: Triton FLA on Blackwell at turn 149. The agent uses the `nvcuda::wmma` fragment API on every NVIDIA arch because that is the only tensor-core path the Triton reference exposes, leaving `tcgen05.mma`, TMEM, and CLC unused.

Hopper reproduces the same WMMA fallback (peaks 17.9, 29.2, 106.3 TFLOPS vs 298 expert), and the agent recognises the ceiling at turn 31 ("WMMA is slow") but its 44-turn attempt to port to `mma.sync` with `ldmatrix` regresses to 1.84% correctness (Figure 26) before reverting to the 17.9 TFLOPS WMMA kernel for the remaining 150 turns. Ampere is the muted case because its tensor-core path *is* `nvcuda::wmma` (27.2 and 36.3 TFLOPS vs 132 expert), which shows the DSL's implicit fallback target rather than DSL competence on Ampere. CDNA4 never passes the correctness gate ($\sim$72% within tolerance vs the required 1.0), the running log at turns 190–193 reads "Still failing with `opaque_abort`" with no further hypothesis, and the MFMA intrinsic `__builtin_amdgcn_mfma_f32_16x16x16bf16_1k` appears nowhere in the Triton reference for the agent to copy.

```
T49: "The mma.sync.aligned.m16n8k16.row.col.f32.bf16.bf16.f32
      instruction compiles successfully. I will use this to replace wmma."

T93: 21.6 TFLOPS reported, correctness = 3.57%
     (the kernel compiles but ldmatrix lane-mapping is wrong)

T107--T131: sustained 21--22 TFLOPS at constant correctness ~1.84%

T232: revert to T30 (WMMA kernel at 17.9 TFLOPS)
```

Figure 26: Triton FLA on Hopper, turns 49–232. The agent identifies that WMMA is the ceiling and tries to hand-roll `mma.sync` with `ldmatrix`, but cannot derive correct lane-to-element addressing without a worked example and reverts to WMMA after 44 turns of incorrect kernels.

GPT-5.4 lands in the same WMMA fallback at lower absolute peaks (Blackwell 34.6, Hopper 16.2, Ampere 4.4, CDNA4 0.2 TFLOPS), confirming the failure is structural rather than a Gemini idiosyncrasy. The agent receives a complete-looking kernel but cannot see which primitives the DSL silently drops on each architecture, and HAWKEYE surfaces those primitives as named cells with composition order baked in, recovering $5.0\times$ on Blackwell against the same agent and model (Table 6).

D.4 Raw PTX and ROCm docs

The `raw_ptx_docs` baseline gives the agent the full PTX ISA (1.22 M tokens) or ROCm ISA ($\sim$0.31 M tokens) on disk, accessed by a productive-turn-counted `read_context` tool. The bottleneck is navigation within budget, not corpus content.

Hopper peaks at 31.6 TFLOPS at turn 82 ($0.11\times$ expert) by re-issuing the same WGMMA-pattern grep across 7 distinct turns (T7, 9, 11, 53, 90, 92, 95), each costing a productive turn while growing trajectory tokens without growing kernel state. The agent retrieves the right symbol `cp.async.bulk.tensor`, the right operand (`tensormap`), and the right reason (swizzle), then declines to construct one and falls back to Ampere-style `cp.async.cg`. Another run reaches 88.6 TFLOPS at turn 118 ($0.30\times$) by adding a WGMMA-shaped block tile but never wires up TMA.

Blackwell is the same pathology made worse, with peaks at 5.6 TFLOPS (turn 4) and 12.0 TFLOPS (turn 84) versus the 424 expert ($0.013\times$ and $0.028\times$). At turn 23 the agent reasons that `tcgen05` is too complex and commits to a WGMMA-first plan it never abandons (Figure 27), then re-issues the same WGMMA-pattern grep across 17 distinct turns (Figure 28), and by turn 124 reverts to the naive kernel (`best_turn = 4`). The string-match bake-off favors the larger WGMMA, `cp.async`, and WMMA chapter cluster over the smaller `tcgen05` addendum. The corpus is correct and the navigation policy is wrong.

```
T17: "I will read the PTX ISA for wgmma first, because it's simpler and
      I'm more familiar with it. WGMMA is still usable on Blackwell."

T23: "The tcgen05.mma instruction is quite complex. It requires TMEM
      allocation, descriptors, and mbarrier synchronization. Since I have
      100 turns, I can try WGMMA first and switch to tcgen05 if needed."

T64: "The kernel is now correct! However the performance is very poor
      (0.9 TFLOPS, 0.00x expert). This is because the kernel is doing a
      lot of scalar operations inside the warp-group loop."

T124: "The current kernel is correct but very slow (0.9 TFLOPS vs 424 for
       the expert)."  -- agent does not pivot to tcgen05
```

Figure 27: Raw PTX docs on Blackwell. The agent reasons that `tcgen05` is too complex, commits to WGMMA from prior knowledge, and across 221 turns attributes the resulting 0.9 TFLOPS to "scalar operations" rather than the wrong instruction family.

```
WGMMA-pattern read_context() calls on the PTX ISA, Blackwell:
T17, T19, T21, T29, T33, T35, T37, T39, T64, T66, T68, T74, T83, T85,
T87, T127, T129  --  17 distinct productive turns

A representative grep:
  read_context("blackwell/full_docs/ptx_isa.md", pattern=r"^### .*wgmma")
  -> 0 hits  (wgmma is buried in section 9.7.15.5.2, not a heading)

T19: "The pattern '^### .*wgmma' didn't match anything. Let's try a
      simpler pattern. Let's search for wgmma.mma_async."
```

Figure 28: Raw PTX docs on Blackwell, the WGMMA grep loop. The PTX ISA's structure is not indexed for the symbols the agent searches, so 17 productive turns are spent re-issuing pattern variants and never reach `tcgen05`.

CDNA4 is the inverted case and the strongest `raw_ptx_docs` result (the workspace name is kept for consistency; on CDNA4 the corpus is the ROCm ISA rather than PTX), peaking at 169.7 TFLOPS at turn 240 ($0.51\times$ expert) because navigation cost is monotone in corpus size. The CDNA4 ISA is $\sim$0.31 M tokens vs PTX's 1.22 M, the MFMA section is one contiguous block, and the agent finds `__builtin_amdgcn_mfma_f32_16x16x16bf16_1k` at turn $\sim$170. Even on the small corpus, navigation still bites in the long tail, with the same run missing `global_load_lds` (in the memory-instruction chapter while the agent anchors in the MFMA chapter) and falsely reporting at turn 197 that the symbol does not exist. GPT-5.4 reproduces the same plateau at lower peaks (Hopper 9.2, Blackwell 19.8, Ampere 19.6, CDNA4 16.0 TFLOPS), with the CDNA4-easiest-to-navigate ranking surviving the model swap, so the bottleneck is navigation rather than content. HAWKEYE closes this by replacing free-form retrieval with named cells the agent addresses by recipe, recovering the largest

single-cell gap in the matrix on Blackwell (46× behind `raw_ptx_docs`, with 12.36 M post-peak tokens spent for zero improvement, Table 11).

D.5 Augmented documentation

The `augmented_ptx_docs` baseline gives the agent curated, hyperlinked PTX or HIP ISA with TL;DR blocks and call-sequence templates per primitive. Curation amortizes navigation but does not provide the cross-primitive recipe, and the result is the most expensive baseline (13.6 M Gemini Hopper tokens, 13.0–13.4 M on Blackwell, 17.7 M on CDNA4) and one of the worst-performing (4.4 / 5.6 / 118.4 Gemini TFLOPS). The failure mode is composition, not navigation or syntax.

The Hopper Gemini run (100 productive turns, 13.97 M tokens, peak 4.4 TFLOPS at turn 4 vs 298 expert) is the cleanest case. The agent identifies WGMMA, TMA, and `mbarrier` primitives correctly through turn 16 and confirms the right WGMMA shape at turn 39, but cannot compose them. The augmented `tensor_cores_and_warpgroup.md` frames the WGMMA call sequence as the canonical path, so the agent commits to that sequence and reverts to the naive kernel after every failed attempt rather than degrading to a simpler shared-memory + WMMA path (Figure 29). The missing piece is the cross-primitive recipe pairing `mbarrier.arrive.expect_tx` with `cp.async.bulk.tensor.....mbarrier::complete_tx::bytes` on the same barrier with phase-bit flips, which the augmented docs treat as separate topics, and the run terminates at `best_turn = 4`.

```
T78:  attempt wgmma -> correctness fail (harness_abort) -> revert
T85:  attempt wgmma -> correctness fail (harness_abort) -> revert
T95:  attempt wgmma -> correctness fail (harness_abort) -> revert
T111: attempt wgmma -> correctness fail (harness_abort) -> revert
...
T199: best_turn still 4 (the naive kernel), 13.97M tokens spent
```

Figure 29: Augmented PTX docs on Hopper, the canonical-sequence over-commitment trap. The augmented `wgmma` doc presents one "canonical" call sequence with no fallback path, so each failed attempt reverts to the naive kernel rather than degrading to a simpler tensor-core path, and the 4.4 TFLOPS naive scalar survives 195 turns.

Blackwell is the same shape with a tcgen05 ceiling. Both runs terminate at `best_turn = 3-4` (peak 5.6 TFLOPS, the naive kernel). The agent writes `tcgen05.mma`, TMEM alloc/dealloc, and cluster TMA boilerplate but cannot decide `cta_group::1` vs `::2`, TMEM column allocation 128 vs 256 vs 512, or phase-bit tracking style. The augmented doc gives the API names but not the descriptor encoding tables at turn-0 inlined level, so each compile fails on argument-mismatch errors the agent cannot resolve from the available context (Figure 30), and every `tcgen05.fence` placement creates a race that reverts the kernel.

```
T163 compile error:
  ptxas /tmp/...: error: Arguments mismatch for instruction 'tcgen05.mma'

The augmented blackwell/tensor_cores_and_tmem.md TL;DR lists the API
names (tcgen05.alloc, tcgen05.mma, tcgen05.commit, tcgen05.ld,
tcgen05.dealloc) but does not inline the descriptor encoding:
  - bits 0--2 : op type
  - bits 3--5 : shape selector
  - bits 6--8 : data type
  - bits 9--15: trans / smulo / block-scaled flags
The agent must read_context() into the doc for these tables, and each
attempt rebuilds the bit layout from scratch.
```

Figure 30: Augmented PTX docs on Blackwell at turn 163. The agent has the `tcgen05` API names from the TL;DR but reconstructs the instruction descriptor bit layout by trial and error, producing repeated `ptxas` argument-mismatch errors and reverting to the naive kernel.

CDNA4 (118.4 TFLOPS at turn 175, 17.7 M tokens, 354 raw turns, the only matrix run to hit `MaxIterationsReached`) is the most diagnostic case because the augmented HIP doc describes `v_mfma_f32_16x16x16bf16_1k` and `v_mfma_f32_32x32x8bf16_1k` side by side without committing to one. The agent peaks at turn 175 with 16×16 then wastes 179 turns (∼9.8 M tokens) re-attempting the 32×32 shape. GPT-5.4 lands on shared-memory tiled kernels without tensor cores at all (Hopper 27.2, Blackwell 26.2, CDNA4 20.0 TFLOPS), the same cluster expressed at a different floor. Across all three architectures the bottleneck is composition rather than navigation, since each excerpt teaches one primitive correctly but none describes the producer-consumer recipe pairing TMA into an mbarrier programmed by `arrive.expect_tx`, decremented by the TMA via `complete_tx::bytes`, and gated by a phase bit that flips per ring slot. That recipe is what HAWKEYE's `pipeline/` module encodes (Section D.8), and `augmented_ptx_docs` spends $+14\%$ of `raw_ptx_docs`'s Hopper tokens for one-seventh the throughput (Table 6).

D.6 Inlined GEMM exemplar

The `inlined_gemm` baseline preloads a hand-written GEMM as one monolithic file (3380 Blackwell lines, ∼2500 on Hopper and CDNA4, 189 on Ampere). The exemplar's density couples algorithmic and hardware concerns tightly enough that the agent treats it as the algorithm and cannot decouple them. This is a missing-algorithm-structure probe.

The Blackwell exemplar is a 470-line 2-CTA cluster BF16 GEMM with the configuration in Figure 31.

```
static constexpr int BM = 256;            // per cluster (128 per CTA)
static constexpr int BN = 256;
static constexpr int BK = 64;
static constexpr int SUPERGROUP_SIZE = 4;

static constexpr int CLUSTER_SIZE    = 2;
static constexpr int NUM_CONSUMERS   = 2;  // non-overlap mode
static constexpr int NUM_PRODUCERS   = 1;
static constexpr int NUM_WARPGROUPS  = NUM_CONSUMERS + NUM_PRODUCERS;  // 3
static constexpr int LOAD_PIPE_DEPTH = 4;
static constexpr int EPI_PIPE_DEPTH  = 8;  // 8 chunks of 32 cols
static constexpr int CTA_M    = BM / CLUSTER_SIZE;   // 128
static constexpr int A_ROWS = CTA_M;                  // 128
static constexpr int B_ROWS = BN / CLUSTER_SIZE;      // 128
static constexpr int K_CHUNKS = BK / 16;              // 4
```

Figure 31: Blackwell BF16 GEMM exemplar configuration.

This configuration is correct for square BF16 GEMM but incorrect for causal Linear Attention, where the tile is S-by-D, D=64 is fixed, and the inter-chunk linear-state recurrence has no GEMM analog. Gemini Blackwell (100 productive turns, 17.27 M tokens, peak 5.8 TFLOPS at turn 173) is the exemplar trap. The agent copies the GEMM TMEM/fence/dealloc and `mbarrier.expect_tx` scaffolding verbatim, crops BM/BN/BK while keeping the GEMM's stride immediates, and falls back to a scalar inner loop for the recurrence. The resulting kernel compiles and is correct (every running line comes from the naive reference), but TMEM is allocated and immediately deallocated, the compute path bypasses it, and the producer/consumer scaffolding adds cost without throughput. Other runs yield `best_tflops = null`.

Hopper exposes the same coupling at the helper-function level. The exemplar ships a `wgmma64` helper with a hardcoded 1024-byte stride for descriptor B, correct for the 64-column GEMM tile but wrong for causal Linear Attention with HEAD_DIM=64 and 128-row K tiles. The agent identifies the mismatch at turn 28 (Figure 32) but cannot edit the inlined helper without rewriting all 2500 lines, and from turn 29 to 208 either the wrong stride silently produces garbage or a from-scratch WGMMA wrapper introduces fresh PTX errors. The run plateaus at `best_turn = 4`, 4.4 TFLOPS, 24.18 M tokens.

The exemplar entangles algorithmic shape, tile geometry, descriptor strides, warpgroup split, and persistent CLC scheduling, and porting to Linear Attention requires moving four of those five in coordination without an abstraction boundary that lets the agent move one lever at a time. GPT-5.4

```
// inlined_gemm helper (immutable, 2500-line monolithic file):
uint64_t desc_a = make_smem_desc(&sA[0], 16, 1024, true);  // <-- 1024
wgmma64<1, 1, 1, 0, 1>(s_acc, sQ + k_it * 16, sK + k_it * 16);

// T28 agent reasoning:
//   The stride of sQ is HEAD_DIM elements = 64 elements = 128 bytes.
//   But make_smem_desc uses 1024 as the stride multiplier.
//   1024 bytes = 512 elements. wgmma64 hardcodes 1024.
//   I need to change the stride in wgmma64 or write my own wrapper.
```

Figure 32: Inlined GEMM on Hopper at turn 28. The `wgmma64` helper hardcodes a 1024-byte descriptor stride for the GEMM tile, but causal Parallel ReBased Linear Attention needs 128 bytes. The helper is inlined and immutable, and 180 turns of from-scratch wrapper attempts produce only fresh PTX errors.

refuses the WGMMA and tcgen05 paths entirely, applies shared-memory tiling to the naive kernel and stops (Blackwell 15.0, Hopper 16.6, CDNA4 2.1 TFLOPS, Table 6). This is the dual of the Triton FLA failure, with hardware paths present but no algorithmic decoupling.

D.7 Vendor FMHA

The `vendor_fmha` baseline gives the agent CUTLASS Blackwell and Hopper FMHA (∼854 KB) and Composable Kernel `ck_tile/01_fmha` (∼287 KB) on the workspace. Both are aggressively abstracted with CuTe layouts, collective builders, and warp-specialized templates that present optimization patterns as objects of composition rather than as exemplars of primitives, so the agent loses turns to navigation. This is a missing-optimization-pattern probe.

The Blackwell run is canonical (Gemini, 328 raw turns, 100 productive, 13.10 M tokens, peak 5.7 TFLOPS at turn 121). The agent identifies the right customization point at turn 11 (replace `softmax_step` with the Taylor feature map), spends turns 25–66 discovering ground-truth violations (CUTLASS not on include path, `blackwell-apis` missing), then reverts both kernel bodies to literal empty stubs at turn 64 (Figure 33) and freezes at 5.7 TFLOPS for the remaining 207 turns. Another run fights 408 raw turns and peaks at 24.1 TFLOPS ($0.043\times$).

```
__global__ void naive_rebased_attention_kernel(
        const bf16* Q, const bf16* K, const bf16* V,
        bf16* O, int S) {
}

void optimized_rebased_attention(
        const bf16* Q, const bf16* K, const bf16* V,
        bf16* O, int B, int H, int S, int D) {
}
```

Figure 33: Vendor FMHA on Blackwell at turn 64. After failing to extract `tcgen05.mma` patterns from CUTLASS source, the agent reverts both kernel bodies to literal empty stubs and the run freezes at the 5.7 TFLOPS naive baseline for 207 more turns.

CDNA4 reduces `vendor_fmha` to `raw_ptx_docs` after 27 turns of CK navigation. The agent grep-cycles for the CK kernel entry point through turns 7–22, then at turn 27 searches CK for "MFMA" and gets zero hits because the calls are hidden inside `ck_tile`, and at turn 28 abandons CK and hand-rolls raw HIP MFMA. Peak is 39.2 TFLOPS ($0.118\times$) at turn 237 from `__builtin_amdgcn_mfma_f32_16x16x16bf16_1k` without CK pipelining. The Hopper CUTLASS run (309 raw / 100 productive turns, 13.77 M tokens, peak 29.7 TFLOPS at turn 260, $0.10\times$) is the only `vendor_fmha` run with a non-trivial kernel. The agent abandons the include-path attempt at turn 75 and hand-rolls a chunked $O(S^2)$ kernel, but lacks the inter-chunk linear-state accumulator that distinguishes causal Linear Attention from FMHA.

The structural symptom across architectures is that every multi-turn reasoning block opens with a navigation verb rather than a kernel edit (Figure 34).

```
T2:   "**Examining the Kernel Code**\n\nOkay, so I need to get a clear
       picture of what's going on with these kernels..."
T41:  "**Analyzing the Implementation for Causal Linear Attention on
       Blackwell**\n\nOkay, so we need to get a hardware-aware kernel..."
T78:  "**My Approach to Implementing a Hardware-Aware Kernel for Causal
       Linear Attention**\n\nOkay, so the challenge is clear..."
T117: "Here's my analysis of the provided files, keeping in mind my
       expert perspective:\n\n**Kernel Optimization Comparison**..."
T229: "**Navigating the PTX Maze for Blackwell Kernel Optimization**
       \n\nOkay, here's the situation. I'm deep in the weeds..."
```

Figure 34: Reasoning openers from `vendor_fmha/blackwell/conversation.jsonl`, turns 2, 41, 78, 117, and 229. Every multi-turn reasoning block opens with a navigation verb. The action the reasoning produces is another `read_context` call, not a kernel edit.

The action each turn produces is another `read_context` call against vendor source rather than a kernel edit. Production GPU libraries are written to be reused, not read, and CUTLASS's `collective::Builder` and CK's `ck_tile::FmhaFwdKernel` hide primitive recipes under reusable layers that are inverted for an LLM that needs to extract those recipes. GPT-5.4 follows the same shape (Hopper 37.6, Blackwell 18.9, CDNA4 0.8 TFLOPS, Table 6), and HAWKEYE's `pipeline/` and `epilogue/` modules expose the same primitives as named single-purpose recipes that the agent does not have to navigate to.

D.8 PTX abstractions ablation

We start from the full HAWKEYE taxonomy and surgically remove the optimization-pattern modules while keeping the primitive-instruction wrappers, so the agent retains thin C++ wrappers around tensor-core, memory, and synchronization intrinsics but loses every multi-instruction composition pattern. Removed modules encode ring-buffered async pipelines, producer-consumer warpgroup splits, decoupled epilogues, persistent tile schedulers, and composed attention helpers. Ampere and Hopper lose only math helpers (`swizzle`), Blackwell loses `epilogue`, `pipeline`, `clc`, `tile_scheduler`, `attention_ops`, `swizzle`, `nvfp4`, and CDNA4 loses `memory`, `gemm`, `attention`, `dpp`, `fp4`, `conv`. The prediction is a small gap on Ampere and Hopper and a large gap on Blackwell and CDNA4, which Tables 7 and 8 confirm.

Table 7: `ptx_abstractions` versus full HAWKEYE on causal Parallel ReBased Linear Attention, Gemini 3.1 Pro backbone, 100-productive-turn budget.

Arch	ptx_abstractions	HAWKEYE full	Gap	Best turn (ptx)
Ampere	84.7 TFLOPS	87.3 TFLOPS	+3%	205
Hopper	240.9 TFLOPS	256.4 TFLOPS	+6%	150
Blackwell	null (compile-fail)	702.0 TFLOPS	catastrophic	—
CDNA4	241.8 TFLOPS	326.5 TFLOPS	+35%	256

Table 8: `ptx_abstractions` versus full HAWKEYE under the GPT-5.4 backbone. The catastrophic Blackwell collapse is reproduced under GPT (9.7 TFLOPS, 44× behind Hawkeye), and a Hopper collapse appears (17.5 vs 327.9) that is more severe than under Gemini.

Arch	ptx_abstractions	HAWKEYE full	Gap	Best turn (ptx)
Ampere	10.1 TFLOPS	135.8 TFLOPS	13.4×	104
Hopper	17.5 TFLOPS	327.9 TFLOPS	18.8×	124
Blackwell	9.7 TFLOPS	418.9 TFLOPS	43.4×	171
CDNA4	11.2 TFLOPS	319.2 TFLOPS	28.6×	114

Under Gemini, `ptx_abstractions` on Ampere and Hopper recovers most of HAWKEYE's performance with a 6–35% residual reconstruction gap, but under GPT the gap explodes by an order of magnitude on every architecture, so the optimization-pattern modules act as an LLM-capability multiplier rather than a fixed bonus.

Gemini `ptx_abstractions` on Hopper reaches 240.9 TFLOPS at turn 150 with a single-warpgroup body that contains the literal admission "`// Single warpgroup for now`" (Figure 35). The agent recognizes that the producer-consumer split is the next move but cannot synthesize the ring-buffer phase-bit protocol from `barrier/` within budget. The kernel is correct but single-warpgroup, single-stage, and serializes both WGMMAs against the same TMA chain, plateauing at ∼241 TFLOPS while HAWKEYE reaches 256.4 by composing through `pipeline/`, a single markdown file the agent burns 100 turns rediscovering 60% of.

```
if (wg_id == 0) {
    // Single warpgroup for now
    warpgroup::reg_alloc<232>();
    // Load Q via TMA, then loop over K, V tiles
    for (int s_block = 0; s_block <= t_block; s_block++) {
        // ... TMA loads K, V into smem ...
        barrier::wait(&smem.full_k, phase_k); phase_k ^= 1;
        barrier::wait(&smem.full_v, phase_v); phase_v ^= 1;

        warpgroup::fence();                                  // S = Q @ K^T
        for (int k = 0; k < HEAD_DIM / 16; k++)
            wgmma_bf16::mma_async<BK, ...>(s_acc, dq, dk);
        warpgroup::commit();
        warpgroup::wait<0>();

        // ... causal mask + feature map ...

        warpgroup::fence();                                  // O += S @ V
        for (int k = 0; k < BK / 16; k++)
            wgmma_bf16::mma_async<HEAD_DIM, ...>(o_acc, ds, dv);
        warpgroup::commit();
        warpgroup::wait<0>();
    }
}
```

Figure 35: `ptx_abstractions` on Hopper at turn 150 (the smoking gun). The agent has manually unpacked the producer-consumer roles into a single warpgroup with explicit `warpgroup::fence/commit/wait` calls between WGMMAs. The Hawkeye `pipeline/` module ships these three calls as a single `ring_advance` helper.

CDNA4 is the cleanest A/B because both kernels compile and produce real performance with no compile-floor confounder, so the 84.7 TFLOPS (35%) gap is purely the six excluded modules. The agent re-derives the LDS XOR swizzle `(col / 8) ^ (row % 8)` at six call sites (Figure 36), each of which would invoke a single helper from HAWKEYE's `memory/` module. The kernel is also single-role (all 512 threads do compute and memory together) where HAWKEYE overlaps K and V loads with S and O via `memory/`'s `async_pipeline`, so the 35% gap is a clean lower bound on the six removed modules.

```
Line 54:  int swizzle_col0  = (col0       / 8) ^ (row0       % 8);
Line 83:  int swizzle_col   = (col_start  / 8) ^ (row        % 8);
Line 121: int swizzle_col   = (global_col / 8) ^ (global_row % 8);
Line 134: int swizzle_col_s = (col_s_start/ 8) ^ (row_s      % 8);
Line 166: int swizzle_col   = (global_col / 8) ^ (global_row % 8);
Line 175: int o_swizzle_col0= (o_col0     / 8) ^ (o_row0     % 8);
```

Figure 36: `ptx_abstractions` on CDNA4 at turn 256. Same XOR swizzle re-derived at six call sites. With Hawkeye's `memory/` module each site invokes a single helper.

The kernel is also single-role (all 512 threads do compute and memory together) while Hawkeye overlaps K/V loads with S and O via `memory/`'s `async_pipeline`. The 35% gap is a clean lower bound on the six removed modules.

Blackwell layers a brittle floor on top of this ceiling, with Gemini hallucinating `__pack_half2` at turn 32 and terminating at `best_tflops = null`, and under GPT the floor softens to 9.7 TFLOPS at turn 171 (vs 418.9 HAWKEYE) but composition still fails. Abstraction-budget scaling curves over $K \in \{1, 2, 4, 6\}$ optimization-pattern modules rise monotonically from `ptx_abstractions` (K=0) to full HAWKEYE (K=6) on Blackwell under both backbones, with the steepest slope in the first three modules, so the contribution is a property of the context rather than the LLM. The wins are not from unit tests, intrinsic wrappers, or the workload guide, all of which are retained in `ptx_abstractions` and still cap at $\sim$241 TFLOPS on Hopper. The only structural difference is the optimization-pattern modules, and under GPT those produce a 13–43$\times$ gap on every architecture, with the cleanest single-arch comparisons being CDNA4 under Gemini (35%) and CDNA4 under GPT (28.6$\times$). Tables 9 and 10 trace turn-by-turn comparisons, and Table 11 reports the two most catastrophic post-peak runs.

Table 9: Trajectory comparison on Blackwell, same agent and model. Tokens are cumulative trajectory tokens. HAWKEYE climbs monotonically through the rung ladder. `raw_ptx_docs` burns 11.6 M tokens after turn 25 with zero improvement.

Turn	**Raw PTX docs**		**HAWKEYE**	
	Tokens	Best TFLOPS	Tokens	Best TFLOPS
1	35k	—	102k	—
25	850k	5.6	1.98M	**576.1**
50	2.06M	5.6	4.99M	576.1
100	4.72M	5.6	12.97M	701.0
123	5.78M	5.6	17.30M	**702.0**
150	7.38M	5.6	22.77M	702.0
200	10.36M	5.6	—	—
221	12.43M	5.6	—	—

Raw-PTX-docs after turn 4 is textbook context pollution. The agent issues 17 distinct `wgmma`-pattern greps and accumulates each result in trajectory text. HAWKEYE climbs monotonically from T25 to T123 (576 to 702 TFLOPS) and then bounded post-peak exploration.

Table 10: Trajectory comparison on Hopper. By turn 100 (half the budget), HAWKEYE is at 184 TFLOPS, already 76% of `ptx_abstractions`'s eventual best.

Turn	`ptx_abstractions`		**HAWKEYE**	
	Tokens	Best TFLOPS	Tokens	Best TFLOPS
1	46k	—	84k	—
100	5.36M	$<$25	9.52M	**183.8**
150	8.10M	240.9	18.99M	193.0
168	—	—	22.94M	**256.4**
203	11.16M	240.9	—	—

Table 11: Two baselines where post-peak tokens are pure context pollution. The trajectory cost grows but the kernel does not.

Run	**Total tokens**	**Best TFLOPS**	**Post-peak waste**
`augmented_ptx_docs` on CDNA4	17.7M	118.4 at T175	9.8M tokens after T175, no gain
`raw_ptx_docs` on Blackwell	12.4M	5.6 at T4	12.36M tokens after T4, no gain

Across the sweep, baselines spend 120k to 2216k trajectory tokens per TFLOPS of best result while Hawkeye spends 24 to 50k.

E Optimization stacking and composition

This appendix shows what HAWKEYE agents do with the taxonomy, anchored on the trajectory waterfall (Figure 37).

E.1 Closed-loop pattern

Across Figure 37 the scalar-to-hardware-aware transition is discontinuous. Hopper GEMM sits at $\sim$23 TFLOPS until one step turns on WGMMA, TMA, mbarrier, and swizzle simultaneously and

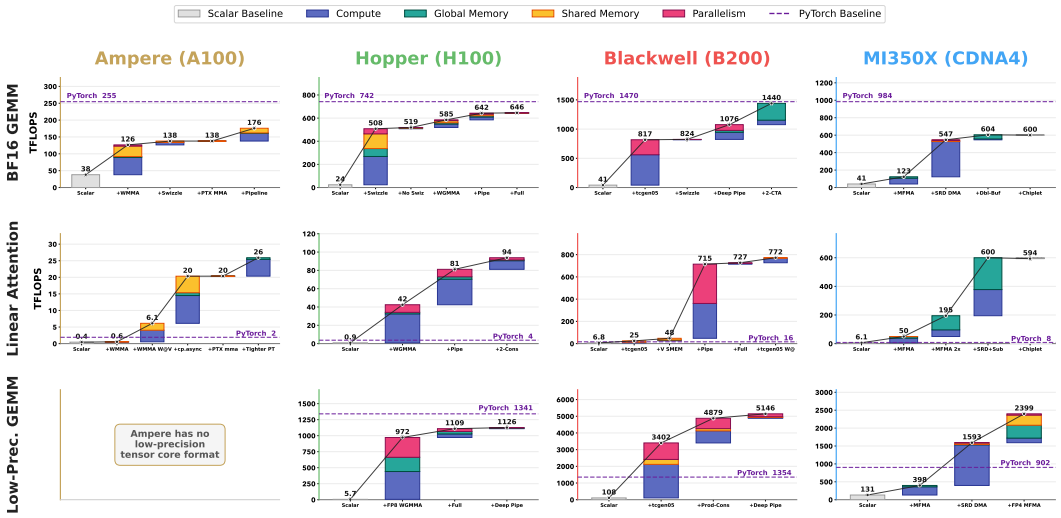


Figure 37: Optimization stacking trajectories across three NVIDIA architectures (Ampere, Hopper, Blackwell) and four workloads (BF16 GEMM, BF16 attention, BF16 causal Parallel ReBased Linear Attention, BF16/low-precision GEMM). Each bar is a cumulative optimization step in the agent's trajectory toward the final kernel; colored segments attribute the throughput delta to a specific hardware-utilization metric drawn from the Hawkeye taxonomy. The waterfalls let us read off not just *whether* the kernel got faster but *which* part of the hardware was newly exercised.

reaches 493 TFLOPS, Hopper FP8 GEMM jumps from 5.8 to 988 TFLOPS in the same step (170×), Hopper attention from 0.15 to 69.0 TFLOPS in one step, and Blackwell causal Linear Attention from 446.2 to 557.0 TFLOPS between turns 49 and 52 on top of an already-converged async pipeline. The discontinuity is structural: the Hopper features are tightly coupled, so a partial stack earns none of the speedup. WGMMA descriptors need swizzle to read coherent data, TMA needs mbarrier for completion signaling, WGMMA needs the warpgroup fence/commit/wait protocol to issue, and WGMMA needs TMA to stay fed. The unit tests and kernel pool expose the full stack in one place, so the agent crosses the gap in a single phase transition rather than incrementally.

Within each trajectory the agent runs the same five-step inner loop on every turn.

1. **Profile.** Run `evaluate_kernel` to obtain TFLOPS, SM-busy, barrier-stall, tensor-core-issue, bank-conflict, and L2-hit-rate counters from the previous kernel snapshot.
2. **Identify the bottleneck.** Compare the counters to the targeted-metric column of the relevant taxonomy cell to name the next bottleneck (`TC underutilized`, `barriers oversynchronized`, `wrong swizzle`, etc.).
3. **Consult the cell.** Read the matching unit-test cell (`naive_kernel.cu`, `optimized_kernel.cu`, `guide.md`) plus any API headers it pulls in.
4. **Modify the kernel.** Lift one or two primitives out of the cell and stack them onto the current kernel via `file_editor` or `terminal` writes. Almost every modification is local.
5. **Re-evaluate.** Run `evaluate_kernel` again, observe the metric delta, and either accept the change (advance to the next rung) or revert (debug). Compile failures and correctness failures route directly back to step 3 with a more focused query.

The loop mirrors what a human kernel author runs, and the taxonomy makes step 3 cheap because the agent reads one cell whose `guide.md` already names the metric being chased.

Table 12 extracts ten representative inflection turns from the canonical Blackwell BF16 causal-Linear-Attention trajectory (Gemini 3.1 Pro, 111 turns, 50 productive, peak 557.0 TFLOPS at turn 52, ∼5.0× over the expert reference and ∼74× over the Torch baseline). The trajectory's `decision` field records the agent's own naming of the bottleneck and the cell it consulted, and `eval_result.tflops` records the metric delta on the next `evaluate_kernel` call.

Turn	Bottleneck	Cell consulted	TFLOPS
T01	Rung 0 hangs / all-zero output	prior-run replay	—
T05	Minimal correct `tcgen05` kernel	`tmem.cuh`	—
T10	TMA tensor-map signature	`tma.cuh`	—
T22–T34	TMA coords + 128B swizzle illegal mem	`tma`, `swizzle`, `descriptor`	—
T44	Barrier phase deadlock	`pipeline` phase-bit guide	—
T49	Rung 0 correct	—	446.2
T50	Triangular work imbalance	reverse query-chunk iteration	—
T52	Schedule reversed	—	**557.0**
T53	Single-WG: no TMA/MMA overlap	`06_producer_consumer`	335.5
T59–T75	Ring-buffer phase-bit semantics	`pipeline/get_phasebit`	—
T87–T103	`barrier::init_array` arity	`general.md`	—
T111	Run ends, T52 best	—	557.0

Table 12: Representative inflection turns from the Blackwell BF16 causal Parallel ReBased Linear Attention trajectory (Gemini 3.1 Pro, 111 turns, peak 557.0 TFLOPS at T52). *Cell consulted* names the unit-test cell or API header the agent cited; TFLOPS shows the next correctness-passing evaluation. Dashes mark debug turns where the kernel still failed.

Every productive jump (T49, T52) is preceded by a bottleneck that is identified and one cell that the agent consults to fix it, while the long T59–T103 ring-buffer debug arc oscillates across `pipeline/`, `barrier/`, and `general.md` because no single cell describes `barrier::init` parity. Profitable phase transitions live where one cell explains one bottleneck.

Optimization is non-monotonic, with roughly a third of trajectories regressing at least once. On Hopper GEMM, CTA swizzle on a shallow QSIZE=1 pipeline drops 504 TFLOPS to 493 (2%) and the same swizzle adds 9% at QSIZE=3, on Ampere attention an early shuffle softmax with the wrong reduction width sits at 2 TFLOPS until the agent finds the 4-lane reduction and jumps to 22 TFLOPS in two steps, and T58 of the Blackwell trajectory above regresses from 557.0 to 335.5 because the producer-consumer split was introduced without the ring buffer. The barrier-stall, tensor-core-issue, and bank-conflict counters let the agent tell a productive transition from a misleading one, since scalar Hopper GEMM reads 73% SM-busy from scalar FMAs and only the tensor-core counter exposes the truth. The optimization moves live on a partial-order lattice where some edges are blocked and others commute, with marginal TFLOPS dropping sharply after three or four cells are stacked, and the rest of this section instantiates this closed loop on four saved trajectories.

E.2 Trajectories across workloads

We walk through four saved Gemini 3.1 Pro trajectories, one per architecture on Parallel ReBased Linear Attention. The same closed loop recurs across all four, and only the activated cells differ.

Ampere BF16 Parallel ReBased Linear Attention (Gemini 3.1 Pro, 63 turns, 41 productive, peak 99.9 TFLOPS at T59). The trajectory walks Rungs 0–6 cleanly, activating `mma/`, `async_copy/`, `swizzle/`, and the Taylor fragment. The biggest jump is at T53, where a register-chained $W \cdot V$ more than doubles throughput from 45.0 to 99.3 TFLOPS in one edit, and the final 99.9 TFLOPS is $57\times$ over Torch.

Turn	Bottleneck	Cell consulted	TFLOPS
T16	WMMA fragment store NaN: PAD too small for HD=64	`mma`, `swizzle`	7.8
T23	Warps racing on shared K,V buffers	`async_copy` (`__syncthreads` after continue)	26.5
T28	Synchronous Q/K/V loads kill MMA throughput	`ampere::async_copy::load_tile_padded`	43.2
T38	Single-buffered K,V; no overlap	double-buffered `cp.async`	45.0
T53	W materialized in SMEM, $W \cdot V$ scalar	`mma_ptx::wv_chain_k8` (register-chained MMA)	**99.3**
T59	Triangular work imbalance for causal mask	causal-mask exit (rung 5)	**99.9**

Table 13: Ampere BF16 Parallel ReBased Linear Attention trajectory inflection turns.

Hopper BF16 Parallel ReBased Linear Attention (Gemini 3.1 Pro, 113 turns, 50 productive, peak 146.9 TFLOPS at T88). The kernel sits at 4.4 TFLOPS for 81 productive turns while the agent works through Rung 1 TMA bugs, and the WGMMA replacement at T87 lifts throughput $33\times$ to 146.9 TFLOPS in one edit. Cells activated are `tma/`, `swizzle/`, `wgmma/wgmma_bf16`, and `wgmma_descriptor/`.

Turn	Bottleneck	Cell consulted	TFLOPS
T6	Per-thread strided GMEM reads	naive baseline → `tma` for Rung 1	4.4
T74	128B swizzle offset reads; TMA coords as element offsets	`swizzle::offset`, `tma::create_tensor_map`	4.4
T88	Two scalar GEMMs; no tensor cores	`wgmma::mma_async` + register-A WGMMA for $P \cdot V$	**146.9**

Table 14: Hopper BF16 Parallel ReBased Linear Attention trajectory inflection turns.

Blackwell BF16 Parallel ReBased Linear Attention (Gemini 3.1 Pro, 111 turns, 50 productive, peak 557.0 TFLOPS at T52). The trajectory activates `tcgen05_mma/`, `tmem/`, `tma/`, `descriptor/`, `swizzle/`, `epilogue/`, `pipeline/`, and `barrier/`, and the peak of 557.0 TFLOPS at T52 is $5.0\times$ over expert and $74\times$ over Torch. Turns T53–T111 attempt Rungs 2 and 3 without converging.

CDNA4 BF16 Parallel ReBased Linear Attention (Gemini 3.1 Pro, 35 turns, 23 productive, peak 299.8 TFLOPS at T27). This is the shortest run because the activated cells are smaller. A correct MFMA baseline reaches 292.6 TFLOPS at T6 ($0.88\times$ expert), incremental LDS double-buffering at T11 lifts throughput to 296.1, and `amdgpu_waves_per_eu(4,4)` at T27 reaches 299.8 TFLOPS.

Turn	Bottleneck	Cell consulted	TFLOPS
T6	Naive global loads; no MFMA	`mfma::compute_2x2_bf16`, Q-in-register	292.6
T11	Single LDS buffer; no overlap	`cdna4-apis/lds` double-buffer	296.1
T27	Underutilized occupancy on chiplets	`amdgpu_waves_per_eu(4,4)`	**299.8**

Table 15: CDNA4 BF16 Parallel ReBased Linear Attention trajectory inflection turns.

Table 16: Cross-workload trajectories from . The same closed-loop pattern from the four Linear-Attention vignettes recurs in each of these runs; full per-turn data is in the run directories.

Workload	Architecture	Precision	Turns	Peak TFLOPS
Linear Attention (v4)	Hopper	BF16	231	174.5
Linear Attention (v4)	CDNA4	BF16	201	316.9
Linear Attention (v4)	Ampere	BF16	157	128.7
Linear Attention	CDNA4	MXFP4	96	384.6
Conv2d	Blackwell	BF16	65	37.5
GEMM	CDNA4	BF16	63	643.4
GEMM	CDNA4	MXFP4	140	2341.4
DeltaNet	CDNA4	MXFP4	86	4.6
Forgetting Attention	CDNA4	BF16	113	25.4

The same closed loop recurs across every workload, architecture, and precision in the saved corpus, with activated cells differing predictably. Ampere stacks `mma/` + `async_copy/` + `swizzle/`, Hopper adds `wgmma/` + `tma/` + `wgmma_descriptor/`, Blackwell adds `tcgen05_mma/` + `tmem/` + `pipeline/` + `epilogue/`, and CDNA4 stacks `mfma/` + `lds/` + `global_load_lds/` + `wave/`. When the workload extends a familiar pattern, the agent adds a thin specialization on top of the shared stack and converges. When it requires composing across cells in a way no single cell describes, the closed loop oscillates without converging.

E.3 Composition across kernels

HAWKEYE agents compose unit-test cells rather than authoring one-off arch-specific blobs. HAWKEYE produces 51 hardware-aware kernels across four architectures, six workloads, and two precision tiers (Figure 38).

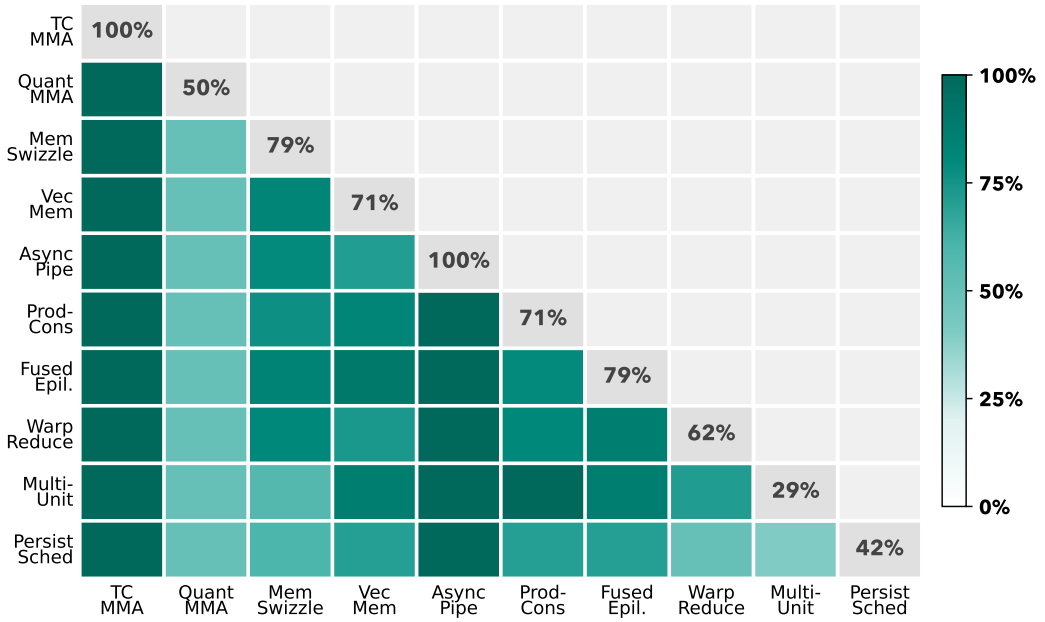


Figure 38: Co-occurrence of unit tests in HAWKEYE kernels across 6 workloads, 2 precisions, and 4 architectures. The dense off-diagonal pattern indicates that performant kernels compose several primitives drawn from across the taxonomy rather than relying on any single optimization in isolation.

Reading the matrix surfaces three structural patterns. An arch-specific bundle of cells co-occurs in nearly every kernel for that architecture (Hopper WGMMA + TMA + mbarrier + 128B swizzle, Blackwell `tcgen05` + TMEM + TMA + producer-consumer, CDNA4 MFMA + LDS swizzle + `global_load_lds` + wave occupancy), forming the shared infrastructure layer that Section E.3.1 traces in detail. The remaining cells (warp or wave reduction, persistent scheduling, epilogue pipeline) attach to that infrastructure in workload-dependent combinations rather than uniformly, with reduction concentrating in attention-family kernels, persistent scheduling in GEMM and Conv2D, and epilogue pipelining where a fused activation or precision conversion lives at the kernel boundary. The matrix is dense off the diagonal in every architecture column, so no HAWKEYE kernel reaches expert performance with a single primitive in isolation, and the subsubsections below trace the recurring multi-cell patterns along the workload, architecture, precision, and invention axes.

E.3.1 Cross-workload reuse

A small set of arch-specific primitives forms an 80–90% shared substrate per architecture (Ampere `mma.sync`/WMMA + three-stage `cp.async`, Hopper WGMMA + TMA + mbarrier + 128B swizzle, Blackwell adds `tcgen05` + TMEM, CDNA4 `mfma_32x32x16_bf16` + `global_load_lds` + LDS swizzle), with only a thin workload-specific layer on top. The Hopper WGMMA $QK^\top$ inner loop is byte-identical between attention and Linear Attention (Figure 39), with only the tile dims and K iteration count changing.

The same pattern recurs on Blackwell ping-pong TMEM (linear and Forgetting Attention), Ampere WMMA (GEMM and attention), and CDNA4 `a_subtile` / `b_subtile` readers (GEMM and Linear Attention). On Blackwell, three cells (`01_tensorcore_mma`, `05_async_pipeline`, `06_producer_consumer`) appear in every attention-family kernel, with the workload-specific layer (Taylor hi/lo split for Linear Attention, β-prescaling for DeltaNet, cumsum-bias factoring for Forgetting Attention, im2col for Conv2D) sitting on top.

E.3.2 Cross-architecture composition

Linear Attention is the cleanest cross-arch test since the algorithm is fixed (Taylor map, causal mask, $W \cdot V$ matmul) but the cell composition shifts at every step. Ampere $\rightarrow$ Hopper activates `producer_consumer` and switches to WGMMA, Hopper $\rightarrow$ Blackwell adds 2-CTA cluster multicast

```
// attention/optimized.cu (lines 219-229)
hopper::wgmma::fence_sync();
hopper::wgmma::wgmma128<0,1,1,0,0>(s_acc, sQ_base + 0*WGMMA_K,
                                   sK_slot + 0*WGMMA_K, 1024);
#pragma unroll
for (int k = 1; k < K_ITERS_S; ++k)
    hopper::wgmma::wgmma128<1,1,1,0,0>(s_acc,
                                       sQ_base + k*WGMMA_K,
                                       sK_slot + k*WGMMA_K, 1024);
hopper::wgmma::commit_group_sync();

// linear_attention/optimized.cu (lines 248-254)
wgmma::fence_sync();
bf16* sK_slot = &s.K[qi * BN * HD];
wgmma::wgmma<BN,1,1,1,0,0>(sa, &sQ_my[0*WK], &sK_slot[0*WK]);
wgmma::wgmma<BN,1,1,1,0,0>(sa, &sQ_my[1*WK], &sK_slot[1*WK]);
wgmma::wgmma<BN,1,1,1,0,0>(sa, &sQ_my[2*WK], &sK_slot[2*WK]);
wgmma::wgmma<BN,1,1,1,0,0>(sa, &sQ_my[3*WK], &sK_slot[3*WK]);
wgmma::commit_group_sync();
```

Figure 39: Hopper $QK^\top$ inner loop. Same fence-commit-wait protocol, same descriptor-based operand binding; only the tile dims and the K iteration count change.

and TMEM with `tcgen05` replacing WGMMA, and Blackwell $\rightarrow$ CDNA4 drops producer-consumer and substitutes LDS double-buffering with `t & 1` stage indexing. The four $QK^\top$ inner loops below (Figure 40) share nothing at the instruction level yet collapse to the same algorithmic skeleton.

```
// --- Ampere: WMMA fragment API ---
#pragma unroll
for (int ki = 0; ki < QKT; ki++) {
    FBc fK; wmma::load_matrix_sync(fK, kc + ni*16*PAD + ki*16, PAD);
    wmma::mma_sync(sf, fQ[ki], fK, sf);
}

// --- Hopper: WGMMA over SMEM descriptors ---
wgmma::fence_sync();
for (int ki = 0; ki < 4; ki++)
    wgmma::wgmma<BN,1,1,1,0,0>(sa, &sQ_my[ki*WK], &sK_slot[ki*WK]);
wgmma::commit_group_sync();

// --- Blackwell: tcgen05 MMA into TMEM ---
tcgen05_mma::smem_fence();
tcgen05_mma::k_loop_ss<K_CHUNKS_QK, CHUNK_SIZE, CHUNK_SIZE>(
    tmem_qk_ab[next_side], q_desc, k_desc, idesc_qk, /*acc=*/true);
tcgen05_mma::commit<1>(&smem.bar_qk_done[next_side]);

// --- CDNA4: MFMA with LDS-resident operands ---
#pragma unroll
for (int ki = 0; ki < BK; ki += 16) {
    bf16x8 k_op;
    k_rd.read_lo<BK>(sm + cur_off, ki, k_op);
    sync::waitcnt_lgkmcnt<0>();
    s_acc = mfma::bf16_32x32x16(q_cached[ki/16], k_op, s_acc);
}
```

Figure 40: Parallel ReBased Linear Attention $QK^\top$ inner loop across Ampere, Hopper, Blackwell, and CDNA4. The four invocations share nothing at the instruction level, yet the agent reuses the algorithmic skeleton with the cell composition shifting to match the available primitive.

The activation set grows monotone Ampere (5 cells) → Hopper (6, adding producer-consumer) → Blackwell (adding warp-reduction cell 08 for TMEM row reduction) before re-shuffling for CDNA4 (drops producer-consumer, keeps cross-lane reduction). Forgetting Attention shows the same monotone-then-shuffle pattern.

E.3.3 Cross-precision cascade

A precision change cascades through cells 02 (`quantized_mma`), 03 (swizzle stride), 05 (descriptor stride), and 07 (epilogue dequant). On Hopper FP8 the WGMMA K grows 16 → 32, the descriptor stride drops 1024 → 512 B, the TMA col-group doubles, QSIZE halves to 2, and the epilogue gains `cvt.rn.satfinite.e4m3x2` (Figure 41). On Blackwell NVFP4 the agent pairs BF16 MMA with offline FP4 dequant, running a preprocessing kernel that converts FP4 Q/K to BF16 with FP8 E4M3 per-16-block scales and pre-scales K by β, then reuses the BF16 deltanet pipeline verbatim. On CDNA4 MXFP4 the agent uses native `mfma_scale_32x32x64` with the E8M0 scale tile transposed to `[K/32, M]` so 512 threads populate it coalesced. Each precision-specific cell exposes the bundle as one change.

```
static constexpr int WGMMA_K_FP8 = 32;       // K=32 instead of 16
static constexpr int COL_GROUP_FP8 = 64;     // 128 bytes / 1 byte per element

#pragma unroll
for (int k_it = 0; k_it < HEAD_DIM / WGMMA_K_FP8; ++k_it) {
    uint64_t dq = wgmma::make_descriptor(
        &sQ_base[k_it * WGMMA_K_FP8], 16, 512, wgmma::Swizzle::B64);
    uint64_t dk = wgmma::make_descriptor(
        &sK[k_it * WGMMA_K_FP8], 16, 512, wgmma::Swizzle::B64);
    wgmma::mma_async_e4m3<BN, 1, 1, 1>(s_acc, dq, dk);
}
```

Figure 41: Hopper FP8 DeltaNet cascade. The K-doubling and the col-group-doubling are simultaneous, and the descriptor stride drops from 1024 to 512.

The CDNA4 MXFP4 row also illustrates how the precision cell catches a correctness-only bug. The first kernel hit peak FLOPs but failed correctness because the E8M0 scale was read from a row-major `[M, K/32]` layout with lane-cache-line collisions, and the transposed `[K/32, M]` fix reaches 87% of unscaled peak and reuses across all five other CDNA4 MXFP4 workloads. The cascade is asymmetric across vendors (Hopper FP8 inverts column-group, Blackwell NVFP4 dequantizes offline, CDNA4 MXFP4 commits to native FP4 MFMA), and each precision-specific cell hides that asymmetry behind a single named change.

E.3.4 Invented patterns

Several agent-invented patterns compose on top of cells without appearing in any single cell. The Ampere INT8 GEMM pre-transposes B to BT[N,K] for col-major WMMA fragments with padded SMEM stride 144, and the Hopper FP8 DeltaNet keeps Q and K as FP8 e4m3 in 8 KB SMEM with the descriptor stride dropped from 1024 to 512. The CDNA4 GEMM MXFP4 kernel uses a transposed `[K/32, M]` LDS layout for E8M0 block scales so 512 threads populate the tile coalesced and the lane-0/32 bank conflict disappears.

The 702 TFLOPS Blackwell Linear-Attention kernel composes five inventions on top of four cells. (a) A three-warpgroup split (compute, MMA-scheduler, TMA-producer) with asymmetric register allocations 256/40/40 instead of the canonical two-warpgroup pattern. (b) A QK lookahead that pre-issues QK[k+1] before waiting for W_ready[k]. (c) A hi/lo BF16 split of the Taylor result for ∼24-bit precision recovery. (d) A reverse chunk-order schedule (`qc = (num_chunks - 1) - blockIdx.y`) that puts heavy causal chunks first for load balance. (e) A Q-aliased-as-O-staging trick that frees ∼16 KB SMEM by having the epilogue stage O through the Q tile region after Q is no longer live.

The Blackwell Forgetting-Attention kernel adds an algebraic invention on top of `08_warp_reduction` (Figure 42). It factors `cumsum_q` out of the additive cumulative-log-decay bias and tracks `effective_max = row_max - cq` in TMEM instead of subtracting the

per-position decay term inside the softmax loop, collapsing one full TMEM read per softmax row into a single scalar update.

```
// Without FoX algebra: every QK entry pays cq[i] - cq[j]
//   for (i,j) in tile: s[i,j] -= cq[q_row + i] - cq[k_row + j];
//   row_max = max_j s[i,j];
//   p[i,j] = exp(s[i,j] - row_max);
//
// With FoX algebra: factor cq out of the bias, track effective_max instead.
// Producer fills cum_K (length BK) per chunk via a small TMA stream.
float effective_max = row_max - cq;        // one scalar update per row
// Inside the softmax: subtract cum_K[j] from s[i,j] (already partial),
// and subtract the per-row effective_max instead of (row_max + cq).
p[i][j] = __expf(s[i][j] - cum_K[j] - effective_max);
```

Figure 42: Blackwell Forgetting-Attention FoX-algebra trick. The cumulative-decay bias `cq` is factored out of the per-element subtraction and folded into a row-wise `effective_max` that the softmax consumes.

F Generalization across porting axes

Hawkeye kernels generalize along four orthogonal porting axes (Sections F.1–F.4), Section F.5 extends the evaluation to workloads without a matrix-multiply component, and Section F.6 places the absolute numbers against the hardware ceiling.

F.1 Cross-workload

Fix a chip, vary the workload, and the hardware-touching cells stay byte-identical across workloads in the same arithmetic family. Five Hopper BF16 kernels (attention, causal Linear Attention, DeltaNet, Forgetting Attention, D=128 Linear Attention) share a one-producer / two-consumer warpgroup partition, TMA loads, an mbarrier-driven K/V ring, the `fence_sync/wgmma/commit_group_sync/wait_group_sync` pipeline, and 128B XOR swizzle, with workload-specific code at 10–20% of the kernel and the inner $\mathrm{Q}{\cdot}\mathrm{K}^{\mathrm{T}}$ loop byte-identical up to BN and the accumulator name (the same comparison demonstrated in Section E.3.1). The same shape recurs on Blackwell across attention, Linear Attention, Forgetting Attention, Conv2D, and DeltaNet with a Blackwell-specific ping-pong TMEM (`NUM_QK_SIDES = 2`) holding pre-issued $\mathrm{QK}[0]$ and lookahead $\mathrm{QK}[k{+}1]$ while $\mathrm{PV}[k]$ drains, and on CDNA4 across six workloads sharing `mfma::bf16_32x32x16`, `global_load_lds` SRD-DMA, `a_subtile/b_subtile` LDS staging, `waitcnt_lgkmcnt` barriers, and `dpp` cross-lane reduction. Shared infrastructure is 80–90% of file size on Hopper and Blackwell and 70–85% on CDNA4, ending at the QK accumulator where online softmax, the Taylor map, and the cumulative-decay broadcast become workload-specific. DeltaNet adds an intra-tile MMA for the recurrent W-carry, and Conv2D replaces the QK descriptor build with an im2col index walk.

F.2 Cross-architecture

Fix the algorithm, vary the chip. The four Linear-Attention $\mathrm{QK}^{\top}$ inner loops in Figure 40 differ along five orthogonal axes (tensor-core call, operand source, sync primitive, pipeline depth, cooperation unit). We triangulate with Forgetting Attention (Hopper, Blackwell, CDNA4), which adds a per-position cumulative-log-decay term $\mathrm{cumlog}_f[i] - \mathrm{cumlog}_f[j]$ before softmax and exposes how each chip handles a broadcast on top of the matmul. Ampere is omitted because the broadcast does not fit SMEM at D=64, S=4096. On Hopper the broadcast lives in an SMEM gather from `s.cumK` into registers that hides under the WGMMA pipe (Figure 43). On Blackwell the consumer warpgroup defers behind `bar_pv_gate` and folds the load into the TMEM read producing P. On CDNA4 the broadcast lives in the wave register file and is consumed in the fused post-MFMA scale loop (Figure 44).

The algorithmic logic is identical across rows but the hardware plumbing is not. The agent picks the right primitive at the right place along three concrete axes. Operand pre-MMA location is registers on Ampere, 128B XOR-swizzle SMEM on Hopper, TMEM-staged descriptor and `idesc`

```
hopper::wgmma::fence_sync();
hopper::wgmma::wgmma<WGMMA_N_S, 0, 1, 1, 0, 0>(
    s_acc, &sQ_my[0 * WGMMA_K], &s.K[qidx * BN * HD + 0 * WGMMA_K]);
#pragma unroll
for (int k = 1; k < K_ITERS_S; ++k) {
    hopper::wgmma::wgmma<WGMMA_N_S, 1, 1, 1, 0, 0>(
        s_acc, &sQ_my[k * WGMMA_K], &s.K[qidx * BN * HD + k * WGMMA_K]);
}
hopper::wgmma::commit_group_sync();

// Prefetch cumK that overlaps both the WGMMA group and the prior PV-half2.
hopper::mbarrier::try_wait_parity(&s.ck_full[qidx], phase_ck[qidx]);
phase_ck[qidx] ^= 1;
float pre_ck[NSTRIPES][4];
#pragma unroll
for (int g = 0; g < NSTRIPES; ++g) {
    int col_base = g * WGMMA_K + col_lane_pre;
    pre_ck[g][0] = s.cumK[qidx * BN + col_base + 0];
    /* ... three more cum-K loads ... */
}
hopper::wgmma::wait_group_sync<0>();
```

Figure 43: Forgetting Attention QK^{T} core, Hopper. One $m64n256k16$ WGMMA chain across K_ITERS_S=4 K-iters, with smem-prefetched cumK overlapping the WGMMA pipeline.

```
fp32x16 s_acc = {};

#pragma unroll
for (int ki = 0; ki < BK; ki += 16) {
    bf16x8 k_op;
    k_rd.read_lo<BK>(sm + cur_off, ki, k_op);
    sync::waitcnt_lgkmcnt<0>();
    s_acc = mfma::bf16_32x32x16(q_cached[ki / 16], k_op, s_acc);
}
#pragma unroll
for (int ki = 0; ki < BK; ki += 16) {
    bf16x8 k_op;
    k_rd.read_lo<BK>(sm + cur_off + LDS_K_HALF, ki, k_op);
    sync::waitcnt_lgkmcnt<0>();
    s_acc = mfma::bf16_32x32x16(q_cached[2 + ki / 16], k_op, s_acc);
}

// Scale + additive cum-log gate + causal mask, fused into post-MFMA registers.
#pragma unroll
for (int i = 0; i < 16; i++) {
    int row, col;
    mfma::output_coord_32x32(lane, i, row, col);
    s_acc[i] = s_acc[i] * SCALE
             + (gate_q_smem[local_m + row] - gate_k_smem[local_n + col]);
    if (q_start + local_m + row < kv_start + local_n + col)
        s_acc[i] = -INFINITY;
}
```

Figure 44: Forgetting Attention QK^{T} core, CDNA4. Cum-log delta and causal mask are fused into the post-MFMA scale stage in registers.

on Blackwell, and `ds_read_b64_tr_b16` LDS on CDNA4. The accumulator is the register file on Ampere, Hopper, and CDNA4 and TMEM on Blackwell, which is what enables the ping-pong lookahead. MMA issue is per-warp on Ampere, warpgroup-leader after `fence_sync` on Hopper, CTA-pair-leader after `smem_fence` on Blackwell, and per-wave on CDNA4. The empirical payoff is in Section F.4, with Linear Attention at $S{=}16384$ reaching 323 TFLOPS on Blackwell, 203 on Hopper, 247 on CDNA4 BF16, and 387 on CDNA4 MXFP4 against `torch.compile` baselines of 18, 7, 15, and 15, and Forgetting Attention at $S{=}16384$ on Blackwell BF16 reaching 255 TFLOPS versus 119 expert ($2.1\times$, the largest seqlen-sweep margin).

F.3 Cross-precision cascade

When the workload and architecture are fixed, changing precision still forces a coordinated rewrite of the kernel. A precision change cascades through five cells (MMA K-dim, SMEM column stride, descriptor swizzle, TMA byte budget, plus a new scale-factor cell for FP4), exposed by the precision-specific unit test as a single bundle. The minimal cascade is BF16 $\rightarrow$ FP8 on Hopper, where K doubles $16 \rightarrow 32$, descriptor stride halves $1024 \rightarrow 512$, and WGMMA switches to `mma_async_e4m3` (the same listing the cross-precision-cascade subsubsection shows in Figure 41).

Blackwell NVFP4 and CDNA4 MXFP4 DeltaNet. NVFP4 keeps the BF16 `tcgen05` structure and adds a scale-factor cell with a separate E4M3 K-scale TMA stream and per-block producer dequantization. CDNA4 MXFP4 uses native `v_mfma_scale`, stages the K-scale tile as transposed `[K/32, M]` LDS, holds K as packed FP4 nibbles, and switches the MFMA to `mfma_fp4::mfma_32x32x64`.

Why the cascade is consistent. All three lower-precision kernels reuse the BF16 schedule and edit only the cascade cells. NVFP4 GEMM on Blackwell hits 5394 TFLOPS at $M{=}N{=}K{=}16384$ (versus 1283 for BF16), and MXFP4 GEMM on CDNA4 reaches 2009 TFLOPS at 4096 (540 for BF16).

The scale-factor cell is genuinely new. FP8 inherits the 16-bit descriptor model and only changes K and stride, while NVFP4/MXFP4 carry a side-channel scale per FP4 block (separate TMA stream on Blackwell, instruction operand on CDNA4) that requires a first-class scale-factor cell.

Order-of-magnitude payoff per cascade. Each cascade buys 1.5–$4\times$ over BF16. NVFP4 GEMM on Blackwell at $M{=}N{=}K{=}16384$ buys $4.2\times$ ($5394 \div 1283$), MXFP4 GEMM on CDNA4 at 4096 buys $3.7\times$ ($2009 \div 540$), FP8 GEMM on Hopper at 4096 buys $1.6\times$ ($1056 \div 675$). The roofline view in Section F.6 re-projects these as a rightward slide on arithmetic intensity plus an upward shift of the compute ceiling.

F.4 Problem-size sweep

Holding workload, architecture, and precision fixed, we sweep GEMM over $M{=}N{=}K \in \{1024, 2048, 4096, 8192, 16384\}$ and three attention variants over $S \in \{1024, 2048, 4096, 8192, 16384\}$ on every arch-precision pair. HAWKEYE holds its speedup over the hardware-unaware baseline across the full range, with the lead widening at the larger sizes (Figure 45).

F.5 Workloads beyond tensor algebra

We apply the unchanged ten-row taxonomy to four workloads with no matrix-multiply component on H100 and B200 (Table 17). RMSNorm, softmax, and a row reduction are BF16 and bandwidth-bound, so we report median latency from `do_bench` rather than TFLOPS. FDTD-2D is the FP32 PolyBench electromagnetic stencil (NX=NY=4096, $T{=}32$) that SwizzlePerf [Tschand et al., 2025] evaluates. All eight kernels match a naive CUDA reference element-wise with a correctness fraction of 1.0.

Which baseline is strongest varies by workload, so we report against the fastest of `torch.compile`, an expert implementation, and the naive CUDA kernel in each cell. For RMSNorm and the reduction the expert implementation is PyTorch's hand-written fused ATen kernel, which beats `torch.compile`; for softmax `torch.compile` is faster than ATen. There is no vendor library for 2D stencils, and the closest expert analogue, a CUDA-graph-captured eager PyTorch sweep, is slower than the naive CUDA kernel, which is in turn the strongest baseline on H100.

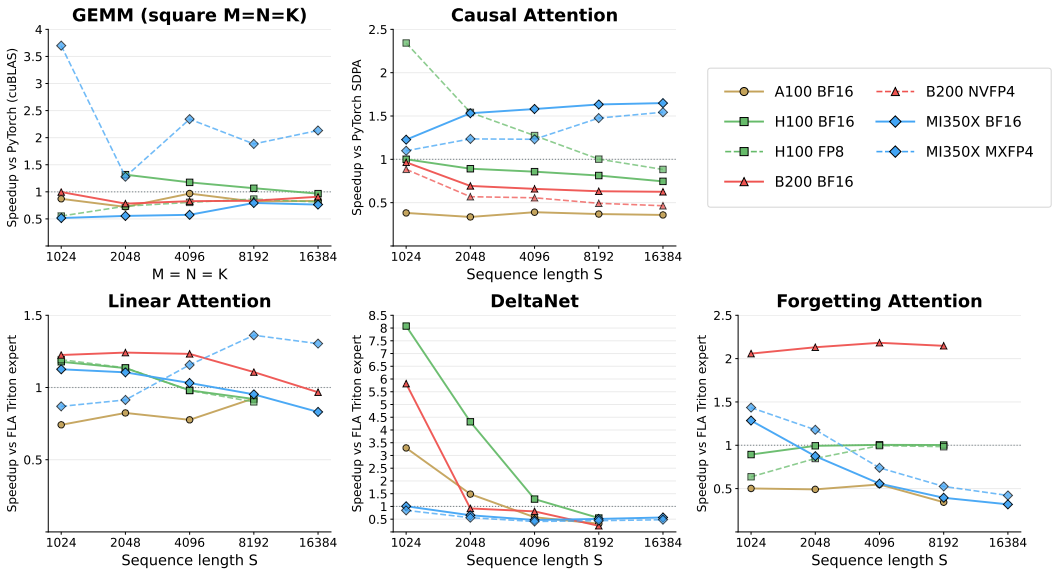


Figure 45: Problem-size sweep across GEMM, Attention, Linear Attention, DeltaNet, Forgetting Attention. HAWKEYE holds its speedup over the hardware-unaware baseline across the full size range on every arch-precision pair, with the lead widening at the larger problem sizes where tensor-core duty cycle and asynchronous data movement matter most.

The bandwidth-bound kernels compose the Vectorized Memory row (128-bit loads and stores) and the Warp/Wave Reduction row (warp-shuffle reductions), and the RMSNorm and reduction kernels add Persistent Scheduling (an occupancy-sized persistent grid). None use tensor cores or asynchronous copies. The FDTD-2D kernels pair 128-bit Vectorized Memory with two workload-specific transformations: fusing the three field updates of a timestep into one pass, and temporal blocking that advances two timesteps per pass, staging the wavefront in shared memory on H100 and in per-lane register windows with warp-shuffle halo exchange on B200. These results are single runs at one problem size per workload.

Table 17: HAWKEYE on workloads beyond tensor algebra. Median latency in ms; Compile is `torch.compile`. Speedups are against `torch.compile` and against the strongest baseline in each cell. The FDTD-2D expert column is a CUDA-graph-captured eager PyTorch sweep, since no vendor stencil library exists.

Workload	GPU	HAWKEYE	Compile	Expert	Naive	Strongest	vs. Compile	vs. strongest
RMSNorm	H100	0.0966	0.2239	0.1407	2.2237	ATen	2.32×	1.46×
RMSNorm	B200	0.0483	0.1432	0.0605	2.2925	ATen	2.97×	1.25×
Softmax	H100	0.0937	0.2734	0.6318	3.0664	Compile	2.92×	2.92×
Softmax	B200	0.0478	0.2208	0.4762	2.7849	Compile	4.62×	4.62×
Reduction	H100	0.0969	0.3341	0.1158	1.0799	ATen	3.45×	1.20×
Reduction	B200	0.0479	0.2197	0.0748	1.1073	ATen	4.59×	1.56×
FDTD-2D	H100	2.5040	9.9689	30.2522	7.9761	Naive	3.98×	3.19×
FDTD-2D	B200	1.1218	5.3970	20.8978	5.8362	Compile	4.81×	4.81×

F.6 Roofline analysis

We profile each kernel with NCU on NVIDIA and rocprof on MI350, extract arithmetic intensity and TFLOPS, and place the result on a roofline [Williams et al., 2009]. The aggregated dataset is 110 profiles (23 HAWKEYE, 12 hardware-unaware baselines, 75 waterfall ablations). Figure 46 shows GEMM, Conv2D, and attention in BF16 and the highest-supported low-precision format per chip, with hardware-unaware baselines sitting 20–40× below the BF16 ceiling, HAWKEYE BF16 kernels approaching the achievable line within 0.6–0.9× (residual gap from DRAM at 40–60% of peak and tensor-core duty cycle at 30–70%), and HAWKEYE low-precision kernels both raising the ceiling (2–4× tensor-core throughput) and sliding rightward (lower bytes per operand).

The profiling data adds detail beyond TFLOPS. Tensor-core duty cycle (`sm__pipe_tensor_op_*_cycles_active`) climbs from 19.3% on hardware-unaware Ampere attention to 50–70% for HAWKEYE, DRAM throughput (`dram__throughput`) climbs from 1.3% to 20–50% as `cp.async` pipelines global loads under the tensor-core pipe, and stall reasons (`smsp__warp_issue_stalled_*`) shift from `long_scoreboard` at 16–30% to `barrier` or `short_scoreboard` at 5–15%, the on-chip signature of producer-consumer engaging.

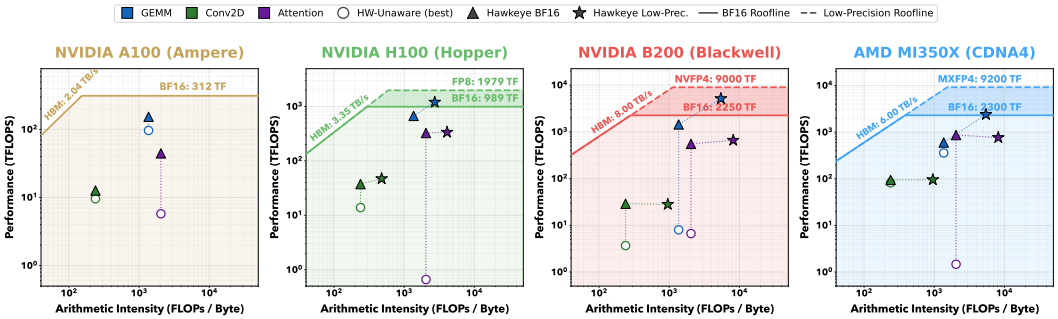


Figure 46: Roofline [Williams et al., 2009] analysis across four GPU architectures (Ampere, Hopper, Blackwell, MI350) on GEMM, Conv2D, and attention in BF16 and a low-precision format. For each workload we plot the best hardware-unaware baseline kernel, the Hawkeye BF16 kernel, and the Hawkeye low-precision kernel. Hawkeye kernels approach the achievable ceiling, and the lower-precision formats both raise that ceiling and increase arithmetic intensity.

G Experimental infrastructure

This section documents the prompts, evaluation pipeline, and compilation toolchain to reproduce every number in Sections D–F.

G.1 Prompt templates

All baselines share the same Gemini 3.1 Pro backbone and per-arch evaluation pipeline, with only the system prompt and workspace files varying. Each run is parameterised by a per-(baseline, arch) configuration that pins the architecture, workload, precision, model, baseline context, turn budget, and eval timeout, and uses one of two harnesses, an API agent with native tool-use (`evaluate_kernel`, `read_header`, `read_validation_test`, `query_replay`) running the system prompt in Figure 47, or an OpenHands agent that collapses the same blocks into a single bash-only task prompt (Figure 48) executed via `remote_exec.py` against a persistent Modal H100 sandbox. The six `baseline_context` values (`triton`, `raw_ptx_docs`, `augmented_ptx_docs`, `inlined_gemm`, `vendor_fmha`, `ptx_abstractions`) share these templates and differ only in the *Available API* block (Triton DSL, raw PTX ISA, curated PTX summaries, hand-optimized GEMM, CUTLASS / CK FMHA listing, or stripped HAWKEYE headers) and in the *Validation* block, which is removed for every baseline except HAWKEYE. Strategy, problem spec, interface contract, and reflections are byte-identical, so plateau differences can be attributed to the baseline-specific context. Each run uses `max_turns` = 100, `max_tokens` = 16,384 (4,096 for the closing reflection), a 900 s eval timeout, and three to five runs per (baseline, arch).

G.2 Evaluation pipeline

The same three-stage harness evaluates every kernel.

1. **Compile.** Standalone CUDA or HIP source. NVIDIA archs use a pybind11 module via `torch.utils.cpp_extension.load`, and CDNA4 uses a C++ binary that includes the kernel via `-DOPTIMIZED_KERNEL=kernel.hip`.
2. **Correctness.** See Section G.3.
3. **Performance.** Median of 100 iterations after 500-iter warmup on NVIDIA or 50 iterations after 10-iter warmup on CDNA4, measured via Triton `do_bench` [Tillet et al., 2019].

```
You are an expert {arch_display} kernel engineer.
Goal: write the highest-performance {problem_type} kernel.
Score = TFLOPS / {baseline_label}.

# Problem
  Type, dimensions, interface contract.
  Correctness: {thr*100:.0f}% of elements within {rtol*100:.0f}% tolerance.

# Tools
  evaluate_kernel       compile + correctness + bench on real {arch_display}
  read_header           {apis_dir}/*.cuh   ({namespace}:: namespace)
  read_validation_test  read working unit-test examples
  query_replay          best previous kernels for this/other problems
  read_triton_reference (workload-dependent) Triton expert kernel
  browse_kernels        optimized kernels for other workloads/precisions

# API
  #include "{apis_dir}/{umbrella}"    # under {namespace}:: namespace

# Workflow
  Phase 1: explore + plan.md (Approach A/B/C, implement A first).
  Phase 2: get correctness.
  Phase 3: profile (smsp__warp_issue_stalled_*, dram__throughput,
           sm__inst_executed_pipe_tensor_op_hmma, l1tex__data_bank_conflicts)
           -> one targeted change -> eval -> revert on regression.
  Final: emit structured JSON reflections, one per insight.
```

Figure 47: API-agent system-prompt skeleton. The `{arch_display}`, `{apis_dir}`, `{namespace}`, and `{umbrella}` placeholders are substituted from `ARCHITECTURES[arch]`.

```
# Task: Write an optimized {problem_type} kernel for {Arch}.
# Score = TFLOPS / {baseline_tflops}.

## Workspace
  kernel.cu, eval.py, {arch}-apis/, validation/
  All GPU work runs on a persistent remote Modal H100 sandbox via
  remote_exec.py. The sandbox persists across commands (up to 24h).

## Available API
  {arch}-apis/{arch}.cuh   # umbrella header
  Modules: barrier, tma, warpgroup, wgmma_descriptor, wgmma, swizzle
  validation/   # {N} read-only unit tests demonstrating correct usage

## Architecture Requirement
  Kernel MUST use WGMMA tensor cores via hopper-apis. wmma / mma.sync
  / scalar FMA will not reach competitive performance on H100.

## Workflow
  Phase 1: Explore + write plan.md (Approach A/B/C ranking).
  Phase 2: Get correctness.
  Phase 3: Profile-optimize cycle (NCU on profile_kernel.py, one
           change per turn, revert on regression). Write
           reflections.json after every eval. Update plan.md when
           changing approach.
```

Figure 48: MSWEA task-prompt skeleton. The agent runs in a Docker container; `remote_exec.py` forwards bash commands to a persistent Modal H100 sandbox.

TFLOPS uses $2MNK$ for GEMM, $2BHS^2D$ for the causal attention variants (standard, Linear, Forgetting), and $4BHSD^2$ for DeltaNet.

Geomean calculation. The two right-most columns of Table 2 report per-row geomeans across the four BF16 columns (A100, H100, B200, MI350) and the three low-precision columns (H100 FP8, B200 NVFP4, MI350 MXFP4). Both `X` (incorrect or runtime errors) and $0.00\times$ (numerically-zero throughput, e.g. correctness-only kernels with negligible speedup) are penalized to $0.01\times$ in the geomean to avoid undefined logarithms and to ensure failed runs depress the aggregate. All other entries enter the geomean at their reported speedup.

G.3 Precision and correctness tolerance

Reference oracle. Each optimized kernel is compared against a same-precision naive reference, an independent FP32-accumulator scalar implementation with no tensor cores or tiling. This is to ensure that kernels stay within a strict correctness bound.

Comparison gate. Element-wise allclose with a fraction threshold:

$$\texttt{frac}\big(|c-r| \le \texttt{atol} + \texttt{rtol}\cdot|r|\big) \ge \texttt{frac_threshold}.$$

We hold $\texttt{frac_threshold} = 1.00$ throughout — every output element must pass. However, lower precision workloads introduce intrinsic noise that widens the per-element bound $(\texttt{atol}, \texttt{rtol})$. The defaults follow KernelBench [Ouyang et al., 2025] and PyTorch's FP8 testing conventions [PyTorch Contributors, 2024].

Per-(workload, precision) tolerance. Table 18 lists the tightest gate every kernel must pass. The default is 0.01; only the three attention-family workloads at FP8 / NVFP4 / MXFP4 widen to 0.10.

Workload	BF16	FP8	NVFP4	MXFP4	Frac
GEMM	0.01	0.01	0.01	0.01	1.00
Conv2D	0.01	0.01	0.01	0.01	1.00
DeltaNet	0.01	0.01	0.01	0.01	1.00
Attention	0.01	**0.10**	**0.10**	**0.10**	1.00
Forgetting Attention	0.01	**0.10**	**0.10**	**0.10**	1.00
Linear Attention	0.01	**0.10**	**0.10**	**0.10**	1.00

Table 18: Per-element bound $\texttt{atol} = \texttt{rtol}$ per (workload, precision); `frac_threshold` stays at 1.00 across every cell — every output element must pass.

Why the attention-family wide bucket exists. The three attention-style workloads compose two low-precision matmul reductions through a non-linearity: softmax in standard attention, `cumsum` of log-decay terms followed by `expf` in Forgetting Attention, and the Taylor feature map $1+qk+\frac{1}{2}(qk)^2$ in ReBased Linear Attention. The non-linearity propagates per-element rounding noise from the first matmul into the second. GEMM, Conv2D, and DeltaNet do not have this composition, so the same per-element noise stays bounded by the default gate.

FP8 numerics (E4M3, $\sim$3-bit mantissa). Summing $K{=}4096$ FP8-quantized products in FP32 yields $\sim 3\%$ inherent relative noise from per-element rounding before accumulation, regardless of summation order. After propagation through the non-linearity this exceeds 0.01 but stays within 0.10.

FP4 numerics (NVFP4 E2M1 with E4M3 block scales; MXFP4 E2M1 with E8M0 scales). FP4's 2–3 effective mantissa bits and tensor-core internal reduction order produce a noise floor that a naive scalar implementation cannot match, so 0.10 is the tightest scalar-reference gate.

Discipline. Tolerance is widened only when the (workload, precision) combination has a documented numerical floor above 0.01; `frac_threshold` stays at 1.00 across every cell.

G.4 Compilation and profiling

Every kernel from HAWKEYE or any baseline runs through the harness in Section G.2. What varies is the baseline being compared and the profilers used to attribute performance.

Baseline matrix. `torch.compile` runs `max-autotune` dispatching to cuBLAS [NVIDIA, 2026], cuDNN [Chetlur et al., 2014], FlashAttention [Dao et al., 2022], FlashInfer, or rocBLAS / MIOpen on MI350. `triton_fla` uses Flash Linear Attention [Yang and Zhang, 2024] autotuned per size, `torch._scaled_mm` and `torch.SDPA` use default PyTorch dispatch, and `hipify` runs `hipify-clang` [AMD ROCm Team, 2024] on the hardware-agnostic CUDA or the HAWKEYE B200 kernel. Agent baselines are `kernelbench` (iterative refinement), `gepa` (single-reflection), `skill_generic`, and `skill_taxonomy` (HAWKEYE), spanning Ampere, Hopper, and Blackwell over six workloads, with CDNA4 adding `triton_fla`.

Profilers. NVIDIA uses `ncu -set full -launch-skip 3 -launch-count 1 python profile_kernel.py` and never runs `ncu` directly on `eval.py`, which performs 600+ launches and overflows the profiler database. MI350 uses `rocprof` with single-launch scripts mirroring NCU via `SQ_INSTS_VALU_MFMA_*`, `TCC_*`, and `SQ_WAVES_*`.

Reproducing a result. Each run is reconstructible from a per-run `config.json` plus the workspace setup (`kernel.cu` or `kernel.hip`, `<arch>-apis` headers, `validation` tests, `eval.py`), with `conversation.jsonl` and `trajectory.jsonl` preserving turn-by-turn tool calls and snapshots.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state our three contributions, namely the taxonomy of unit tests, the bootstrapping of coding agents with these tests, and the porting evaluation across architectures, vendors, and precisions, all of which are demonstrated in Section 4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 6 discusses HAWKEYE's limitations (the requirement of expert-authored unit tests per architecture, dependence on workload quantization tolerance for low-precision results, compounding precision loss forcing mixed precision in attention variants, and remaining gaps on heavily vendor-tuned workloads). Section 4 notes that HAWKEYE does not exceed cuDNN-backed SDPA on standard attention. Detailed failure-mode analysis of hardware-aware coding agents is in Appendix B.2 and Appendix D.2.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not include theoretical results.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 3 specifies the hardware platforms, six workloads with exact problem sizes, the four agent baselines, the agent harness with backbone model and turn budget, and the correctness and performance metrics, while Appendix G details prompt templates, evaluation pipeline, workspace structure, and per-baseline experimental setups. We will also release the taxonomy unit tests, agent prompts, and evaluation harness.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide an anonymized code repository linked in the paper containing the taxonomy unit tests, reference solution kernels, agent prompts, and the evaluation harness, along with instructions to reproduce the main experimental results across the four GPU architectures and precision formats reported in the paper.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: All training and evaluation details, including hardware platforms, workload configurations, agent backbone selection, turn budget, and metric definitions, are reported in Section 3, with additional infrastructure details in Appendix G.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: Performance is reported as the median over 100 runs from Triton's `do_bench`. We report the mean, standard deviation, and range over five independent agent trajectories for causal Linear Attention on MI350 and B200 (Section 4.3), but not for every cell, due to the compute cost of running additional trajectories per cell.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Each HAWKEYE optimization trajectory runs approximately 100 productive turns of the OpenHands loop, with every turn issuing one shell action (compile, execute, or profile) on a dedicated A100, H100, B200, or MI350 GPU and one inference call against the Gemini 3.1 Pro backbone. A complete trajectory takes about 40 minutes of wall time and costs about $4–5 in agent inference. Total GPU spend over the project, including all baselines, failed iterations, and re-measurements across the four architectures, six workloads, two precision tiers, four agent baselines, and the preliminary model-selection sweep, was $14,629, or roughly $287 per released kernel across the 51 kernels.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The research conforms with the NeurIPS Code of Ethics in every respect.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: HAWKEYE can lower the human porting tax for emerging accelerators and accelerate day-0 software support for new GPU architectures, expanding access to high-performance kernels beyond a small set of expert authors. We note potential negative impacts as well. Cheaper kernel optimization can induce additional compute demand and increase aggregate energy consumption despite per-kernel efficiency gains, and faster day-0 support for emerging accelerators is dual-use in that it equally accelerates safety-positive and safety-negative model training pipelines.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release high-risk models or scraped datasets, only GPU kernels and unit tests for hardware-aware kernel optimization.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All external libraries and baselines are properly cited, including PyTorch, Triton, OpenHands, the Flash Linear Attention library, Hipify, KernelBench, NVIDIA CUTLASS, and the HuggingFace CUDA kernels project, and we use them under their respective open-source licenses.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The HAWKEYE taxonomy of unit tests and reference solution kernels are documented in Section 2, with concrete code examples and architecture-specific differences in Appendix A and Appendix A.4, and we submit them as supplementary material with accompanying README and instructions.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are used as the underlying coding agent backbone in our methodology, and Section 3 describes the model selection process and the agent harness used in all experiments. The authors designed the taxonomy of unit tests, hand-authored every reference solution kernel, built the evaluation harness, and analyzed the agent trajectories that ground our findings.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.