HIPKITTENS: FAST AND *Furious* AMD KERNELS

William Hu [1] **Drew Wadsworth** [1] **Sean Siddens** [2] **Stanley Winata** [2] **Daniel Y. Fu** [3] **Ryan Swann** [2]
Muhammad Osama [2] **Christopher Ré** [1] **Simran Arora** [1]

ABSTRACT

AMD GPUs offer state-of-the-art compute and memory bandwidth; however, peak performance AMD kernels are written in raw assembly. To address the difficulty of mapping AI algorithms to hardware, recent work proposes C++ embedded and PyTorch-inspired domain-specific languages like THUNDERKITTENS (TK) to simplify high performance AI kernel development on NVIDIA hardware. We explore the extent to which such primitives — for explicit tile-based programming with optimized memory accesses and fine-grained asynchronous execution across workers — are NVIDIA-specific or general. We provide the first detailed study of the programming primitives that lead to performant AMD AI kernels, and we encapsulate these insights in the HIPKITTENS (HK) programming framework. We find that tile-based abstractions used in prior DSLs generalize to AMD GPUs, however we need to rethink the algorithms that instantiate these abstractions for AMD. We validate the HK primitives across CDNA3 and CDNA4 AMD platforms. In evaluations, HK kernels compete with AMD's hand-optimized assembly kernels for GEMMs and attention, and consistently outperform compiler baselines. Moreover, assembly is difficult to scale to the breadth of AI workloads; reflecting this, in some settings HK outperforms all available baselines by $1.2 - 2.4\times$ ($d = 64$ attention, GQA backwards, memory-bound kernels). HK demonstrates a portable tile-based programming model that abstracts over hardware-specific implementation details. These findings help pave the way for a single, tile-based software layer for high-performance AI kernels across GPU vendors. HIPKITTENS has been productionalized in AITER and is released at: https://github.com/HazyResearch/HipKittens.

1 INTRODUCTION

While AI has largely used a single hardware vendor in the past (Krizhevsky et al., 2012; AI@Meta, 2024; OpenAI et al., 2025), AMD GPU hardware now offer state-of-the-art peak compute and memory bandwidth (Table 2). However, the lack of mature software support creates a hardware lottery ("CUDA moat") (Sara Hooker, 2021; SemiAnalysis, 2024). Peak-performance AMD kernels are *written in raw assembly* by a handful of experts (i.e., the AITER library (AMD, 2025a)), which is slow to scale to the breadth of AI workloads. For instance, AITER and PyTorch Llama GQA backwards achieve just 30% and 24% of SoTA performance respectively on AMD MI355X GPUs (Section 4).

Developing NVIDIA kernels also required painstaking effort a few years ago. For instance, using low level CUDA/CUTLASS, it took *two years* between the H100 GPU's release and the release of peak performance open-source attention kernels (Shah et al., 2024). Compilers like Triton (Tillet et al., 2019) are simpler to use, but sacrifice performance and struggle to quickly support new hardware features (Spector et al., 2024; Triton, 2025). AI-designed kernels are showing early promise (Baronio et al., 2025; Lange et al., 2025), but current models also struggle to use new hardware features (Ouyang et al., 2025) and are susceptible to reward hacking (Baronio et al., 2025). Recently, lightweight C++ embedded DSLs like THUNDERKITTENS (TK) (and successors like CuTe DSL (NVIDIA, 2025b) and Gluon (Zhou et al., 2025)) consider simplifying development by encoding kernel design in terms of a small set of opinionated primitives that give the developer full control:

1. **Tiles.** The basic data type is a *tile* with optimized memory access patterns. TK exposes lightweight PyTorch-inspired bulk compute operators (`mma`, `exp`, etc.) over tiles, wrapping PTX. Tiles help developers explicitly manage data at each level of the GPU memory hierarchy.

2. **Overlapping.** A few basic kernel patterns help developers achieve high occupancy, or schedule workers (*waves* on AMD, *warps* on NVIDIA) onto the different hardware execution units. Modern NVIDIA kernels have consolidated around wave specialization (producer-consumer) scheduling patterns (Shah et al., 2024; Spector et al., 2024; 2025; Zhang et al., 2025; Wang et al., 2025).

[1]Department of Computer Science, Stanford University [2]Advanced Micro Devices, Inc. [3]University of California, San Diego. Correspondence to: William Hu <willhu@stanford.edu>, Simran Arora <simarora@stanford.edu>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

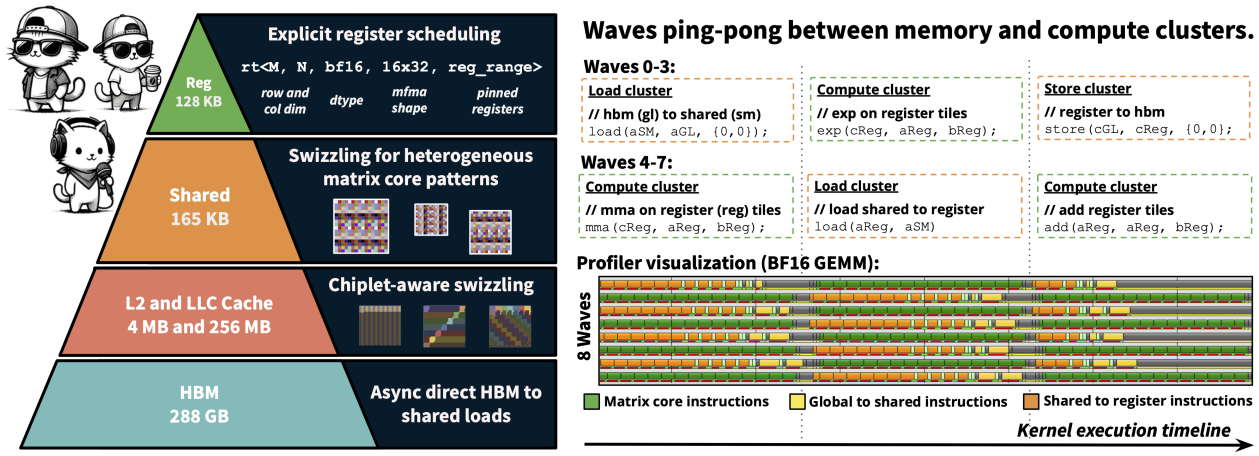


Figure 1. We study whether existing tile based programming primitives suffice for AMD kernels, or whether entirely new primitives are needed. Our study led to HIPKITTENS: a minimal and opinionated set of primitives for fast and *furious* AMD kernels. HK introduces a general 8-WAVE PING-PONG SCHEDULE to overlap compute and memory, programmer controlled register allocation, and efficient shared memory and chiplet-aware swizzling algorithms to enable a suite of high performance AMD AI kernels. In the profiler visualization, each row corresponds to a wave and two consecutive waves occupy the same SIMD. While one wave executes memory instructions, the other executes compute instructions.

3. **Grid scheduling.** By assigning work to thread blocks in the appropriate order, a developer can maximize the reuse of non-programmable cache memory.

Our work asks whether entirely new programming primitives are needed to simplify AMD kernel development, or whether existing primitives suffice. Ideally, we would have a simple framework that helps developers write a breadth of high performance kernels. Our exploration led to HIPKITTENS (HK), a minimal collection of C++ embedded programming primitives for AMD:

1. **Optimized access patterns for programmable GPU memory.** Careful register memory management is critical for peak performance kernels. HK retains the tile data structures of prior DSLs to help developers manage memory (Spector et al., 2024). However, optimizing the tiles for AMD introduces new challenges.

   Compilers such as Triton and HIPCC routinely impact the kernel developer's ability to carefully schedule register allocations and lifetimes (Sec. 3.2). For instance, HIPCC prevents the HIP developer from using certain types of registers (AGPRs) as input operands to matrix instructions. [1] We thus introduce a way for the developer to *bypass* the compiler altogether, and explicitly pin the registers belonging to each tile.

   As for memory access patterns, different NVIDIA matrix instruction shapes are all built from the same underlying *core matrix* structure, which makes it easy to use a single tile swizzle strategy for all shapes as in TK and Linear Layouts (Zhou et al., 2025). AMD matrix instructions lack this compositional structure, leading to an explosion in the number of tile layouts. Further, the shared memory bank structure and order in which threads in a wave run differ based on the memory instruction on AMD (Sec. 3.2). HK handles this complexity for the developer.

2. **Schedules to overlap compute and memory.** Ideally we would have simple, reusable patterns for scheduling the compute and memory within kernels that generalize across AI workloads. The wave specialization pattern dominates NVIDIA kernels and DSLs; *producer waves* handle memory operations while *consumers* execute bulk compute operations over large tiles. However, we find that this pattern underperforms on AMD CDNA3 and CDNA4 GPUs due to architectural differences—AMD's static register allocation means that producer waves consume registers without contributing to computation, which limits the output tile size that gets computed per thread block and thus the kernel's arithmetic intensity. On the MI355X, wave specialization achieves just 80% of peak BF16 GEMM performance (Tab. 2).

   We identify two alternate scheduling patterns that consistently achieve peak AMD performance: 8-WAVE PING-PONG and 4-WAVE INTERLEAVE, which tradeoff programmability with performance (Tab. 3). The 8-WAVE pattern employs two waves per SIMD that alternate between compute and memory roles, each executing large bulk tile operations (Fig. 1), while the 4-WAVE pattern employs one wave per SIMD and finely interleaves in-

[1] AMD CDNA includes 512 registers per SIMD, which are evenly partitioned across waves co-located on the SIMD. For kernels with a single SIMD per wave, the hardware splits the registers into 256 vector general-purpose registers (VGPRs) and 256 accumulator registers (AGPRs).

SPEC	NVIDIA B200 SXM5	AMD MI355X OAM
BF16 MATRIX / TENSOR	2.2 PFLOPS	2.5 PFLOPS
MXFP8 MATRIX / TENSOR	4.5 PFLOPS	5.0 PFLOPS
MXFP6 MATRIX / TENSOR	4.5 PFLOPS	10.1 PFLOPS
MXFP4 MATRIX / TENSOR	9.0 PFLOPS	10.1 PFLOPS
MEMORY CAPACITY	180 GB	288 GB
MEMORY BANDWIDTH	8.0 TB/S	8.0 TB/S

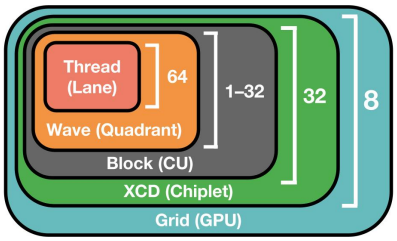


Figure 2. **Hardware overview.** (Left) Peak memory and compute speeds for the latest generation GPU platforms (AMD, 2025b; NVIDIA, 2025a). (Right) Diagram of the AMD GPU software and hardware hierarchy.

structions over small tile sizes. Remarkably, the simple 8-WAVE pattern is sufficient to match AMD's hand-optimized assembly kernels across BF16 GEMM, FP8 GEMM, and attention forward, and outperforms baselines by $1.8\times$ on GQA non-causal backward.

3. **Optimized access patterns for non-programmable GPU memory.** Chiplet architectures are becoming the dominant path to GPU scaling—NVIDIA Blackwell uses 2 chips, AMD MI355X uses 8—but existing frameworks ignore their hierarchical cache structures, leaving performance untapped. Each AMD CDNA4 chiplet contains 32 processors with private L2 cache, while all chiplets share a last-level cache (LLC) between L2 and global memory. These hierarchical caches have orthogonal preferences for how work is parallelized across thread blocks. Table 4 demonstrates an instance where a naive row-major ordering for allocating work to thread blocks achieves only a 36% L2 hit rate for BF16 GEMMs. We show that optimizing solely for L2 reuse (e.g., to 79% hit rate) degrades LLC performance and overall bandwidth. HK introduces an algorithm that models both cache levels when scheduling thread blocks. This improves performance by 19% over the naive row-major baseline (Tab. 4).

Evaluation. We validate HIPKITTENS on AMD CDNA3 MI325X and CDNA4 MI355X GPUs. On the most widely used and optimized workloads in AI, HK competes with or outperforms all AMD baselines (BF16 FP8 GEMM, GQA/MHA Attention forward and backward, RoPE, LayerNorm). HK outperforms all available AMD baselines on average, including AMD's *hand-optimized kernels that have been written in raw assembly*. However, assembly is not a scalable method for kernel development and leaves many important AI workloads unsupported — in such settings (e.g., some attention shapes, GQA backwards, memory bound kernels), HK outperforms the available AMD baselines by $1.2-10\times$. Furthermore, HK consistently outperforms compiler approaches (e.g., by up to $3\times$ the Triton BF16 GEMM and $2\times$ the Mojo MHA forwards).

We contribute (1) principles for writing high performance AMD kernels, (2) HK, a collection of opinionated C++ programming primitives for the AI community, and (3) a suite of performant AMD kernels. We further show that the tile primitives proposed in the TK DSL transfer to AMD, providing evidence that one unified and performant programming model is possible across AI accelerators. Scaling kernel support across *multiple silicon platforms* is key to unlocking the "compute capacity needed to achieve AI's full potential" (OpenAI, 2025). We hope this work helps open AI's hardware landscape.

2 BACKGROUND

In this section, we provide background on AMD GPU hardware in Section 2.1 and discuss related work in Section 2.2. Section B provides an extended discussion of related work.

2.1 GPU fundamentals

GPU kernels are small programs that load data, perform work on it, and write the results back out to memory. We generally adopt AMD terminology in this work and provide a mapping between AMD and NVIDIA terminology in Appendix A.

1. **Compute hierarchy.** Kernels are executed by tens of thousands of threads across hundreds of processors, called "compute units" (CUs). CUs organize their hardware resources in 4 "single instruction, multiple data" (SIMD) units. Threads are arranged in a hierarchy: threads are the smallest units of execution, "waves", groups of 64 threads, execute in lockstep on individual SIMDs, and "thread blocks", groups of waves, are jointly scheduled on the CUs. AMD MI355X GPUs contain 256 CUs organized into 8 accelerator complex dies (XCDs) of 32 CUs in a chiplet layout.

2. **Memory hierarchy.** Memory is organized in a hierarchy: small amounts of quick-access and large amounts of slow-access memory. A single SIMD contains 512 32-bit vector registers (for a total of 512KB per CU). Each CU has an L1 cache and shared memory that can be accessed by multiple waves in the same thread block. Each XCD shares a non-programmable 4MB L2 cache. All CUs share a large, slow global memory (HBM) and a last level cache (LLC) sits between L2 and HBM.

3. **Occupancy.** Threads execute instructions on physical execution units (ALU, FMA, matrix cores), which are

specialized for different types of compute. Instructions performed by these units each have a fixed issue latency and limited amount of bandwidth. Different waves can occupy different units simultaneously to avoid saturating any single unit. Each unit imposes different constraints on the memory layouts, or the mapping of logical data elements to physical thread ownership (AMD, 2025a).

Software overview. Developers can write kernels at different levels of the software stack. Raw assembly provides maximal control over register usage and instruction selection and ordering. CUDA / HIP C++ gets compiled (via NVCC, HIPCC) to assembly, and the compiler may introduce its own instruction reordering and register lifetime tracking. LLVM accepts compiler hints, which let the developer guide the compiler's behavior. Some compilers expose high level interfaces on top of C++ (e.g., Python (Paszke et al., 2019), Triton (Tillet et al., 2019)).

2.2 Related work

Currently, peak-performance AMD kernels finely interleave compute and memory instruction issues in raw assembly (see the AITER and Composable Kernel libraries) (AMD, 2025a;b). In contrast, to simplify and accelerate the kernel development process, the AI community recently adopted bulk programming operators over optimized *tile* primitives for kernel development, as proposed in THUNDERKITTENS (Spector et al., 2024) [2] and successors (e.g., CuTe DSL (NVIDIA, 2025b)[3], Gluon (Zhou et al., 2025)[4]). However, these existing C++ based DSLs only run on NVIDIA GPUs, wrapping PTX and CUDA. Compiler libraries such as Triton, TileLang, and Mojo are built on top of LLVM/MLIR (Lattner et al., 2020; Lattner & Adve, 2004; LLVM Compiler Infrastructure, 2025) and can compile for AMD GPUs. However, these works have neither provided reusable principles or primitives for AMD, nor released comprehensive suites of high performance AMD kernels. For instance, Mojo's MHA kernel suffers from expensive bank conflicts and achieves just 50% of the peak kernels' performance on the MI355X. [5]. HIPKITTENS provides the first systematic set of AMD AI kernel primitives towards opening up the hardware landscape.

[2] https://github.com/HazyResearch/ThunderKittens (May 2024)

[3] https://docs.nvidia.com/cutlass/media/docs/pythonDSL/cute_dsl.html (Sept 2025)

[4] https://github.com/triton-lang/triton/tree/main/python/tutorials/gluon (June 2025)

[5] Measured using `rocprofv3 --pmc SQ_LDS_BANK_CONFLICT,SQ_INSTS_LDS --output-format csv --output-file profiles_3 -d out -- mojo bench_mha.mojo` at https://github.com/modular/modular/tree/main/max/kernels/benchmarks/gpu in the nightly Modular build on a MI355X GPU on 11/06/2025.

3 HIPKITTENS

This section describes HIPKITTENS (HK), a framework for programming AI kernels on AMD GPUs. HK builds on the THUNDERKITTENS framework (Spector et al., 2024), which uses C++ embedded tile-based programming primitives to simplify high-performance and flexible AI kernel development (discussed in Section 3.1). We describe HK's principles for optimizing the access patterns of programmable GPU memory in Section 3.2, maximizing occupancy in Section 3.3, and optimizing the access patterns of non-programmable cache memory in Section 3.4.

3.1 Tile programming interface

Like existing kernel frameworks, HK adopts tiles as the basic data structure and provides a suite of optimized operators over tiles. The tile design and suite of operators is heavily inspired by PyTorch and NumPy (Paszke et al., 2019; Harris et al., 2020), given their familiarity to the AI community.

- **Memory.** The developer can initialize tiles in register or shared memory. A tile is parametrized by a `dtype` (FP32, BF16, FP16, FP8, FP6), `rows`, `columns` and a `layout` (row or column major). Tile rows and columns are restricted to be a multiple of the matrix core shape. HK provides operators to `load` and `store` tiles across different levels of the GPU memory hierarchy.
- **Compute.** HK provides a suite of bulk compute operators over tiles, inspired by the set of operators in PyTorch (e.g., `mma`, `exp`, `add`). The functions are lightweight and do not add overhead as they directly wrap raw AMD CDNA assembly/HIP (NVIDIA PTX/CUDA for TK).

Given these familiar programming primitives, HK automatically optimizes the memory access patterns for tiles. Memory management on AMD GPUs raises key challenges at each level of the hierarchy, discussed next.

3.2 Optimizing programmable memory access

We now discuss the specifics of HIPKITTENS tiles.

3.2.1 Developer-controlled register scheduling

Careful register management is critical for high performance. However, compilers either prevent (e.g., Triton) or impede (e.g., in the HIPCC compiler) the developer's ability to maximally control register allocations.

For instance, in kernels with a single wave per SIMD, the AMD hardware splits the SIMD's 512 registers into 256 vector general-purpose registers (VGPRs) and 256 accumulator registers (AGPRs). However, while the hardware does support using AGPRs as input to matrix core instructions, HIPCC does not. For workloads that involve both matrix and vector operations (e.g., attention backwards), kernels compiled via HIPCC would need to generate redun-

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Figure 3. **Swizzle pattern for a 16x32 tile of BF16s.** Shared memory on AMD CDNA4 GPUs have different banking behavior depending on the instruction. `ds_read_b128` accesses shared memory through 64 banks, each 32-bits wide, and correspond the individual cells and numbers in the figure. The shaded cells represent banks that are accessed by the first phase of a `ds_read_b128` instruction for a 16x32 row layout register tile. On the left is an unswizzled layout suffering from 2-way bank conflicts. On the right is a swizzled layout with no bank conflicts. The swizzle applied here swaps the first 8 columns with the last 8 starting from the 8th row. This swizzling strategy simultaneously enables bank-conflict free accesses from column-major reads using `ds_read_b64_tr_b16`. Details can be found in D.1.

dant `v_accvgpr_read` instructions that move data from AGPRs to VGPRs prior to issuing matrix instructions.

Table 1. **EXPLICIT REGISTER SCHEDULING enables increased developer control.** A 4-wave MHA non-causal backwards kernel implemented in HIP underperforms AMD's raw assembly kernel (AITER) due to compiler limitations. We match AITER by bypassing the compiler and pinning register tiles to explicit registers. We use batch size 16, heads 16 and head dim 128.

METHOD	SEQ. LENGTH	TFLOPS
HK	4096	855
HK WITH PINNED REGISTERS	4096	**1024**
AMD ASSEMBLY (AITER)	4096	1018
HK	8192	909
HK WITH PINNED REGISTERS	8192	1091
AMD ASSEMBLY (AITER)	8192	**1169**

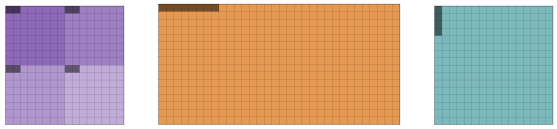
EXPLICIT REGISTER SCHEDULING. The compiler constraints motivate a feature in HK that gives developers the ability to fully control register scheduling. The developer pins the registers belonging to each tile, rather than letting HIPCC manage the registers. By bypassing the compiler, the developer can use AGPRs as inputs to matrix instructions, resulting in our SoTA-level backwards attention kernel (Tab. 1). The interface for programming with pinned register tiles exactly matches that of using standard compiler-managed register tiles. We leave both options so developers can choose the level of control they want.

3.2.2 *Tiles for heterogeneous matrix core shapes*

AI kernels use different matrix core instruction shapes (MxNxK), depending on the workload properties, in order to carefully manage register pressure. However, it is challenging to use multiple shapes on AMD GPUs.

Matrix layout complexity. Recall that GPU matrix instructions impose rules as to which thread owns each data element in its registers. Further, shared memory accesses result in bank conflicts if multiple threads in a wave attempt to access the same bank simultaneously. Waves (and NVIDIA warps) execute shared memory accesses in *phases*; i.e., a *subset* of threads in a wave accesses shared memory concurrently (Forum, 2023).

The complexity of AMD matrix layouts relative to NVIDIA layouts impacts the access patterns at each level of the GPU memory hierarchy. First, NVIDIA matrix instructions use a *regular* pattern (Fig. 4a); all shapes are composed from an underlying 16×16 *core matrix* building block that is stamped out multiple times depending on the overall matrix instruction shape. Thus, prior frameworks like TK (Spector et al., 2024) and Linear layouts (Zhou et al., 2025), can use a unified swizzling strategy that generalizes across matrix shapes. Meanwhile, each AMD matrix instruction uses an entirely different layout without a similar underlying structure.



(a) NVIDIA core matrices (b) AMD 16×16×32 MFMA A/B matrix (c) AMD 16×16×32 C/D matrix

Figure 4. Matrix layouts on NVIDIA and AMD GPUs. The shaded cells in each matrix represent elements owned by thread 0.

Second, NVIDIA instructions sequentially assign threads to phases (e.g., threads 0-7 in phase one, 8-15 in phase two). Meanwhile on AMD, the phases are non-sequential and differ based on the memory instruction. [6]

[6] For e.g., the threads in a wave execute the `ds_read_b128` instruction in 4 phases and load data from 64 shared memory banks, each 4-bytes wide, while a `ds_read_b96` executes over 8 phases and loads from 32 banks. The phases are undocumented in the CDNA ISA so we create a solver to determine them, and document the phases in Tab. 7.

Optimized tile memory. We discuss how HK abstracts away this complexity from the AMD kernel developer:

1. **Register.** By default, register tiles in HK default to using the smallest MFMA instruction since this provides maximal scheduling control as highlighted in Section 3.3. However, for the edge case kernels that use alternate sizes, HK lets the developer parameterize the desired register tile by the MFMA instruction shape.

2. **Shared.** On AMD GPUs, it is not possible to use a single swizzle pattern for all layouts (a simple counter-example is provided in Appendix D.1). While we could implement unique swizzle patterns for every matrix layout, this adds code complexity. Instead, we identify the layouts that commonly co-occur and support swizzle patterns that are bank conflict free for these instances. Figure 3 shows one such swizzle that is bank conflict free for both the 16×32 row layout and column layout load.

3. **Global.** AMD GPUs support direct asynchronous HBM to shared memory loads. Like TMA, these loads bypass the register file. The instruction takes as input per-thread addresses in HBM from which each thread will read data. While DSLs like TK directly swizzle the shared memory addresses, swizzling shared memory on AMD is instead accomplished by swizzling on the HBM addresses.

3.3 Overlapping compute and memory utilization

We study the principles for scheduling instructions within AMD AI kernels and identify two high-performance patterns that lead to peak utilization across diverse workloads.

Current approaches and their limitations. State-of-the-art AI kernels and DSLs have consolidated around *wave specialization*—a pattern where specialized producer waves handle memory movement while consumer waves handle computation. This approach dominates in NVIDIA implementations including FlashAttention-3 (Shah et al., 2024), COMET for MoE (Zhang et al., 2025), and GEMMs (Bikshandi & Shah, 2023), and kernel DSLs like TK (Spector et al., 2024) and TileLang (Wang et al., 2025). In this paradigm, waves occupy specific hardware units for long periods of time, so they can issue bulk operations over large tile primitives. This tile-based programming makes the code size compact and readable.

However, wave specialization struggles to generalize to modern AMD devices due to fundamental architectural differences. Instead, state-of-the-art AMD kernels (AITER (AMD, 2025a), CK (AMD, 2025b)) resort to raw assembly to finely interleave instruction issues—an approach orthogonal to tile-based programming. While it might seem that AMD requires bespoke schedules for each AI workload, we identify simple general principles that achieve high performance across diverse applications.

# P / # C	MFMA SHAPE	OUTPUT	TFLOPS
HK 4 / 8	$16 \times 16 \times 32$	128×256	893
HK 4 / 12	$16 \times 16 \times 32$	192×256	1278
HK 0 / 8	$16 \times 16 \times 32$	192×256	1281
HK 0 / 8	$16 \times 16 \times 32$	256×256	**1610**
TK	$256 \times 256 \times 16$	256×256	1538
CUTLASS	$256 \times 256 \times 16$	256×256	1570

Table 2. **Producer consumer comparisons.** We report results for a series of producer consumer BF16 GEMM kernels of shape $M = N = K = 8192$. We denote the number of producers and consumers as P and C respectively. We denote the underlying matrix instruction size, output tile size computed per thread block, and TFLOPS measured (500 iterations of warmup, 100 iterations of measurement on inputs from $\mathcal{N}(0, 1)$). AMD kernels run on an MI355X and NVIDIA kernels (TK, CUTLASS) on a B200.

3.3.1 *Wave specialization underperforms on AMD*

NVIDIA kernels implement wave specialization using dedicated memory access hardware (*tma*), asynchronous matrix multiplies which accept operands directly from shared or tensor memory (*wgmma*, *tcgnen05*), deep pipelines enabled by large shared memory per processor (B200 has 40% larger SRAM than AMD MI355X per processor), register reallocation (where the register-efficiency of TMA lets producers give their registers to consumers), and hardware synchronization primitives (*mbarriers*). AMD lacks these architectural features, changing the kernel design space.

To evaluate how these differences impact performance, we vary the synchronization mechanism, pipeline depth, and producer-consumer ratios (Tab. 2). Our experiments reveal two principles. We need to maximize the output tile size computed per thread block to increase the arithmetic intensity (operations per byte moved), and we need to maximize the pipeline depth to hide the latency of memory loads.

Peak performance open-source TK and CUTLASS profiler-selected GEMMs use wave specialization and an output tile size of 256×256 on the B200. [7] Our best AMD GEMM achieves comparable performance when computing a 256×256 output tile per thread block only when using *no wave specialization* (i.e., zero producers) and degrades as the number of producers increases (Tab. 2). This is because AMD hardware statically divides registers across all waves (AMD, 2025), meaning producers consume registers without contributing to output computation. This limits the usable output tile size when using wave specialization.

Tradeoffs. NVIDIA's larger shared memory enables the use of deep pipelines while using large matrix instruction shapes (e.g., $256 \times 256 \times 16$). However, AMD's smaller tensor core shapes (e.g., $16 \times 16 \times 32$) provide an alternative

[7] The profiler sweeps and tunes the suite of CUTLASS GEMMs, selecting the best one for the shape and dtype.

Table 3. **Scheduling patterns for AMD.** We identify two primary paradigms—8-WAVE and 4-WAVE—that generalize across workloads. Both patterns can leverage HK's tile primitives. We report the hot loop code size and TFLOPs, showing how these patterns trade off programmability and performance.

KERNEL	PATTERN	LOC	TFLOPS
FP8 GEMM	8-WAVE	48	3222
FP8 GEMM	4-WAVE	183	3327
MHA BACKWARDS	8-WAVE	331	894
MHA BACKWARDS	4-WAVE	989	1091

path to establish deep pipelines by using finer-granularity load and compute stages.

NVIDIA's matrix multiply instructions, which accept operands from shared or tensor memory, helps alleviate register pressure, and it may be surprising that AMD can match performance without this. However, AMD devices have a $2\times$ larger register file to compensate.

We further validate that using shared memory atomics instead of *mbarriers* adds negligible overhead; we find the 192×256 producer consumer kernel, which uses atomics, and non-wave specialized kernel, which is atomic-free, perform similarly, emphasizing that the output tile shape is the dominant factor impacting performance (Tab. 2).

3.3.2 *Performant scheduling patterns for AMD AI kernels*

AMD GPUs have four SIMD units per CU, and waves scheduled on the same SIMD can overlap compute and memory instructions. We identify two scheduling patterns that consistently achieve peak performance across AI workloads by exploiting this parallelism differently:

1. **8-WAVE PING-PONG (balanced workloads).** This pattern employs eight waves per thread block—two resident per SIMD. The waves are split into two groups of four, with each group containing one wave per SIMD. Within each SIMD, the two waves alternate the type of work each does: one issues only compute instructions while the other issues only memory instructions, and then they swap roles, flipping back and forth between compute and memory as shown in Figure 1. A conditional barrier such as the one in Figure 5 controls the alternation.

```
if (kittens::warpid() / 4 == 1) {
    __builtin_amdgcn_s_barrier();
}
```

Figure 5. **Conditional Barrier.** This control-flow primitive introduces the alternating pattern of work between waves resident on the same SIMD. In 8-WAVE PING-PONG, the conditional barrier causes half of the waves - one on each of the four SIMDs - to stall temporarily so that, at any given point in time, they are performing different work than the waves that were left unimpeded.

   This pattern excels when compute and memory durations are roughly balanced. A SIMD's compute wave executes matrix fused multiply add (MFMA) instructions while its paired memory wave prefetches the next data, hiding memory effectively.

2. **4-WAVE INTERLEAVE (imbalanced workloads).** This pattern places exactly one wave on each of the processor's four SIMDs. Each wave issues both compute and memory instructions in a carefully staggered sequence to maximize occupancy of the hardware units.

   This fine-grained pattern better saturates both MFMA and LDS pipelines when workloads are imbalanced (either compute-heavy or memory-heavy). The wave per SIMD can adapt its instruction mix dynamically.

These schedules tradeoff programmability and performance. HK lets developers use tile-based primitives to implement either of these patterns, albeit at different tile *granularities*. 8-WAVE PING-PONG allows for large tile primitives similar to the ones used in *wave specialization*. On the other hand, 4-WAVE INTERLEAVE requires developers to program with small base tile primitives, extending the code size due to finer grained instruction issues. This tradeoff is captured in Table 3. Surprisingly, we find that 8-WAVE is *sufficient* to match or outperform AMD's raw assembly kernels across BF16 GEMM, FP8 GEMM, and attention forwards workloads. On GQA non-causal attention backwards, our 8-WAVE kernel outperforms the baselines (PyTorch SDPA, CK, and AITER) by $1.8\times$, and our 4-WAVE kernel delivers an even larger $2.3\times$ speedup.

3.4 Optimizing the access patterns of non-programmable GPU memory

Modern GPUs—AMD and NVIDIA—are moving towards chiplet, rather than monolithic, architectures (e.g., Blackwell is comprised of two chips). This results in a disaggregated cache hierarchy, where distinct *clusters* of processors are attached to distinct slices of the GPU cache (see Figure 2). Here, we explore principles for disaggregated cache scheduling and introduce HK's algorithm for cache reuse.

Cost model. AMD devices use two types of caches – L2 and LLC – where cache misses have a worst case miss penalty of 300ns for the L2 cache and 500ns for the LLC cache. AMD devices assign clusters of 32 (CDNA4) or 38 (CDNA3) compute units to a cluster (accelerated complex die, or XCD), and include 8 clusters per GPU. The hardware scheduler assigns thread blocks to XCDs in round-robin order. The grid schedule, or order of work assigned to thread blocks, impacts the cache reuse and achieved bandwidth:

$$\begin{aligned}\text{Bandwidth} = \text{LLC Bandwidth } &\times \text{LLC Hit \%} \\ +\text{L2 Bandwidth } &\times \text{L2 Hit \%}\end{aligned} \tag{1}$$

Algorithm 1 `XCD swizzle for cache reuse on GEMMs`

Require: grid block indices $(b.x, b.y, b.z)$; grid dimensions $(g.x, g.y, g.z)$; number of XCDs `nXCD`; problem sizes M, N with tile sizes $\texttt{BLOCK}_M, \texttt{BLOCK}_N$; window height W, chunk size C

Ensure: remapped block indices $(b.x', b.y', b.z)$

```
blocks ← g.x × g.y                                   ▷ blocks per batch (a single b.z slice)
xy ← b.x + g.x × b.y                                 ▷ flatten (b.x, b.y) within the batch
blocks_per_cycle ← nXCD × C
limit ← ⌊blocks / blocks_per_cycle⌋ × blocks_per_cycle   ▷ largest full (nXCD × C)-aligned prefix
if xy > limit then
    xy ← xy                                          ▷ tail region: leave order unchanged
else
    xcd ← xy mod nXCD                                ▷ which XCD this block belongs to (round-robin)
    local ← ⌊xy / nXCD⌋                              ▷ local index after de-interleaving by XCD
    chunk_idx ← ⌊local / C⌋
    pos ← local mod C
    xy ← chunk_idx × blocks_per_cycle + xcd × C + pos
end if
num_rows ← M / BLOCK_M                               ▷ tile rows along M
num_cols ← N / BLOCK_N                               ▷ tile cols along N
tid_per_group ← W × num_cols                         ▷ one window (height W) across all columns
group_id ← xy / tid_per_group                        ▷ which window of rows
first_row ← group_id × W
win_h ← min(num_rows − first_row, W)                 ▷ tail-safe window height
ℓ ← xy mod tid_per_group                             ▷ local index within the window
row ← first_row + (ℓ mod win_h)                      ▷ fast index: go down within the column
col ← ℓ / win_h                                      ▷ slow index: move to next column after win_h rows
return (row, col, b.z)                               ▷ logical tile coordinates (+ batch)
```

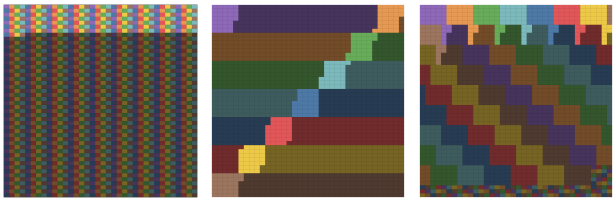
In a GEMM kernel ($D = AB + C$), each thread block computes a distinct tile of the output matrix D. When thread blocks are scheduled in naïve row-major order, cache reuse is suboptimal (≈55%) because blocks that share the same L2 cache often load different, non-overlapping tiles of A and B. Thus, their memory accesses fail to exploit spatial locality, leading to redundant data movement. This behavior is illustrated in Fig. 6a and Tab. 4 (Row 1). To mitigate this, we use two key principles to improve cache reuse:

1. **L2 Reuse.** Thread blocks mapped to the same XCD (and thus sharing an L2 cache) should cover a rectangular region of the output matrix—an "L2 tile." This layout ensures that consecutive blocks reuse both the same rows of A and the same columns of B. However, optimizing purely for L2 locality can cause each XCD to fetch disjoint portions of A and B, leading to redundant loads at the next cache level.
2. **LLC Reuse.** To further improve reuse at the last-level cache (LLC), we must coordinate accesses across XCDs. Ideally, the combined access footprint of all XCDs—the "LLC tile"—should overlap in both A and B. In other words, multiple XCDs should work on nearby or identical regions of the input matrices, so that shared data remains resident in the LLC.

By jointly optimizing these two principles, we can raise both L2 and LLC hit rates, leading to higher effective bandwidth (Figure 6c, Table 4, Row 3). For instance, Table 4 shows that an L2/LLC-aware schedule achieves up to 15% higher performance than the default grid order. The benefit is particularly pronounced when the output matrix width (in tiles) is coprime with the number of XCDs—for example, 57 tiles across 8 XCDs on an AMD MI355X—since the default schedule causes worst case reuse patterns (Tab. 4).

HIPKITTENS chiplet swizzling algorithm. To make cache-aware scheduling accessible to developers, HIPKITTENS provides a simple and tunable strategy for maximizing cache reuse across a wide range of GEMM problem sizes. Algorithm 1 implements this strategy in two steps:

1. **XCD grouping.** Flatten the 2D-grid into a linear sequence and remap block ID's such that chunks of C



(a) Row-major (b) XCD (W7, C216) (c) XCD (W5, C25)

BLOCK ORDER	L2 %	LLC %	MEM. BW	TFLOPS
MATRIX MULTIPLY (M=N=K=9216, MT 192x256x64)				
ROW-MAJOR	55%	95%	15.1 TB/s	1113
XCD ($W7/C216$)	79%	24%	14.9 TB/s	991
XCD ($W5/C25$)	75%	93%	18.3 TB/s	1145
MATRIX MULTIPLY (M=N=K=14592, MT 192x256x64)				
ROW-MAJOR	36%	76%	10.7 TB/s	900
XCD ($W8/C542$)	79%	7%	13.9 TB/s	980
XCD $W8/C64$	78%	55%	16.6 TB/s	1068

Table 4. **Chiplet swizzling for cache reuse.** Visualization of three different grid schedules for the output matrix of a $M = N = K = 9216$ BF16 GEMM. The color represents the XCD assignment for the first set of thread blocks scheduled across the GPU (256 CUs). Schedule 6a (Table Row 1) assigns blocks to the grid according to block ID. Schedules 6b (Table Row 2) and 6c (Table Row 3) apply Algorithm 1 with different window and chunk size parameters. Table 4 shows how these schedules trade off L2 and LLC reuse to gain performance. Figure 19a provides the corresponding visualization for the 14592 shape.

consecutive IDs are resident on the same XCD. This reduces cross-chiplet traffic.

2. **Hierarchical windowed traversal.** Instead of processing the grid row by row, we process it in vertical windows of height W. This has the effect of "folding" the input block ID space into rectangular tiles, optimizing L2 cache reuse.

The two parameters, W and C, control the trade-off between L2 and LLC reuse. Since L2 bandwidth is roughly $3\times$ higher than LLC bandwidth, W should be chosen to maximize L2 hit rate. On AMD MI355X, each XCD contains 32 CUs, and empirical results show that L2 tiles of shape 8×4 or 4×8 achieve the best hardware utilization. Tuning the chunk size C further improves LLC efficiency by coordinating access patterns across XCDs so that they operate on similar rows of the input matrix.

4 EXPERIMENTS

In this section, we validate that HK enables peak performance kernels, using simple and reusable tile-based primitives, across a breadth of AI operations.

Baselines. We compare to the best performing baseline kernels across PyTorch (compiled and SDPA), AITER (AMD, 2025a), Composable Kernel (AMD, 2025b), ROCm Library Triton (AMD, 2025), and HipBLASLT (AMD, 2025). We evaluate on both MI325 CDNA3 and MI355 CDNA4. We benchmark HK kernels in Python scripts using Python bindings (except FP8 where AMD PyTorch support remains experimental). For each kernel, we use 500 warmup runs and report the average TFLOPs/s performance over 100 runs over randomly generated input tensors from the standard normal distribution. All kernels are benchmarked in AMD's recently released beta Docker using ROCm 7.0 (`rocm/7.0-preview:rocm7.0_preview_pytorch_training_mi35x_beta`).

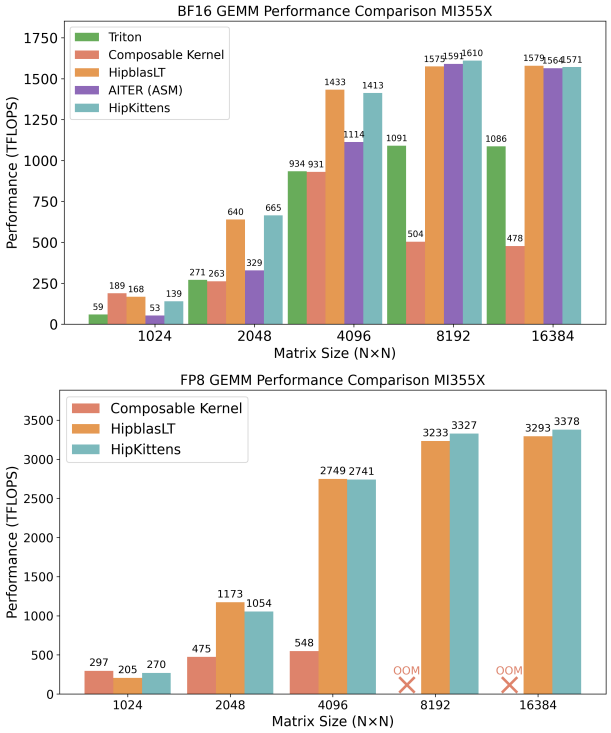


Figure 7. **GEMM.** We compare HK BF16 and FP8 GEMMs to the strongest available baselines.

HK provides a comprehensive suite of peak-performance AMD AI kernels, written using reusable tile-based abstractions. We also include code listings in Appendix E and additional results in Appendix C:

1. **BF16 and FP8 GEMM.** HK competes with the AMD baseline kernels that are written in assembly (AITER, HipBLASLT/PyTorch). HK outperforms the Triton compiler by $1.3 - 3.0\times$. Further, we obtain these results using a single 8-wave kernel schedule that generalizes across the evaluated problem shapes.

2. **Attention forwards.** We evaluate multi-head attention (MHA) and group-query attention (GQA) kernels in causal and non-causal settings, and for head dimen-

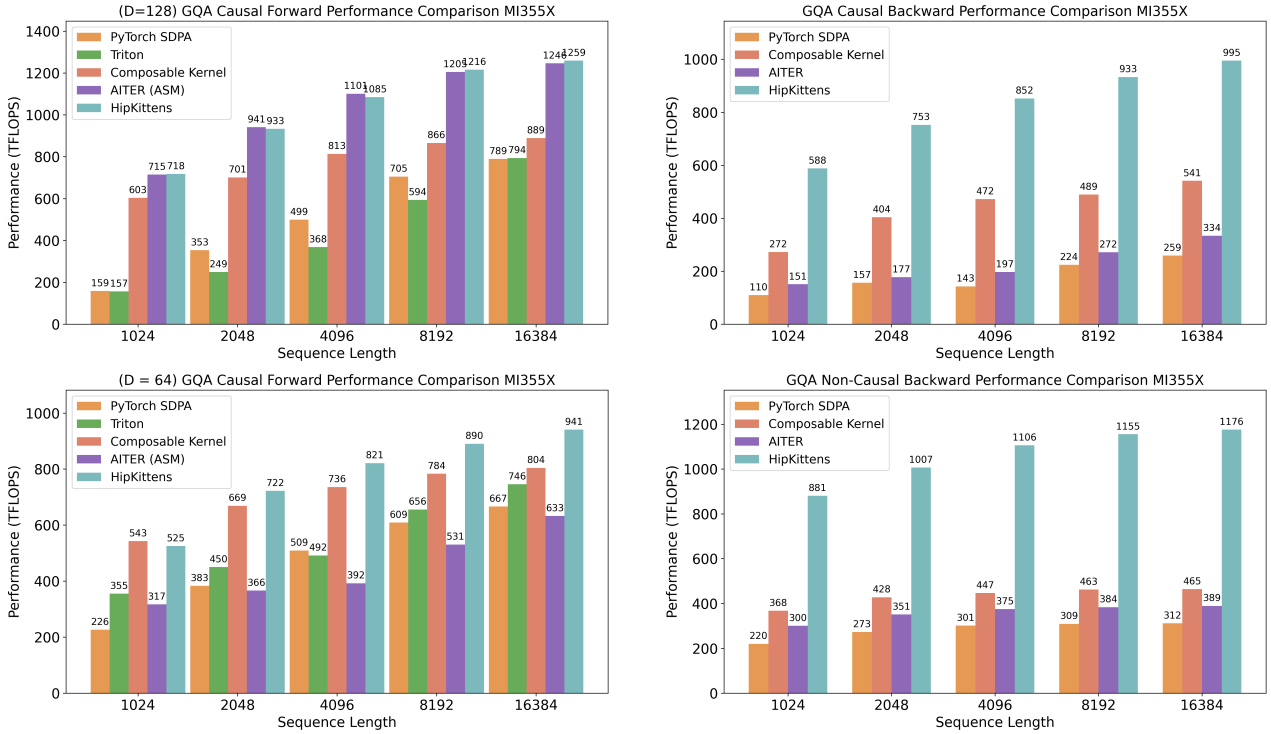


Figure 8. **Attention forwards.** We compare HIPKITTENS GQA and MHA (Figure 17) to the strongest available baselines. We use batch 16, query heads 64, key value heads 8, head dim 64 and 128.

Figure 9. **Attention backwards.** We compare HIPKITTENS GQA and MHA (Figure 16) to the strongest available baselines. We use batch 16, query heads 64, key value heads 8, and head dim 128.

sions 64 and 128. HK outperforms all available AMD baselines on average, including the AITER kernels, which are written in hand-optimized raw assembly by AMD engineers. HK is $1.0 - 2.1\times$ faster than AITER, $1.3 - 4.5\times$ faster than PyTorch (SDPA), $1.0 - 1.4\times$ CK, and $1.2 - 4.5\times$ Triton kernels in Figure 8.

HK's attention forward kernel uses an 8-WAVE PING-PONG. Within compute clusters, each wavefront interleaves online-softmax vector ops (max/subtract/exp2/accumulate) with MFMA instructions. Despite substantial scheduling and hardware differences between MI355X and NVIDIA B200, the kernel is competitive with FlashAttention-3 under comparable settings (Shah et al., 2024).

3. **Attention backwards.** Our GQA causal and non-causal backwards attention kernels outperform the baselines by $1.8 - 2.5\times$ across settings (Fig. 9). Our MHA kernels compete with the strongest available baselines, which are written in assembly (Fig. 16).

   Attention backwards is a notoriously register heavy workload. Our efficient HK kernel uses multiple MFMA instruction shapes ($16 \times 16 \times 32$ and $32 \times 32 \times 16$), different shared memory access patterns (e.g., both row and column layout loads to registers from the same shared tile), and explicit register pinning.

4. **Memory bound results.** We consider a fused dropout-residual-layernorm kernel (from prenorm Transformer architectures) and a rotary positional encoding kernel in Figure 10. HK outperforms both AITER and PyTorch compiled kernels by $1.1 - 2.2\times$ across settings.

The inconsistent performance of AMD libraries and the difficulty of scaling assembly-engineered kernels (e.g., reinforced by head dim. 64 attention and GQA non-causal backwards) reflects the value of having a simple set of kernel programming abstractions to accelerate AMD kernel development. Finally, to validate kernel stability, we use our kernels to pretrain Llama 1B (AI@Meta, 2024) and BERT 110M (Devlin et al., 2019) on the Pile (Gao et al., 2020), matching the perplexity of models trained using PyTorch and AITER after 10B tokens of training.

5 DISCUSSION AND CONCLUSION

Ideally, AI systems can leverage the full diversity of modern hardware. AMD CDNA4 GPUs offer state-of-the-art compute and memory bandwidth, but the "CUDA moat" limits adoption. While prior systems such as Triton aim for multi-silicon portability, our study shows that these compilers (and sometimes even C++ compilers) often fail to enable peak AMD performance.

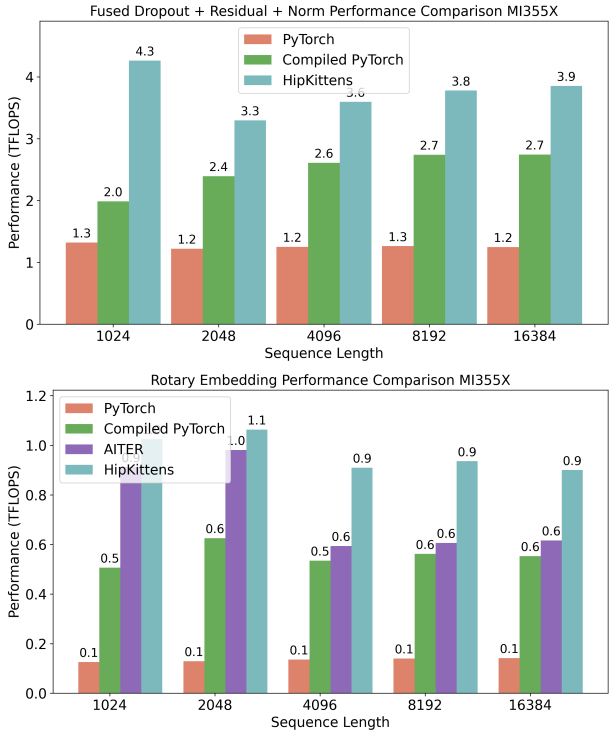


Figure 10. **Memory bound.** We compare HIPKITTENS fused dropout-residual-layernorm and rotary kernels to the strongest available baselines at batch 16, heads 16, and head dim 128.

This work provides the first systematic analysis of the principles that enable high-performance AMD AI kernels and introduces HIPKITTENS, a minimal set of C++ embedded programming primitives that capture those principles. Though the abstractions and front-end *interface*—tiles and PyTorch-inspired bulk operations over tiles—remains the same across NVIDIA and AMD, the instantiation of those abstractions—in terms of schedules, memory movement, and cache optimizations—differ due to fundamental hardware differences. We evaluate the ideas presented in HK by implementing a suite of representative AI workloads and find that we can achieve peak performance across them.

By codifying the principles for AMD kernels into composable, open abstractions, these findings move the community closer to the long-standing vision of a universal software stack that performs well across diverse hardware platforms.

6 ACKNOWLEDGEMENTS

We thank Simon Guo, Michael Zhang, Stuart Sul, Arya Tschand, Tom Papatheodore, the Hazy Research Lab and Stanford AI Lab for providing feedback on this work. We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF2247015 (Hardware-Aware), CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); US DEVCOM ARL under Nos. W911NF-23-2-0184 (Long-context) and W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under Nos. N000142312633 (Deep Signal Processing); Stanford HAI under No. 247183; NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), and members of the Stanford DAWN project: Meta, Google, and VMWare. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, or the U.S. Government.

7 CONTRIBUTIONS

WH, DW, and SA designed and implemented HIPKITTENS, kernels, micro experiments, and baselines. SS designed the cache reuse strategy. WH and SA wrote the manuscript. SW, DF, RS, MO, CR advised and SA supervised the project.

REFERENCES

Helion, 2025. URL https://github.com/pytorch/helion.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

AMD. AITER, 2025a. URL https://github.com/ROCm/aiter.

AMD. Composable Kernel, 2025b. URL https://github.com/ROCm/composable_kernel.

AMD. Rocm hardware features, 2025. URL https://rocm.docs.amd.com/projects/HIP/en/docs-develop/reference/hardware_features.html.

AMD. AMD Matrix Instruction Calculator, 2025a. URL https://github.com/ROCm/amd_matrix_instruction_calculator.

AMD. AMD Instinct™ MI355X GPUs, 2025b. URL https://www.amd.com/en/products/accelerators/instinct/mi350/mi355x.html.

AMD. Rocm libraries, 2025. URL https://github.com/ROCm/rocm-libraries.

Baronio, C., Marsella, P., Pan, B., Guo, S., and Alberti, S. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

Bikshandi, G. and Shah, J. Developing cuda kernels for accelerated matrix multiplication on nvidia hopper architecture using the cutlass library, 2023. URL https://research.colfax-intl.com/wp-content/uploads/2023/12/colfax-gemm-kernels-hopper.pdf.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT 2019*, 2019.

Forum, N. D. How to understand the bank conflict of shared memory, 2023. URL https://forums.developer.nvidia.com/t/how-to-understand-the-bank-conflict-of-shared-mem/260900.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, September 2020.

Kim, Y. J., Henry, R., Fahim, R., and Awadalla, H. H. Who says elephants can't run: Bringing large scale moe models into cloud scale production. *Proceedings of the Third Workshop on Simple and Efficient Natural Language Processing (SustaiNLP)*, 2022.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, 2012.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. Towards robust agentic cuda kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: A compiler infrastructure for the end of moore's law, 2020. URL https://arxiv.org/abs/2002.11054.

LLVM Compiler Infrastructure. User Guide for AMDGPU Backend, 2025. URL https://llvm.org/docs/AMDGPUUsage.html.

NVIDIA. Cuda templates for linear algebra subroutines, 2017. URL https://github.com/NVIDIA/cutlass.

NVIDIA. NVIDIA CuTe, 2024. URL https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/00_quickstart.md.

NVIDIA. Nvidia blackwell architecture technical brief. https://resources.nvidia.com/en-us-blackwell-architecture, 2025a.

NVIDIA. nvidia-cutlass-dsl, 2025b. URL https://pypi.org/project/nvidia-cutlass-dsl/.

OpenAI. AMD and OpenAI announce strategic partnership to deploy 6 gigawatts of AMD GPUs, 2025. URL https://openai.com/index/openai-amd-strategic-partnership/.

OpenAI, :, Agarwal, S., Ahmad, L., Ai, J., Altman, S., Applebaum, A., Arbus, E., Arora, R. K., Bai, Y., Baker, B., Bao, H., Barak, B., Bennett, A., Bertao, T., Brett, N., Brevdo, E., Brockman, G., Bubeck, S., Chang, C., Chen, K., Chen, M., Cheung, E., Clark, A., Cook, D., Dukhan, M., Dvorak, C., Fives, K., Fomenko, V., Garipov, T., Georgiev, K., Glaese, M., Gogineni, T., Goucher, A., Gross, L., Guzman, K. G., Hallman, J., Hehir, J., Heidecke, J., Helyar, A., Hu, H., Huet, R., Huh, J., Jain, S., Johnson, Z., Koch, C., Kofman, I., Kundel, D., Kwon, J., Kyrylov, V., Le, E. Y., Leclerc, G., Lennon, J. P., Lessans, S., Lezcano-Casado, M., Li, Y., Li, Z., Lin, J., Liss, J., Lily, Liu, Liu, J., Lu, K., Lu, C., Martinovic, Z., McCallum, L., McGrath, J., McKinney, S., McLaughlin, A., Mei, S., Mostovoy, S., Mu, T., Myles, G., Neitz, A., Nichol, A., Pachocki, J., Paino, A., Palmie, D., Pantuliano, A., Parascandolo, G., Park, J., Pathak, L., Paz, C., Peran, L., Pimenov, D., Pokrass, M., Proehl, E., Qiu, H., Raila, G., Raso, F., Ren, H., Richardson, K., Robinson, D., Rotsted, B., Salman, H., Sanjeev, S., Schwarzer, M., Sculley, D., Sikchi, H., Simon, K., Singhal, K., Song, Y.,

Stuckey, D., Sun, Z., Tillet, P., Toizer, S., Tsimpourlas, F., Vyas, N., Wallace, E., Wang, X., Wang, M., Watkins, O., Weil, K., Wendling, A., Whinnery, K., Whitney, C., Wong, H., Yang, L., Yang, Y., Yasunaga, M., Ying, K., Zaremba, W., Zhan, W., Zhang, C., Zhang, B., Zhang, E., and Zhao, S. gpt-oss-120b gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Ouyang, A., Guo, S., Arora, S., L. Zhang, A., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels? In *International Conference on Machine Learning (ICML)*, 2025. doi: 10.48550/arXiv.2502.10517. URL https://arxiv.org/abs/2502.10517.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

Sara Hooker. The Hardware Lottery. *Communications of the ACM*, 2021.

SemiAnalysis. MI300X vs H100 vs H200 Benchmark Part 1: Training – CUDA Moat Still Alive, 2024. URL https://semianalysis.com/2024/12/22/mi300x-vs-h100-vs-h200-benchmark-part-1-training/.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Spector, B., Juravsky, J., Sul, S., Dugan, O., Lim, D., Fu, D., Arora, S., and Ré, C. Look ma, no bubbles! designing a low-latency megakernel for llama-1b, 2025. URL https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles.

Spector, B. F., Arora, S., Singhal, A., Fu, D. Y., and Ré, C. Thunderkittens: Simple, fast, and adorable ai kernels. *International Conference on Learning Representations (ICLR)*, 2024.

Tillet, P., Kung, H. T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

Triton. Gluon?, 2025. URL https://github.com/triton-lang/triton/issues/7392.

Wang, L., Cheng, Y., Shi, Y., Tang, Z., Mo, Z., Xie, W., Ma, L., Xia, Y., Xue, J., Yang, F., and Yang, Z. Tilelang: A composable tiled programming model for ai systems, 2025. URL https://arxiv.org/abs/2504.17577.

Zhang, S., Zheng, N., Lin, H., Jiang, Z., Bao, W., Jiang, C., Hou, Q., Cui, W., Zheng, S., Chang, L.-W., Chen, Q., and Liu, X. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. *Proceedings of the 8th MLSys Conference (MLSys)*, March 2025.

Zhou, K., Lezcano, M., Goucher, A., Rakhmati, A., Niu, J., Lebar, J., Szczerbuk, P., Bell, P., Tillet, P., Raoux, T., and Moudallal, Z. Linear layouts: Robust code generation of efficient tensor computation using $\mathbb{F}_2$. *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume (ASPLOS '26)*, 2025.