HipKittens: Fast and Furious AMD Kernels

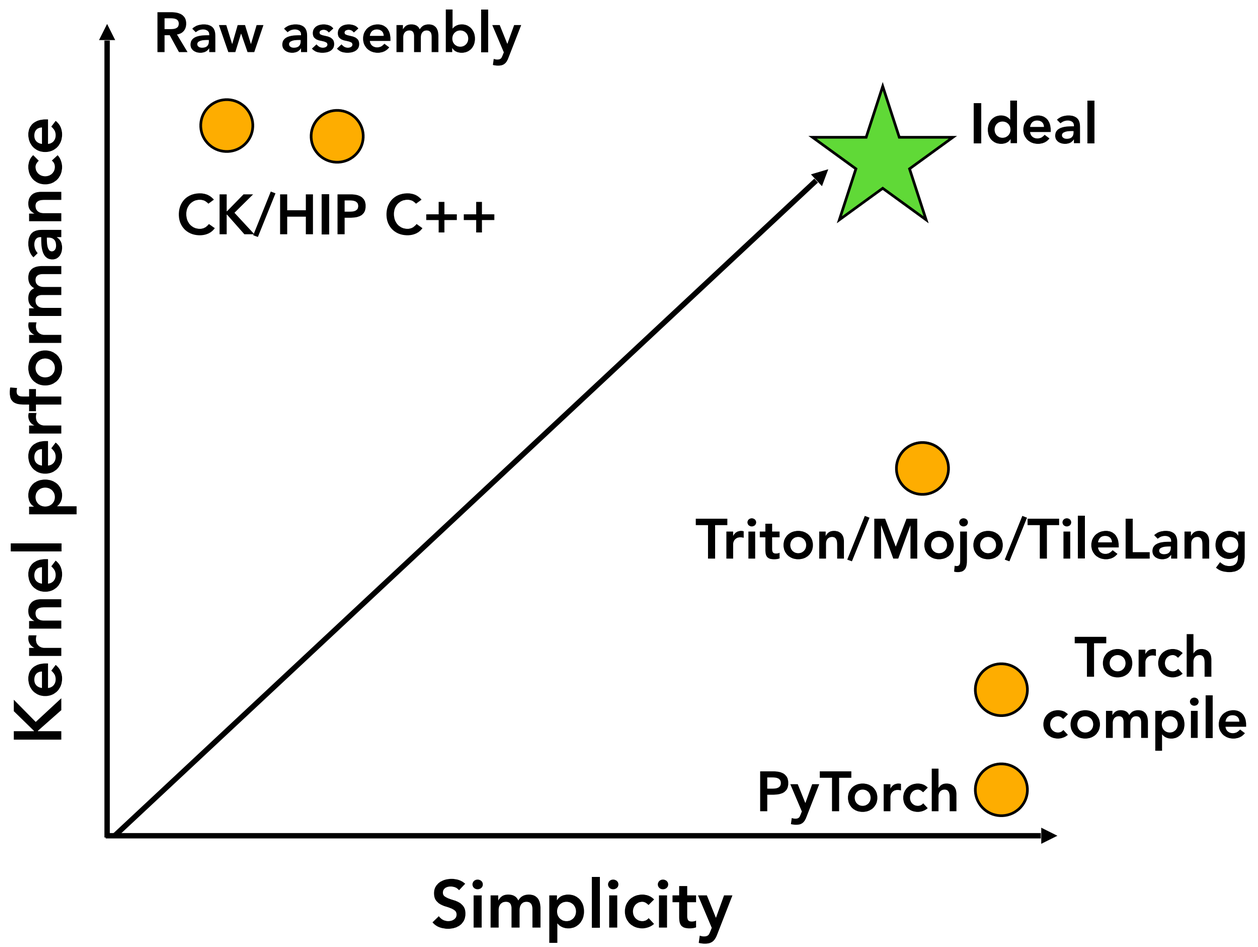
William Hu, Drew Wadsworth, Sean Siddens, Stanley Winata, Dan Fu, Ryan Swann, Muhammad Osama, Chris Ré, Simran Arora

AMD offers state-of-the-art peak compute and memory!

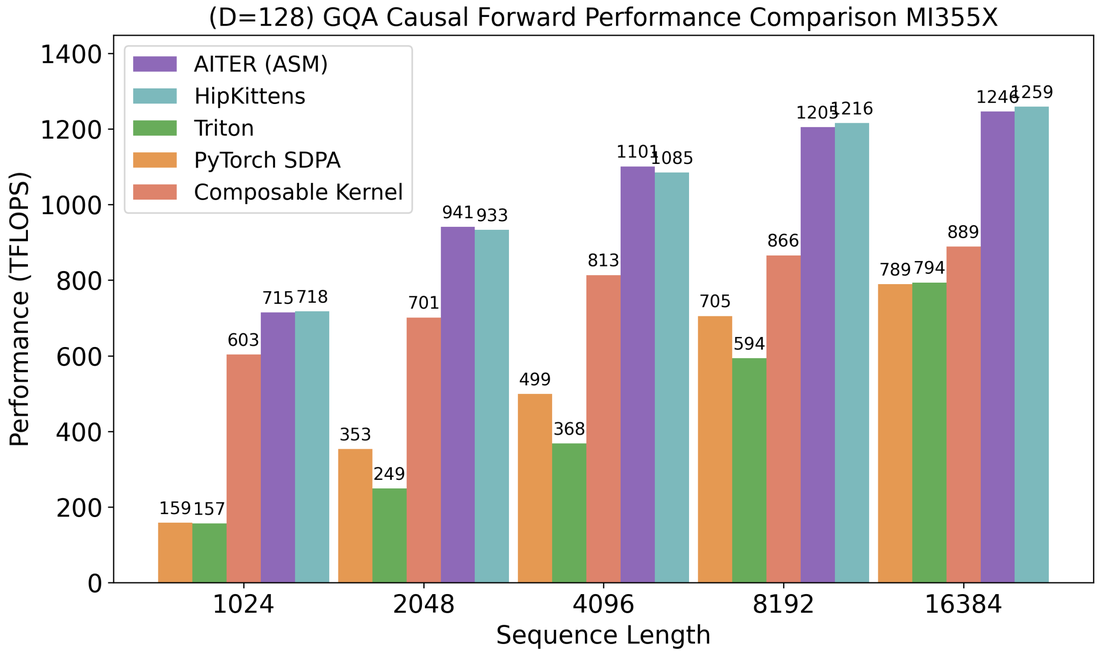
Specification	NVIDIA B200 SXM5	AMD MI355X OAM
BF16 matrix / tensor	2.2 PFLOPs	2.5 PFLOPs
MXFP8 matrix / tensor	4.5 PFLOPs	5.0 PFLOPs
MXFP6 matrix / tensor	4.5 PFLOPs	10.1 PFLOPs
MXFP4 matrix / tensor	9.0 PFLOPs	10.1 PFLOPs
Memory capacity	180 GB	288 GB
Memory bandwidth	8.0 TB/s	8.0 TB/s

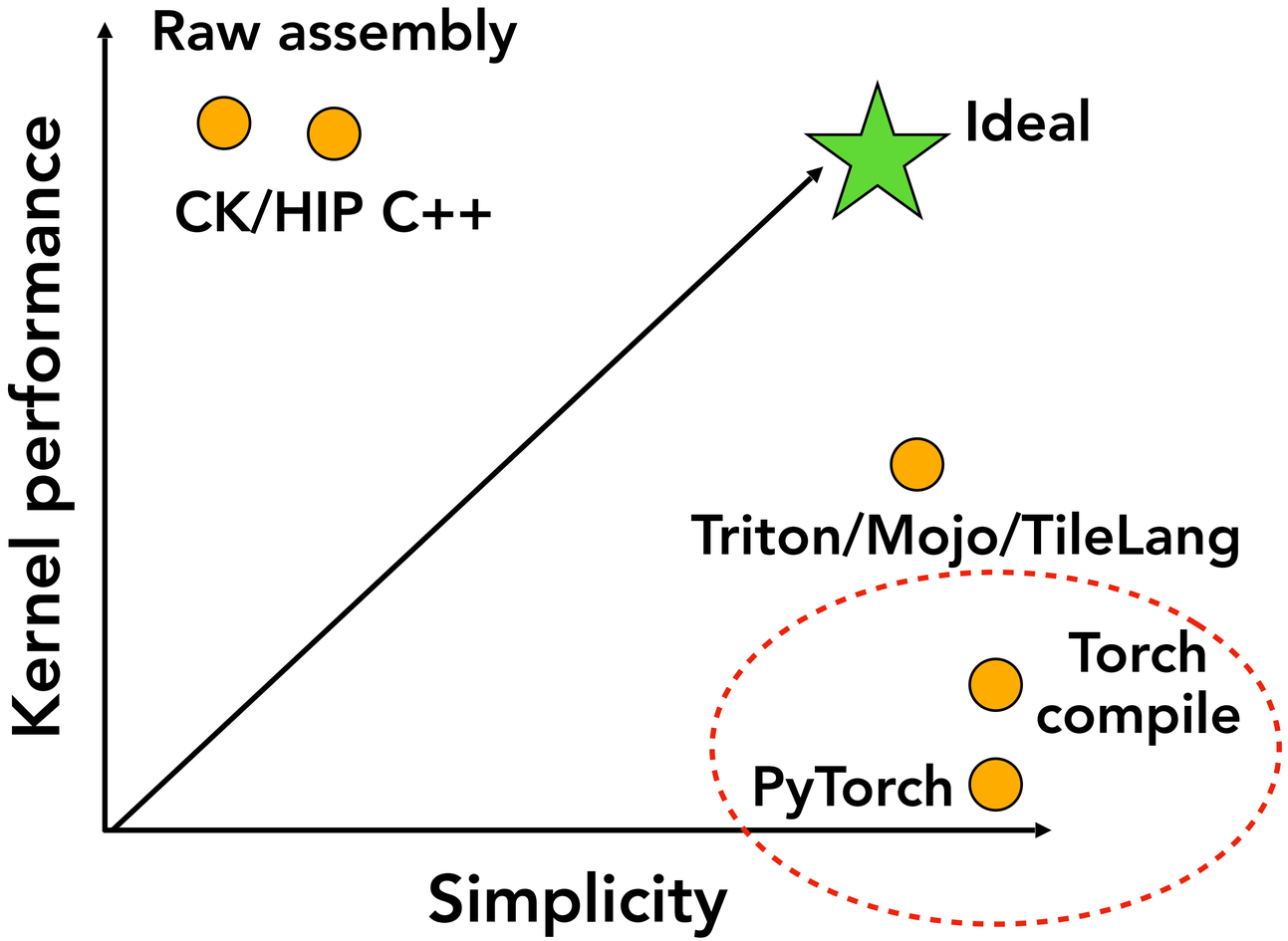
But, this performance is locked away from AI workflows if we lack mature software.

The current AMD software landscape lacks a simple and performant framework for developing a breadth of AI kernels.



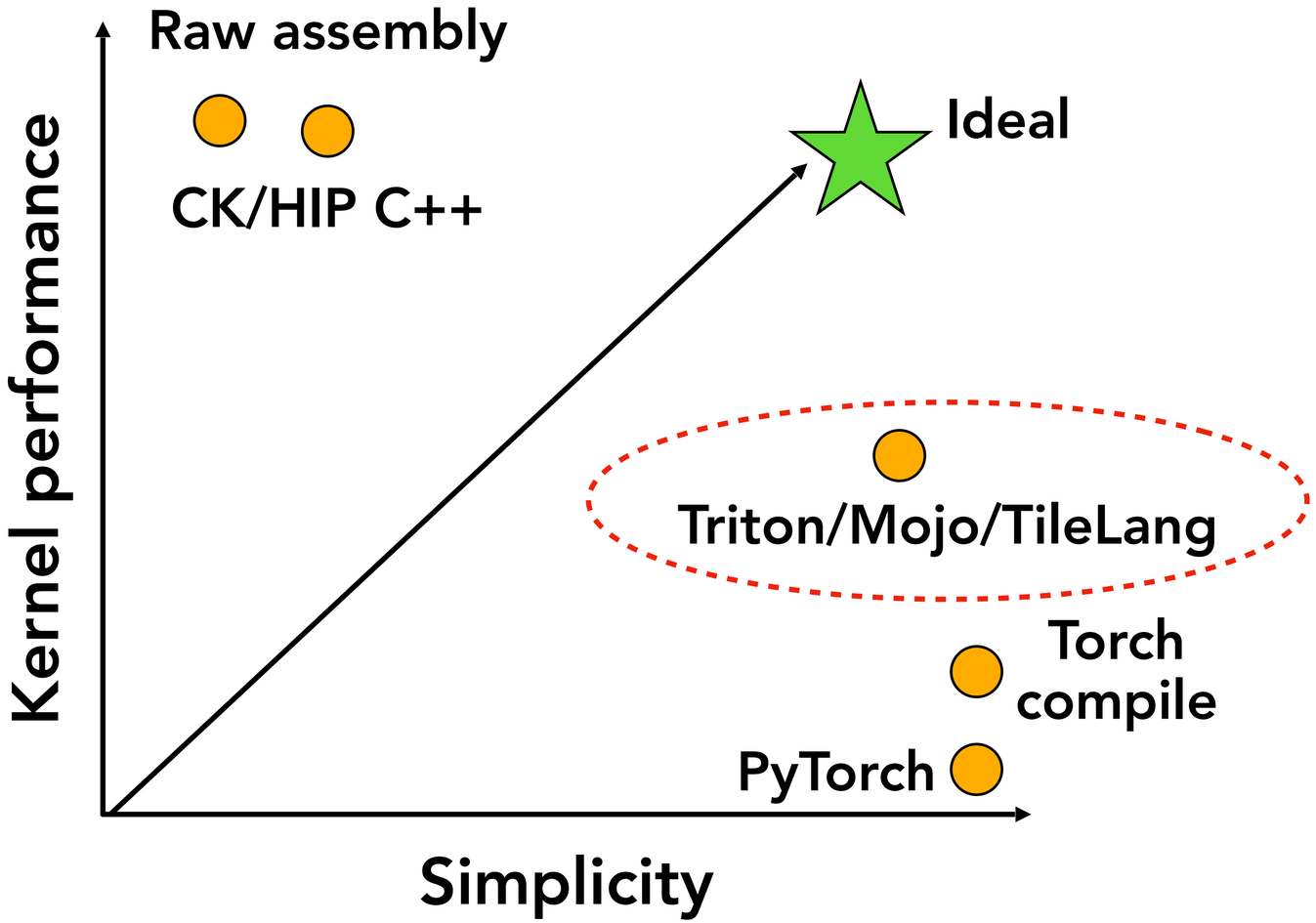
PyTorch does not provide sufficient control over the hardware. Current AMD backends are slow.



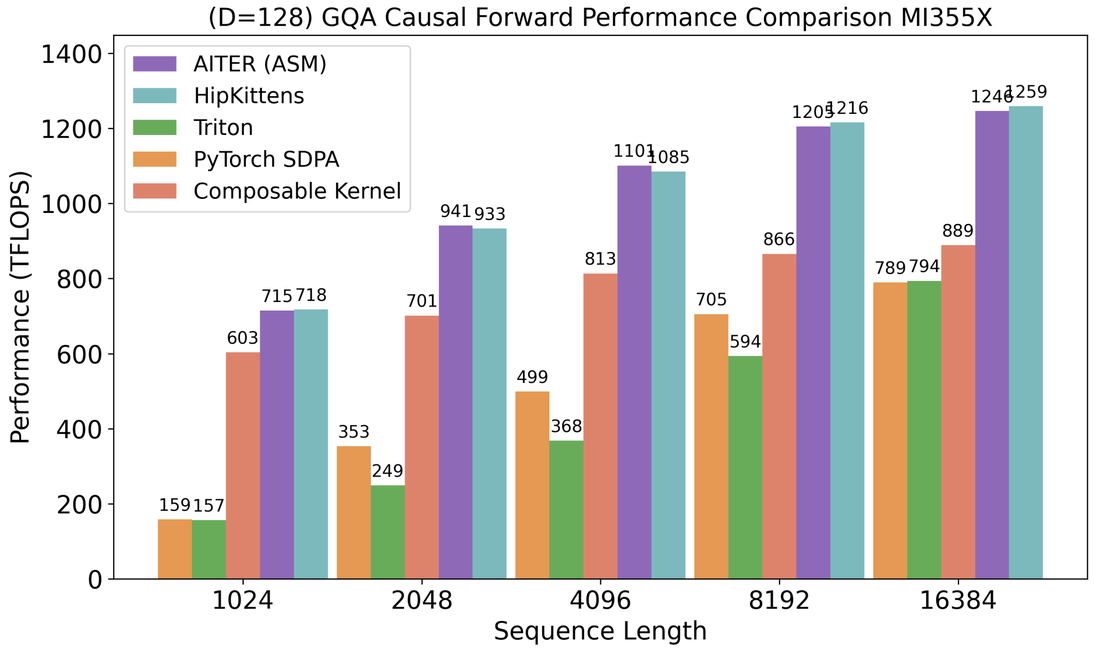


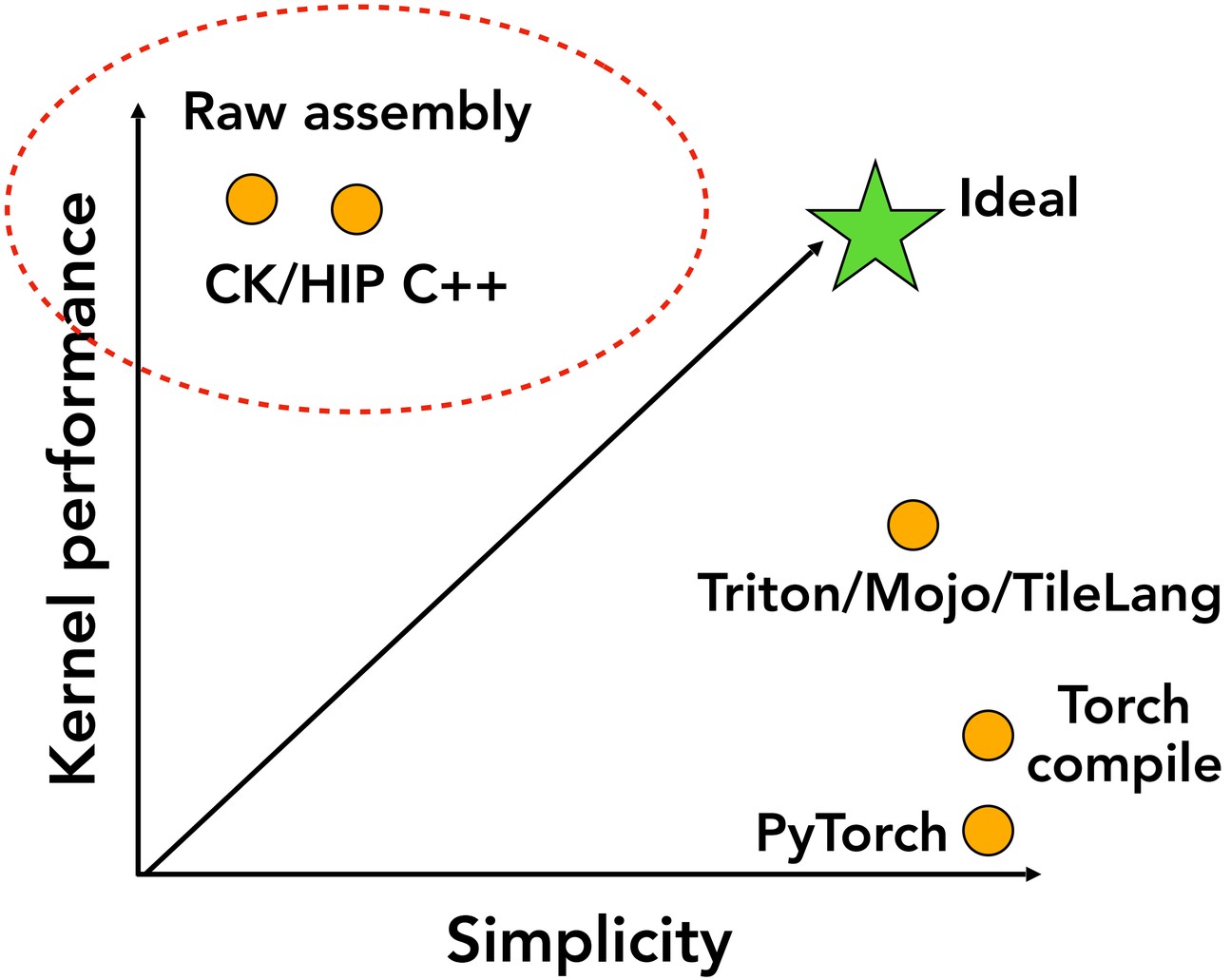
Compilers obfuscate understanding and prevent the developer from using the full hardware.

- The compiler, not the developer, controls the register pool
- Triton struggles with **register lifetime tracking** and **lowering memory accesses** to the most performant intrinsics



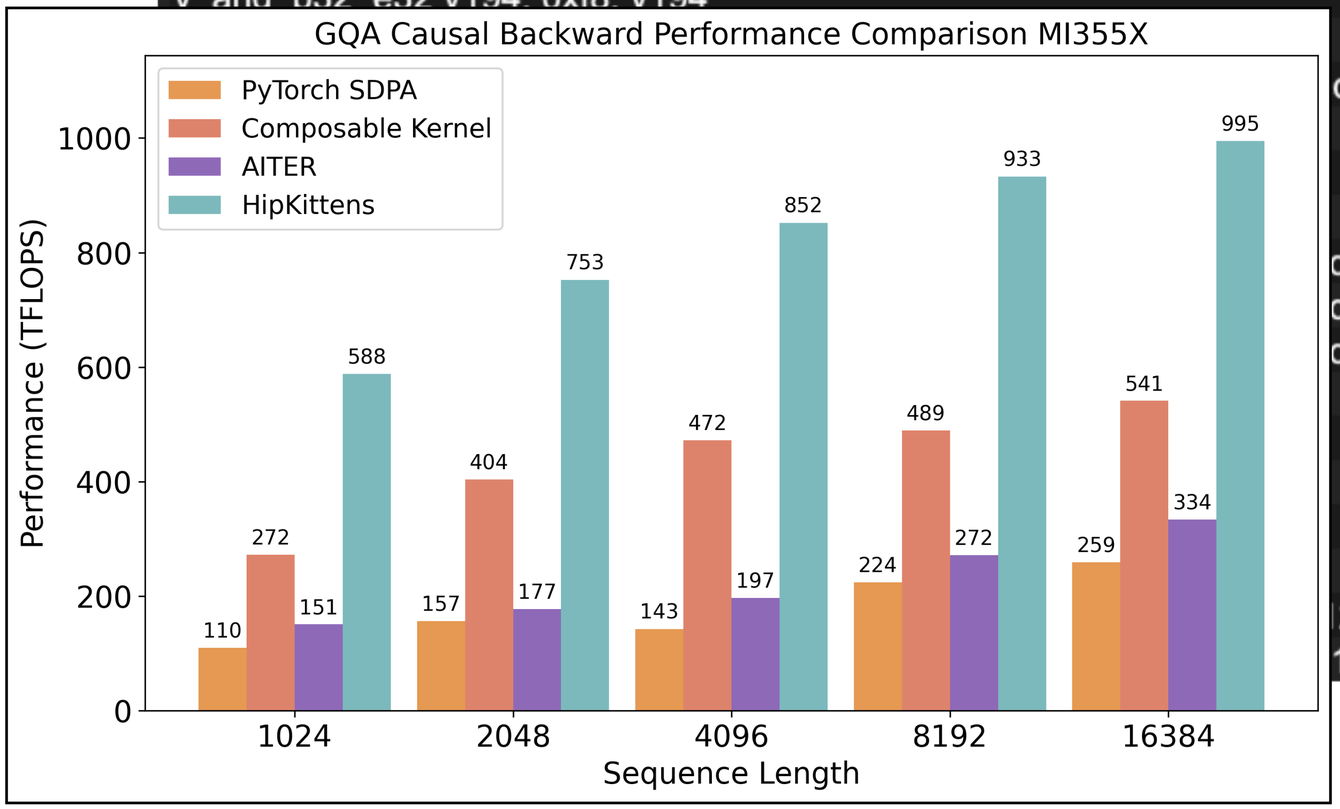
Raw assembly gives developers full control over the hardware and yields fast kernels!



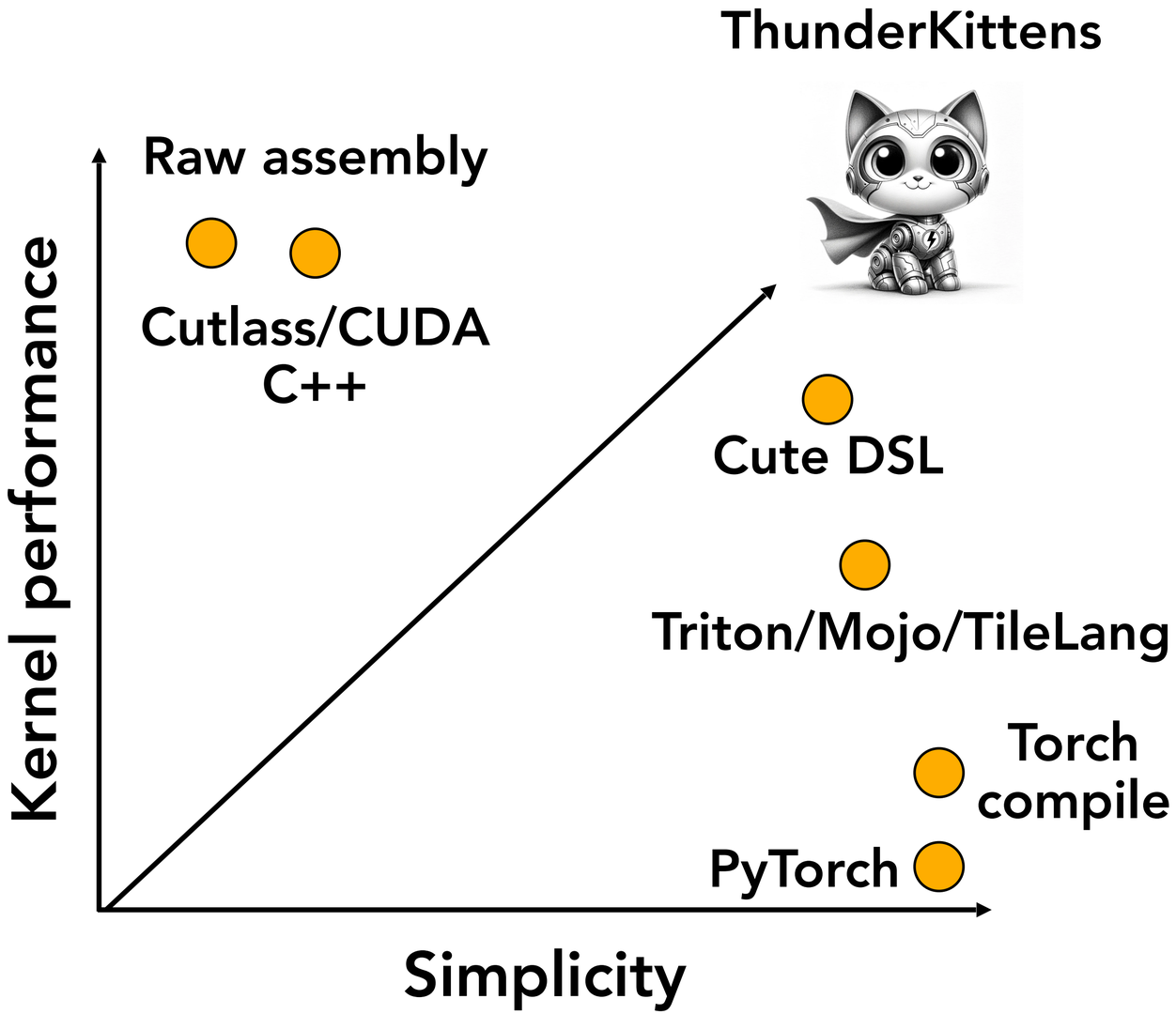


Raw assembly is hard to scale to the breadth of AI workloads. It's hard to read and modify.

```
v_sub_u32_e32 v164, v164, v194
v_lshrrev_b16_sdwa v194, v255, sext(v164) dst_sel:DWORD dst_unused:UNUSED_PAD src0_sel:DWORD src1_sel:BYTE_0
v_and_b32_e32 v194, 7, v194
v_add_u16_e32 v194, v164, v194
v_ashrrev_i16_sdwa v197, v250, sext(v194) dst_sel:DWORD dst_unused:UNU
v_and_b32_e32 v194, 0xf8, v194
```



The NVIDIA software landscape includes similar frameworks that trade off performance and simplicity.

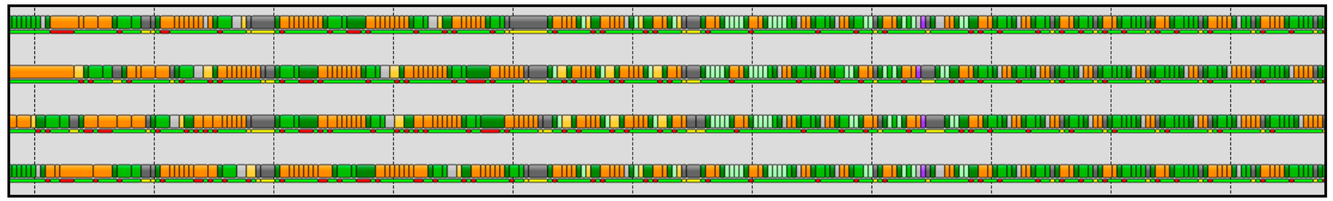


Research question

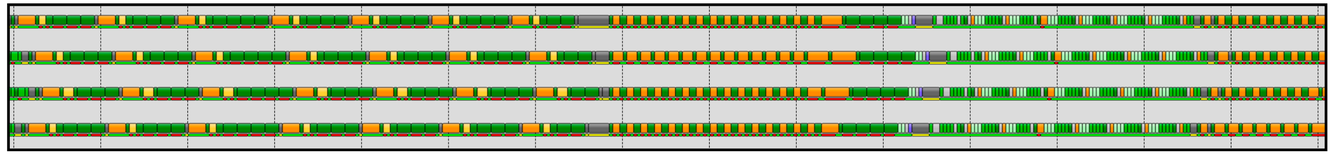
Are entirely new programming primitives needed to simplify AMD kernel development, or do existing primitives suffice?

Tiles continue to be an important primitive for AI kernels

AMD Hipblaslt BF16 GEMM



AMD AITER FA3



Nvidia CUTLASS FA3

```
1962 00007fd1 b7be0890        FMUL.FTZ R86, R86, R2
1963 00007fd1 b7be08a0        F2FP.F16.F32.PACK_AB R20, R41, R20
1964 00007fd1 b7be08b0        F2FP.F16.F32.PACK_AB R21, R43, R21
1965 00007fd1 b7be08c0        F2FP.F16.F32.PACK_AB R22, R45, R22
1966 00007fd1 b7be08d0        F2FP.F16.F32.PACK_AB R23, R47, R23
1967 00007fd1 b7be08e0        IADD3 R3, R93, -UR4, RZ
1968 00007fd1 b7be08f0        F2FP.F16.F32.PACK_AB R12, R49, R12
1969 00007fd1 b7be0900        STSM.16.M88.4 [R9], R28
1970 00007fd1 b7be0910        F2FP.F16.F32.PACK_AB R13, R51, R13
1971 00007fd1 b7be0920        F2FP.F16.F32.PACK_AB R14, R53, R14
1972 00007fd1 b7be0930        STSM.16.M88.4 [R8], R24
1973 00007fd1 b7be0940        F2FP.F16.F32.PACK_AB R15, R55, R15
1974 00007fd1 b7be0950        IADD3 R2, R92, -UR4, RZ
1975 00007fd1 b7be0960        STSM.16.M88.4 [R3], R20
1976 00007fd1 b7be0970        F2FP.F16.F32.PACK_AB R4, R57, R4
1977 00007fd1 b7be0980        F2FP.F16.F32.PACK_AB R5, R59, R5
1978 00007fd1 b7be0990        STSM.16.M88.4 [R2], R12
1979 00007fd1 b7be09a0        F2FP.F16.F32.PACK_AB R6, R61, R6
1980 00007fd1 b7be09b0        F2FP.F16.F32.PACK_AB R7, R63, R7
1981 00007fd1 b7be09c0        IADD3 R0, R94, -UR4, RZ
1982 00007fd1 b7be09d0        F2FP.F16.F32.PACK_AB R16, R65, R64
1983 00007fd1 b7be09e0        F2FP.F16.F32.PACK_AB R17, R67, R17
1984 00007fd1 b7be09f0        STSM.16.M88.4 [R0], R4
```

```
1937 00007fd1 b7be0700        FMUL.FTZ R5, R58, R2
1938 00007fd1 b7be0710        FMUL.FTZ R63, R63, R2
1939 00007fd1 b7be0720        FMUL.FTZ R7, R62, R2
1940 00007fd1 b7be0730        F2FP.F16.F32.PACK_AB R28, R16, R9
1941 00007fd1 b7be0740        FMUL.FTZ R67, R67, R2
1942 00007fd1 b7be0750        F2FP.F16.F32.PACK_AB R29, R3, R29
1943 00007fd1 b7be0760        FMUL.FTZ R17, R66, R2
1944 00007fd1 b7be0770        F2FP.F16.F32.PACK_AB R30, R10, R8
1945 00007fd1 b7be0780        FMUL.FTZ R71, R71, R2
1946 00007fd1 b7be0790        F2FP.F16.F32.PACK_AB R31, R0, R31
1947 00007fd1 b7be07a0        FMUL.FTZ R19, R70, R2
1948 00007fd1 b7be07b0        IADD3 R9, R91, -UR4, RZ
1949 00007fd1 b7be07c0        BAR.SYNC.DEFER_BLOCKING 0x6, 0x100
1950 00007fd1 b7be07d0        F2FP.F16.F32.PACK_AB R24, R33, R24
1951 00007fd1 b7be07e0        FMUL.FTZ R75, R75, R2
1952 00007fd1 b7be07f0        F2FP.F16.F32.PACK_AB R25, R35, R25
1953 00007fd1 b7be0800        FMUL.FTZ R74, R74, R2
1954 00007fd1 b7be0810        F2FP.F16.F32.PACK_AB R26, R37, R36
1955 00007fd1 b7be0820        FMUL.FTZ R79, R79, R2
1956 00007fd1 b7be0830        F2FP.F16.F32.PACK_AB R27, R39, R27
1957 00007fd1 b7be0840        FMUL.FTZ R11, R78, R2
1958 00007fd1 b7be0850        IADD3 R8, R90, -UR4, RZ
1959 00007fd1 b7be0860        FMUL.FTZ R83, R83, R2
```

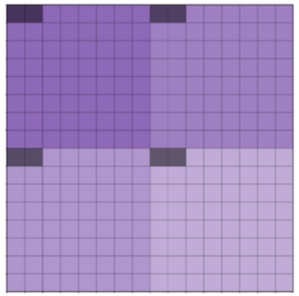
Modular Mojo AMD GEMM on MI300X

```
# Stage 3: Main computation loop - Pipelined execution with double buffering
for _ in range(2, K // BK):

    @parameter
    for k_tile_idx in range(1, num_k_tiles):
        load_tiles_from_shared[k_tile_idx]()

    mma[0, swap_a_b=True](a_tiles, b_tiles, c_reg_tile)

    barrier()

    copy_tiles_to_shared()
    load_tiles_from_dram()

    @parameter
    for k_tile_idx in range(1, num_k_tiles):
        mma[k_tile_idx, swap_a_b=True](a_tiles, b_tiles, c_reg_tile)

    barrier()

    load_tiles_from_shared[0]()

    amd_scheduling_hints[
        BM=BM,
        BN=BN,
        BK = Int(BK),
        num_m_mmas=num_m_mmas,
        num_n_mmas=num_n_mmas,
        num_k_tiles=num_k_tiles,
        simd_width=simd_width,
        num_threads = Int(config.num_threads()),
        scheduler_hint = config.scheduler_hint,
    ]()
```

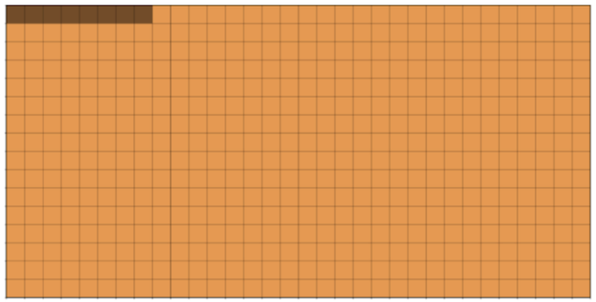
Differences of AMD register and shared memory tiles

AMD matrix cores and shared memory bank behavior impose restrictions on how register and shared memory tiles are instantiated.

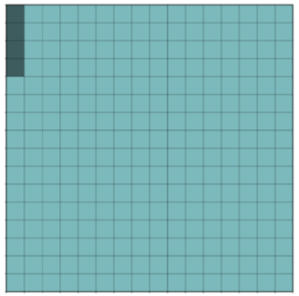
Instr.	Banks	Phase	Active threads
DS_READ_B128	64	0	0-3, 12-15, 20-27
		1	4-11, 16-19, 28-31
		2	32-35, 44-47, 52-59
		3	36-43, 48-51, 60-63
DS_READ_B96	32	0	0-3, 20-23
		1	4-7, 16-19
		2	8-11, 28-31
		3	12-15, 24-27
		4	32-35, 52-55
		5	36-39, 48-51
		6	40-43, 60-63
		7	44-47, 56-59
DS_WRITE_B64	32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



(a) NVIDIA core matrices



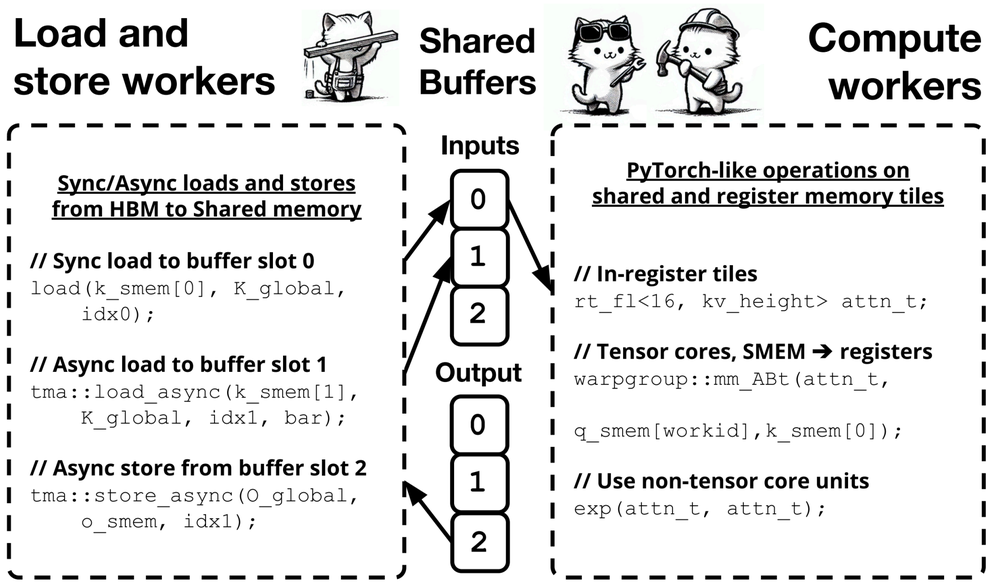
(b) AMD 16×16×32 MFMA A/B matrix



(c) AMD 16×16×32 C/D matrix

Does wave specialization work for AMD?

Peak performance Nvidia kernels and DSLs rely on wave specialization, but does the same pattern translate to AMD GPUs?

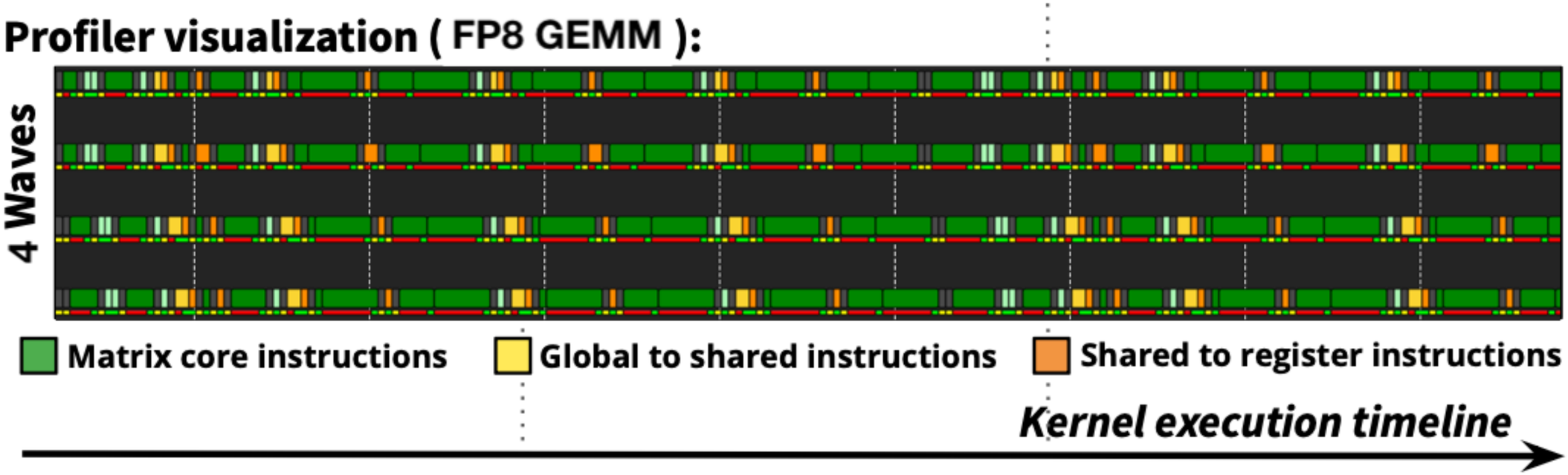


# P / # C	MFMA Shape	Output	TFLOPS	Hardware
HK 4 / 8	16x16x32	128x256	893	MI355X
HK 4 / 12	16x16x32	192x256	1278	MI355X
HK 0 / 8	16x16x32	192x256	1281	MI355X
HK 0 / 8	16x16x32	256x256	**1605**	MI355X
TK	256x256x16	256x256	1538	B200
CUTLASS	256x256x16	256x256	1570	B200

What now?

The most optimized AMD kernels rely on 2 scheduling patterns to achieve good occupancy:

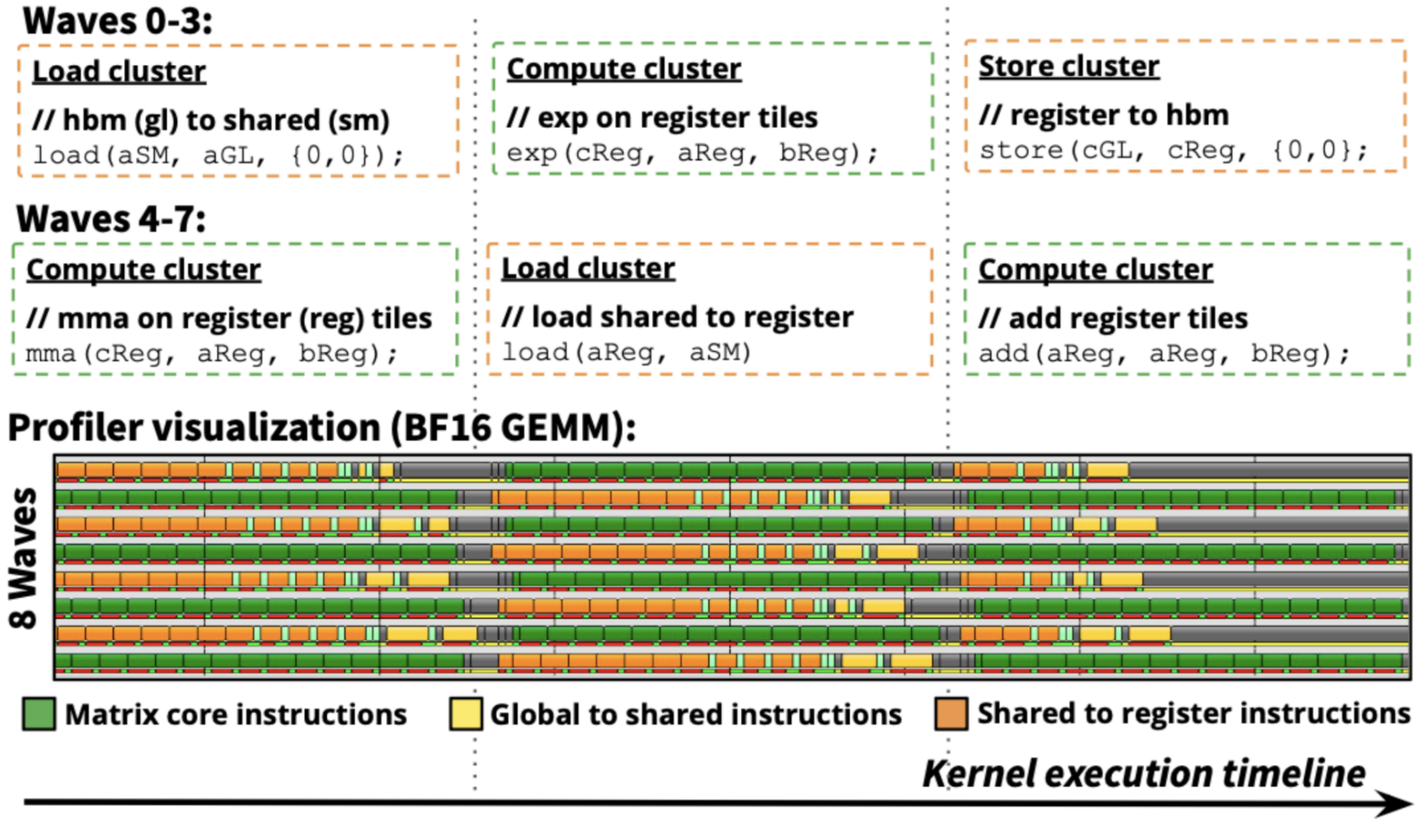
- 4-wave finely interleaved
- 8-wave ping-pong



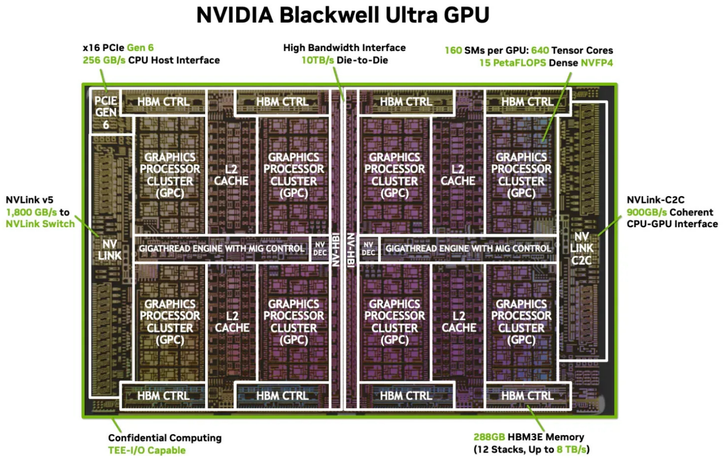
What now?

The most optimized AMD kernels rely on 2 scheduling patterns to achieve good occupancy:

- 4-wave finely interleaved
- 8-wave ping-pong

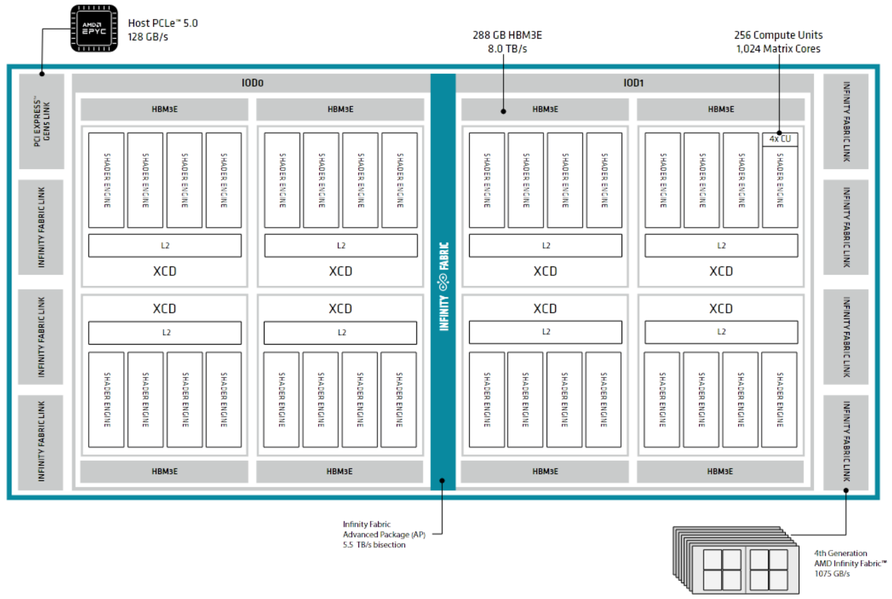


Chiplet architectures and larger scale-up domains

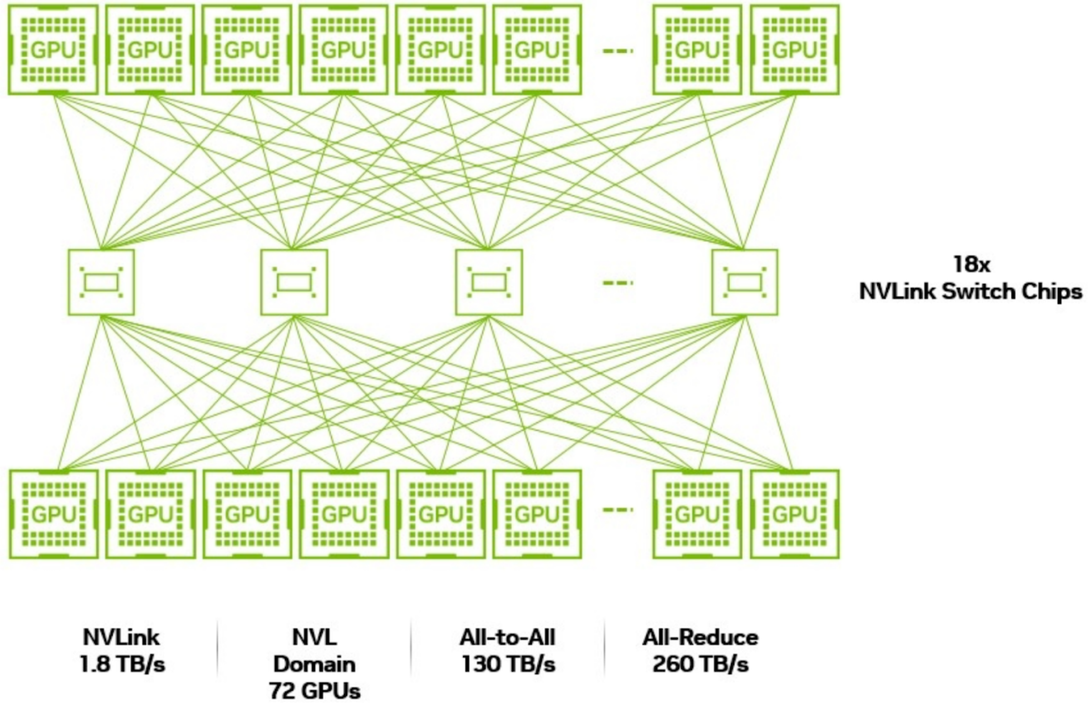


Lam, C. (2025). *Nvidia's B200: Keeping the CUDA juggernaut rolling*. Chips and Cheese. Retrieved from chipsandcheese.com

Figure 5. AMD Instinct MI350 Series GPU Multi-Die Chiplet, Memory and I/O System



.AMD. (2025). *AMD CDNA 4 architecture whitepaper*. Advanced Micro Devices.



Mitra et al. (2025). *How NVIDIA GB200 NVL72 and NVIDIA Dynamo boost inference performance for MoE models*. NVIDIA Technical Blog.

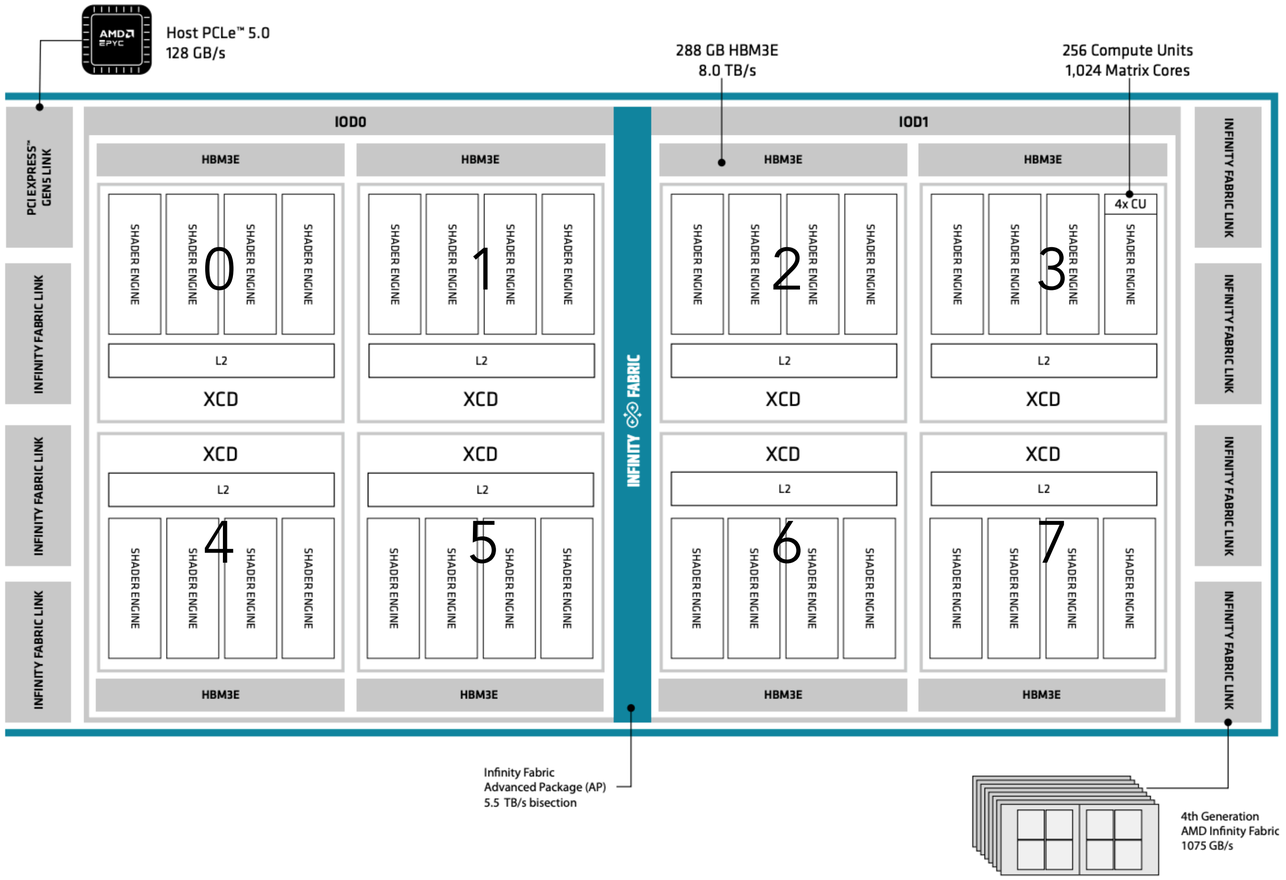
Disaggregated memory

Modern AMD GPUs use a chiplet architecture - 8 accelerator complex dies (XCDs) are brought together in a GPU - and has a **disaggregated cache hierarchy**

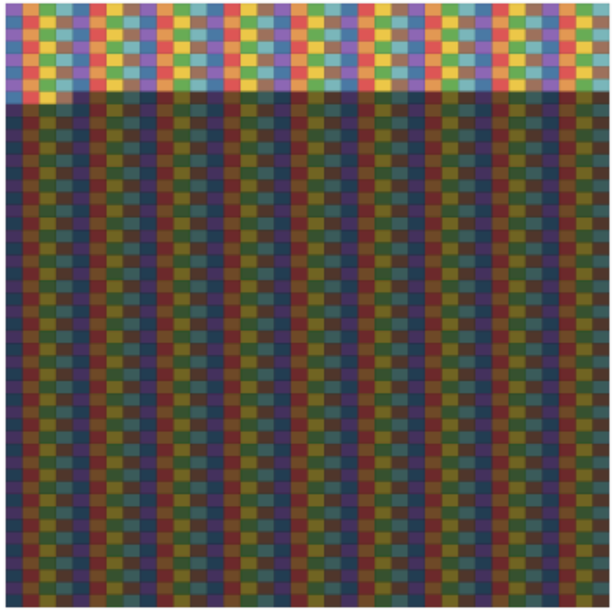
- Private L2 caches
- Shared last level cache (LLC)

Work is assigned round-robin between the different XCDs

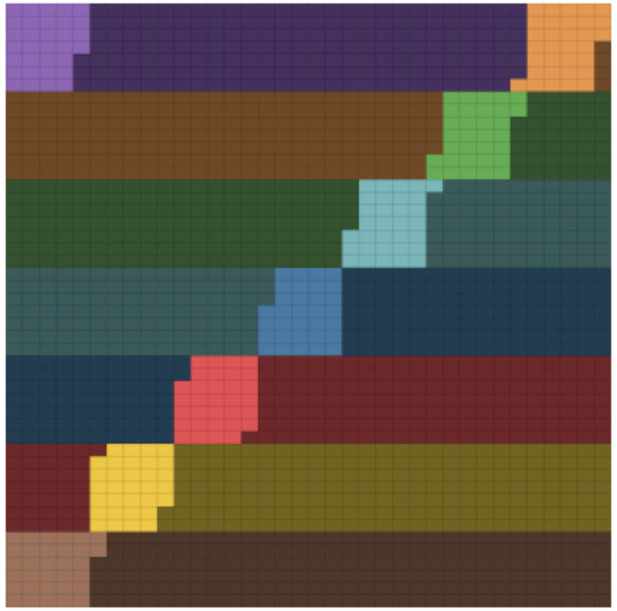
Figure 5. AMD Instinct MI350 Series GPU Multi-Die Chiplet, Memory and I/O System



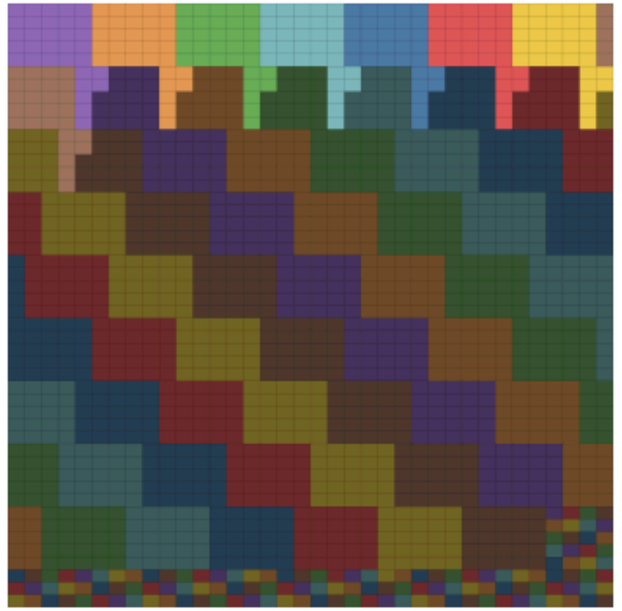
Achieving good LLC + L2 reuse on GEMMs



(a) Row-major



(b) XCD (W7, C216)



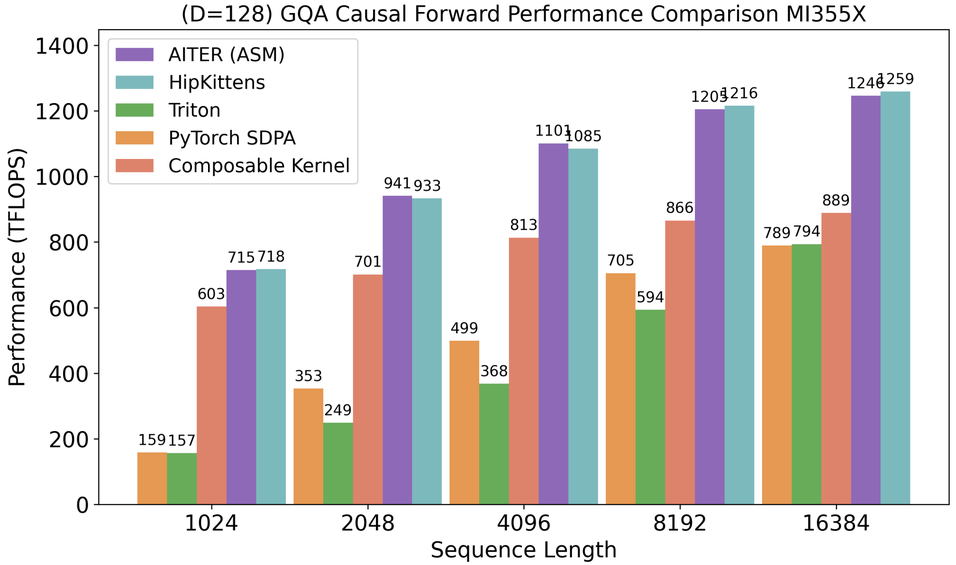
(c) XCD (W5, C25)

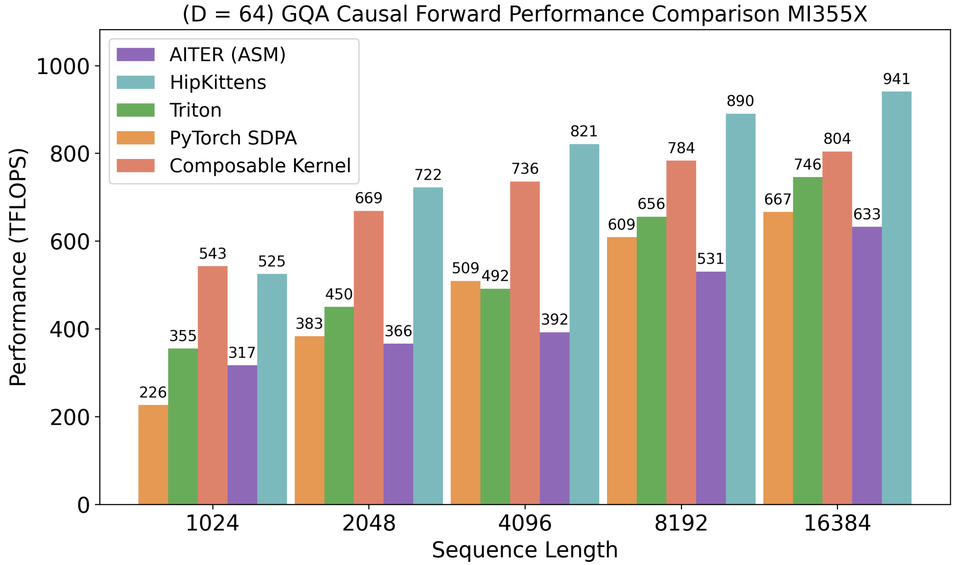
Block Order	L2 %	LLC %	Mem. BW	TFLOPS
Matrix Multiply (M=N=K=9216, MT 192x256x64)				
Row-major	55%	95%	15.1 TB/s	1113
XCD ($W7/C216$)	79%	24%	14.9 TB/s	991
XCD ($W5/C25$)	75%	93%	18.3 TB/s	1145

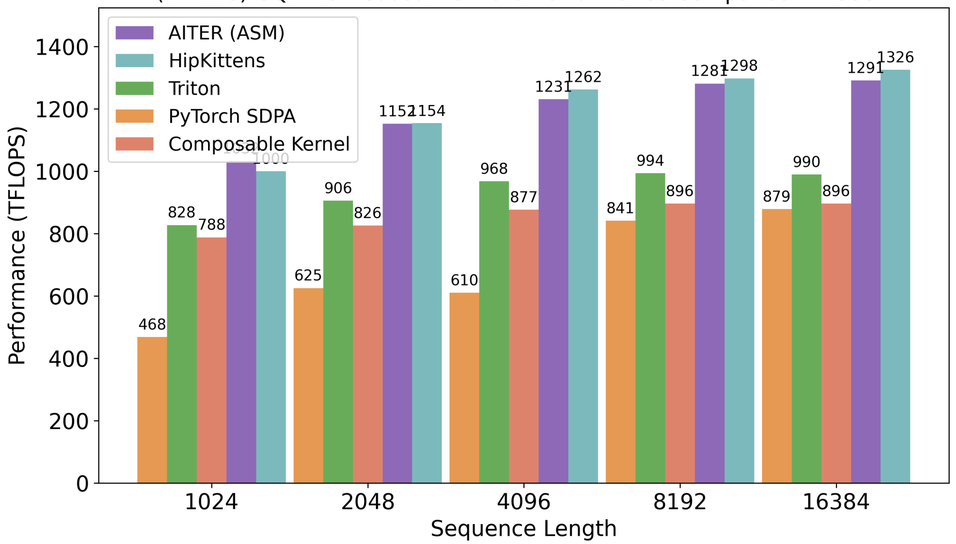
HipKittens: Fast and Furious AMD AI Kernels

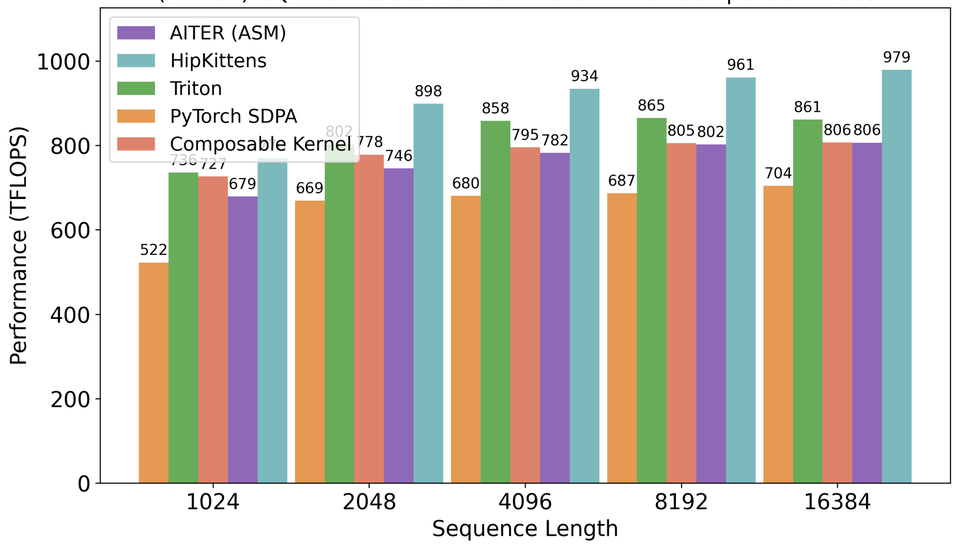
Attention forwards.

~500 LoC and outperforms all the baselines on average … including raw assembly kernels!



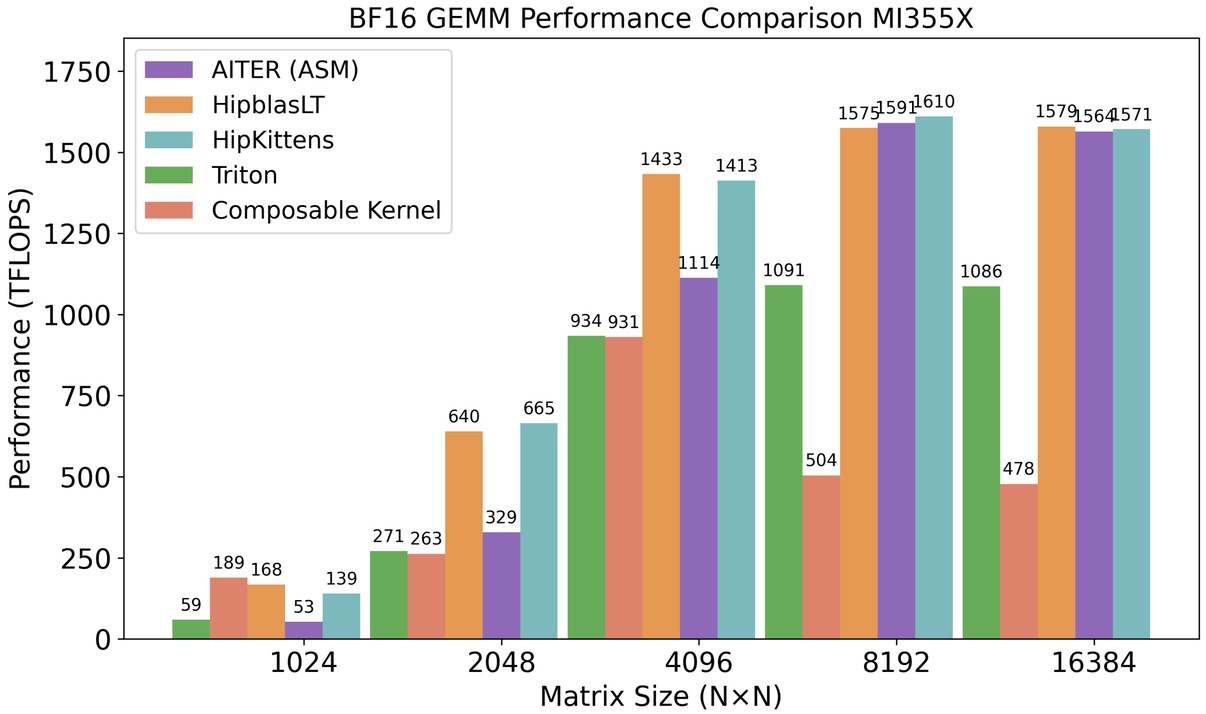


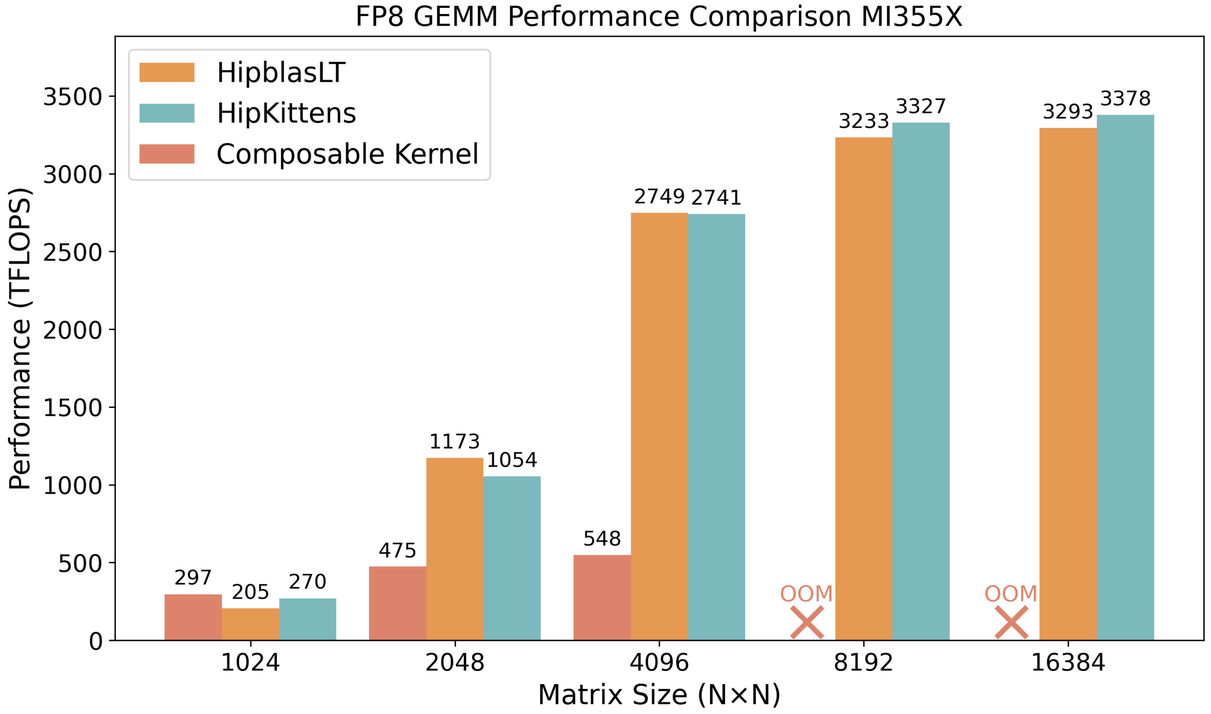




HipKittens: Fast and Furious AMD AI Kernels

BF16 and FP8 GEMM: Our kernel hot loop is <100 lines of code and achieves peak performance. Again, AITER / HipBLASLT kernels are programmed in ... raw assembly!

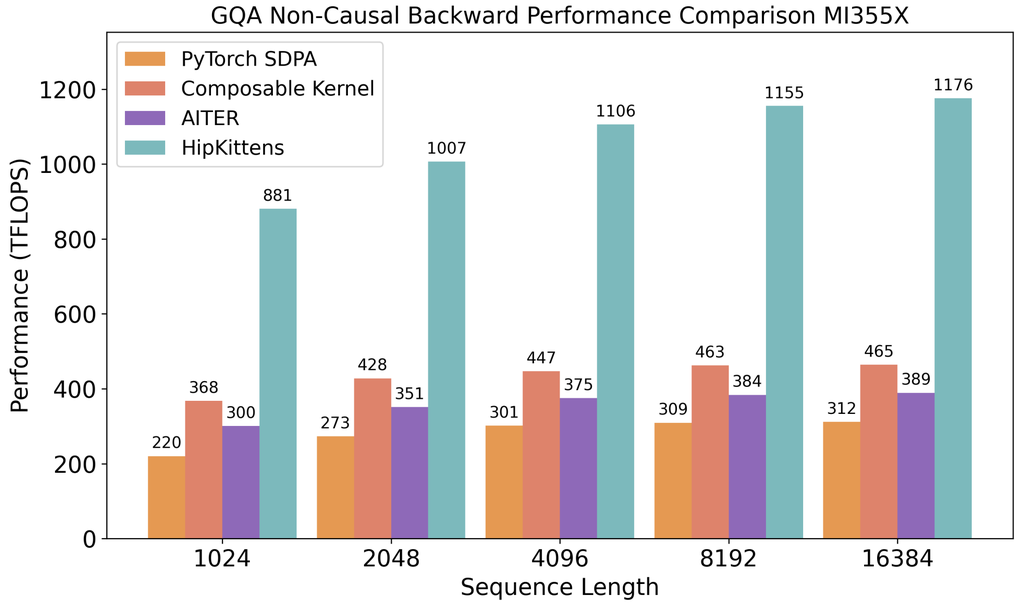


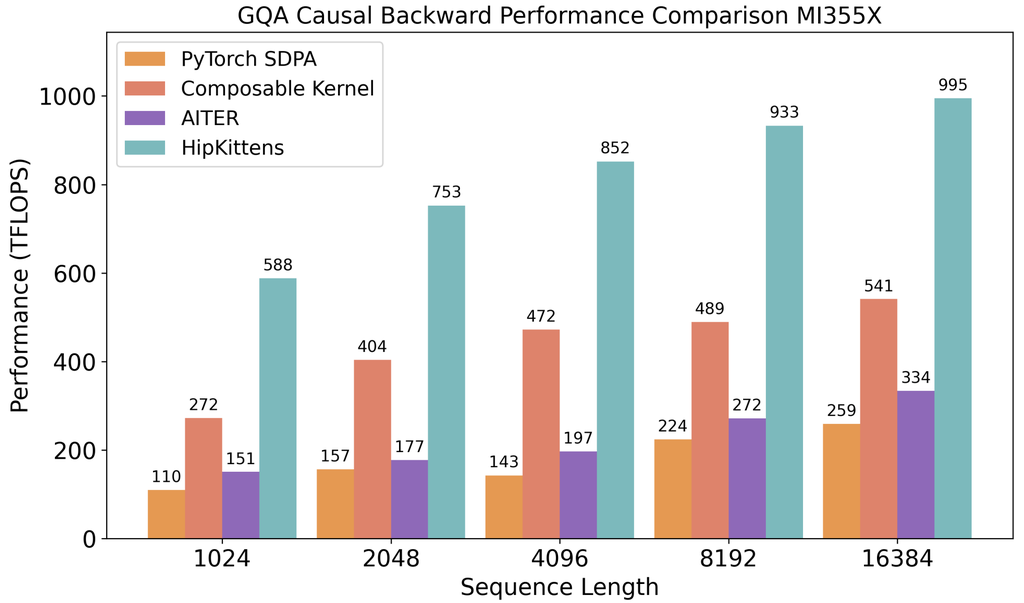


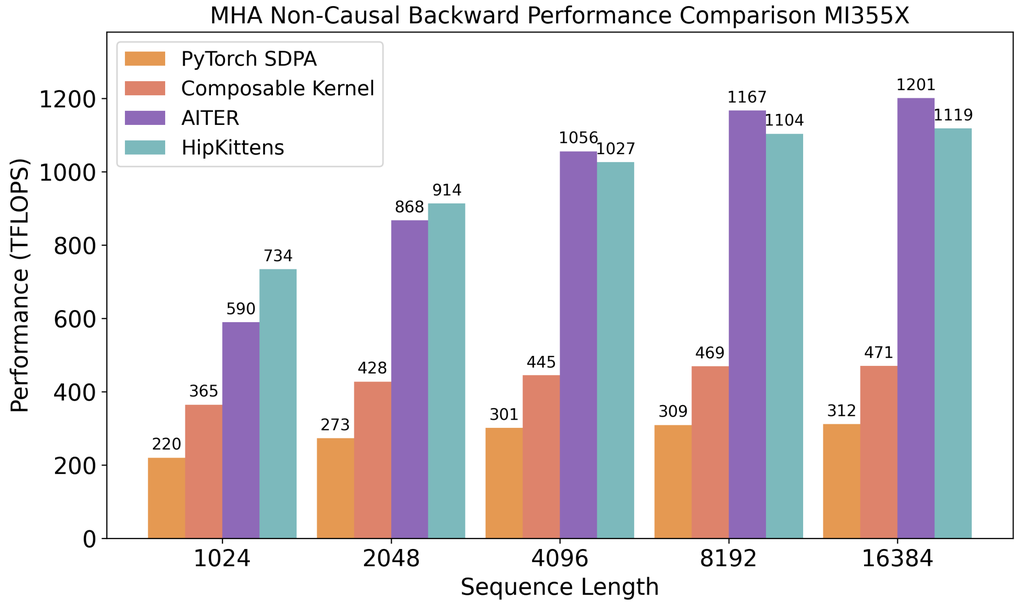
HipKittens: Fast and Furious AMD AI Kernels

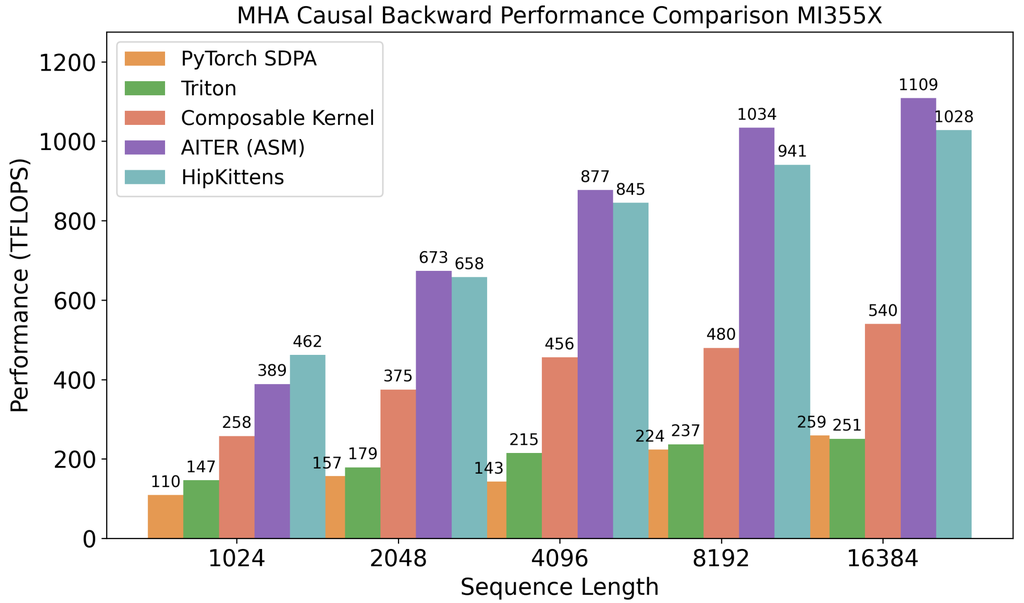
Attention backwards.

Speedy kernels on one of AI's most register-intensive workloads.



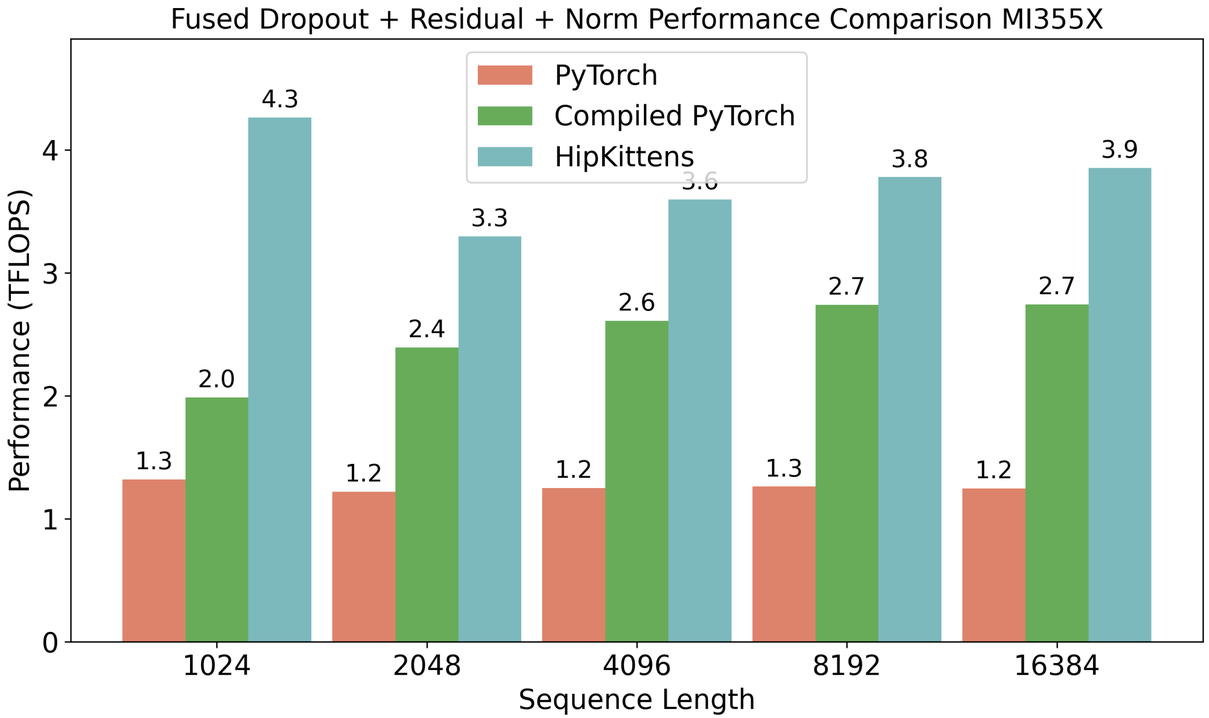


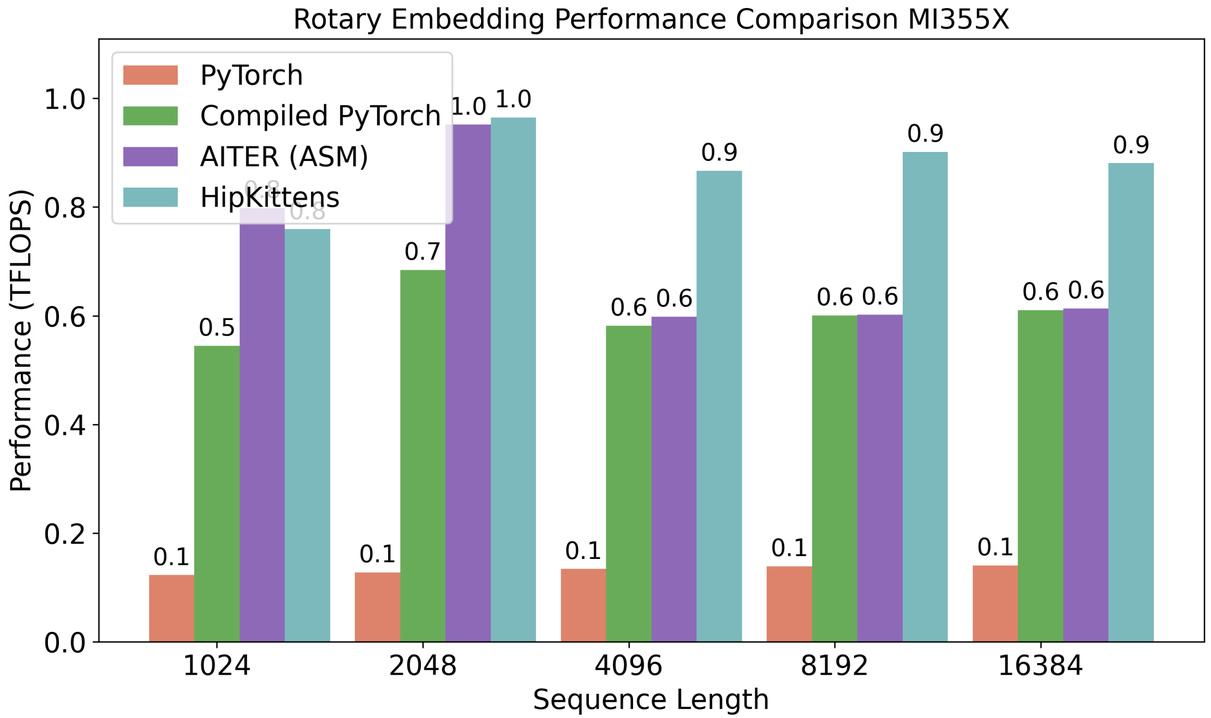




HipKittens: Fast and Furious AMD AI Kernels

Memory bound kernels: speedy rotary and fused dropout-residual-layernorm kernels compared to the strongest available baselines!





HipKittens: Fast and Furious AMD AI Kernels

Takeaways:

Tiles continue to be an important primitive when writing AMD AI kernels, but hardware differences require them to be instantiated differently

Wave specialization works better on modern NVIDIA GPUs for instruction scheduling. AMD GPUs rely on fine grained interleaving (4-wave schedule) or an 8-wave ping pong schedule.

AMD, as well as NVIDIA, GPUs are trending towards larger scale-up domains and disaggregated memory hierarchies. Understanding NUMA effects will be a first-class concern for writing GPU kernels moving forward.

Thank you! Questions?

Checkout the resources:
https://github.com/HazyResearch/HipKittens
Contact: **willhu@stanford.edu** and **simarora@stanford.edu**

Community adoption:
https://github.com/ROCm/aiter
https://github.com/ROCm/TransformerEngine