The appendix is organized as follows:

1. Appendix A compares key NVIDIA and AMD terminology.
2. Appendix B contains a discussion of extended related work.
3. Appendix C provides extended results and ablations.
4. Appendix D discusses library implementation details.
5. Appendix E provides sample kernel listings written using HK.
6. Appendix F provides a case study of FP6 kernels on AMD.

CONCEPT	NVIDIA TERM	AMD TERM
EXECUTION UNIT	WARP (32 THREADS)	WAVE (64 THREADS)
THREAD BLOCK	THREAD BLOCK (CTA)	WORKGROUP
PROCESSOR	STREAMING MULTIPROCESSOR	COMPUTE UNIT
SHARED MEMORY	SHARED MEMORY (SMEM)	LOCAL DATA SHARE (LDS)
REGISTERS	REGISTERS	ACCUMULATOR / VECTOR REGISTERS (AGPRS/VGPRS)
GLOBAL MEMORY	HBM	HBM
CACHE	L2 (GPU-WIDE)	L2 CACHE (CHIPLET-WIDE) + LLC (GPU-WIDE)
MATRIX COMPUTE	TENSOR CORE	MATRIX CORE
MATRIX INSTRUCTION	WGMMA / WMMA / TCGEN05	MFMA
ASYNC MEMORY OPS	TMA	BUFFER LOAD TO LDS
COMPILER / TOOLCHAIN	CUDA, NVCC	HIP, HIPCC

A TERMINOLOGY

Although both NVIDIA and AMD GPUs follow a broadly similar SIMT (single-instruction, multiple-thread) execution model, their hardware and software stacks use distinct terminology. The table above summarizes the key correspondences.

B EXTENDED RELATED WORK

B.1 Libraries and frameworks for AI kernels

High-performance AMD kernel libraries AMD provides AITER (AMD, 2025a), a library of high-performance kernels, but its fastest implementations are written in raw assembly. While effective, these kernels do not expose reusable abstractions and are brittle to extend across workloads. CUDA and HIP expose kernels at the level of individual threads, whereas ML workloads are built from larger reusable computational patterns that benefit from coarser abstractions. Several libraries and compilers have attempted to bridge this gap. Composable Kernel (like CUTLASS) uses deeply nested C++ templates, creating a level of complexity that makes it difficult to use and extend (NVIDIA, 2017; AMD, 2025b).

Compilers Compiler-based systems such as TVM and Triton (Chen et al., 2018; Tillet et al., 2019; hel, 2025) target a broader ML audience with higher-level Python-like DSLs. While more familiar to researchers, these frameworks both sacrifice fine-grained control over registers and synchronization and have been slow to support new hardware features (see Section B.2). As a result, on AMD, developers often resort to inline assembly in Triton kernels to recover performance (Kim et al., 2022). Even on NVIDIA, where there has been relatively more investment into the compiler systems, C++-based DSLs provide up to $10\times$ performance improvements (Spector et al., 2024).

There are also more recent compilers:

1. **TileLang and Mojo** can compile to AMD via LLVM IR, but they lack abstractions for AMD's architectural constraints (e.g., flexible tile sizing under register pressure, thread block scheduling, cache-aware grid ordering). Their evaluations on AMD are limited: TileLang reports a single attention kernel at 257 TFLOPs on AMD MI300X GPU. TileLang also depends on backend calls to CUTLASS/CK, while Mojo relies on compiler hints rather than reusable abstractions. Mojo's attention kernel for the MI300X also shows bank conflicts. Neither framework systematically supports AMD thus far.

2. **Linear Layouts** formalizes MMA/WGMMA/MFMA layouts as linear maps and implements automatic, optimized conversions between them in Triton's backend (via warp shuffles and swizzled shared memory), with measured speedups. However, it does not define abstractions for thread-block or grid-level scheduling, and its evaluations do not demonstrate kernels that mix tensor core shapes or discuss the different phase orderings of memory instructions (as in Section 3.2).

Programming frameworks for peak performance Recently, a wave of DSLs has proposed C++ embedded tile-based primitives for AI kernels including THUNDERKITTENS (TK) (Spector et al., 2024) and successors (TileLang (Wang et al., 2025), CuTe (NVIDIA, 2024), Linear Layouts (Zhou et al., 2025)). These approaches demonstrate that small, opinionated sets of abstractions can yield both simplicity and high performance. However, none provides a comprehensive set of AMD kernel abstractions: THUNDERKITTENS and CuTe support and validate only on NVIDIA hardware, with kernel templates such as producer–consumer scheduling tied to NVIDIA-specific features. In contrast, our work identifies a minimal and principled set of abstractions that suffice for performant AMD kernels across warp, block, and grid levels. We demonstrate their sufficiency by implementing end-to-end kernels across diverse workloads, including attention backward, which requires mixing tensor core shapes.

B.2 AMD software ecosystem is brittle

Many AMD libraries are forks of NVIDIA libraries. Given the hardware differences, this risks sub-optimal performance. We also find that despite the investment into AMD software, the current ecosystem is brittle, motivating HK:

- **PyTorch kernels:** The built-in scaled dot product attention (SDPA) backend achieves just 259 TFLOPS on AMD MI355X GPUs for Llama GQA backwards (as of October 2025 on ROCm 7.0.0). Attention is a workhorse operation in modern AI workloads, and this performance gap highlights limited backend maturity.

- **Assembly kernels:** AITER (AMD, 2025a) includes high-performance kernels, but its fastest implementations are written directly in raw assembly. This approach is difficult to scale across the breadth of AI workloads; we can see the lack of full-fledged kernel support for GQA backwards on the AMD MI355X where AITER achieves just 272/384 TFLOPS at a sequence length of 8192 for causal and non-causal respectively.

- **Triton kernels:** The Triton compiler on AMD struggles with register lifetime tracking and lowering memory accesses to the most performant intrinsics. For example, it may fail to reclaim registers (code snippet) or lower vectorized loads (code snippet). Torch-compiled kernels can deliver competitive performance on memory-bound workloads (Figure 10), but the optimizations are black-box and may miss optimal intrinsics. For example, a compiled LayerNorm kernel on Llama-like dimensions exhibits a 23% lower L2 hit rate than our HK kernel. The time between new CDNA/PTX features and their integration into compilers is also slow; for instance as of September 2025, buffer loads are not the default for Triton loads/stores on AMD (code snippet).

This ecosystem motivates HK, which is designed to help simplify and accelerate high performance AMD kernel development.

C EXTENDED ANALYSIS

This section provides details on our setup and supplemental results.

Setup details. AMD provides multiple docker containers at https://hub.docker.com/u/rocm. We use the recent AMD provided Docker containers to benchmark kernels: docker.io/rocm/7.0-preview:rocm7.0_preview_pytorch_training_mi35x_beta on MI350X/MI355X and docker.io/rocm/pytorch on the MI300X/MI325X. A sample command to launch the container is provided below:

```
podman run -it \
    --ipc=host \
    --network=host \
    --privileged \
    --cap-add=CAP_SYS_ADMIN \
    --cap-add=SYS_PTRACE \
    --security-opt seccomp=unconfined \
    --device=/dev/kfd \
    --device=/dev/dri \
    -v $(pwd):/workdir/ \
    -e USE_FASTSAFETENSOR=1 \
    -e SAFETENSORS_FAST_GPU=1 \
    rocm/7.0-preview:rocm7.0
        ↪ _preview_pytorch_training_mi35x_beta \
    bash
```

Baselines. For the AITER baselines, we use Figure 11. If AITER does not automatically come with the Docker, we install from source. For the Composable Kernel baselines, we use the installation process and kernels indicated in Figure 13. For the PyTorch baselines, we use Figure 12. For the HipBLASLT baselines, we use the command from Figure 14 within the AMD provided Dockers.

```
// Attention
import aiter
out_aiter, softmax_lse = aiter.flash_attn_func(Q_aiter,
    ↪  K_aiter, V_aiter, causal=causal, return_lse=
    ↪ True, deterministic=False)
out_aiter.backward(dO_aiter)

// GEMM
from aiter.tuned_gemm import tgemm
C_aiter = tgemm.mm(A, B, None, None, None)
```

Figure 11. AITER benchmarking.

```
out_pt = torch.nn.functional.
    ↪ scaled_dot_product_attention(q_pt, k_pt, v_pt,
    ↪ attn_mask=None, dropout_p=0.0, is_causal=causal)
out_pt.backward(dO_bhnd)
```

Figure 12. PyTorch benchmarking.

```
// Build
[~] git clone https://github.com/rocm/composable_kernel
[~] cd composable_kernel
[~/composable_kernel] mkdir build && cd build
[~/composable_kernel/build] ../script/cmake-ck-dev.sh
    ↪ .. gfx950 -G Ninja
[~/composable_kernel/build] ninja
    ↪ tile_example_gemm_basic
[~/composable_kernel/build] ninja tile_example_fmha_fwd
[~/composable_kernel/build] ninja tile_example_fmha_bwd

// Run
./bin/tile_example_gemm_basic -prec=fp8 -m=1024 -n=1024
    ↪  -k=1024 -warmup=500 -repeat=100 -v=1
./bin/tile_example_fmha_fwd -prec=bf16 -b=16 -h=16 -d
    ↪ =128 -s=1024 -mask=1 -warmup=500 -repeat=100 -
    ↪ kname=1
./bin/tile_example_fmha_bwd -prec=bf16 -b=16 -h=16 -d
    ↪ =128 -s=1024 -mask=1 -warmup=500 -repeat=100 -
    ↪ kname=1
```

Figure 13. CK benchmarking. For each dimension of the GEMM, we report the best performance across the CK tile example streamk gemm basic, tile example gemm basic, and tile example gemm universal.

```
// bf16
hipblaslt-bench --batch_count 1 --a_type bf16_r --
    ↪ b_type bf16_r --c_type bf16_r --d_type bf16_r --
    ↪ rotating 512 --iters 100 --cold_iters 500 -m
    ↪ 1024 -n 1024 -k 1024


// fp8
hipblaslt-bench --api_method c --stride_a 0 --stride_b
    ↪ 0 --stride_c 0 --stride_d 0 --alpha 1.000000 --
    ↪ beta 0.000000 --transA T --transB N --
    ↪ batch_count 1 --scaleA 1 --scaleB 1 --a_type
    ↪ f8_r --b_type f8_r --c_type bf16_r --d_type
    ↪ bf16_r --scale_type f32_r --bias_type f32_r --
    ↪ compute_type f32_r --rotating 4 --iters 100 --
    ↪ cold_iters 500 -m 8192 -n 8192 -k 8192 --lda
    ↪ 8192 --ldb 8192 --ldc 8192 --ldd 8192 --
    ↪ initialization norm_dist


// fp6. after installing from source
./clients/hipblaslt-bench --api_method c -m 1024 -n
    ↪ 1024 -k 1024 --alpha 1 --beta 0 --transA T --
    ↪ transB N --batch_count 1 --scaleA 3 --scaleB 3
    ↪ --a_type f6_r --b_type f6_r --c_type f16_r --
    ↪ d_type f16_r --compute_type f32_r --rotating 0
    ↪ --cold_iters 500 --iters 100
```

Figure 14. HipBLASLT benchmarking.

C.1 HIPKITTENS kernels

We include the remaining kernel plots from Section 4. We include BF16 GEMM results for the MI325X and MI350X GPUs in Figure 15. We include MHA results on the MI355X GPUs in Figure 17 for the forwards pass and Figure 16 for the backwards pass.

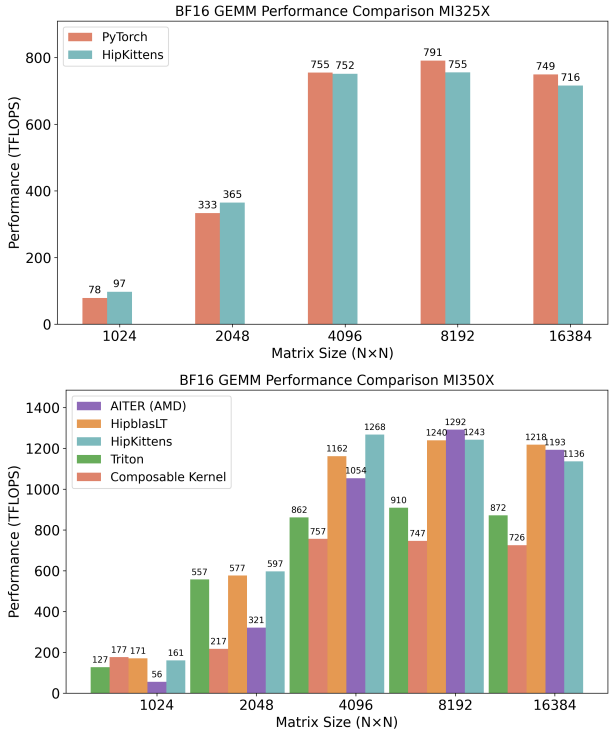


Figure 15. **BF16 GEMM.** We compare HIPKITTENS to the strongest available baselines on the MI325X and MI350X. For these kernels, we use dimensions $M = N = K$.

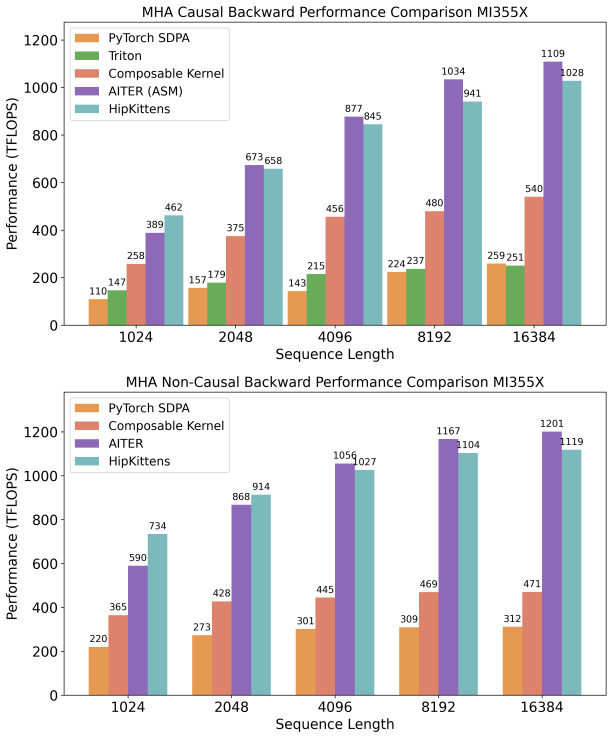


Figure 16. **Attention backwards.** MI355X results for causal and non-causal attention. We use batch size 16, heads 16, head dim 128.

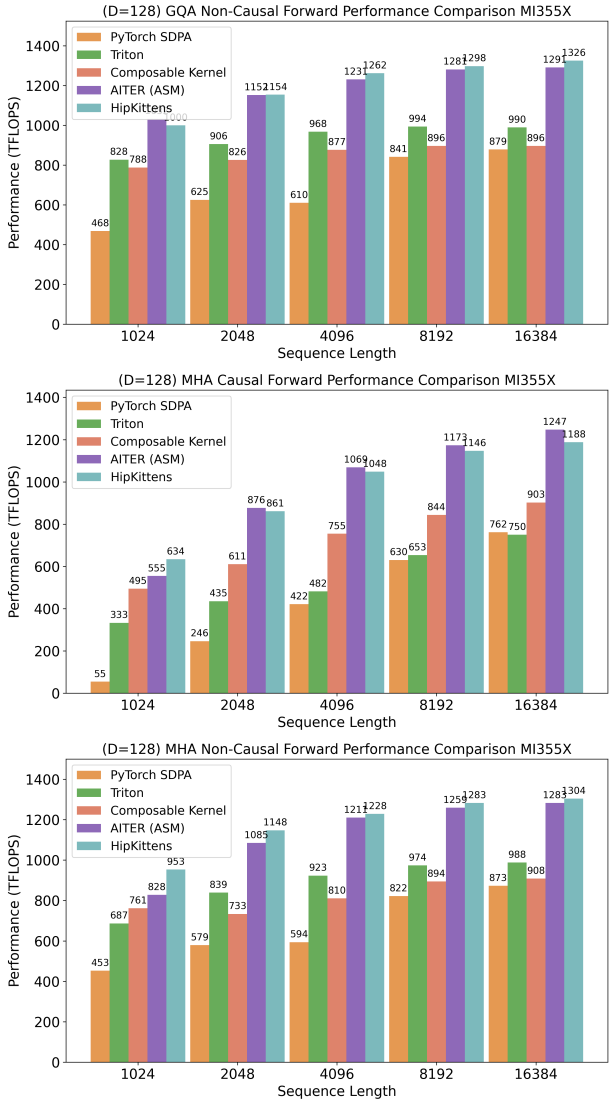


Figure 17. **Attention forwards.** MI355X results for causal and non-causal attention. We use batch size 16, heads 16, head dim 128.

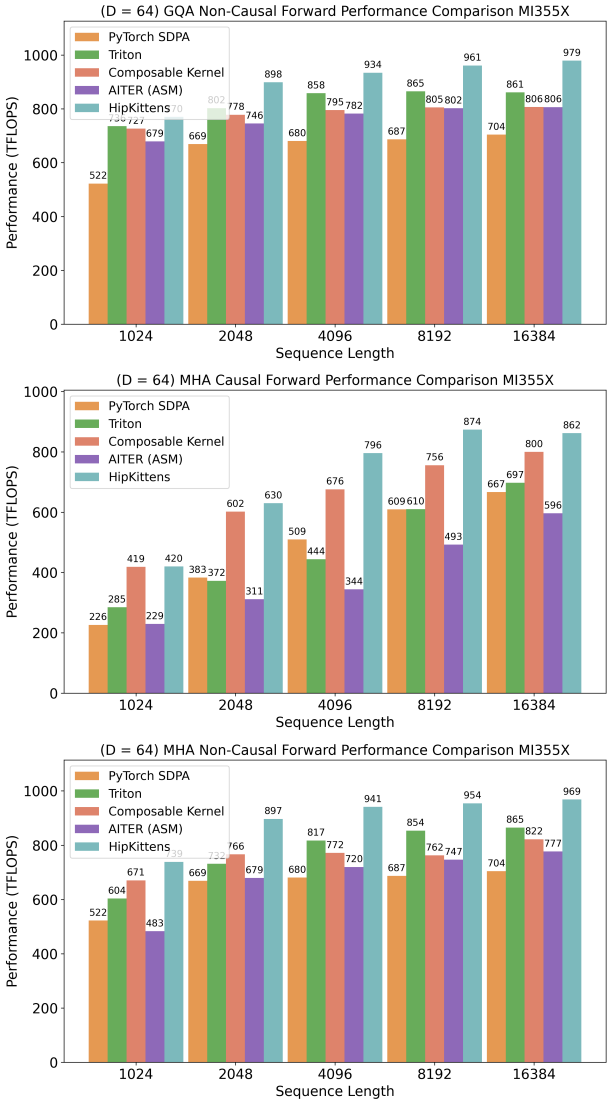
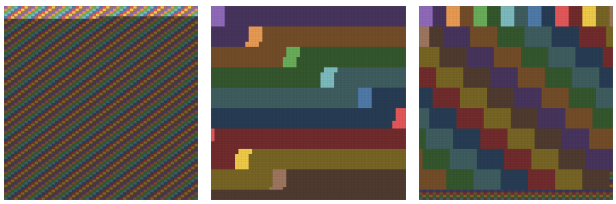


Figure 18. **Attention forwards.** MI355X results for causal and non-causal attention. We use batch size 16, heads 16, head dim 64.

C.2 Grid schedules

In Section 3.4, we include Table 4 to discuss the chiplet swizzling strategy to optimize L2 and LLC reuse. In Figure 19 we provide the corresponding grid order visualization for the 14592 dimension GEMM.



(a) (b) (c)

Figure 19. Visualization of three different grid schedules for the output D matrix of a M=N=K=14592 BF16 GEMM. Color represents XCD assignment. Highlighted is the first timestep of thread blocks scheduled across the device (256 CUs). Schedule 19a assigns blocks to the grid according to block ID. Schedules 19b and 19c apply algorithm 1 with different chunk sizes and the same window size. Table 4 showcases the performance for each of these schedules. This GEMM setting is especially sensitive to these optimizations due to the default schedule resulting in worst case L2 reuse and the large memory footprint making LLC reuse even more important.

C.3 End-to-end training performance

We report end-to-end training performance from October 1st, 2025. Training with custom HipKittens attention kernels results in a 26% speedup compared to a PyTorch-only implementation. We trained a model using the Llama architecture on 5 billion tokens of the Pile, while matching the same perplexity (2048 model dimension, 16 heads, 2048 sequence length, 6 layers).

C.4 THUNDERKITTENS performance

We benchmark TK (Spector et al., 2024) and CuBLASLT kernels on inputs drawn from $\mathcal{N}(0, 1)$ with 500 iterations of warmup and 100 iterations of measured runs, remaining consistent with our protocol on the AMD GPUs. Results are shown in Figure 20. [8] We include this to highlight how the TK philosophy, which we extend in this work, leads to performant NVIDIA kernels.

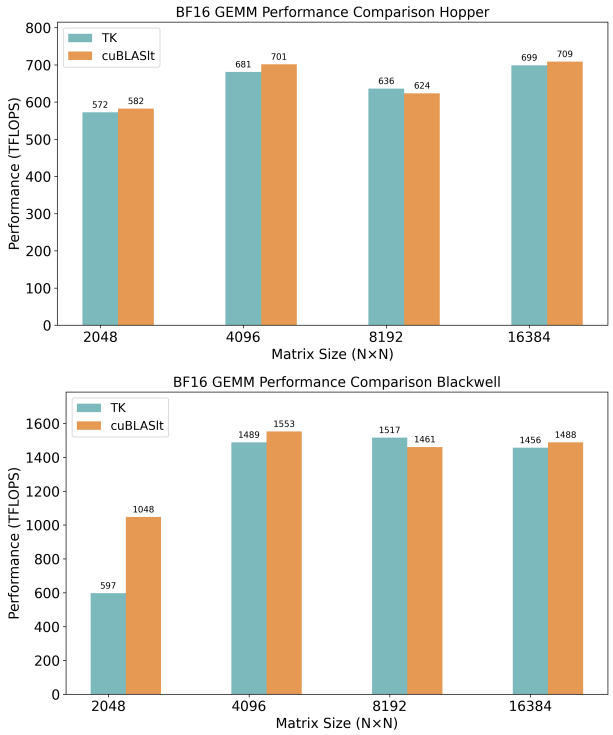


Figure 20. **THUNDERKITTENS performance.** We compare TK to NVIDIA CublasLT BF16 GEMM performance, finding that TK kernels offer competitive performance, despite the TK kernels being released ≈ 8-12 months ago.

C.5 HIPKITTENS compile time

We measure the end-to-end compile times (in seconds) of HK kernels. Results are show in Table 5. Despite aggressive specialization and extensive template usage, compile times remain within a few seconds for most kernels, with more complex attention backward kernels incurring higher compilation cost.

[8] We evaluate both on NVIDIA H100 and NVIDIA B200 GPUs. We generally observe that the NVIDIA kernel performance degrades as the number of iterations increases and we note that Spector et al. (2024) reports using fewer iterations.

KERNEL	COMPILE TIME (S)
CAUSAL ATTENTION BACKWARDS	17.27
ATTENTION BACKWARDS	17.25
CAUSAL ATTENTION FORWARDS	4.53
ATTENTION FORWARDS	5.04
FUSED DROPOUT + LAYERNORM + RESIDUAL	4.61
ROPE	4.05
BF16 GEMM	4.19
FP8 GEMM	2.32

Table 5. **Compile times of representative kernels.** We measure compile times (in seconds) for a range of kernels implemented using HK.

C.6 HIPKITTENS debugging effort and code complexity

We evaluate HK kernels based on debugging effort and code complexity. In Table 6, we show the number of lines of code for a range of kernels implemented in both HK and TK. HK kernels implemented using an 8-WAVE PING-PONG SCHEDULE (i.e. attention and GEMM) are roughly twice as long as the corresponding TK kernel. However, 8-WAVE PING-PONG kernels relies on limited inter-wave communication primitives (i.e. conditional barriers) while NVIDIA warp-specialized kernels rely on memory barriers and fine-grain communication between warps to coordinate producers and consumers. 4-WAVE INTERLEAVED kernels are over ten times longer but do not rely on conditional barriers at all - barriers are used to ensure that all waves participating in a global memory to shared memory has finished. As a result, we claim that HK makes a complexity tradeoff between race-condition-prone code and code length.

KERNEL	HK	TK
CAUSAL ATTENTION BACKWARDS	3303	277 (H100)
ATTENTION BACKWARDS	3113	277 (H100)
CAUSAL ATTENTION FORWARDS	510	180 (H100)
ATTENTION FORWARDS	472	180 (H100)
FUSED DROPOUT + LAYERNORM + RESIDUAL	61	89
ROPE	30	86
BF16 GEMM	289	193 (B200)
FP8 GEMM	237	120 (B200)

Table 6. **Lines of code (LoC) comparison.** We compare the implementation complexity of representative kernels written in HK (HK) and ThunderKittens (TK). TK implementations are hardware-specific, targeting either Hopper or Blackwell GPUs.

D LIBRARY IMPLEMENTATION DETAILS

D.1 Shared and Register Memory

Register Layouts. The layout of a register tile dictates whether threads hold elements in a row-major or column-major order. [9] Since threads hold consecutive elements in the reduction dimension of MFMA instructions, the register layout decides whether we are reducing over rows or columns of memory. When loading data from shared memory to registers, we typically use two different types of instructions for BF16:

- **Row layouts (ds_read_b128):** `ds_read*` instructions take in 3 arguments: 1) vector registers to serve as the destination of the data, 2) a shared memory address, and 3) a constant offset from the shared memory address to read data. The `ds_read_b128` instruction is carried out in 4 phases where subsets of the 64 threads execute in each phase (Tab. 7). As a result, ensuring that no two threads belonging to the same phase access the same shared memory bank eliminates bank conflicts for this instruction. With 16 threads participating in each phase, each thread accessing 128 bits or 4 banks, all 64 banks should be read and we maximize shared memory throughput.

- **Column layouts (ds_read_b64_tr_b16):** Normally, reading data in a column-major format involves issuing multiple loads for each individual row we access. Using the `ds_read_b64_tr_b16` instruction allows us to perform these column-major loads much more efficiently by having threads access shared memory at a greater granularity. Take the 16x32 register tile for example in Figure 21).

  In a 16x32 column layout register tile where each thread holds 8 contiguous elements in the reduction dimension (i.e., stride 8), thread 0 holds the first element in rows 0-7. They are shaded in the two tables too. The `ds_read_b64_tr_b16` instruction accomplishes this load by having different threads read data that is placed in another thread's vector register lane. For example, thread 4 technically reads the first element in the second row, but instead of placing it in its own register lane, it puts it into thread 0's register. This instruction executes in two phases where the first 32 threads read during the first phase and the remaining ones read during the second phase. If this SMEM tile only needed to support reads from column-major 16x32 register tiles, an unswizzled pattern would be sufficient to eliminating bank conflicts. However, shown in Figure 3, a swizzle is necessary to support reads from a row-major 16x32 register tile.

[9] https://github.com/ROCm/amd_matrix_instruction_calculator. This is a useful resource to learn more about the register tile layouts and shapes.

Shared Memory and Register Tile Shapes. While the previous subsection focused on loads from a 16x32 shared memory tile shape to a 16x32 register tile, different workloads could warrant other shared memory and register tile shapes mapping to different MFMA instructions. HK supports loads and stores between shared memory and register tile shapes as long as one is a multiple of the other. For example, loading from a 16x32 shared memory tile into a 32x16 register tile is not supported, but loading from a 16x16 shared memory tile into a 32x16 register tile is permitted. Each shared memory tile shape is also equipped with a default swizzling pattern that is a best-effort attempt to eliminate bank conflicts for common access patterns.

A single swizzle is not possible. To show why a single swizzling pattern is insufficient across different register tile shapes and layouts on AMD GPUs, consider the following two access patterns that surface in attention backwards:

1. A row-layout 16x16 bf16 tile is written to shared memory. For this tile configuration, each thread holds 4 contiguous bf16 values - 64 bits in memory - and the most optimal instruction to issue this write is `ds_write_b64`. Avoiding bank conflicts for this access requires a swizzle pattern that respects the phase ordering and bank behavior as listed in Table 7. In this case, a swizzle that abides by these constraints is `offset ^= ((offset % 512) >> 7) << 3`, where 64-bit chunks of memory is shifted around memory using an XOR swizzle.

2. A row-layout 16x32 bf16 tile is read from shared memory. For this tile, each thread holds 8 contiguous bf16 values - 128 bits in memory - and the most optimal instruction to issue this read is `ds_read_b128`.

Regardless of the swizzling pattern required for `ds_read_b128`, the granularities of these two instructions are in conflict with each other. `ds_read_b128` requires at least 128 bits of memory to be contiguous in shared memory, and the swizzle pattern for `ds_write_b64` breaks apart memory into 64-bit chunks. As a result, different swizzling patterns need to be used for each.

D.2 Phases and Banks

Since per-instruction phase and bank behavior is not well documented, we create simple solvers for both. The phase solver iterates over every pair of threads in a wave and performs the shared memory instruction on the same bank. If a shared memory bank occurs, the two threads belong to the same phase. The bank solver takes two threads belonging to the same phase, fixes one thread to access bank zero, and accesses other banks using the other thread. The number of

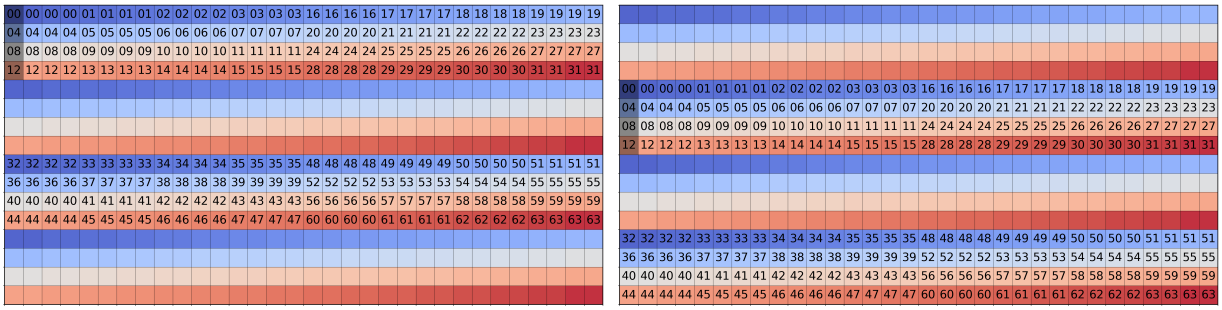


Figure 21. Shared memory access pattern for `ds_read_b64_tr_b16` for reading in a 16x32 column layout register tile. Each cell represents a single 16 bit values and the numbers represent threads. Different colors represent different shared memory banks (note that a bank spans two cells).

INSTR.	BANKS	PHASE	ACTIVE THREADS
DS_READ_B128	64	0	0-3, 12-15, 20-27
		1	4-11, 16-19, 28-31
		2	32-35, 44-47, 52-59
		3	36-43, 48-51, 60-63
DS_READ_B96	32	0	0-3, 20-23
		1	4-7, 16-19
		2	8-11, 28-31
		3	12-15, 24-27
		4	32-35, 52-55
		5	36-39, 48-51
		6	40-43, 60-63
		7	44-47, 56-59
DS_WRITE_B64	32	0	0-15
		1	16-31
		2	32-47
		3	48-63
DS_READ_B64	64	0	0-31
		1	32-63

Table 7. **Phase-bank table.** The number of banks available to each shared memory instruction and the number of phases (and participating threads per phase) each instruction requires.

banks between bank zero and the first bank where a bank conflict occurs represents the number of banks accessible by the shared memory instruction.

D.3 Shared memory swizzle patterns

HK defines a library of XOR swizzle patterns. Each shared tile is defined with an underlying subtile shape - a 256x256 shared tile might be composed of 16x32 subtiles. Each subtile shape automatically defines a swizzle that memory operations (i.e. loads and stores between global memory, share memory, and registers) apply. By only considering common shared memory access patterns for each subtile shape, each swizzle pattern ensures bank-conflict-free accesses.

For example, a BF16 shared tile defined with ST_16X32 as the underlying subtile shape is primarily associated with two matrix core instructions: V_MFMA_F32_16X16X32_BF16 where the reduction dimension is contiguous in memory (row layout) and V_MFMA_F32_32X32X16_BF16 where the reduction dimension is non-contiguous in memory (column layout). The dimensions of these instructions match those of the subtile. Each layout rely on specific shared memory instructions (Section D.1) and each shared memory instruction has different phase and bank behavior (Section D.2) swizzle patterns need to consider. For each subtile shape, HK defines a swizzle pattern that ensures bank-conflict-free accesses for the access patterns specific to it. Being C++ embedded, HK also allows developers to extend the language and define new swizzle patterns; memory operations will handle them automatically. The following code-snippet shows bank-conflict-free loads from a 16x32 shared tile into two 16x32 register tiles with different layouts:

```
    using ST_A = st_bf<16, 32, st_16x32_s>;
    ST_A (&As) = al.allocate<ST_A>();

    rt_bf<16, 32, row_l, rt_16x32_s> A_reg;
    rt_bf<16, 32, col_l, rt_32x16_s>
        ↪ A_reg_transposed;

    load(As, g.a, {0, 0, 0, 0});
    __builtin_amdgcn_s_barrier();
    // bank-conflict-free
    load(A_reg, As);
    // bank-conflict-free
    load(A_reg_transposed, As);
```

D.4 Pinned register tiles

HK lets developers control the registers assigned to different register tiles through the concept of register ranges. For example:

```
using Q_ranges =
  split_many_t<type_list<range<24, 39>>,
      ↪ 4>;
```

This defines a list of register ranges where each range contains exactly 4 registers. The register ranges here are `v[24:27]`, `v[28:31]`, `v[32:35]`, and `v[36:39]`. Each register range corresponds to the registers required to hold a single base tile in a register tile, and we specify a list of register ranges when defining a register tile like:

```
rt<bf16, 16, 128, row_l,
   rt_16x32_s, Q_ranges> Q_i;
```

Developers can call the same functions in HK, but now have them operate on specific registers instead. As mentioned in Section 3.2, this allowed us to pin AGPRs as the A or B matrix inputs to MFMA instructions when writing our attention backwards kernel.

HK implements explicit register scheduling through C++ template metaprogramming. Pinned register tiles are defined with a range of registers as part of its type. Operations on these tiles produce inlined assembly instructions based on their type (i.e. what dtype, MFMA shape, and pinned registers the tile was defined with). This has the caveat of removing compiler-imposed safety guaranties in exchange for the level of control developers need to write peak performance AMD AI kernels.

D.5 Compiler hints

The LLVM compiler accepts developer-provided hints to guide instruction scheduling on AMD GPUs.[10] We use these some of these hints in our kernels to augment the scheduling that we apply at the HIP level. There are three sets of intrinsics that we find useful.

1. The `llvm.amdgcn.sched.barrier` intrinsic accepts a mask, which tells the compiler which types of instructions can cross the intrinsic in the compiled schedule. Masks exist for all instructions, VALU (vector ALU) instructions, SALU (scalar ALU) instructions, VMEM (global memory) instructions, MFMA (matrix) instructions, and so on, as described in the documentation. This intrinsic is used to establish hard boundaries between clusters of instructions in our clusters. For instance, see `__builtin_amdgcn_sched_barrier(0)` in our Appendix E kernel listings.

2. The `llvm.amdgcn.sched.group.barrier` intrinsic is used to establish scheduling pipelines. The developer considers a group of instructions and specifies to the compiler precisely how to order them. A call of the builtin accepts a `mask` that specifies the instruction type, `size` indicating the number of instructions of this type that calls the builtin applies to, and a `sync id` serves as an identifier.

   This builtin creates a "super group" of "instruction groups". The `sync id` identifies the super group; order is enforced between instruction groups with the same `sync id`, and instruction groups are only scheduled relative to other groups with the same `sync id`.

   The `mask` is a bitmask. Here are some frequently used ones:

   ```
   #define MFMA_MASK 0x08
   #define VMEM_MASK 0x20
   #define DS_MASK 0x100
   ```

   Each call of this intrinsic looks backward and finds the most recent of the corresponding type of instruction which are not already part of a group created by a previous `__builtin_amdgcn_sched_group_barrier`

   For example:

   ```
   __builtin_amdgcn_sched_group_barrier(
       ↪ VMEM_MASK, 4, 0);
   __builtin_amdgcn_sched_group_barrier(
       ↪ MFMA_MASK, 4, 0);
   __builtin_amdgcn_sched_group_barrier(
       ↪ DS_MASK, 8, 0);
   __builtin_amdgcn_sched_group_barrier(
       ↪ MFMA_MASK, 4, 0);
   ```

   This finds the last 4 global memory (VMEM) loads and schedules those first, then finds the last 4 matrix (MFMA) instructions and schedules those after the global memory loads, then finds the last 8 shared to register (DS READ) loads and schedules those after the 4 MFMAs, then finds the previous 4 MFMAs before the last 4 and schedules those last.

3. The `__builtin_amdgcn_s_setprio` intrinsic lets us specify the priority (0-3) for a wave relative to other waves that are competing for hardware resources. We use this around compute clusters in the 8-WAVE ping-pong schedule as shown in our GEMM and attention forwards kernels.

The limitation of using these hints for scheduling is that any code wrapped in `asm volatile` is black-box to the compiler, and for some instructions (e.g., `v_cvt_pk_bf16_f32` to convert from BF16 to FP32), LLVM builtins are missing.

It is worth noting that the current Modular AI GEMM kernels (as of October 2025) rely on compiler hints (`sched_group_barrier`). This approach could work since Modular is replacing the compiler as well; however, it requires the developer to think about every single instruction issue rather than providing the option to think about bulk

[10] https://llvm.org/docs/AMDGPUUsage.html

tile primitives. Our opinion is that using scheduling hints at the cluster scope and tile primitives to form the top level kernel schedule (as in our attention forwards kernel) may help simplify programmability and maintain performance.

D.6 Synchronization

Loads Similar to asynchronous tensor memory acceleration (TMA), AMD CDNA3 and CDNA4 GPUs have direct global memory to LDS (shared memory) load instructions called `buffer_load_dword`. These instructions can load one `dword` (4 bytes, one bank), three `dwordx3` (12 bytes), or four `dwordx4` (16 bytes). The instructions skip the register file and accept constant offsets that also help mitigate address calculation overheads. Once the load is issued, the instruction `vmcnt(x)` specifies to wait until only `x` global memory load instructions remain in flight, and `vmcnt(0)` indicates to wait on all outstanding loads. Ideally, we can separate the distance between load issues and these waits (as shown in our GEMM and attention kernels (Sec. E)).

Similarly, there are asynchronous shared to register memory loads called `ds_read_b32` (or `b64` for 8 bytes, `b96` for 12 bytes, `b128` for 16 bytes). The instruction `lgmkcnt(x)` specifies to wait until `x` shared to register instructions remain in flight, and `lgmkcnt(0)` indicates to wait on all outstanding shared to register loads.

Execution An `__builtin_amdgcn_s_barrier()` functionally matches `syncthreads`. Note that AMD has a SIMD model and NVIDIA follows SIMT, so we do not need to sync the threads within the warp on AMD. As a result, there is no equivalent of `syncwarp` on AMD.

D.7 Extending XCD swizzling to other workloads

An XCD swizzle can be viewed as a mapping from logical workgroups to physical XCDs that aims to co-locate workgroups with shared input reuse, improving L2 residency and reducing cross-XCD traffic. For a given workload, the key step is to identify which inputs are reused across workgroups and cluster those workgroups onto the same XCD within a short execution window. A simple first cut is to map nearby workgroups in the logical index space to the same XCD, which often preserves spatial and temporal locality. In GEMM, this corresponds to clustering workgroups that share tiles of one operand (e.g., fixed N,K for B reuse or fixed M,K for A reuse), effectively assigning a small tile of the M×N workgroup grid to each XCD. For GQA-style workloads, for example, many workgroups share the same K/V inputs, so clustering workgroups that operate on the same K/V tiles onto the same XCD yields similar L2 benefits. The CHIPLET_CHUNK transform is a general pattern that's useful for the programmer in implementing an XCD swizzle optimization for any workload. This is because it "undos" the hardware schedule and gives you nice logical chunks of workgroups that share the same L2 and makes it easier as a programmer to reason about input reuse.

E HIPKITTENS KERNEL LISTINGS

This section demonstrates HIPKITTENS kernel examples and discusses the algorithmic details of our kernel implementations.

E.1 Matrix Multiply

The BF16 GEMM kernel (Section E.1) decomposes the problem into computing a 256×256 output tile per thread block (denoted by `BLOCK_SIZE`). In the prologue, the kernel pre-loads the A and B input matrices from global to shared memory. The kernel inserts a conditional barrier to stall half the waves (one wave per SIMD) while the other half begins performing additional loads. When this leader wavegroup finishes its additional loads, it unblocks the follower wavegroup through the `s_barrier` invocation. Thereafter, the two wavegroups alternate between the compute and memory clusters shown in the hotloop, where the end of a cluster is always demarked by an `s_barrier`. This represents the 8-WAVE PING PONG kernel schedule introduced in Section 3.3.

For the MI325X version of the kernel, we maintain the same 8-WAVE structure, however the hardware only has 65 KB of shared memory so we cannot double buffer in shared memory. Instead, we double buffer using the register file. We do not use the direct HBM to LDS buffer loads, and instead load from HBM to a register buffer, while the waves compute MFMAs on the previously stored register tiles. When the compute completes, the data from the register buffers gets stored down to shared memory using a `ds_write`.

```
constexpr int BLOCK_SIZE       = 256;
constexpr int HALF_BLOCK_SIZE  = BLOCK_SIZE / 2;
constexpr int K_STEP           = 64;
constexpr int WARPS_M          = 2;
constexpr int WARPS_N          = 4;
constexpr int REG_BLOCK_M      = BLOCK_SIZE / WARPS_M;
constexpr int REG_BLOCK_N      = BLOCK_SIZE / WARPS_N;
constexpr int HALF_REG_BLOCK_M = REG_BLOCK_M / 2;
constexpr int HALF_REG_BLOCK_N = REG_BLOCK_N / 2;
constexpr int DOT_SLICE        = 32;

__global__ void GEMM_BF16(const micro_globals g) {
    /**
    * Set up shared memory buffers for double buffering.
    * This allows us to create a memory pipeline where
    * we consume subtiles of A and B while we load in
    * the next subtiles to hide memory latency.
    */
    extern __shared__ alignment_dummy __shm[];
    shared_allocator al((int*)&__shm[0]);
    using ST_A = st_bf<HALF_BLOCK_SIZE, K_STEP, st_16x32_s>;
    using ST_B = st_bf<HALF_BLOCK_SIZE, K_STEP, st_16x32_s>;
    ST_A (&As)[2][2] = al.allocate<ST_A, 2, 2>();
    ST_B (&Bs)[2][2] = al.allocate<ST_B, 2, 2>();

    rt_bf<HALF_REG_BLOCK_M, K_STEP, row_l, rt_16x32_s> A_tile;
    rt_bf<HALF_REG_BLOCK_N, K_STEP, row_l, rt_16x32_s> B_tile_0;
    rt_bf<HALF_REG_BLOCK_N, K_STEP, row_l, rt_16x32_s> B_tile_1;
    rt_fl<HALF_REG_BLOCK_M, HALF_REG_BLOCK_N, col_l, rt_16x16_s> C_accum[2][2];
    zero(C_accum[0][0]);
    zero(C_accum[0][1]);
    zero(C_accum[1][0]);
    zero(C_accum[1][1]);

    /**
    * L2 and LLC Cache swizzling (Section 3.4)
    * This optimization reorders the assignment of thread blocks
    * to subtiles of the output matrix to maximize memory bandwidth
    * through higher L2/LLC cache hit rates.
    */
    int wgid = (blockIdx.y * gridDim.x) + blockIdx.x;
    const int NUM_WGS  = gridDim.x * gridDim.y;
    const int WGM = 8;
    // Swizzle chiplet so that wgids are in the same XCD.
    wgid = chiplet_transform_chunked(wgid, NUM_WGS, NUM_XCDS, WGM*WGM);
    // Swizzle for better L2 within the same XCD.
    const int num_pid_m = ceil_div(M, BLOCK_SIZE);
    const int num_pid_n = ceil_div(N, BLOCK_SIZE);
    const int num_wgid_in_group = WGM * num_pid_n;
    int group_id = wgid / num_wgid_in_group;
    int first_pid_m = group_id * WGM;
    int group_size_m = min(num_pid_m - first_pid_m, WGM);
    int pid_m = first_pid_m + ((wgid % num_wgid_in_group) % group_size_m);
    int pid_n = (wgid % num_wgid_in_group) / group_size_m;
    // Assign the tile's row/column based on the pid_m and pid_n.
    int row = pid_m;
    int col = pid_n;

    const int warp_id = kittens::warpid();
    const int warp_row = warp_id / 4;
    const int warp_col = warp_id % 4;
    const int num_tiles = K / K_STEP;

    // Used to swap between shared memory buffers
    int tic = 0;
    int toc = 1;

   // Pre-load the first subtile of A and B
    G::load(Bs[tic][0], g.b, {0, 0, col*2, 0});
    G::load(As[tic][0], g.a, {0, 0, row*2, 0});
    G::load(Bs[tic][1], g.b, {0, 0, col*2 + 1, 0});
    G::load(As[tic][1], g.a, {0, 0, row*2 + 1, 0});

    /**
    * Conditional barrier that stalls half of the waves.
    * This indicates that we are using the 8-wave ping-pong
    * schedule introduced in Section 3.2.
    */
    if (warp_row == 1) {
        __builtin_amdgcn_s_barrier();
    }
```

```
// When the unstalled waves hit this s_barrier, the
// stalled waves are unblocked and allowed to resume
// execution.
asm volatile("s_waitcnt vmcnt(4)");
__builtin_amdgcn_s_barrier();

// Pre-load part of the second subtile of A and B
G::load(Bs[toc][0], g.b, {0, 0, col*2, 1});
G::load(As[toc][0], g.a, {0, 0, row*2, 1});
G::load(Bs[toc][1], g.b, {0, 0, col*2 + 1, 1});

asm volatile("s_waitcnt vmcnt(6)");
__builtin_amdgcn_s_barrier();

#pragma unroll
for (int tile = 0; tile < num_tiles - 2; ++tile, tic^=1, toc^=1) {

    /**
    * Cluster 0: memory
    * We load in a subtile of A and B from shared memory into registers.
    * We load in the next subtile of A from global memory into shared memory.
    * This cluster uses two hardware resources: HBM to shared memory bandwidth
    * and shared memory to register bandwidth.
    */
    auto st_subtile_b = subtile_inplace<HALF_REG_BLOCK_N, K_STEP>(Bs[tic][0], {warp_col, 0});
    load(B_tile_0, st_subtile_b);
    auto st_subtile_a = subtile_inplace<HALF_REG_BLOCK_M, K_STEP>(As[tic][0], {warp_row, 0});
    load(A_tile, st_subtile_a);
    G::load(As[toc][1], g.a, {0, 0, row*2 + 1, tile + 1});
    asm volatile("s_waitcnt lgkmcnt(8)");
    __builtin_amdgcn_s_barrier();

    /**
    * Cluster 1: compute
    * We perform a matrix multiplication between the A and B subtiles
    * loaded in from the previous cluster and accumulate the result into C_accum.
    * This cluster uses the matrix cores.
    */
    asm volatile("s_waitcnt lgkmcnt(0)");
    __builtin_amdgcn_s_setprio(1);
    mma_ABt(C_accum[0][0], A_tile, B_tile_0, C_accum[0][0]);
    __builtin_amdgcn_s_setprio(0);
    __builtin_amdgcn_s_barrier();
    __builtin_amdgcn_sched_barrier(0);

    /**
    * Cluster 2: memory
    * We load in a subtile of B from shared memory into registers and
    * load in a future subtile of B from global memory into shared memory.
    */
    st_subtile_b = subtile_inplace<HALF_REG_BLOCK_N, K_STEP>(Bs[tic][1], {warp_col, 0});
    load(B_tile_1, st_subtile_b);
    G::load(Bs[tic][0], g.b, {0, 0, col*2, tile + 2});
    __builtin_amdgcn_s_barrier();

    /**
    * Cluster 3: compute
    * We perform a matrix multiplication between a subtile of A and B and
    * accumulate the result.
    */
    asm volatile("s_waitcnt lgkmcnt(0)");
    __builtin_amdgcn_s_setprio(1);
    mma_ABt(C_accum[0][1], A_tile, B_tile_1, C_accum[0][1]);
    __builtin_amdgcn_s_setprio(0);
    __builtin_amdgcn_s_barrier();

    /**
    * Cluster 4: memory
    * We load in a subtile of A from shared memory into registers and
    * load in a future subtile of A from global memory into shared memory.
    */
    st_subtile_a = subtile_inplace<HALF_REG_BLOCK_M, K_STEP>(As[tic][1], {warp_row, 0});
    load(A_tile, st_subtile_a);
    G::load(As[tic][0], g.a, {0, 0, row*2, tile + 2});
    __builtin_amdgcn_s_barrier();
```

```
        /**
        * Cluster 5: compute
        * We perform a matrix multiplication between subtiles of A and B and
        * accumulate the result.
        */
        asm volatile("s_waitcnt lgkmcnt(0)");
        __builtin_amdgcn_s_setprio(1);
        mma_ABt(C_accum[1][0], A_tile, B_tile_0, C_accum[1][0]);
        __builtin_amdgcn_s_setprio(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        /**
        * Cluster 6: memory
        * We load in a future subtile of B from global memory into shared memory.
        */
        G::load(Bs[tic][1], g.b, {0, 0, col*2 + 1, tile + 2});
        asm volatile("s_waitcnt vmcnt(6)");
        __builtin_amdgcn_s_barrier();

        /**
        * Cluster 7: compute
        * We perform a matrix multiplication between subtiles of A and B and
        * accumulate the result.
        */
        __builtin_amdgcn_s_setprio(1);
        mma_ABt(C_accum[1][1], A_tile, B_tile_1, C_accum[1][1]);
        __builtin_amdgcn_s_setprio(0);
        __builtin_amdgcn_s_barrier();
    }

    // Epilogue not shown

    // This conditional barrier restores the stalled wave to be back in-step.
    if (warp_row == 0) {
        __builtin_amdgcn_s_barrier();
    }

    // Store out results
    store(g.c, C_accum[0][0], {0, 0, (row * 2) * WARPS_M + warp_row, col * 2 * WARPS_N + warp_col});
    store(g.c, C_accum[0][1], {0, 0, (row * 2) * WARPS_M + warp_row, col * 2 * WARPS_N + WARPS_N + warp_col});
    store(g.c, C_accum[1][0], {0, 0, (row * 2) * WARPS_M + WARPS_M + warp_row, col * 2 * WARPS_N + warp_col});
    store(g.c, C_accum[1][1], {0, 0, (row * 2) * WARPS_M + WARPS_M + warp_row, col * 2 * WARPS_N + WARPS_N +
        ↪ warp_col});
}
```

Figure 22. HK BF16 GEMM, which is competitive with AITER on CDNA4.

E.2 Fused Dropout + Residual + Layernorm

A very simple HIPKITTENS kernel that processes a chunk of vectors along the sequence dimension per thread block. This kernel listing demonstrates HK operators and vectors, which resemble those in PyTorch (e.g., `sum`, `add`, `mul`, `div`).

```
__global__ void fused_layernorm(const norm_globals g) {

    auto warpid = kittens::warpid();

    const int batch = blockIdx.y;
    const int seq_start = blockIdx.x*g.n_per_tile;
    constexpr int d_model = 128;

    rv_naive<bf16, d_model> residual_s_reg, x_s_reg, norm_weight_s_reg, norm_bias_s_reg;
    load(x_s_reg, g.x, {0, batch, seq_start + warpid, 0});
    asm volatile("s_waitcnt vmcnt(0)");
    load(residual_s_reg, g.residual, {0, batch, seq_start + warpid, 0});
    bf16 mean = __float2bfloat16(0.0f);
    bf16 var  = __float2bfloat16(0.0f);
    if constexpr (DROPOUT_P > 0.0f) {
        dropout_mask(x_s_reg, DROPOUT_P);
    }
    if constexpr (DROPOUT_P > 0.0f) {
        constexpr float scale = 1.0f / (1.0f - DROPOUT_P);
        mul(x_s_reg, x_s_reg, __float2bfloat16(scale));
    }
    asm volatile("s_waitcnt vmcnt(0)");
    load(norm_weight_s_reg, g.norm_weight, {0,0,0,0});
    load(norm_bias_s_reg, g.norm_bias, {0,0,0,0});
    add(residual_s_reg, residual_s_reg, x_s_reg);
    store(g.o_resid, residual_s_reg, {0, batch, seq_start + warpid, 0});

    // mean and variance
    sum(mean, residual_s_reg);
    constexpr float dim_scale = 1.0f / d_model;
    mean = mean * __float2bfloat16(dim_scale);
    sub(residual_s_reg, residual_s_reg, mean);
    mul(x_s_reg, residual_s_reg, residual_s_reg);
    sum(var, x_s_reg);
    var = var * __float2bfloat16(dim_scale);
    var = __float2bfloat16(sqrt(__bfloat162float(var + __float2bfloat16(1e-05f))));

    // compute norm
    div(residual_s_reg, residual_s_reg, var);
    asm volatile("s_waitcnt vmcnt(0)");
    mul(residual_s_reg, residual_s_reg, norm_weight_s_reg);
    add(residual_s_reg, residual_s_reg, norm_bias_s_reg);
    store(g.o, residual_s_reg, {0, batch, seq_start+warpid, 0});
}
```

Figure 23. Fused Dropout + Residual + Layernorm kernel outperforming `torch.compile`.

E.3 Attention

The HIPKITTENS attention kernel uses an 8-WAVE PING PONG schedule. Each wave computes a 32×128 tile of the output for a single head and batch. In the prologue, all eight waves first collaboratively load tiles of keys and values, and their own personal tiles of queries. The threads perform the initial query-key matrix multiply and first half of softmax. Then the kernel uses a conditional barrier to stall half the waves (one wave per SIMD). The leader wavegroup proceeds ahead, loading the next tiles of keys and values and upon completion, unlocks the follower wavegroup. In the hotloop, the two wavegroups alternate between compute clusters (each involving matrix multiplies and vector operations) and load clusters (involving global to shared and shared to register loads).

In compute clusters, the compiler interleaves vector ops and matrix ops. We can also use `sched_barrier` hints to guide the LLVM compiler to interleave in custom patterns.

```
template<int D>
__global__ void attend_ker(const attn_globals<D> g) {

    extern __shared__ alignment_dummy __shm[];
    shared_allocator al((int*)&__shm[0]);
    st_bf<KV_BLOCK_SIZE, ATTN_D, st_32x32_s> (&k_smem)[2] = al.allocate<st_bf<KV_BLOCK_SIZE, ATTN_D, st_32x32_s>,
        ↪ 2>();
    st_bf<KV_BLOCK_SIZE, ATTN_D, st_8x32_s> (&v_smem)[2] = al.allocate<st_bf<KV_BLOCK_SIZE, ATTN_D, st_8x32_s>, 2>()
        ↪ ;

    const int head_idx = (blockIdx.x % GROUP_SIZE) * GROUP_SIZE + (blockIdx.x / GROUP_SIZE);
    const int batch_idx = blockIdx.z;
    const int head_idx_kv = head_idx / GROUP_SIZE;
    const int block_tile_idx = blockIdx.y;
    const int tile_idx = block_tile_idx * NUM_WARPS + warpid();
    const int stagger = warpid() / 4;

    const int num_tiles = ATTN_N / KV_BLOCK_SIZE;
    constexpr float TEMPERATURE_SCALE = (D == 128) ? 0.08838834764f*1.44269504089f : 0.125f*1.44269504089f;

    // Initialize all of the register tiles.
    qo_tile<D, bf16> q_reg; // Q and K are both row layout, as we use mma_ABt.
    qo_tile_transposed<D, bf16> q_reg_transposed;
    kv_tile<D, bf16> k_reg;
    kv_tile_transposed<D, bf16> k_reg_transposed;

    kv_tile<D, bf16, col_l, rt_32x32_s> v_reg;
    qo_tile_transposed<D, float, col_l, rt_32x32_s> o_reg; // Output tile.
    attn_tile<D, float, col_l, rt_32x32_s> att_block[2]; // attention tile, in float.
    attn_tile<D, bf16, col_l, rt_32x32_s> att_block_bf16;
    typename attn_tile<D, float, col_l, rt_32x32_s>::row_vec max_vec, norm_vec, max_vec_prev;

    G::load<1, false>(k_smem[0], g.Kg, {batch_idx, 0, head_idx_kv, 0});
    __builtin_amdgcn_s_waitcnt(0);
    __builtin_amdgcn_sched_barrier(0);
    __builtin_amdgcn_s_barrier();

    qo_tile<D, float> q_reg_fl;
    load<1, qo_tile<D, float>, _gl_QKVO>(q_reg_fl, g.Qg, {batch_idx, tile_idx, head_idx, 0});
    mul(q_reg_fl, q_reg_fl, TEMPERATURE_SCALE);  // Use sqrtf for clarity
    copy(q_reg, q_reg_fl);
    swap_layout_and_transpose(q_reg_transposed, q_reg);

    zero(o_reg);
    zero(norm_vec);
    neg_infty(max_vec_prev);

    // All warps then collaboratively load in the first slice of V (V0) and the second slice of K (K1) into shared
        ↪ memory
    G::load<1, false>(k_smem[1], g.Kg, {batch_idx, 1, head_idx_kv, 0});
    // All warps then load in the first slice of K (K0)
    G::load<1, false>(v_smem[0], g.Vg, {batch_idx, 0, head_idx_kv, 0});
    load(k_reg, k_smem[0]);
    asm volatile("s_waitcnt lgkmcnt(0)");
    asm volatile("s_waitcnt vmcnt(2)");
    __builtin_amdgcn_sched_barrier(0);
    __builtin_amdgcn_s_barrier();

    // each warp performs QK0
    zero(att_block[0]);
    swap_layout_and_transpose(k_reg_transposed, k_reg);
    mma_AtB(att_block[0], k_reg_transposed, q_reg_transposed, att_block[0]);

    // each warp performs a partial softmax of QK0
    col_max(max_vec, att_block[0]);
    sub_col(att_block[0], att_block[0], max_vec);
    exp2(att_block[0], att_block[0]);
    sched_barrier_pairs<8, 6, 1>();
    sched_barrier_exp_pairs<8, 4, 1>();

    // conditional stagger
    if (stagger) {
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
    }
```

```
    // All warps then load in the second slice of K (K1)
    load(k_reg, k_smem[1]);
    // All warps then collaboratively load in the third slice of K (K2) into shared memory
    G::load<1, false>(k_smem[0], g.Kg, {batch_idx, 2, head_idx_kv, 0});
    // All warps then collaboratively load in the second slice of V (V1) into shared memory
    G::load<1, false>(v_smem[1], g.Vg, {batch_idx, 1, head_idx_kv, 0});
    asm volatile("s_waitcnt lgkmcnt(0)");
    asm volatile("s_waitcnt vmcnt(4)");
    __builtin_amdgcn_sched_barrier(0);
    __builtin_amdgcn_s_barrier();


    // hot loop
    for (int j = 3; j < num_tiles - 1; j += 2) {
        // Cluster 0:
        //      QK1
        zero(att_block[1]);
        swap_layout_and_transpose(k_reg_transposed, k_reg);
        mma_AtB(att_block[1], k_reg_transposed, q_reg_transposed, att_block[1]);
        //      Finish softmax for QK0
        sub(max_vec_prev, max_vec_prev, max_vec);
        exp2(max_vec_prev, max_vec_prev);
        mul(norm_vec, norm_vec, max_vec_prev);
        col_sum(norm_vec, att_block[0], norm_vec);
        copy(att_block_bf16, att_block[0]);
        sched_barrier_pairs<16, 3, 2>();
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        // Cluster 1:
        //      Load K3 into shared
        G::load<1, false>(k_smem[1], g.Kg, {batch_idx, j, head_idx_kv, 0});
        //      Load V0 into registers
        load(v_reg, v_smem[0]);
        asm volatile("s_waitcnt lgkmcnt(0)");
        asm volatile("s_waitcnt vmcnt(4)");
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        // Cluster 2:
        //      A0V0
        __builtin_amdgcn_s_setprio(1);
        mul_col(o_reg, o_reg, max_vec_prev);
        __builtin_amdgcn_sched_barrier(0);
        mma_AtB(o_reg, v_reg, att_block_bf16, o_reg);
        //      Partial softmax for QK1
        copy(max_vec_prev, max_vec);
        col_max(max_vec, att_block[1], max_vec);
        sub_col(att_block[1], att_block[1], max_vec);
        exp2(att_block[1], att_block[1]);
        sched_barrier_pairs<8, 6, 3>();
        sched_barrier_exp_pairs<8, 4, 3>();
        __builtin_amdgcn_s_setprio(0);
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        // Cluster 3:
        //      Load V2 into shared
        G::load<1, false>(v_smem[0], g.Vg, {batch_idx, j - 1, head_idx_kv, 0});
        //      Load K2 into registers
        load(k_reg, k_smem[0]);
        asm volatile("s_waitcnt lgkmcnt(0)");
        asm volatile("s_waitcnt vmcnt(4)");
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);
```

```
        // Cluster 4:
        //      QK2
        zero(att_block[0]);
        swap_layout_and_transpose(k_reg_transposed, k_reg);
        mma_AtB(att_block[0], k_reg_transposed, q_reg_transposed, att_block[0]);
        //      Finish softmax for QK1
        sub(max_vec_prev, max_vec_prev, max_vec);
        exp2(max_vec_prev, max_vec_prev);
        mul(norm_vec, norm_vec, max_vec_prev);
        col_sum(norm_vec, att_block[1], norm_vec);
        copy(att_block_bf16, att_block[1]);
        sched_barrier_pairs<16, 3, 4>();
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        // Cluster 5:
        //      Load K4 into shared
        G::load<1, false>(k_smem[0], g.Kg, {batch_idx, j + 1, head_idx_kv, 0});
        //      Load V1 into registers
        load(v_reg, v_smem[1]);
        asm volatile("s_waitcnt lgkmcnt(0)");
        asm volatile("s_waitcnt vmcnt(4)");
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        // Cluster 6:
        //      A1V1
        __builtin_amdgcn_s_setprio(1);
        mul_col(o_reg, o_reg, max_vec_prev);
        __builtin_amdgcn_sched_barrier(0);
        mma_AtB(o_reg, v_reg, att_block_bf16, o_reg);
        //      Partial softmax for QK2
        copy(max_vec_prev, max_vec);
        col_max(max_vec, att_block[0], max_vec);
        sub_col(att_block[0], att_block[0], max_vec);
        exp2(att_block[0], att_block[0]);
        sched_barrier_pairs<8, 6, 5>();
        sched_barrier_exp_pairs<8, 4, 5>();
        __builtin_amdgcn_s_setprio(0);
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);

        // Cluster 7:
        //      Load V3 into shared
        G::load<1, false>(v_smem[1], g.Vg, {batch_idx, j, head_idx_kv, 0});
        //      Load K3 into registers
        load(k_reg, k_smem[1]);
        asm volatile("s_waitcnt lgkmcnt(0)");
        asm volatile("s_waitcnt vmcnt(4)");
        __builtin_amdgcn_sched_barrier(0);
        __builtin_amdgcn_s_barrier();
        __builtin_amdgcn_sched_barrier(0);
    }

    // Epilogue not shown

    // Conclusion
    if (!stagger) {
        __builtin_amdgcn_s_barrier();
    }

    qo_tile<D, float, row_l, rt_32x32_s> o_reg_transposed;
    swap_layout_and_transpose(o_reg_transposed, o_reg);
    store<1>(g.Og, o_reg_transposed, {batch_idx, tile_idx, head_idx, 0});

    // multiply by ln(2)
    mul(max_vec, max_vec, 0.69314718056f);
    log(norm_vec, norm_vec);
    add(norm_vec, norm_vec, max_vec);
    store(g.L_vec, norm_vec, {batch_idx, head_idx, 0, tile_idx});
}
```

Figure 24. HIPKITTENS non-causal attention forwards kernel that competes with the assembly kernel provided in AMD's AITER library.

F CASE STUDY: PRELIMINARY FINDINGS FOR THE FP6 GEMM

We discuss our **initial empirical observations** of FP6 hardware behavior on AMD's MI350X and MI355X GPUs. The goal is to characterize how FP6 memory movement and matrix-core operations behave in practice. AMD's own CK library baselines are **unoptimized** at the time of writing, and our results should be interpreted as preliminary. FP6 is exciting because AMD matrix cores achieve twice the peak FLOPS for FP6 as NVIDIA devices. However, in practice, we incur challenges in loading FP6 values to and from memory, which impact our ability to achieve high utilization.

Memory Loads. Our FP6 GEMM kernel uses 4 waves to cooperatively load a 128×128 tile from global memory in each memory cluster. With the FP6 datatype, that tile is $128 \times 128 \times 6/8 = 12{,}288$ bytes, or 48 bytes per thread. We consider the following CDNA4 instructions as options to load from global memory:

- `buffer_load_dwordx4` minimizes the number of instructions issued per thread (at 3 per tile). However, naively using this instruction to load to shared memory and then using `ds_read_b128` followed by `ds_read_b64` to read the 24 consecutive bytes owned by each thread from shared memory into registers causes shared memory alignment issues, because `ds_read_b128` must be 16-byte aligned for maximum performance. For example, the second thread on each row of the tile performs a `ds_read_b128` at an offset of 24 bytes into the row. Our kernel becomes shared-memory bound in this configuration.

  One solution is for every second thread (`laneid % 2 == 1`) to flip the two shared memory load instructions, so `ds_read_b64` loads the first 8 bytes of the thread's data, and `ds_read_b128` loads the next 16 bytes. We find that loading into a register destination that depends on the thread ID requires the use of slow scratch storage. Instead, for threads with flipped `ds_read_*` instructions, we continue reading `ds_read_b128` into the first four registers and `ds_read_b64` into the second two registers, regardless of thread. This approach then requires a shuffle, where we conditionally swap the two regions in registers based on thread ID. This "breaks the wave," resulting in two jump instructions in addition to the VALU instructions required to move the memory. We find that the incurred jump + VALU from shuffling comprises 49% of the cycles in the kernel's hot loop, resulting in a kernel that achieves only 2430 TFLOPs.

  Alternatively, we can use two `ds_read_b96` instructions. This approach causes misalignment between the 16-byte chunks read by `buffer_load_dwordx4` from global to shared memory and the 12-byte chunks read by `ds_read_b96`. As a result, we are unable to swizzle the data in shared memory, resulting in 4-way bank conflicts. Given these issues with `buffer_load_dwordx4`, we look at other options for loading FP6 values from global to shared memory.

- `buffer_load_dwordx3` is appealing for FP6 because it allows a warp to exactly load an 8×128 tile from global memory. It also supports swizzling. Unfortunately, this instruction for FP6 does not outperform FP8 in instruction issue count: each thread issues 4 instructions to load a 128×128 tile, which is the same number used in our FP8 GEMM kernel. This instruction also loads each thread's 12 bytes of data at a stride of 16 bytes, wasting 25% of the resulting shared memory tile and rendering 8 of the 32 banks available to `ds_read_b96` (Table 7) unused. Despite these drawbacks, we find this is likely the most compelling global-to-shared load instruction for our FP6 GEMM use case. `ds_read_b96` is the natural LDS-to-register load instruction for data loaded to shared memory using `buffer_load_dwordx3`. We find that `ds_read_b96` works well on 16-byte aligned shared memory addresses.

- `buffer_load_dwordx1` avoids shared memory waste, alignment issues, and swizzling limitations, but the kernel becomes bottlenecked by the number of instructions issued, resulting in a slower kernel than the one built with `buffer_load_dwordx3`.

Matrix Core Operation. With the BF16 and FP8 GEMMs, we found the fastest MFMA instruction is the smaller one available for the given dtype (Figure 3), which in this case is `v_mfma_f32_16x16x128_f8f6f4`. In this instruction, each thread owns 32 consecutive elements, or 24 consecutive bytes, of each FP6 operand matrix. In our kernel, each thread issues two `ds_read_b96` instructions: one for the first 12 bytes, and one for the latter 12 bytes. `ds_read_b96` constrains the destination base address in the register file to a 16 byte aligned address, so in order to achieve continuity between the 24 bytes of data, we must issue three `v_mov_b32_e32` instructions to shuffle each of the three registers written to by the second `v_mov_b32_e32` down one register. Note that this shuffle is not as expensive as the shuffle required when using `buffer_load_dwordx4` with `ds_read_b128` and `ds_read_b64` because less data is shuffled and no waves break due to this shuffle. We note that HIPCC does not handle this shuffle requirement well, and this kernel at size 16384x16384x16384 spills 54 registers to scratch memory, resulting in a slow and incorrect kernel. To remedy

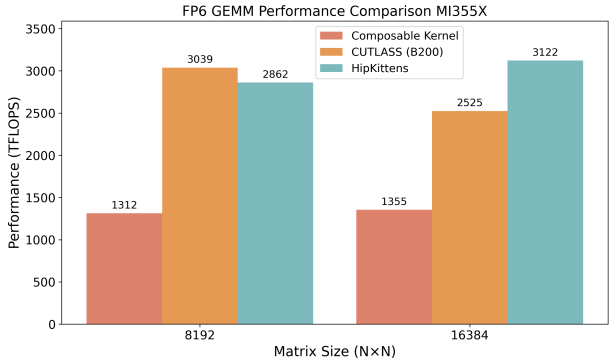


Figure 25. **FP6 GEMM.** We compare performance of FP6 GEMM on square shapes 8192 and 16384 across AMD's CK library, NVIDIA's CUTLASS on B200, and HIPKITTENS. We use 500 iterations of warmup and 100 iterations of measurement.

this, we rewrote our FP6 GEMM while explicitly scheduling registers, allowing us to intelligently set which registers are used in this shuffle and completely removing register spills. This approach was not without hazards. We need to account for the instruction latency of `v_mov_b32_e32`, so we manually ensure that at least 8 cycles of instructions are between the `v_mov_b32_e32` instructions and the dependent MFMA instruction. In some cases, this involves manually inserting `v_nop` instructions.

Results. FP6 matrix core speed is a standout feature of MI350X and MI355X GPUs. Our FP6 GEMM kernel outperforms AMD's CK implementation and attains performance comparable to our own FP8 GEMM. These results reflect our initial observations of FP6 hardware behavior; we expect additional improvements in future work.