Towards simple abstractions for hardware-efficient AI kernel programming

Simran Arora

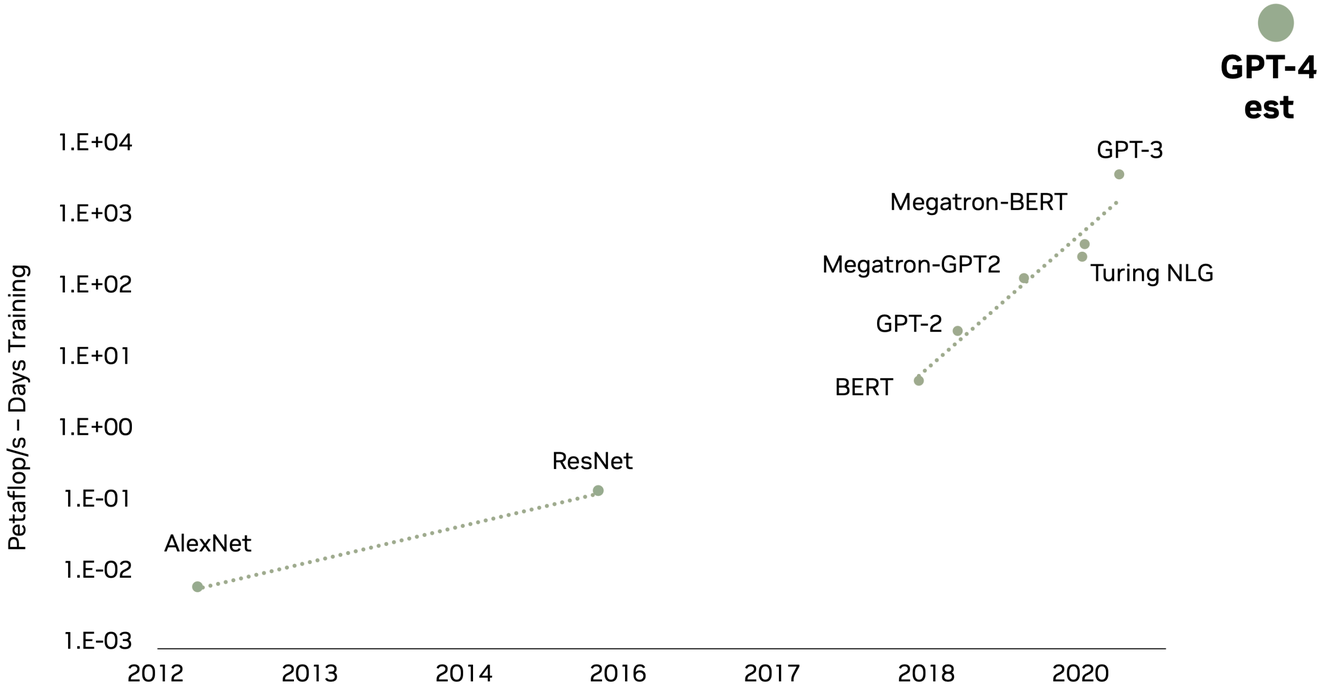
ScaleML 2025

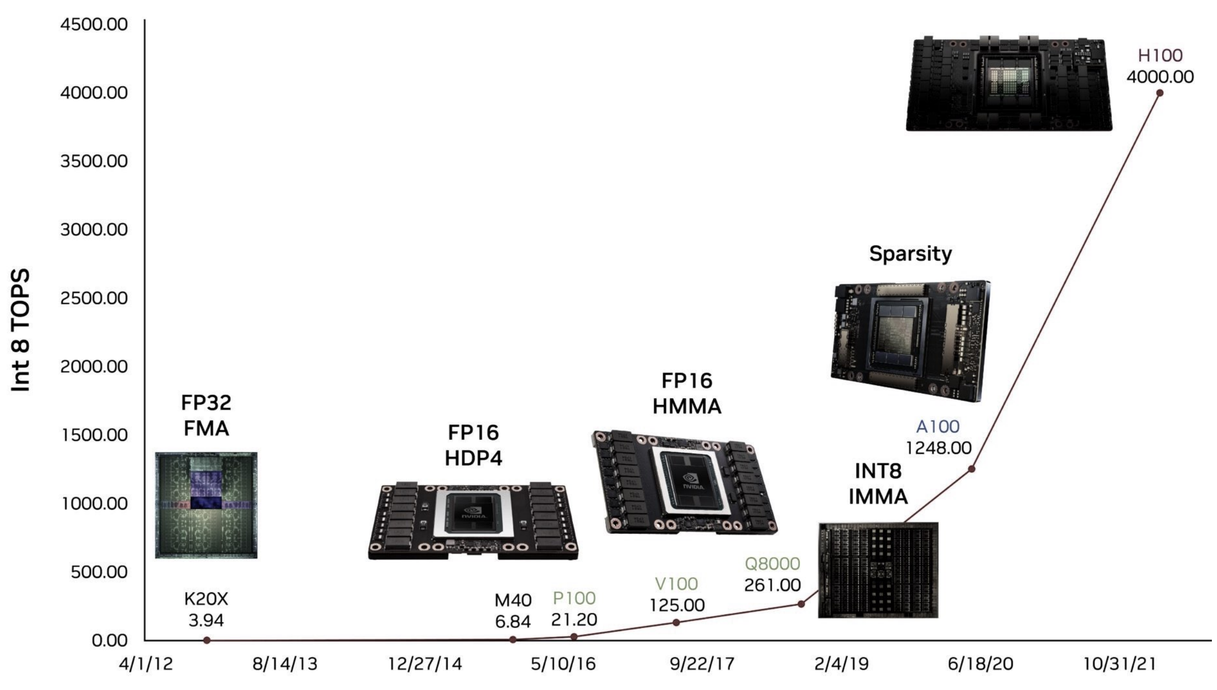
Why should we care?

- Nvidia's 2025 Q2 revenue: >$47Bn
- AI kernels often run on clusters of $100Ms of GPUs for multiple months (across both training and inference)
- Hardware-efficient algorithms that for one hardware generation have completely degraded on the next generation hardware (e.g., Ampere to Hopper, Hopper to Blackwell)

Poor software costs billions of dollars worth of compute! AI compute is expensive! 

We have made exciting progress in AI through a specific AI recipe: **massive models trained on massive compute.**





Massive models…

Massive compute…

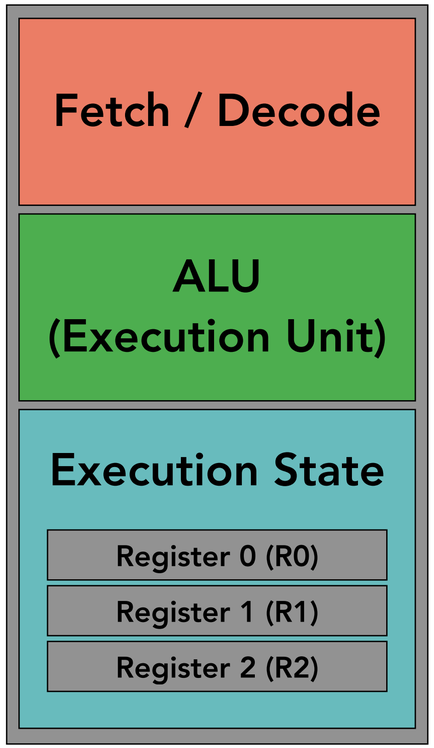
Bill Dally, NVIDIA, 2023.

Overview

1. **Introduction to AI hardware**
2. ThunderKittens: Tile-based programming for AI kernels
3. What architecture does *the hardware* prefer?
4. Key directions

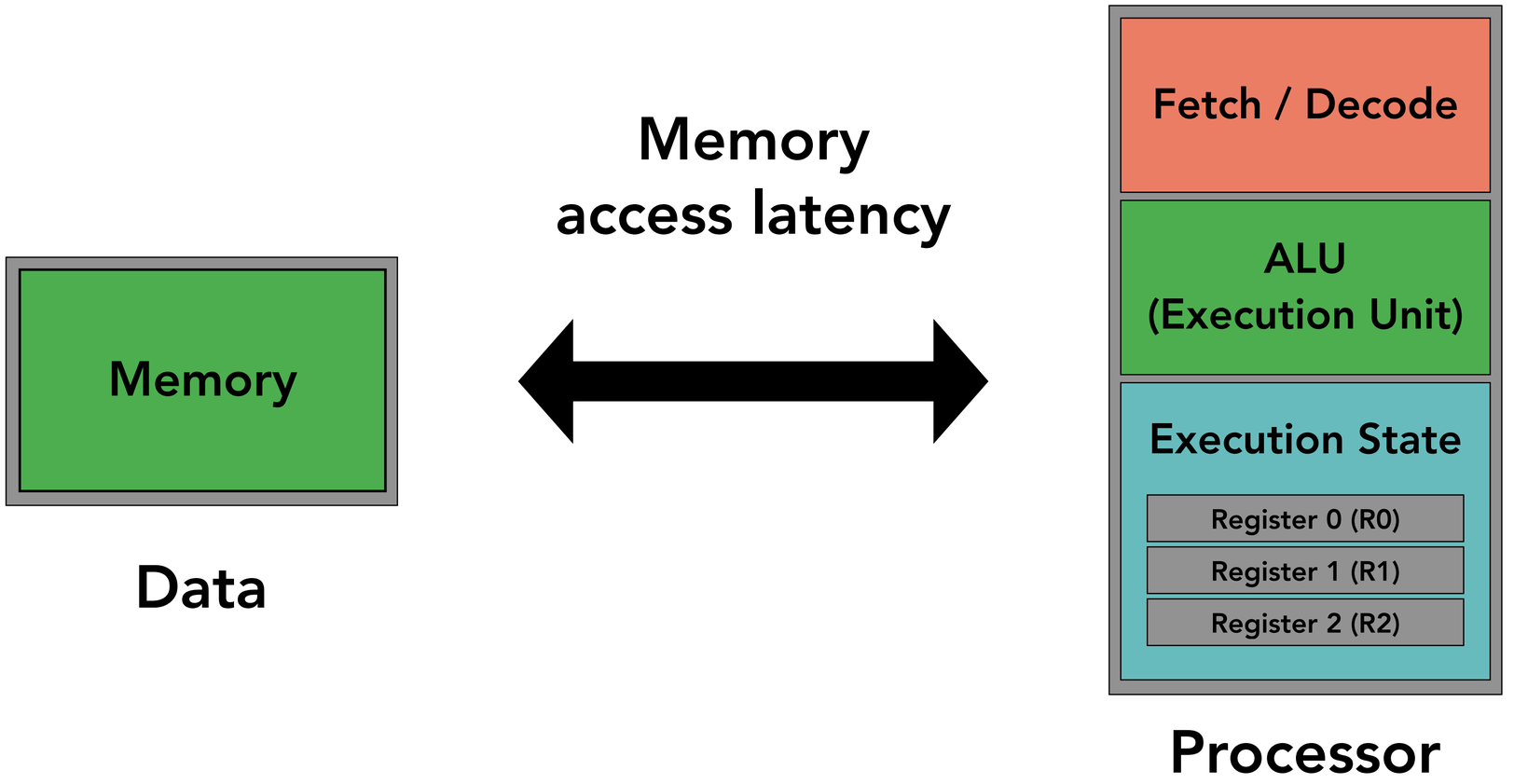
Computer hardware

- A **processor** executes **instructions** using different **threads** (executors)
- An instruction modifies the computer's **execution state**
- A **kernel** is the basic unit of software that runs on the hardware. It loads data, computes operations, and stores the results

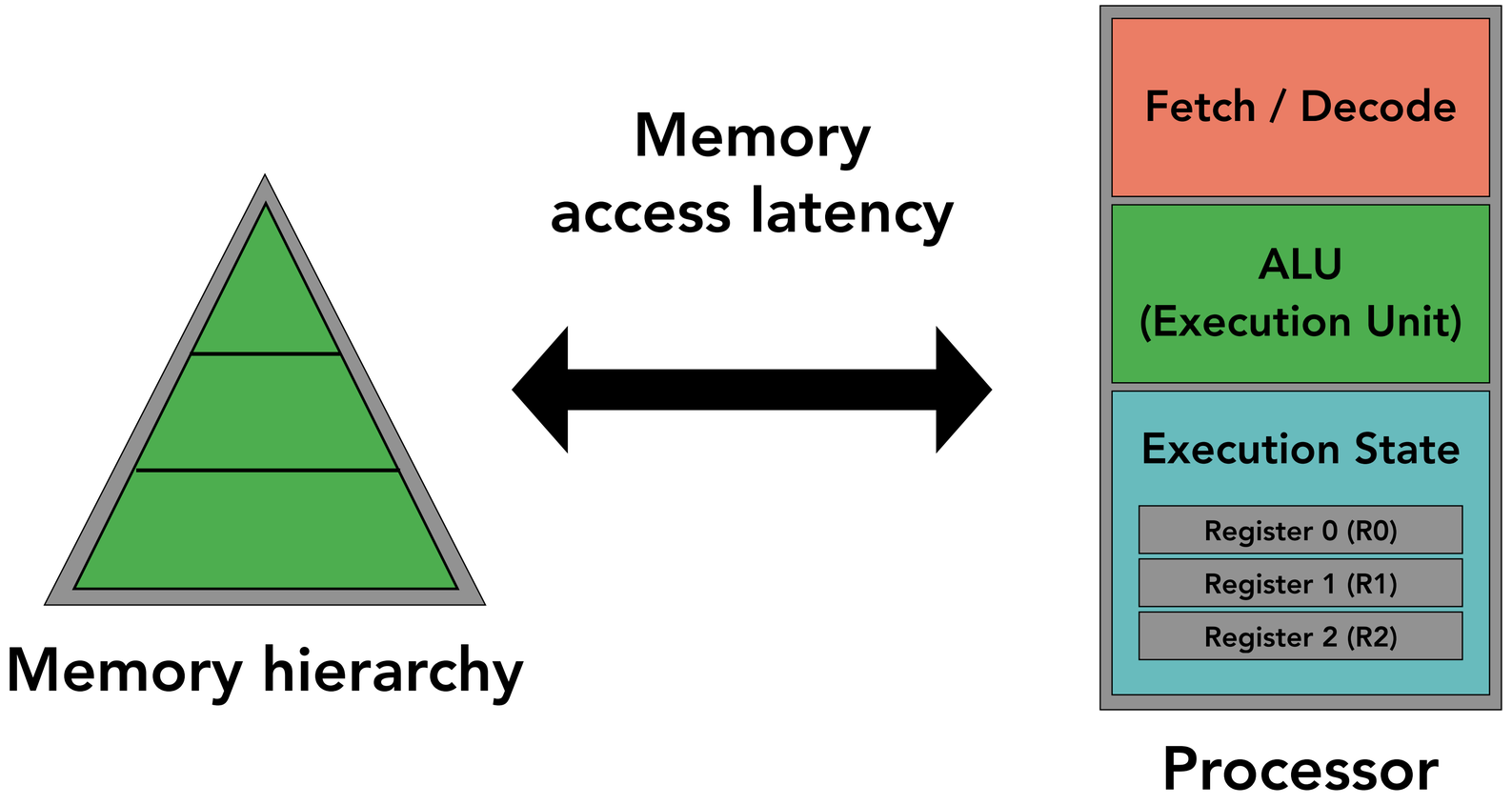


Processor

Computer memory



Computer memory



Caching = reduce the access latency for frequently reused data

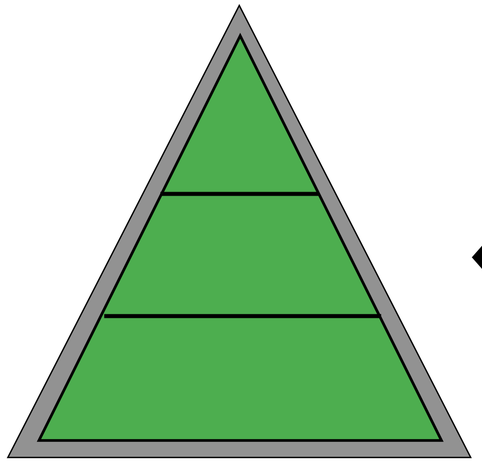
Hardware performance measures

- **Memory bandwidth** = items that can be moved to/from processor from memory per second

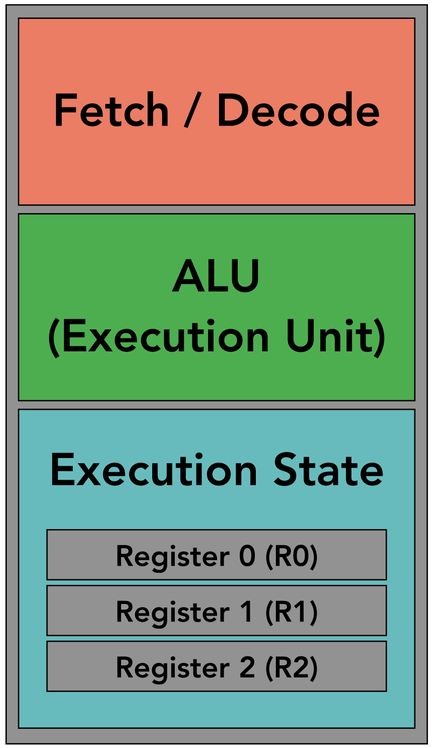
- **Compute bandwidth** = floating point operations that can complete on a processor per second

- **Comms bandwidth** = bytes that can be moved to/from other devices per second

*Note: these are **hardware**, not software/ algorithm properties.



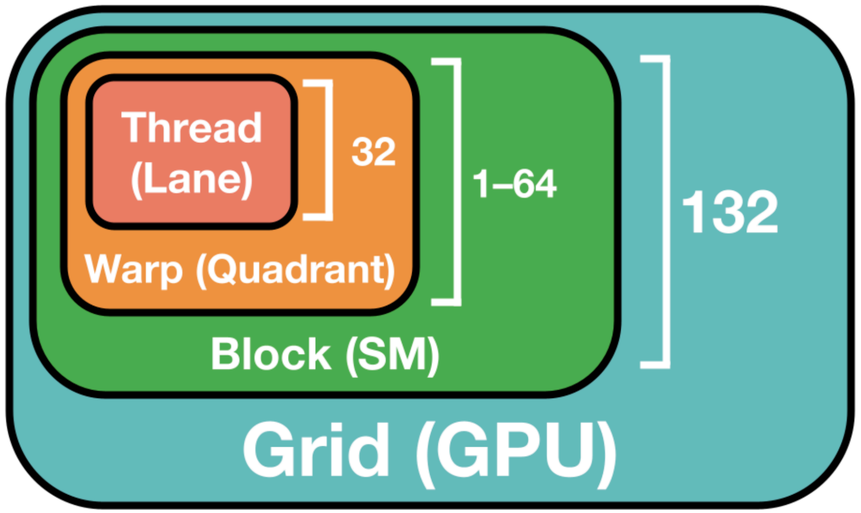
Memory hierarchy



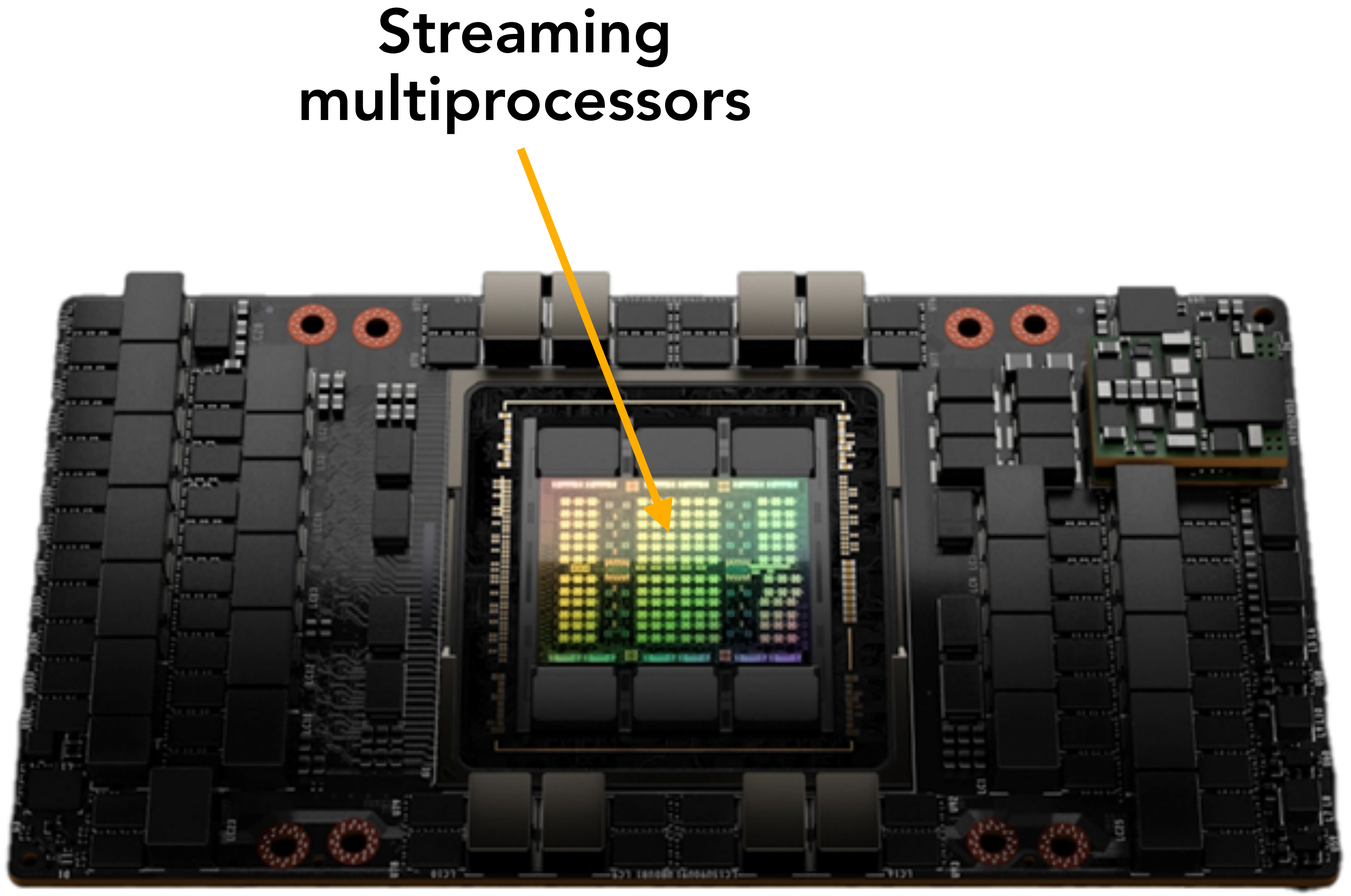
Processor

AI hardware

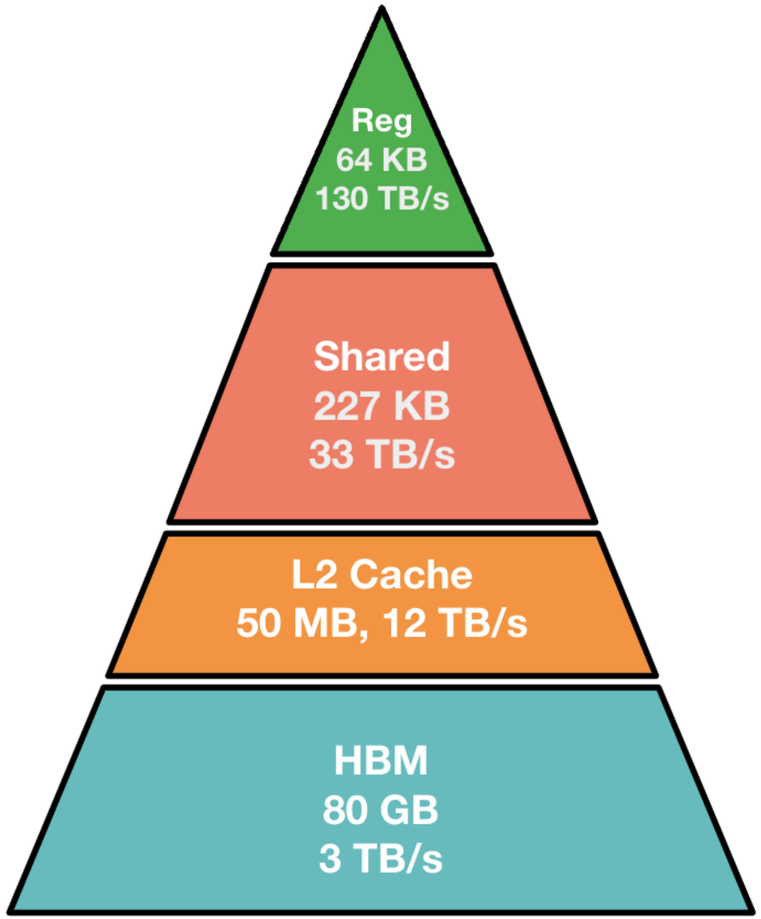
“Kernel” = GPU program.



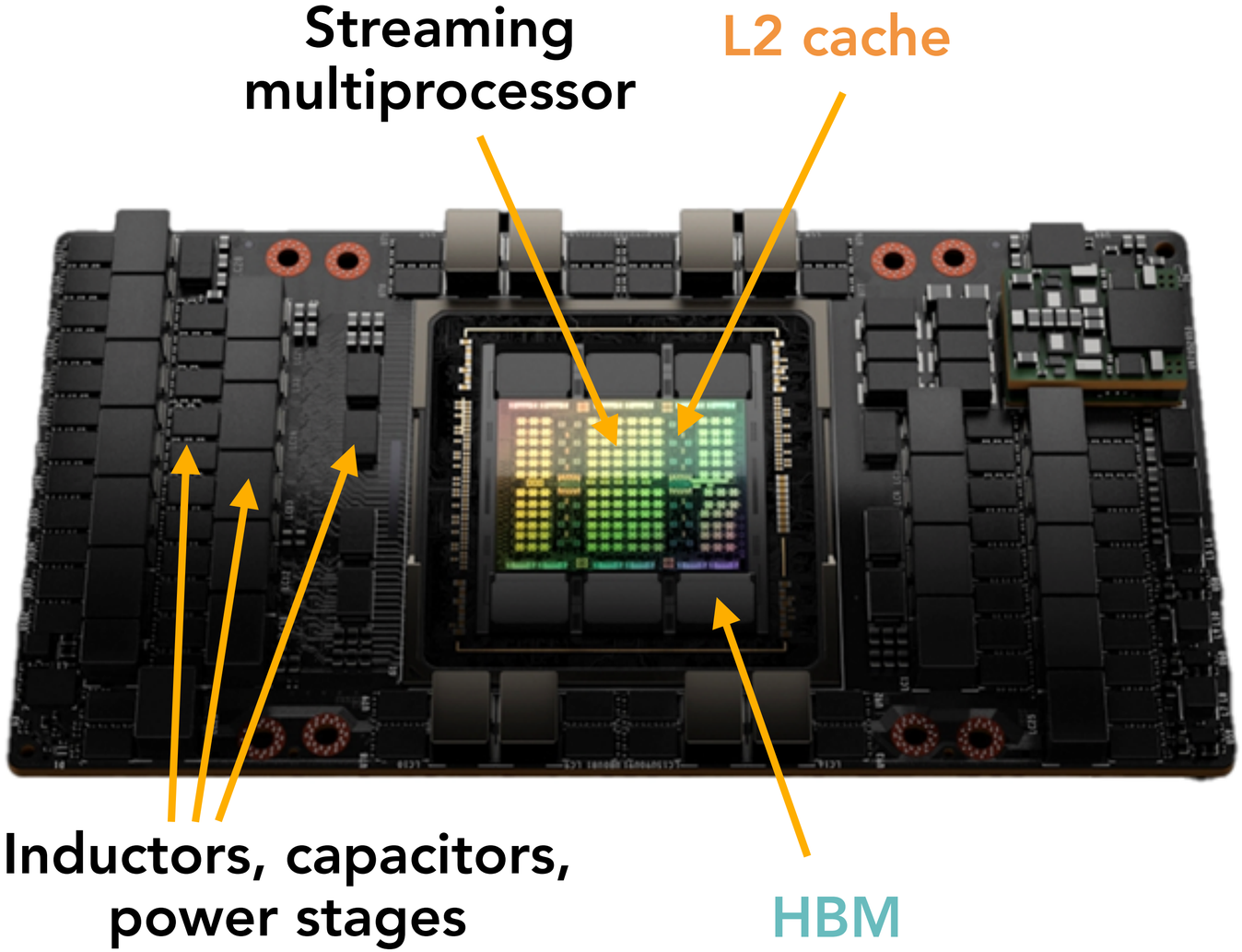
GPU Compute Hierarchy



AI hardware

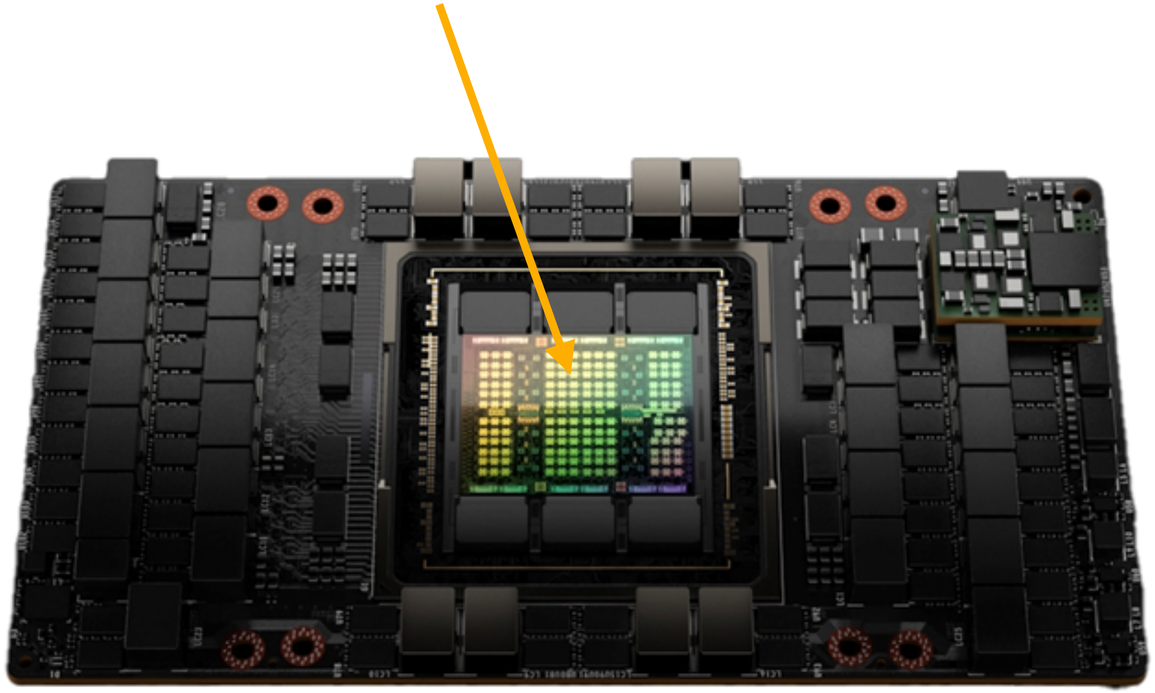


GPU Memory Hierarchy



AI hardware

Streaming multiprocessor



There are multiple types of execution units per SM.

All the compute is in the tensor cores

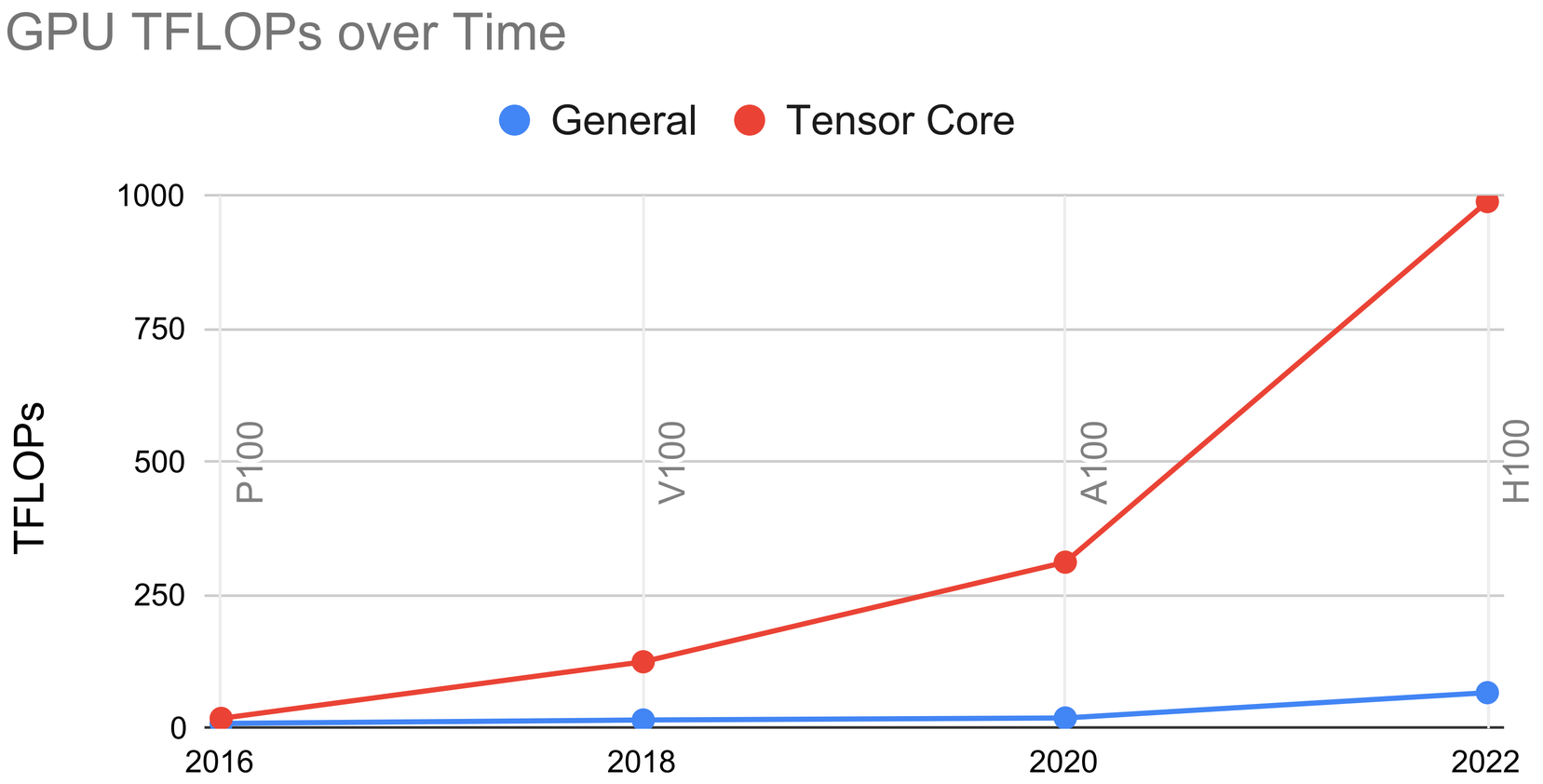
On an Nvidia H100:

- **SFU:** 7.5 TFLOPs of special functions like exp
- **FMA:** 60 FLOPs
- **Tensor cores:** 989 TLOPs for BF16

Essentially, the GPU runs at 100% when the tensor cores are running and 0% when they're not.

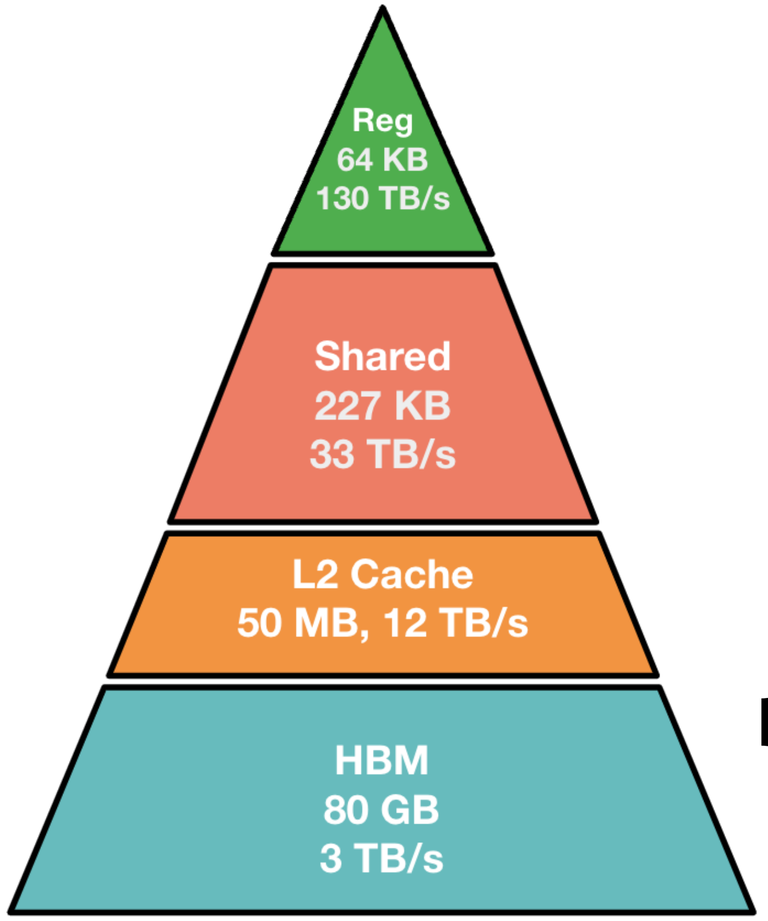
There are multiple types of execution units per SM.

Golden rule of hardware-aware AI: utilize the tensor cores!



Tensor cores are 16x faster than other compute on H100 GPUs (in BF16)!

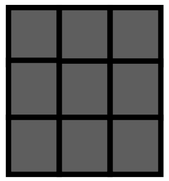
The challenge is minimizing the latencies of non tensor core operations.



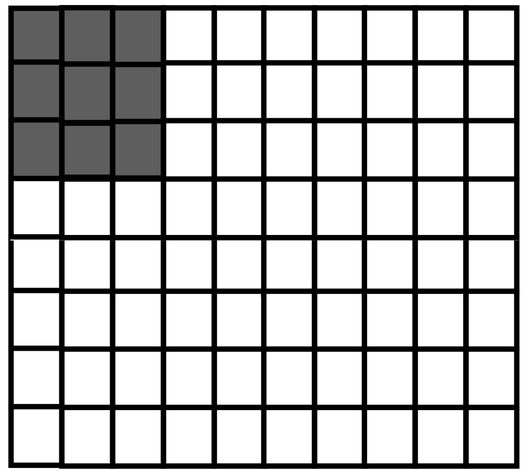
GPU Memory Hierarchy

Tile data!

$X' =$



$X =$



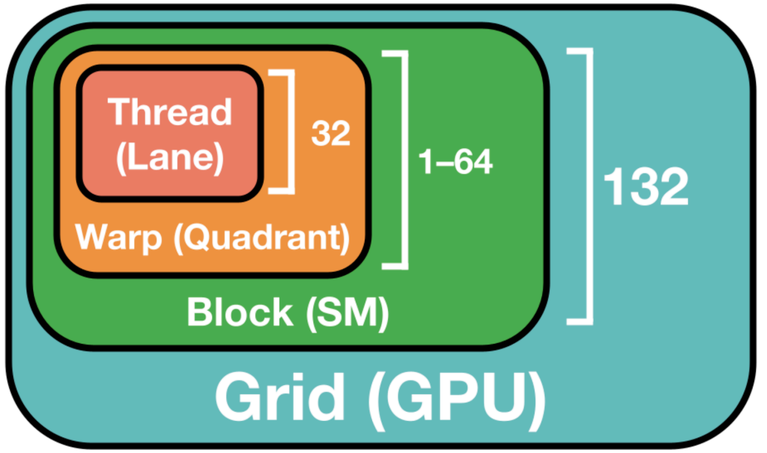
Fuse ops!

```
Z1 = Cumsum(X')
Z2 = Matmul(Z1, Y)
Z3 = exp(X')
Z4 = Add(Z2, Z3)
```

Tensor core op!

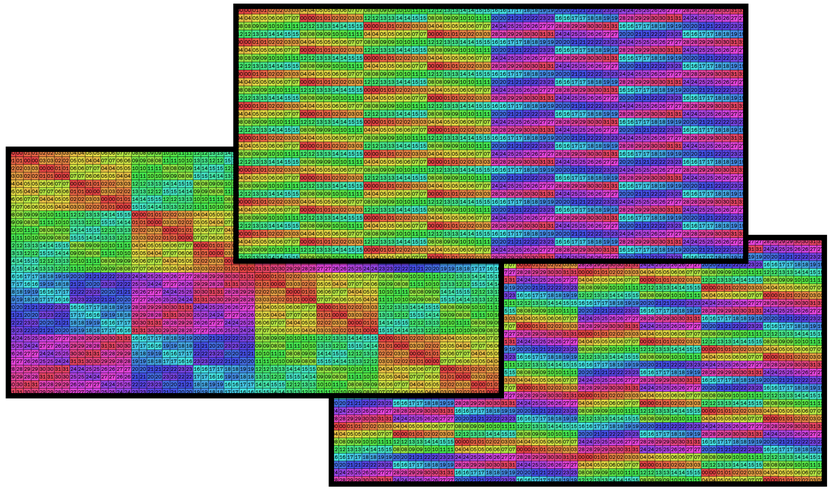
Non tensor cores ops

A simplified model of GPU parallelism.



GPU Hierarchy

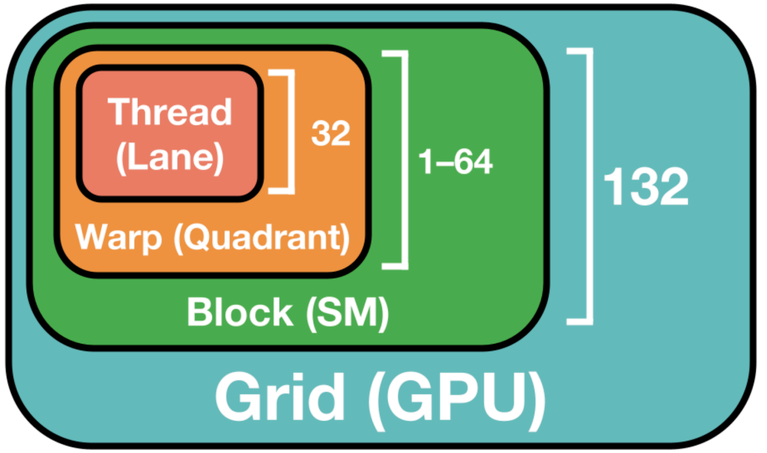
- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together



Memory layouts: how logical data elements map to physical thread ownership

A simplified model of GPU parallelism.

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data

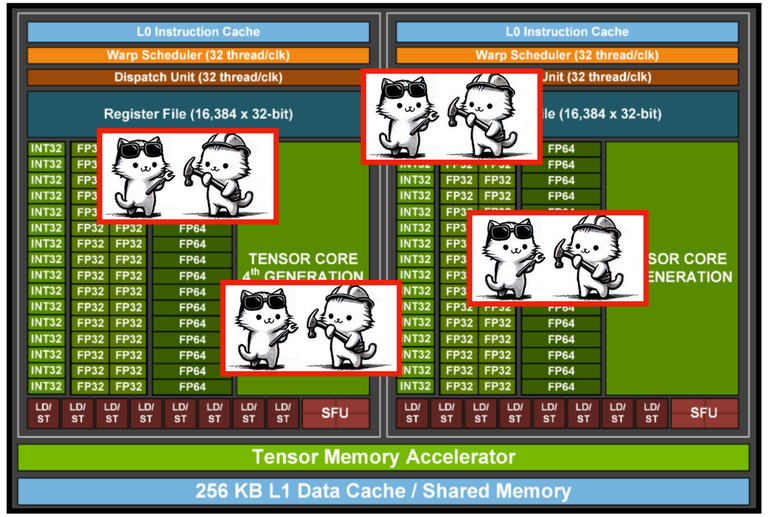


GPU Hierarchy

Occupancy tradeoff

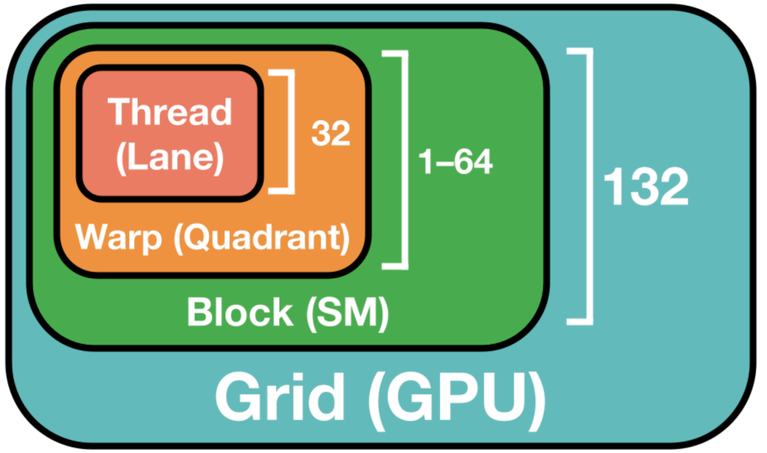
Threads executing in parallel

Resources available per thread

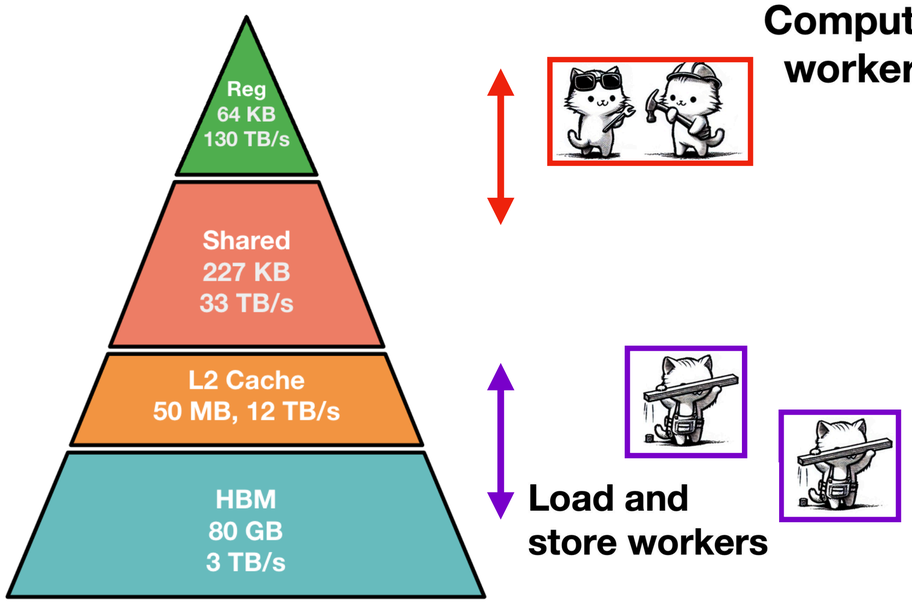


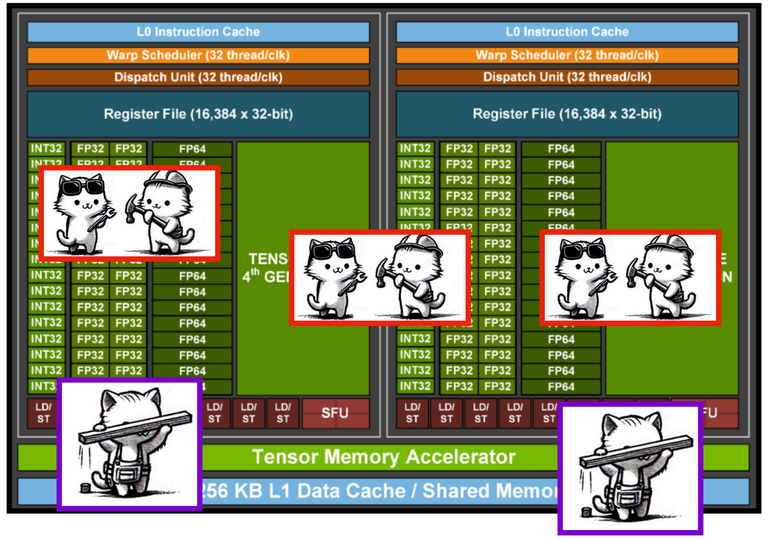
A simplified model of GPU parallelism.

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data

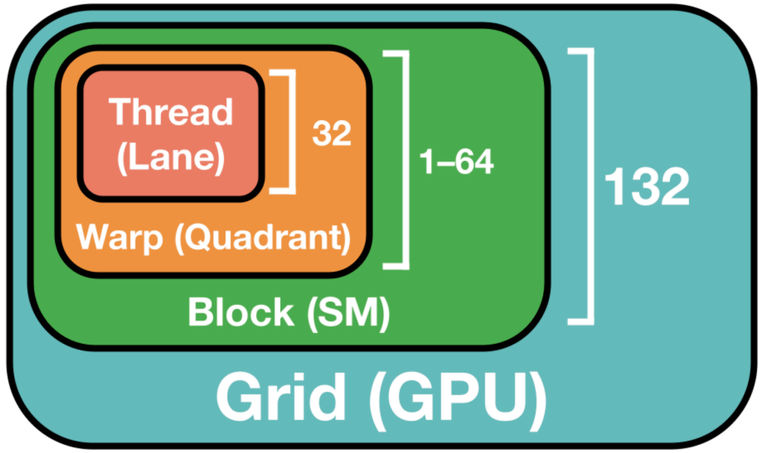


GPU Hierarchy



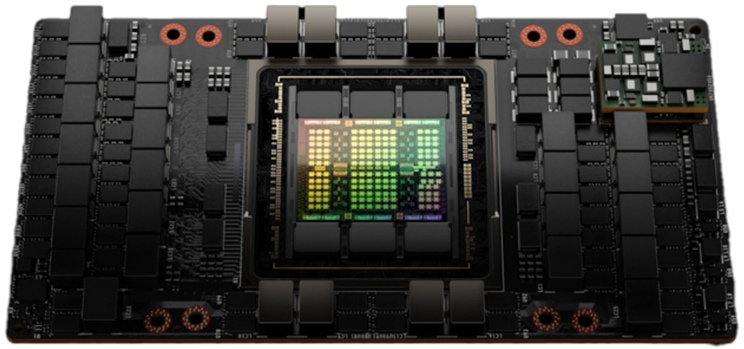


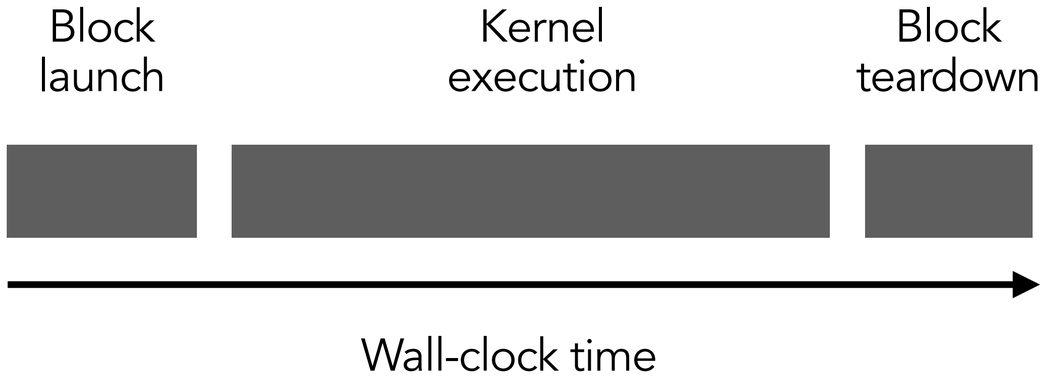
A simplified model of GPU parallelism.



GPU Hierarchy

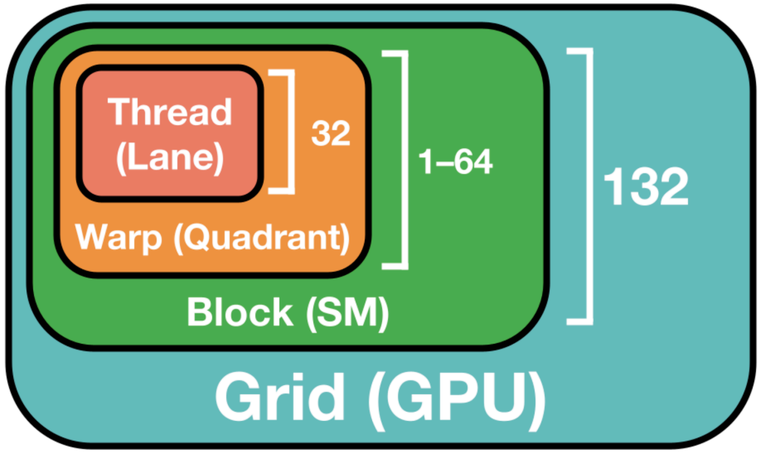
- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data
- **Grid**: GPUs run many blocks of threads at once



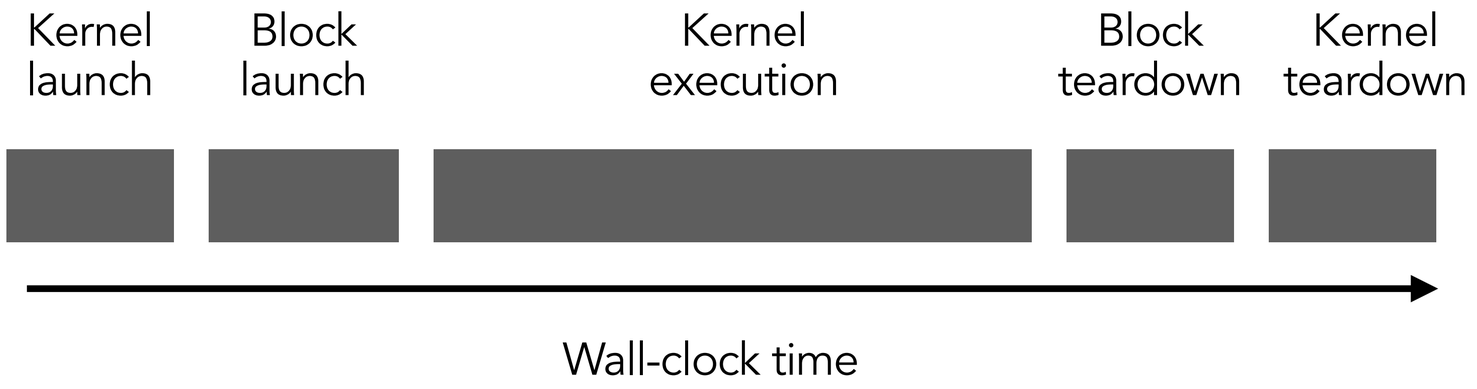


A simplified model of GPU parallelism.

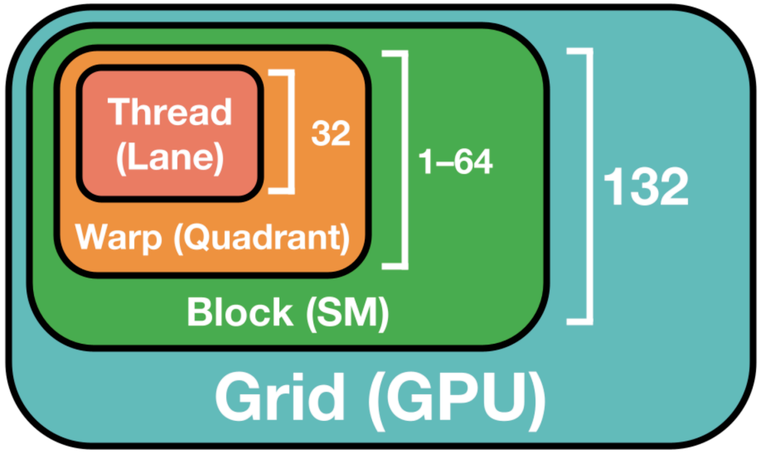
- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data
- **Grid**: GPUs run many blocks of threads at once



GPU Hierarchy



A simplified model of GPU parallelism.

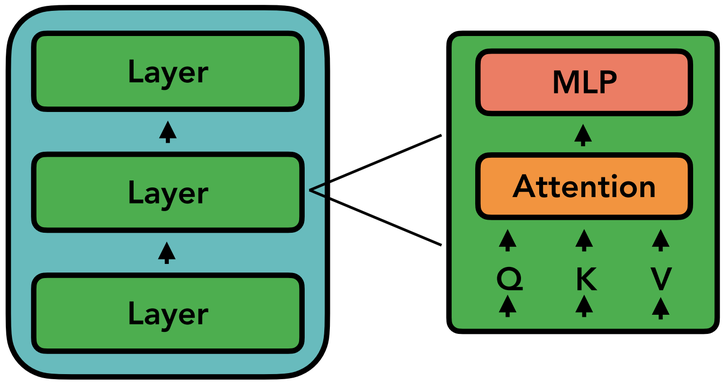


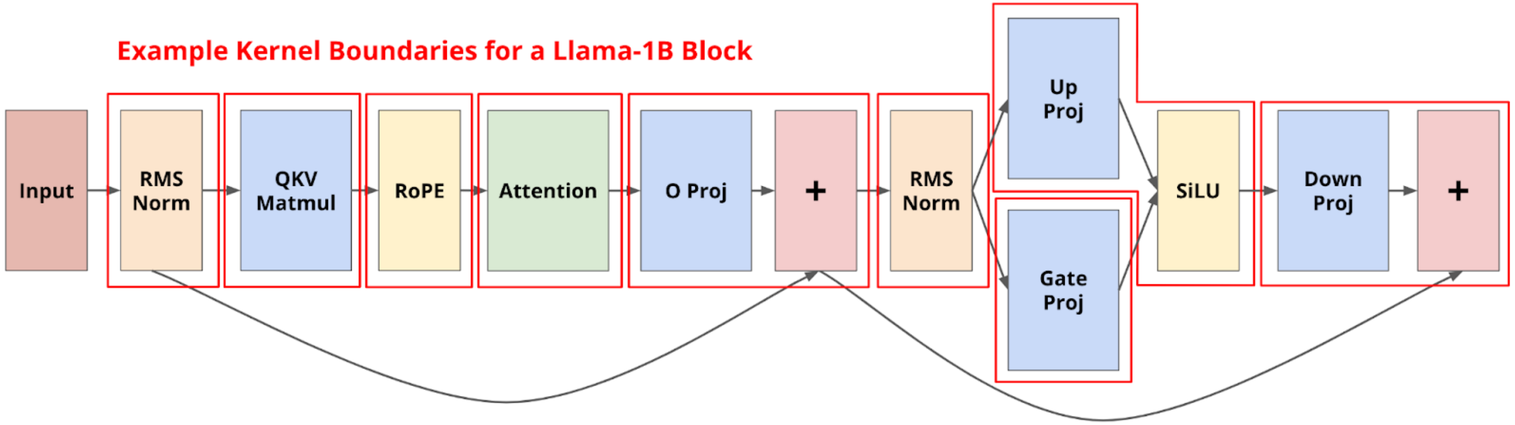
GPU Hierarchy

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data
- **Grid**: GPUs run many blocks of threads at once

A ML model has many layers

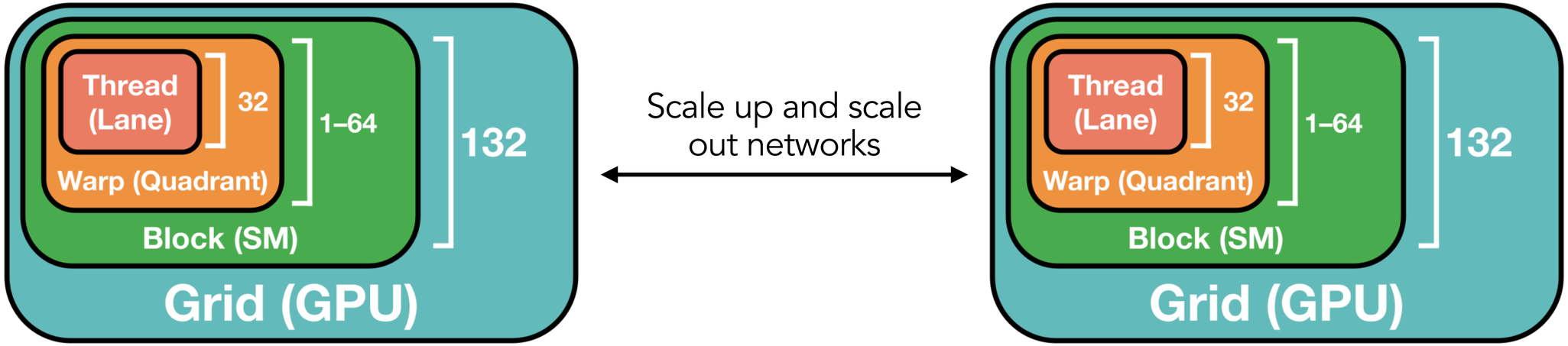
And many ops per layer





A simplified model of GPU parallelism.

- Threads: Tens of thousands of threads run on GPUs
- Warps: groups of threads that run instructions together
- Blocks: groups of warps, which can quickly share data
- Grid: GPUs run many blocks of threads at once
- GPU: multiple devices collaborate to run large-scale AI



GPU Hierarchy

GPU Hierarchy

Overview

1. Introduction to AI hardware
2. **ThunderKittens: Tile-based programming for AI kernels**
3. What architecture does *the hardware* prefer?
4. Key directions

A critical problem in AI is mapping algorithms and architectures to hardware.

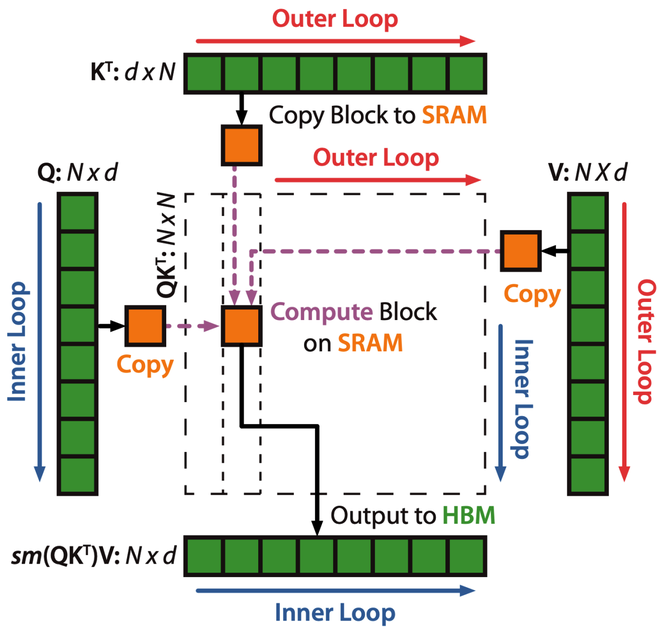
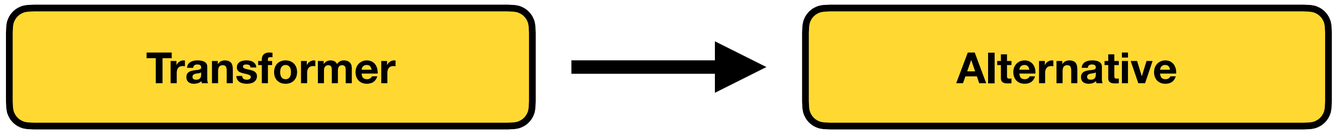


Figure: Dao, Fu et al., FlashAttention, 2022

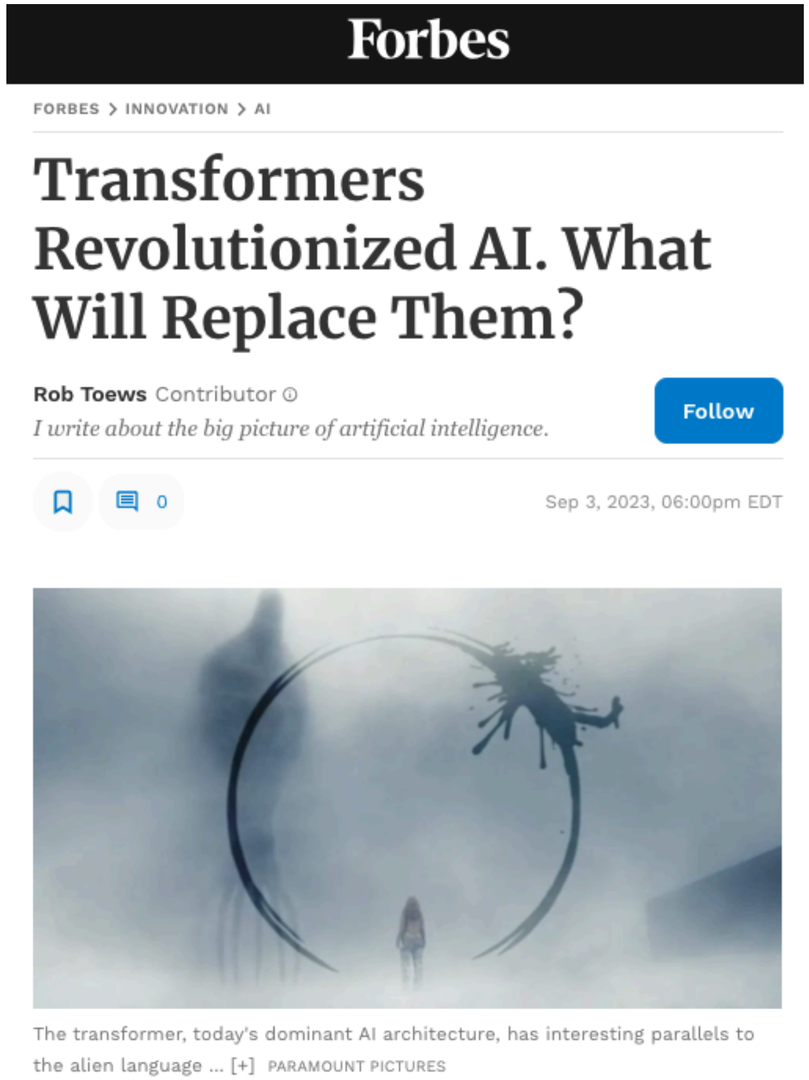
- Widely used FlashAttention GPU kernel degraded 47% in TFLOPS from A100 to H100 GPUs
- Took 2 years from the H100's release to obtain performant kernels

It's been hard to obtain peak-performance for Transformers, let alone new architectures!

Can we replace Transformers with improved alternatives?



S4 [Gu et al.]**, DSS** [Gupta], **GSS** [Mehta et al.], **S4D** [Gu et al.], **Liquid S4** [Hasani et al.], **H3** [Fu et al.], **S5** [Smith et al.] **BIGS** [Wang et al.], **Hyena** [Poli et al.], **RWKV** [Peng et al.], **RetNet** [Sun et al.], **M2** [Fu et al.], **Mamba** [Gu et al.], **Based** [Arora et al.,], **GLA** [Yang et al.], **GateLoop** [Kastch et al.], **Hawk/Griffin** [De et al.], **Transformers are RNNs** [Katharopoulos et al.], and many more candidates!


Forbes

FORBES > INNOVATION > AI

Transformers Revolutionized AI. What Will Replace Them?

Rob Toews Contributor

I write about the big picture of artificial intelligence.

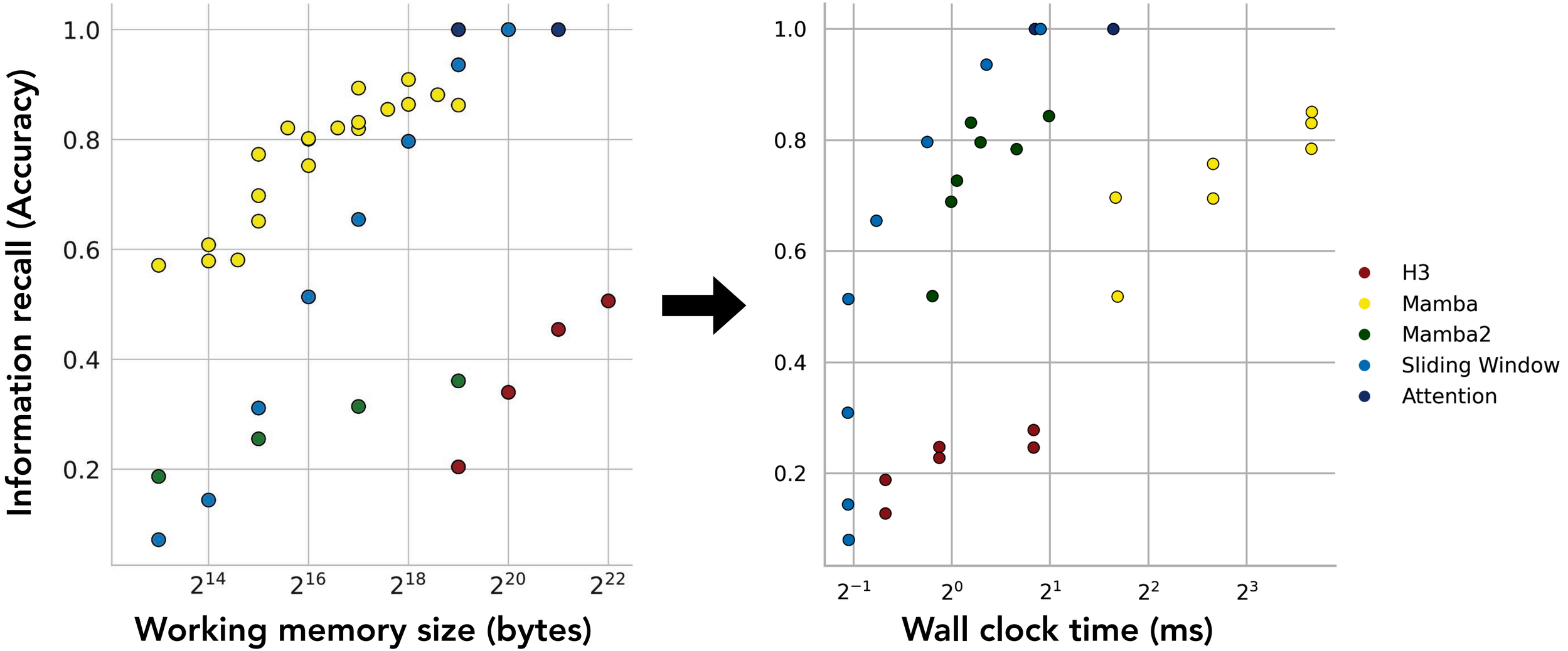
Follow

0

Sep 3, 2023, 06:00pm EDT

The transformer, today's dominant AI architecture, has interesting parallels to the alien language ... [+] PARAMOUNT PICTURES

Theoretical efficiency ≠ wall-clock efficiency.



Sequence length 4096; batch size 256.

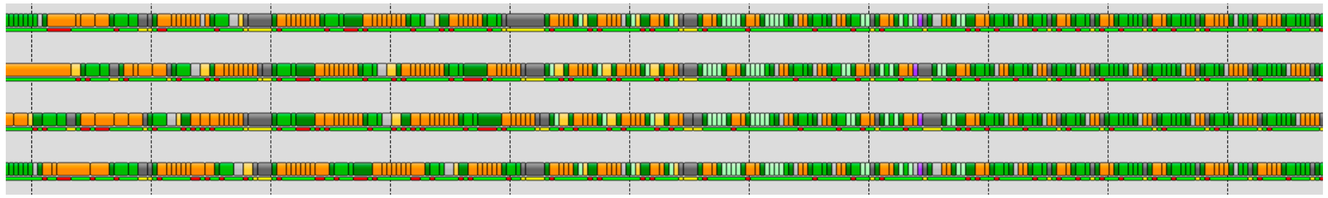
https://github.com/HazyResearch/zoology

Research question

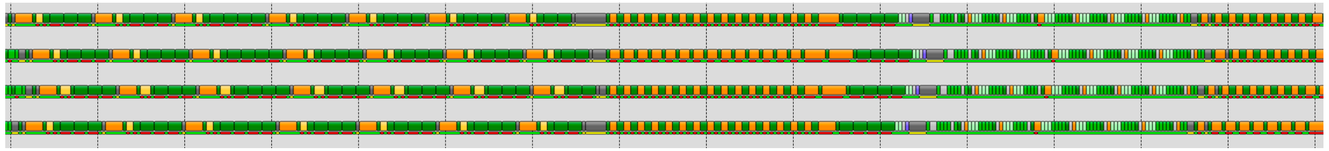
How concise a set of programming abstractions can we use to support fast, simple kernels for a breadth of AI workloads?

Motivating tile-based kernel dev

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

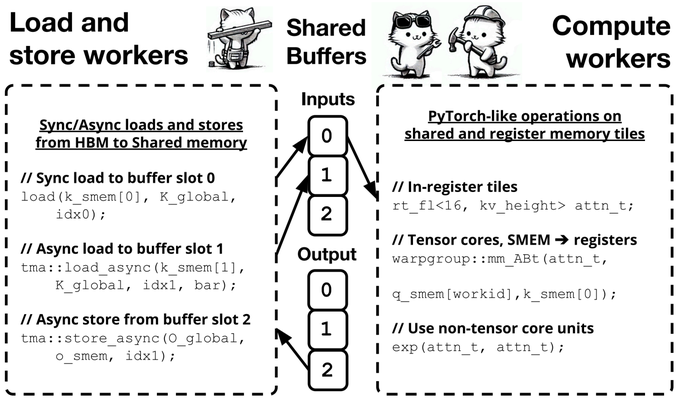
```
# Stage 3: Main computation loop - Pipelined execution with double buffering
for _ in range(2, K // BK):

    @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,
    ]()
```

Kernel programming abstractions

```
if constexpr (std::is_same_v<T_D, float> && std::is_same_v<T_AB, bf16>) {
    asm volatile (
        "{\n"
        ".reg .pred p;\n" \
        "setp.ne.b32 p, %37, 0;\n" \
        "wgmma.mma_async.sync.aligned.m64n64k16.f32.bf16.bf16 " \
        "{%0, %1, %2, %3, %4, %5, %6, %7, %8, %9, %10, %11, %12, %13, %14,
        "{%32, %33, %34, %35}, " \
```

C++ and inline assembly

Compilers

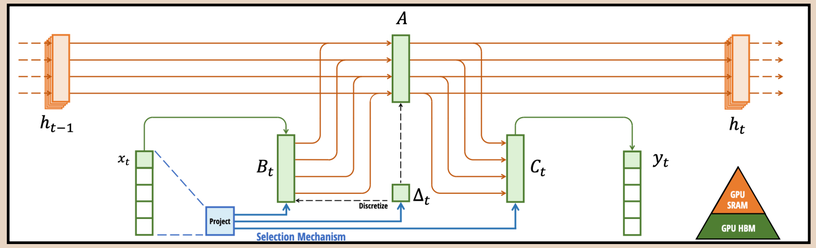


Kernel schedules

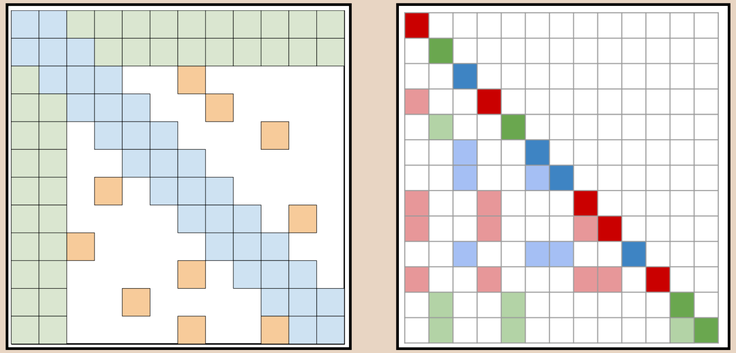
Where should we set the programming boundaries?

(Across basic blocks (tile layouts), across schedules (producer-consumer), and across entire models (megakernels). What insights scale to our multi-silicon future?

Two challenges of hardware-aware AI.



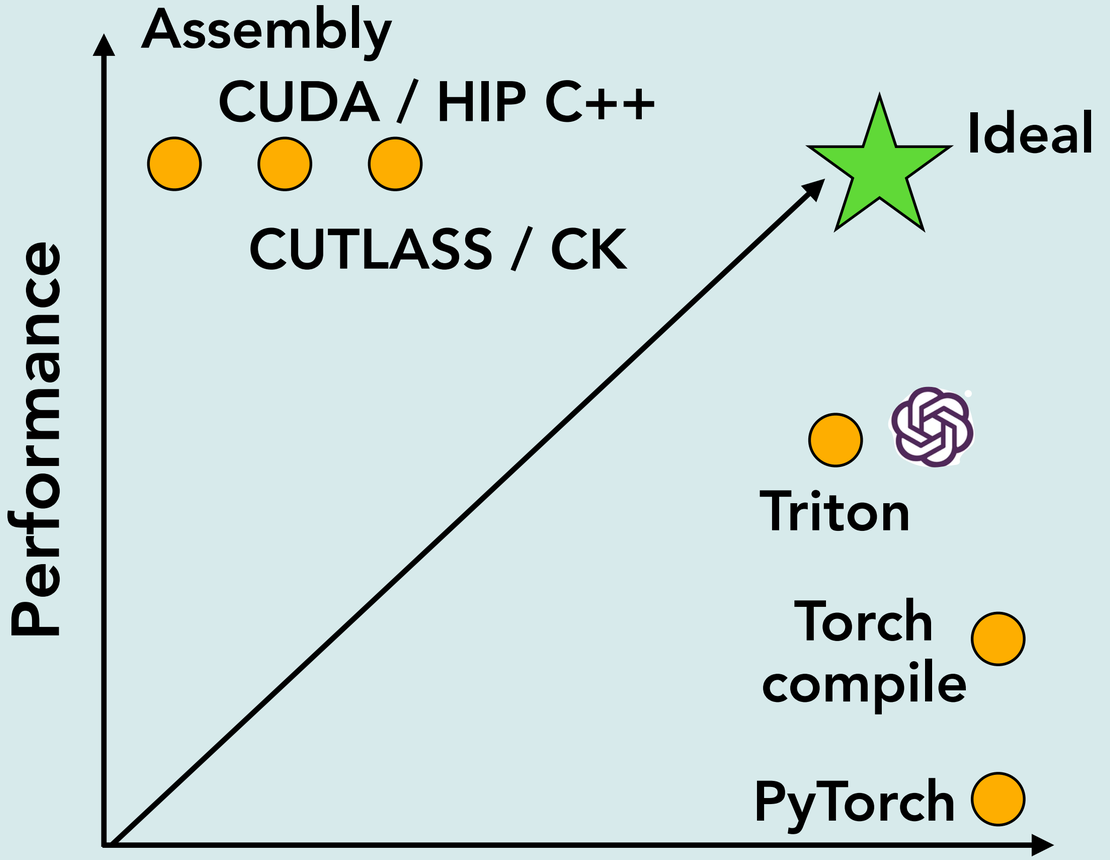
Parallel scans (matmul-free and high materialization costs)



Unstructured memory accesses

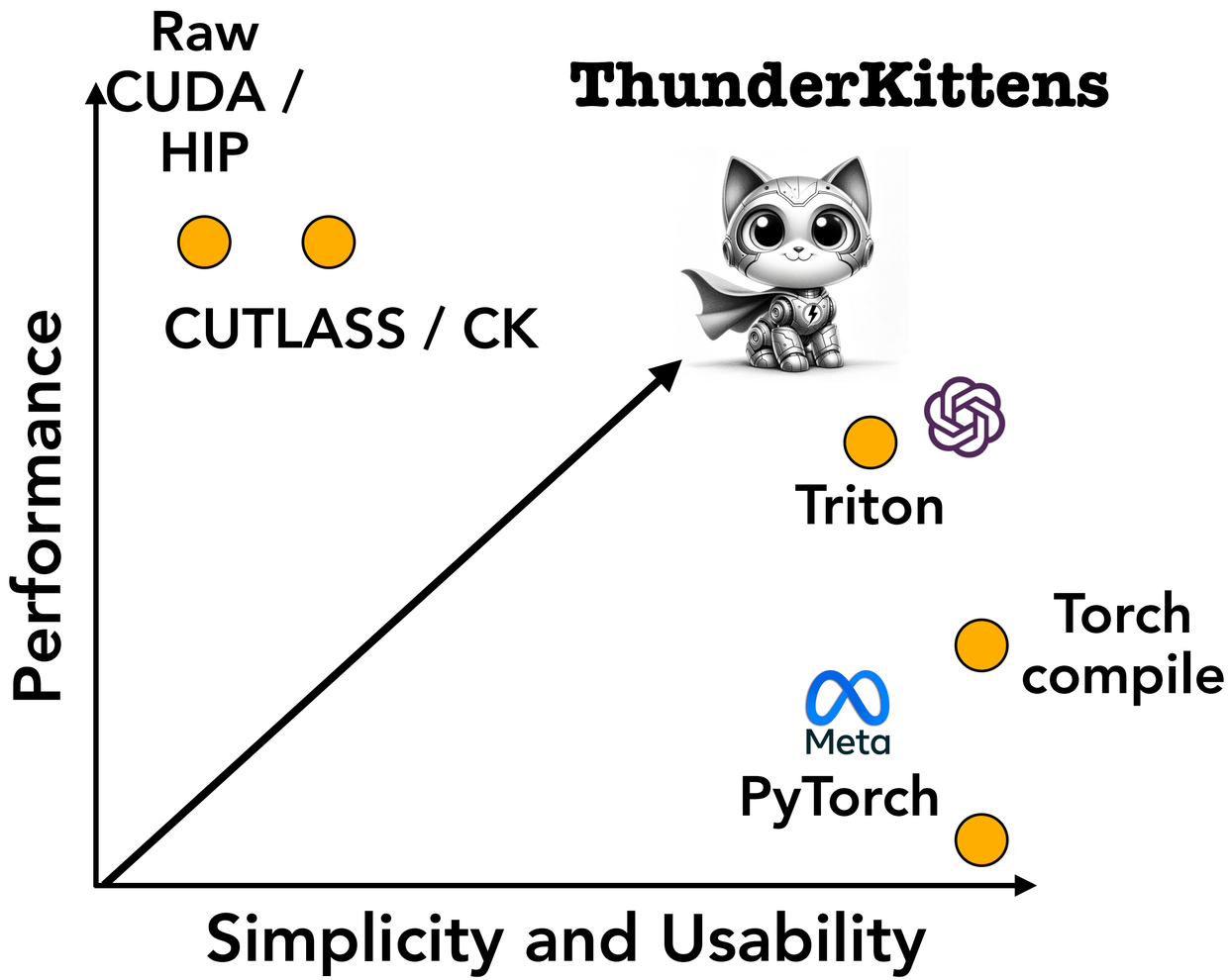
And more!

Architectural choices



Programming abstraction choices

ThunderKittens: tile-based programming for AI kernels

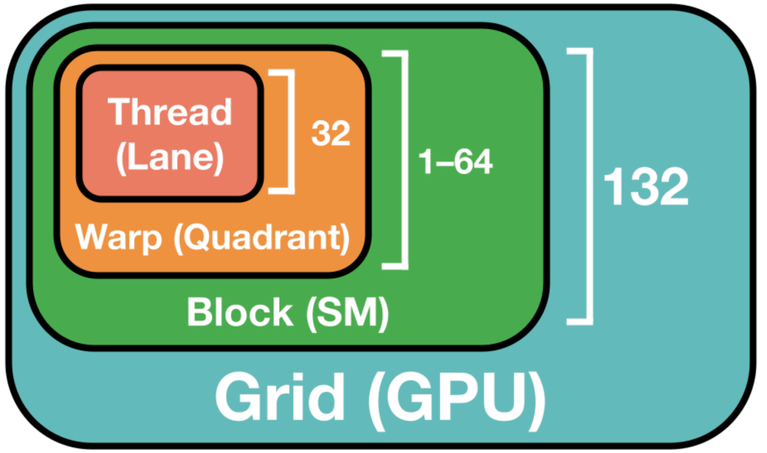


Library sizes

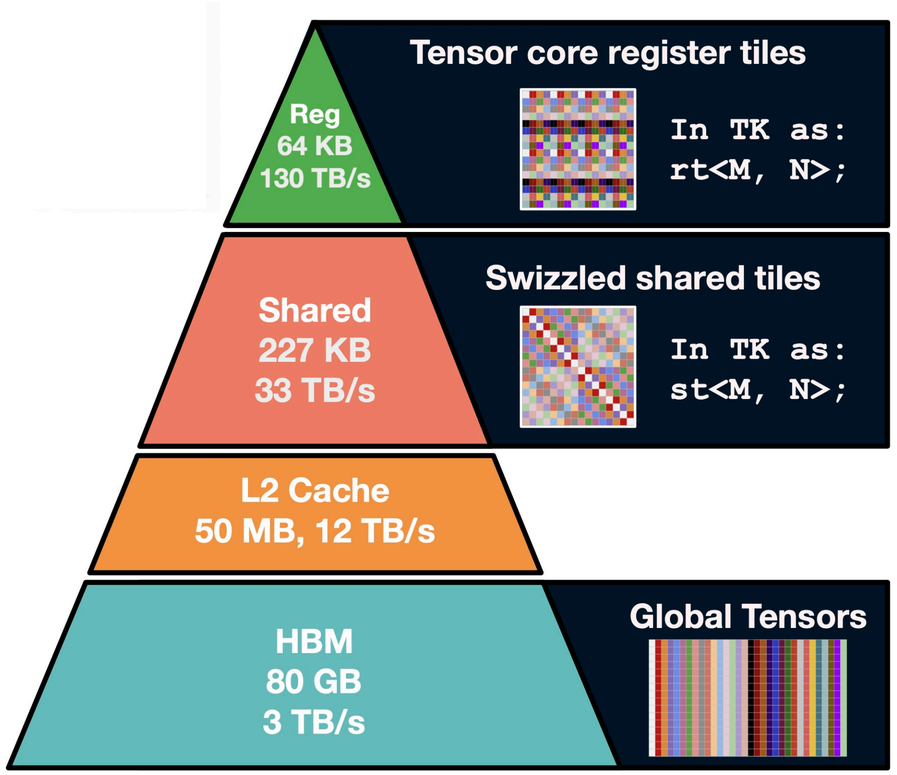
Framework	Size	Date
CUTLASS	22 MB	10/22
Triton	12.6MB	10/22
TK	<1MB	10/22

ThunderKittens library

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together



GPU Hierarchy



Tile data abstraction:
16x16 tiles with managed memory layouts to minimize bank conflicts and encourage tensor core use

Principles of memory layouts

Global memory: We want threads to use **large coalesced aligned** global to shared memory loads (minimize instruction issues, hardware friendly).

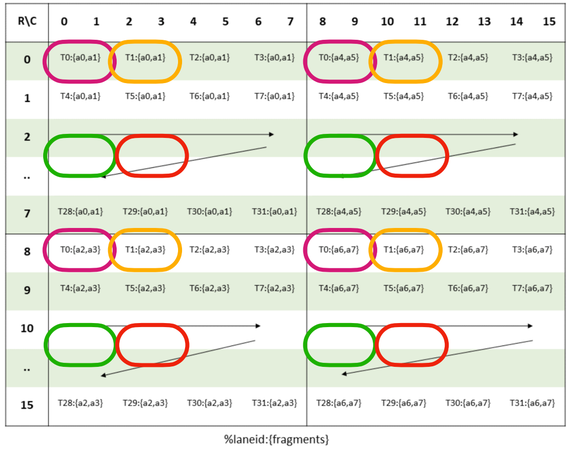
Shared memory: We want to minimize **bank conflicts** as threads pull data from shared memory. The granularity of global loads and registers layouts govern our ability to avoid bank conflicts.

Register memory: We want threads to eagerly store the elements that the **tensor core** expects them to own for matmuls (“eager register layouts").

Thread 0 Thread 1 Thread 16 Thread 17

A[M][K]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	v0{0}	v0{0}	v0{0}	v0{0}	v1{0}	v1{0}	v1{0}	v1{0}	v0{16}	v0{16}	v0{16}	v0{16}	v1{16}	v1{16}	v1{16}	v1{16}	v0{32}	v0{32}	v0{32}	v0{32}	v1{32}	v1{32}	v1{32}	v1{32}	v0{48}	v0{48}	v0{48}	v0{48}	v1{48}	v1{48}	v1{48}	v1{48}
1	v0{1}	v0{1}	v0{1}	v0{1}	v1{1}	v1{1}	v1{1}	v1{1}	v0{17}	v0{17}	v0{17}	v0{17}	v1{17}	v1{17}	v1{17}	v1{17}	v0{33}	v0{33}	v0{33}	v0{33}	v1{33}	v1{33}	v1{33}	v1{33}	v0{49}	v0{49}	v0{49}	v0{49}	v1{49}	v1{49}	v1{49}	v1{49}
2	v0{2}	v0{2}	v0{2}	v0{2}	v1{2}	v1{2}	v1{2}	v1{2}	v0{18}	v0{18}	v0{18}	v0{18}	v1{18}	v1{18}	v1{18}	v1{18}	v0{34}	v0{34}	v0{34}	v0{34}	v1{34}	v1{34}	v1{34}	v1{34}	v0{50}	v0{50}	v0{50}	v0{50}	v1{50}	v1{50}	v1{50}	v1{50}
3	v0{3}	v0{3}	v0{3}	v0{3}	v1{3}	v1{3}	v1{3}	v1{3}	v0{19}	v0{19}	v0{19}	v0{19}	v1{19}	v1{19}	v1{19}	v1{19}	v0{35}	v0{35}	v0{35}	v0{35}	v1{35}	v1{35}	v1{35}	v1{35}	v0{51}	v0{51}	v0{51}	v0{51}	v1{51}	v1{51}	v1{51}	v1{51}
4	v0{4}	v0{4}	v0{4}	v0{4}	v1{4}	v1{4}	v1{4}	v1{4}	v0{20}	v0{20}	v0{20}	v0{20}	v1{20}	v1{20}	v1{20}	v1{20}	v0{36}	v0{36}	v0{36}	v0{36}	v1{36}	v1{36}	v1{36}	v1{36}	v0{52}	v0{52}	v0{52}	v0{52}	v1{52}	v1{52}	v1{52}	v1{52}
5	v0{5}	v0{5}	v0{5}	v0{5}	v1{5}	v1{5}	v1{5}	v1{5}	v0{21}	v0{21}	v0{21}	v0{21}	v1{21}	v1{21}	v1{21}	v1{21}	v0{37}	v0{37}	v0{37}	v0{37}	v1{37}	v1{37}	v1{37}	v1{37}	v0{53}	v0{53}	v0{53}	v0{53}	v1{53}	v1{53}	v1{53}	v1{53}
6	v0{6}	v0{6}	v0{6}	v0{6}	v1{6}	v1{6}	v1{6}	v1{6}	v0{22}	v0{22}	v0{22}	v0{22}	v1{22}	v1{22}	v1{22}	v1{22}	v0{38}	v0{38}	v0{38}	v0{38}	v1{38}	v1{38}	v1{38}	v1{38}	v0{54}	v0{54}	v0{54}	v0{54}	v1{54}	v1{54}	v1{54}	v1{54}
7	v0{7}	v0{7}	v0{7}	v0{7}	v1{7}	v1{7}	v1{7}	v1{7}	v0{23}	v0{23}	v0{23}	v0{23}	v1{23}	v1{23}	v1{23}	v1{23}	v0{39}	v0{39}	v0{39}	v0{39}	v1{39}	v1{39}	v1{39}	v1{39}	v0{55}	v0{55}	v0{55}	v0{55}	v1{55}	v1{55}	v1{55}	v1{55}
8	v0{8}	v0{8}	v0{8}	v0{8}	v1{8}	v1{8}	v1{8}	v1{8}	v0{24}	v0{24}	v0{24}	v0{24}	v1{24}	v1{24}	v1{24}	v1{24}	v0{40}	v0{40}	v0{40}	v0{40}	v1{40}	v1{40}	v1{40}	v1{40}	v0{56}	v0{56}	v0{56}	v0{56}	v1{56}	v1{56}	v1{56}	v1{56}
9	v0{9}	v0{9}	v0{9}	v0{9}	v1{9}	v1{9}	v1{9}	v1{9}	v0{25}	v0{25}	v0{25}	v0{25}	v1{25}	v1{25}	v1{25}	v1{25}	v0{41}	v0{41}	v0{41}	v0{41}	v1{41}	v1{41}	v1{41}	v1{41}	v0{57}	v0{57}	v0{57}	v0{57}	v1{57}	v1{57}	v1{57}	v1{57}
10	v0{10}	v0{10}	v0{10}	v0{10}	v1{10}	v1{10}	v1{10}	v1{10}	v0{26}	v0{26}	v0{26}	v0{26}	v1{26}	v1{26}	v1{26}	v1{26}	v0{42}	v0{42}	v0{42}	v0{42}	v1{42}	v1{42}	v1{42}	v1{42}	v0{58}	v0{58}	v0{58}	v0{58}	v1{58}	v1{58}	v1{58}	v1{58}
11	v0{11}	v0{11}	v0{11}	v0{11}	v1{11}	v1{11}	v1{11}	v1{11}	v0{27}	v0{27}	v0{27}	v0{27}	v1{27}	v1{27}	v1{27}	v1{27}	v0{43}	v0{43}	v0{43}	v0{43}	v1{43}	v1{43}	v1{43}	v1{43}	v0{59}	v0{59}	v0{59}	v0{59}	v1{59}	v1{59}	v1{59}	v1{59}
12	v0{12}	v0{12}	v0{12}	v0{12}	v1{12}	v1{12}	v1{12}	v1{12}	v0{28}	v0{28}	v0{28}	v0{28}	v1{28}	v1{28}	v1{28}	v1{28}	v0{44}	v0{44}	v0{44}	v0{44}	v1{44}	v1{44}	v1{44}	v1{44}	v0{60}	v0{60}	v0{60}	v0{60}	v1{60}	v1{60}	v1{60}	v1{60}
13	v0{13}	v0{13}	v0{13}	v0{13}	v1{13}	v1{13}	v1{13}	v1{13}	v0{29}	v0{29}	v0{29}	v0{29}	v1{29}	v1{29}	v1{29}	v1{29}	v0{45}	v0{45}	v0{45}	v0{45}	v1{45}	v1{45}	v1{45}	v1{45}	v0{61}	v0{61}	v0{61}	v0{61}	v1{61}	v1{61}	v1{61}	v1{61}
14	v0{14}	v0{14}	v0{14}	v0{14}	v1{14}	v1{14}	v1{14}	v1{14}	v0{30}	v0{30}	v0{30}	v0{30}	v1{30}	v1{30}	v1{30}	v1{30}	v0{46}	v0{46}	v0{46}	v0{46}	v1{46}	v1{46}	v1{46}	v1{46}	v0{62}	v0{62}	v0{62}	v0{62}	v1{62}	v1{62}	v1{62}	v1{62}
15	v0{15}	v0{15}	v0{15}	v0{15}	v1{15}	v1{15}	v1{15}	v1{15}	v0{31}	v0{31}	v0{31}	v0{31}	v1{31}	v1{31}	v1{31}	v1{31}	v0{47}	v0{47}	v0{47}	v0{47}	v1{47}	v1{47}	v1{47}	v1{47}	v0{63}	v0{63}	v0{63}	v0{63}	v1{63}	v1{63}	v1{63}	v1{63}

AMD 16x16x16 BF16 A matrix register layout.



Nvidia BF16 A matrix register layout.

Memory layout challenges

Towards an FP6 GEMM on MI355X:

- **Global**: for loading global to shared memory we want to use the max 16-byte, 4-bank load granularity
- **Shared**: shared memory has 64 banks
- **Register**: for 32x32x64 FP6 matrix instructions each thread needs to own 6 registers (6 banks worth of data)

Threads 0-15, 32-47 [Conflict]

0	1	2	3	4	5	6	7	8	9	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 !	0	1	2	3	4	5	6	7
8	9	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	29	40	41	42	43
44	45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63 !	0	1	2	3
4	5	6	7	8	9	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 !

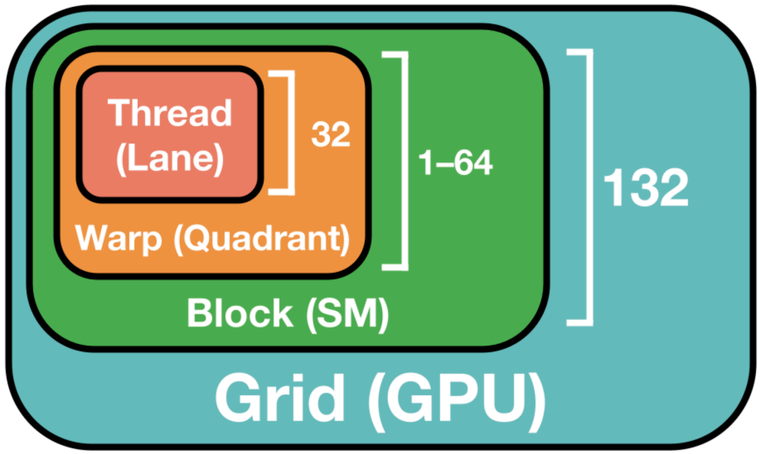
Threads 16-31, 48-63

0	1	2	3	4	5	6	7	8	9	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 !	0	1	2	3	4	5	6	7
8	9	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	29	40	41	42	43
44	45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63 !	0	1	2	3
4	5	6	7	8	9	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: Register layout for 32x32x64 shaped FP6 tensor core instructions on AMD. Each register owns 6 banks

ThunderKittens library

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together



GPU Hierarchy

```python
# imports
import torch
import torch.nn.functional as F


# compute Q@K.T
att = torch.matmul(
    q, k.transpose(2, 3))

# compute softmax
att = F.softmax(
    att, dim=-1,
    dtype=torch.float32)

# convert back to bf16
att = att.to(q.dtype)

# mma att@V
attn_output = torch.matmul(att, v)
```

PyTorch operations over tensors

```cpp
// imports
using namespace kittens;
rt_bf<16, 64> k_reg, v_reg;
// load k from shared memory to register
load(k_reg, k_smem[subtile]);
// compute Q@K.T
zero(att);
mma_ABt(att, q_reg, k_reg, att);
// compute softmax
sub_row(att, att, max_vec);
exp(att, att);
div_row(att, att, norm_vec);
// convert to bf16 for mma_AB
copy(att_mma, att);
// load v from shared memory to register
load(v_reg, v_smem[subtile]);
auto &v_reg_col = swap_layout_inplace(v_reg);
// mma att@V onto o_reg
mma_AB(o_reg, att_mma, v_reg_col, o_reg);
```

TK operations over tiles

Function abstractions:
Templated NumPy and PyTorch like library functions applied over the basic tiles

Tile-based operations

Functions in TK have **no side effects** and are almost an ISA:

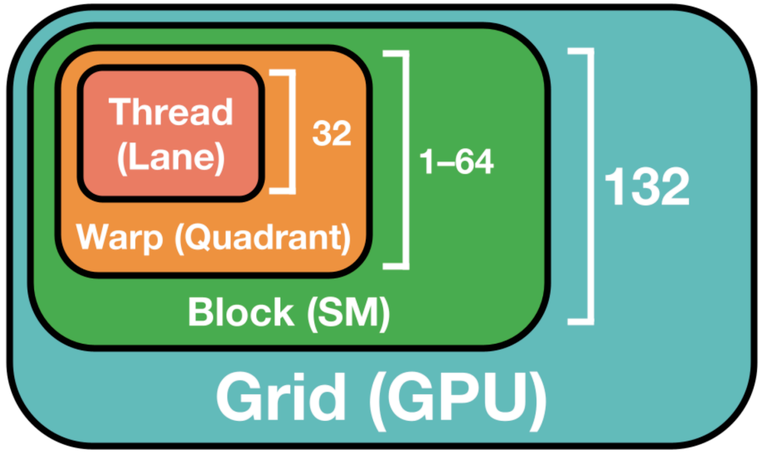
- Functions are simple wrappers over assembly instructions (PTX / CDNA ISA)
- All TK functions take the form `func(dest, src1, src2)`

The function set in TK is **extensible**:

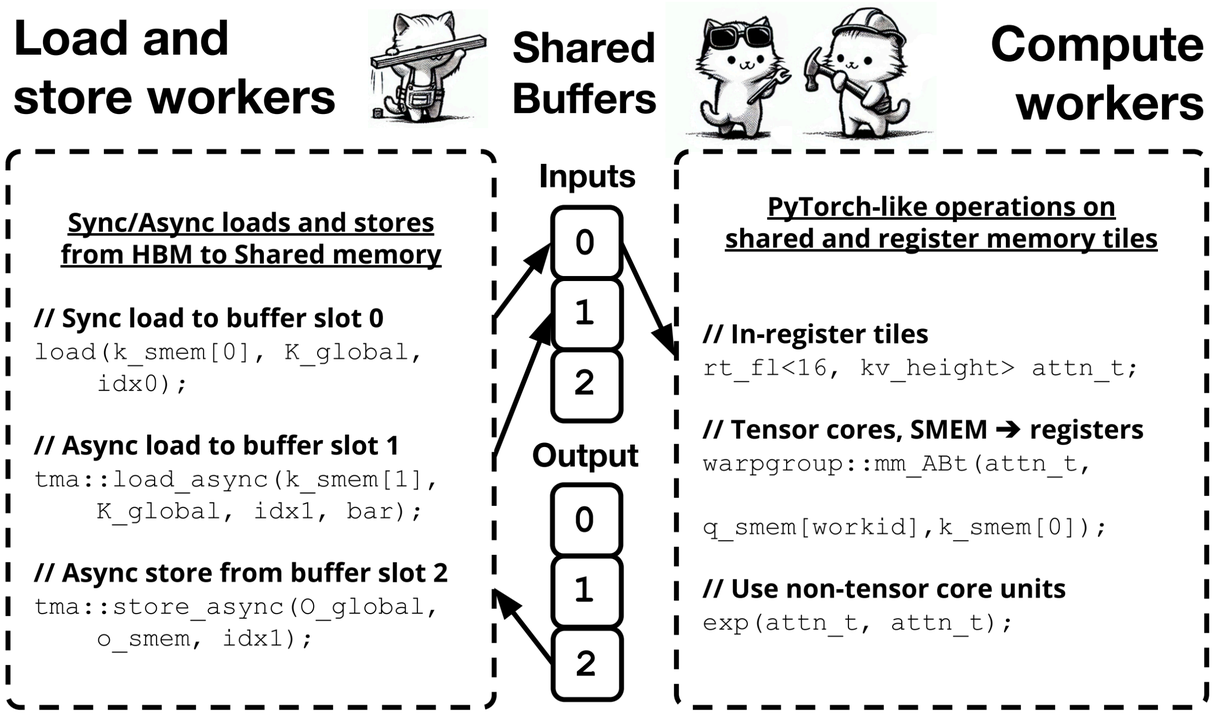
- Categories of functions (e.g., rowmax, rowmin, rowsum) are implemented using a minimal set of template patterns (e.g., row reduction pattern).
- If a function (or hardware platform) is not yet supported in TK, a developer can write their own inline function in CUDA / HIP. TK fails gracefully.

ThunderKittens library

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data

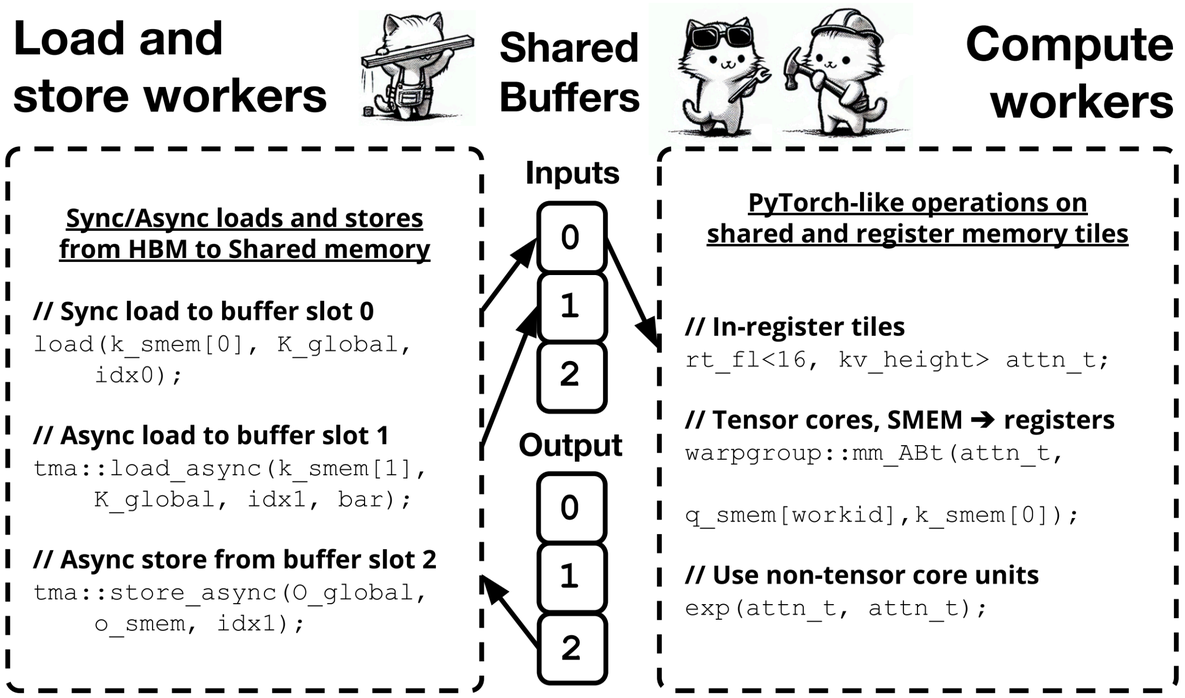


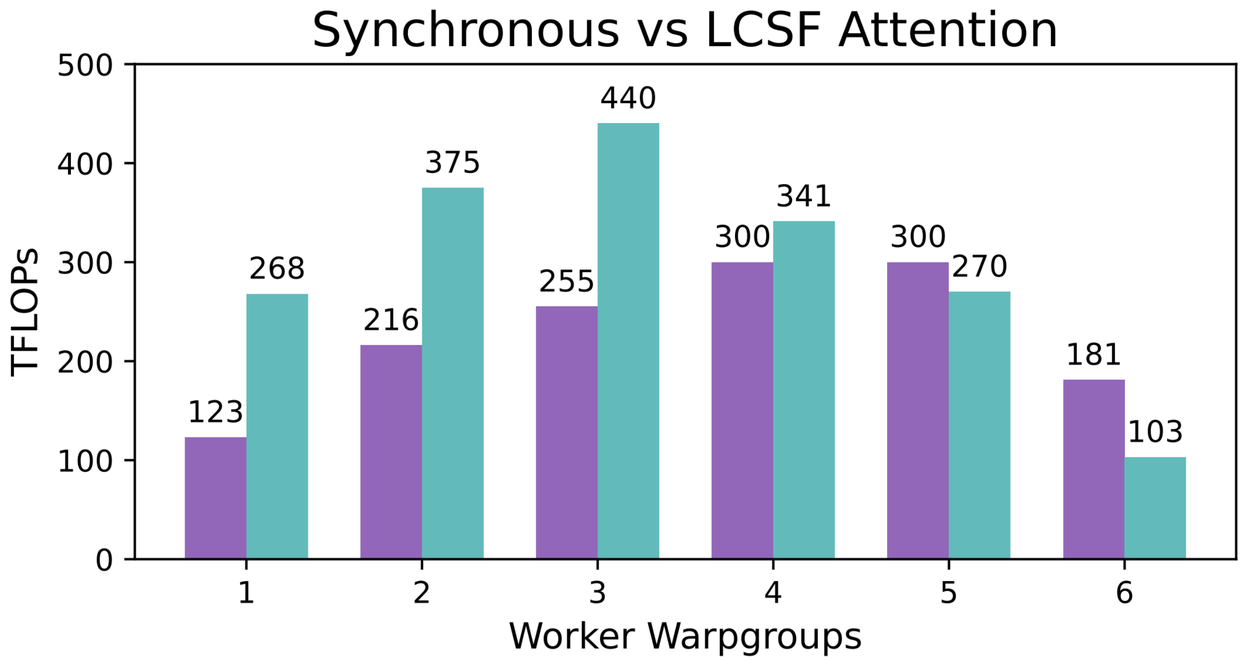
GPU Hierarchy



Data flow kernel schedules:
There are a few simple data flow patterns (software pipelining, worker specialization) for AI kernels. Let's templatize them for the developer!

Example: Standard synchronous GPU code versus a TK template.



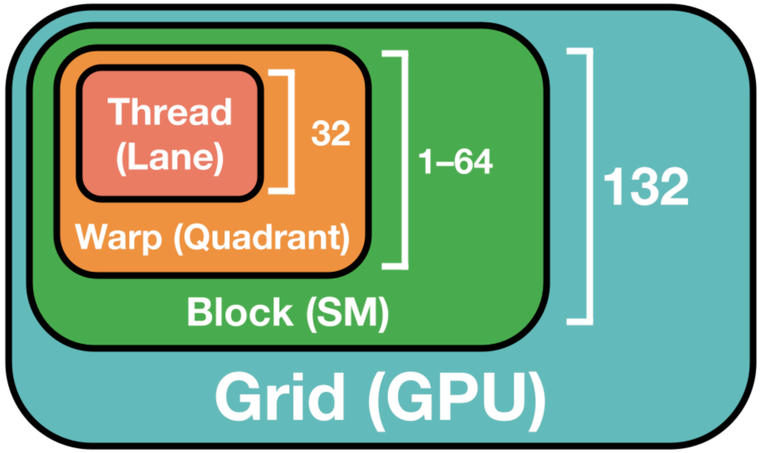


M = N = K	Stages	TFLOPS
4096	1	260
4096	2	484
4096	3	683
4096	4	760

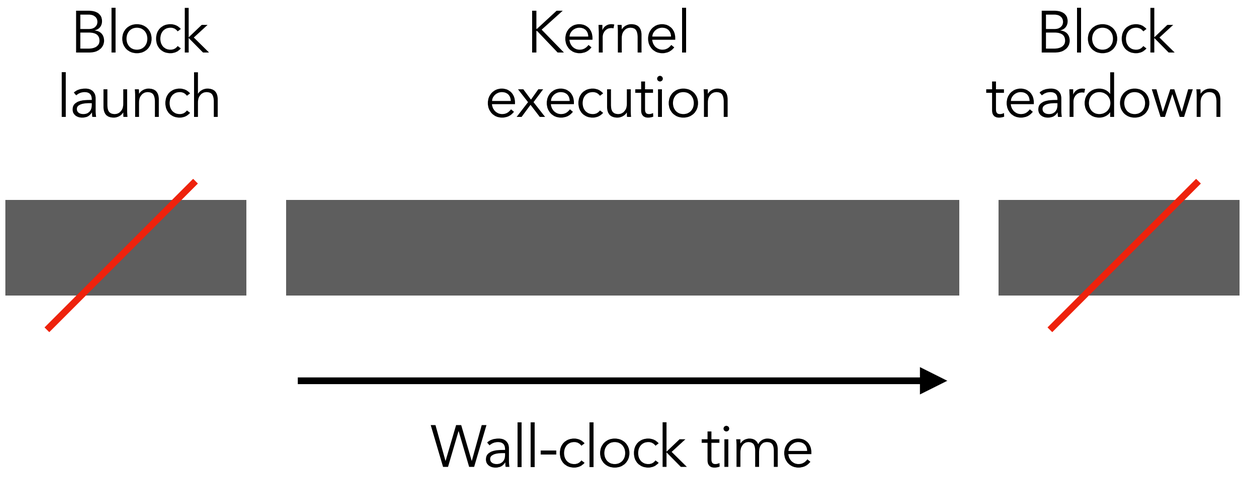
Table 1: **Pipeline buffer stages** We measure efficiency in TFLOPS for our GEMM kernels as we vary the number of pipeline buffer stages in the TK template.

ThunderKittens library

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data
- **Grid**: GPUs run many blocks of threads at once



GPU Hierarchy

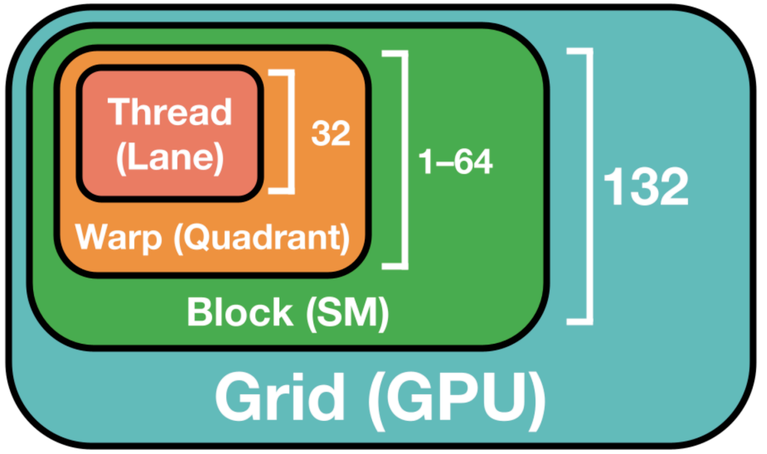


M=N, K	TK-No	TK-Yes	CuBLAS
4096, 64	93	108	69
4096, 128	161	184	133
4096, 256	271	309	242
4096, 512	414	450	407
4096, 1024	565	600	633

Table 2: **Persistent block launch** TFLOPS for TK GEMM kernels with (**yes**) persistent and without (**no**) persistent launch as we vary K.

ThunderKittens library

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data
- **Grid**: GPUs run many blocks of threads at once



GPU Hierarchy

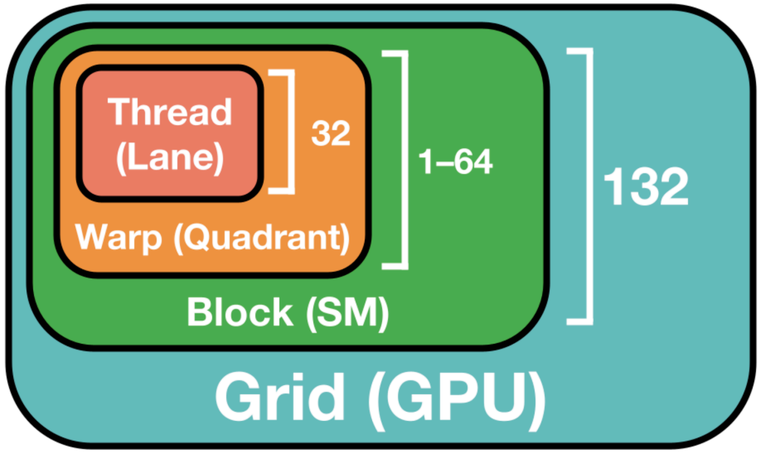
Matrix Multiply (M=N=K=16384)		
Block Order	HBM GB/s	TFLOPS
{8, N, M/8}	982	805
{N, M}	3,070	392

Attention Forward (D=128)		
Block Order	HBM GB/s	TFLOPS
{N, H, B}	213	600
{B, H, N}	2,390	494

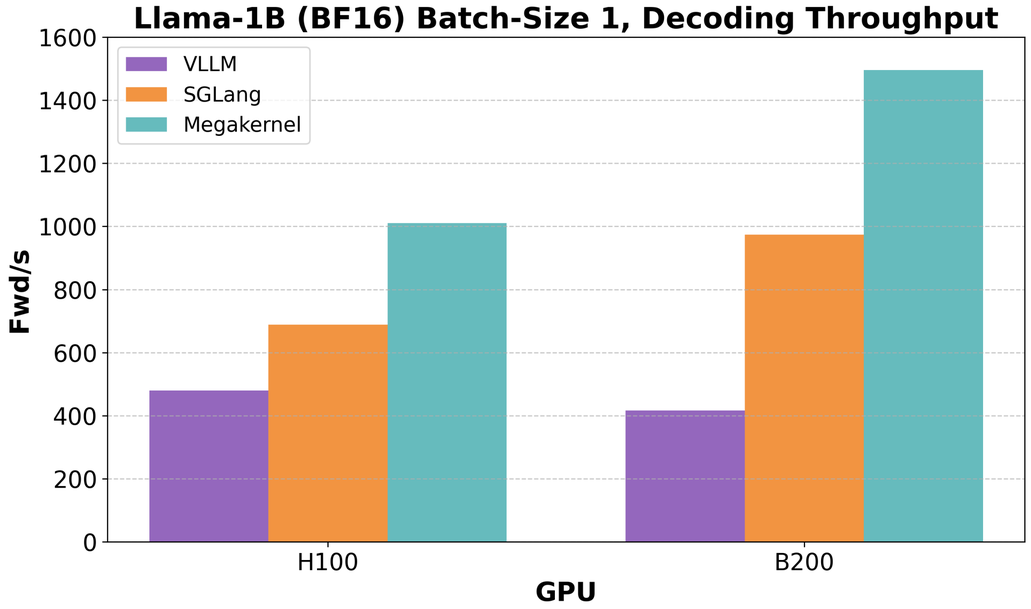
Table 3: **L2 reuse** We vary the block orders and measure both consumed bandwidth from HBM (GB/s) and efficiency (TFLOPs). For attention, we consider an optimized kernel, with an internal tiling of 8 rows of blocks, versus a naive that schedules blocks in row-major order. For attention, we compare block order (1) sequence length N, heads H, and outermost batch B vs. (2) innermost B, H, then outermost N. Different block orders have significant performance implications.

MegaKernels built on top of TK

- **Threads**: Tens of thousands of threads run on GPUs
- **Warps**: groups of threads that run instructions together
- **Blocks**: groups of warps, which can quickly share data
- **Grid**: GPUs run many blocks of threads at once
- **GPU**: multiple devices collaborate to run large-scale AI



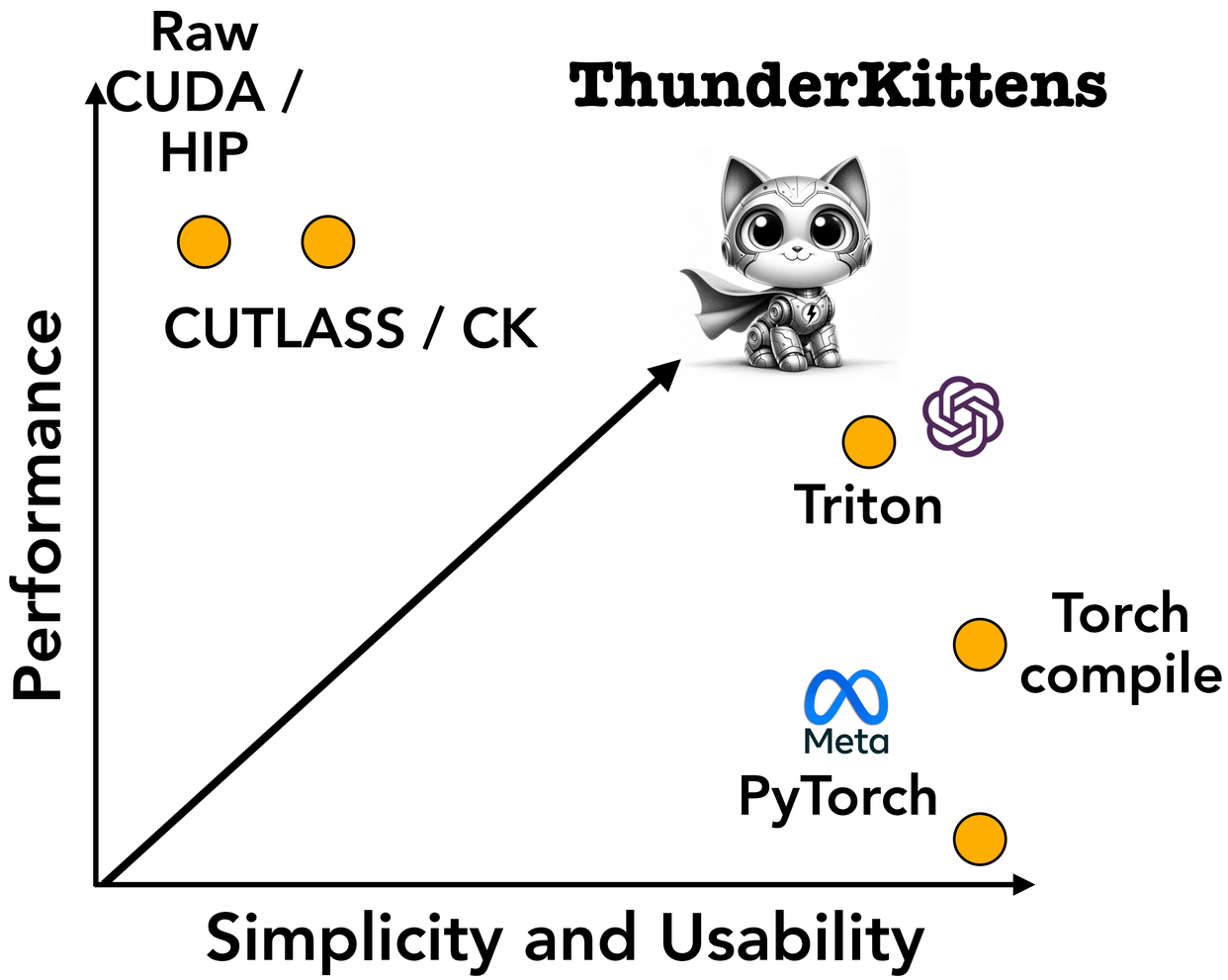
GPU Hierarchy



Research question

How concise a set of programming abstractions can we use to support fast, simple kernels for a breadth of AI workloads?

ThunderKittens: tile-based programming for AI kernels

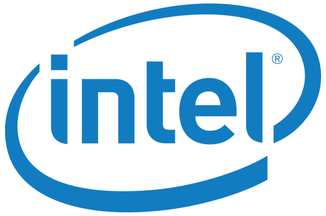


- AI workloads follow a few patterns; TK can us a small number of primitives.
- Primitives are transparent to the underlying hardware.
- Set of primitives are restricted to the patterns that yield high performance.

ThunderKittens: simple, fast, and adorable AI kernels

Workload	TK LoC	Reference LoC	Speed up (min-max)
H100 Attention inference	217	2325 (CUTLASS FA3)	0.87-1.14×
H100 BF16 GEMM	84	463 (CUTLASS)	0.98-2.05×
H100 FP8 GEMM	91	Closed Source (CuBLASLt)	0.93-1.10×
H100 Convolution	131	624 (CUDA FlashFFTConv)	4.6-4.7×
H100 Based linear attention	282	89 (Triton)	3.7-14.5×
H100 Hedgehog linear attention	316	104 (Triton)	4.0-6.5×
H100 Mamba-2	192	532 (Triton)	3.0-3.7×
H100 Rotary	101	119 (Triton)	1.1-2.3×
4090 Attention ($D = 64$)	93	1262 (CUTLASS FA2)	0.96-0.98×
4090 Attention ($D = 128$)	93	1262 (CUTLASS FA2)	0.89-0.95×
Apple M2 Attention ($D = 64$)	47	343 (Apple MLX)	1.12-1.15×
Apple M2 GEMM	27	412 (Apple MLX)	1.04-1.12×

TK kernels use few lines of code (~simple) and are fast.

ThunderKittens: simple, fast, and adorable AI kernels

Open-source:

- 2.6K+ GitHub stars in 1 year
- One of 3 DSLs of choice in Meta and GPU MODE's kernel competition!

Research:

- ICLR 2025 Spotlight (top 5.1% of 12k papers)
- Used in published work: Stanford, Nvidia, UCSD, UT Austin, UC Berkeley, University of Italy, etc.
- Rapid portability to new hardware platforms

Industry:

- Used to write custom production kernels at Cursor, Cruise, Jump Trading, Together AI, and more!
- Led a wave of simplified documentation and simpler AI DSLs over the past year

CURSOR cruise Meta GPU MODE jumptrading together.ai

Python-like abstractions for peak-performance kernels

✨ThunderKittens✨
(May 2024)

TileLang
(December
2024)

TileLang
TILE LANGUAGE

Triton Linear
Layouts
(January 2025)

Nvidia Pythonic CuTe
DSL / CUTLASS 4.0
(May 2025)

NVIDIA

Max Kernels Library

ThunderKittens builds on the wonderful work of many: Triton, Halide, CUTLASS, Flash Attention…

Overview

1. Introduction to AI hardware
2. ThunderKittens: Tile-based programming for AI kernels
3. **What architecture does *the hardware* prefer?**
4. Key directions

Attention

For $Q, K, V \in \mathbb{R}^{N \times d}$:

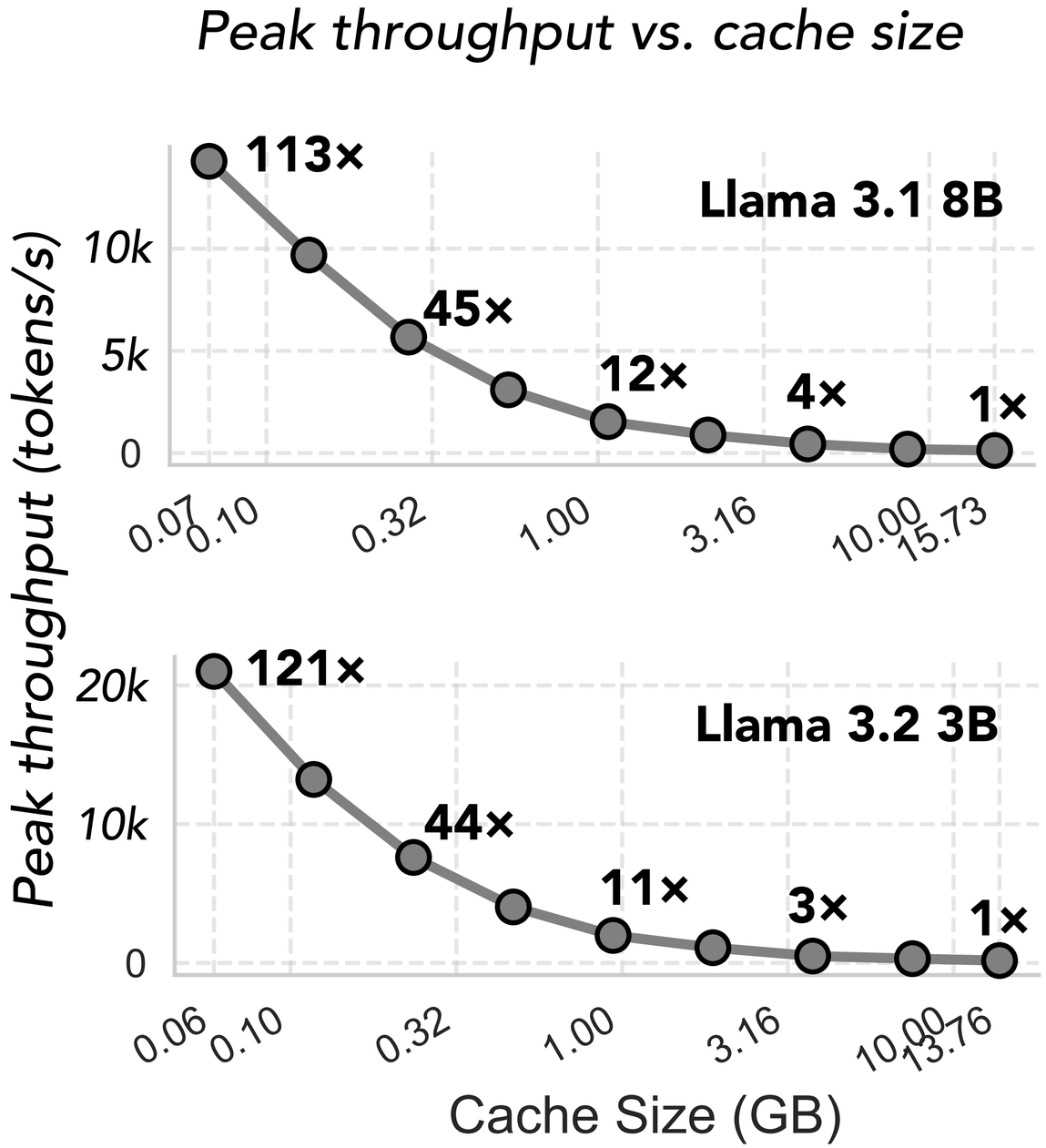
$$y_i = \sum_i \frac{e^{Q_i \cdot K_i}}{\sum_j^i e^{Q_i \cdot K_j}} V_i$$

$$y = \text{Softmax}(QK^T)V$$

Memory complexity is $O(Nd)$ and compute complexity is $O(N^2 d)$.

Attention is expensive

Memory complexity is $O(Nd)$ and compute complexity is $O(N^2d)$.



Linear attention

For $Q, K, V \in \mathbb{R}^{N \times d}$ and feature map $\phi() : \mathbb{R}^{N \times d} \rightarrow \mathbb{R}^{N \times f}$:

$$y_i = \frac{\sum_{j=1}^{i} \phi(Q_i^T)\phi(K_j)}{\sum_{j=1}^{i} \phi(Q_i)^T\phi(K_j)} V_j$$

$$y = \phi(Q)\phi(K^T)V$$

Use an alternate similarity function such that: $\phi(Q)\phi(K^T) \approx \exp(QK^T)$.

Linear attention

For $Q, K, V \in \mathbb{R}^{N \times d}$ and feature map $\phi() : \mathbb{R}^{N \times d} \rightarrow \mathbb{R}^{N \times f}$:

Multiply keys and values first: $$y_i = \phi(Q_i^T)(\sum_{j=1}^{i} \phi(K_j)V_j)$$

Memory: $O(df)$

Compute: $O(Ndf)$

Multiply keys and values first: $$y_i = (\phi(Q_i^T)\phi(K_{0:i}))V_{0:i}$$

Memory: $O(Nd)$

Compute: $O(N^2 d)$

Linear attention

For $Q, K, V \in \mathbb{R}^{N \times d}$ and feature map $\phi() : \mathbb{R}^{N \times d} \rightarrow \mathbb{R}^{N \times f}$:

$$y_i = \phi(Q_i^T)(\sum_{j=1}^{i} \phi(K_j)V_j)$$

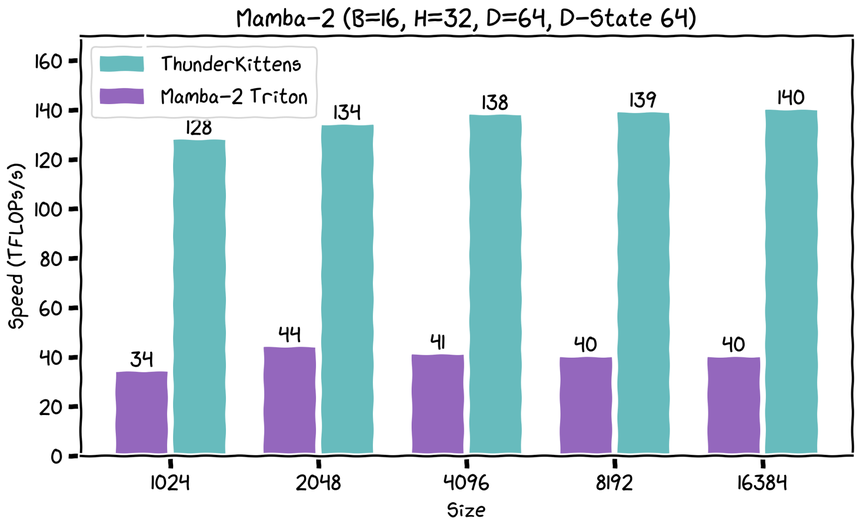
Memory: $O(df)$

Compute: $O(Ndf)$

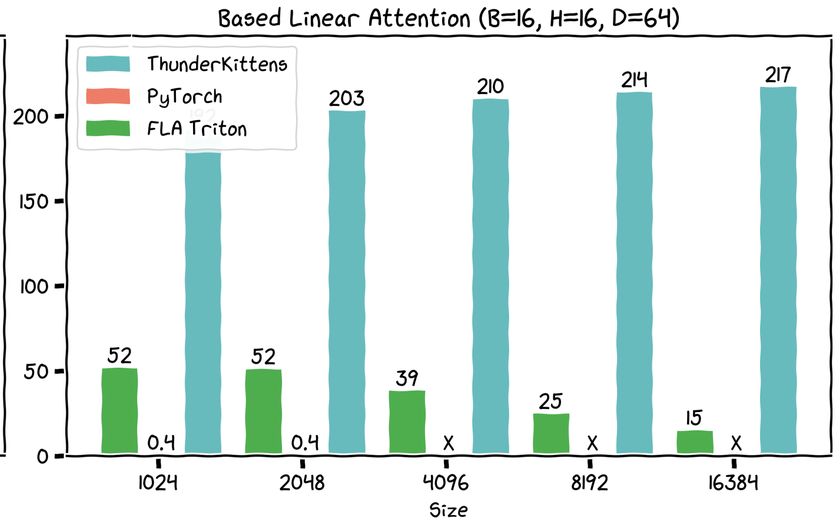
How should we choose the feature map $\phi()$ to balance quality and efficiency?

There are *many* linear attention and efficient model variants.

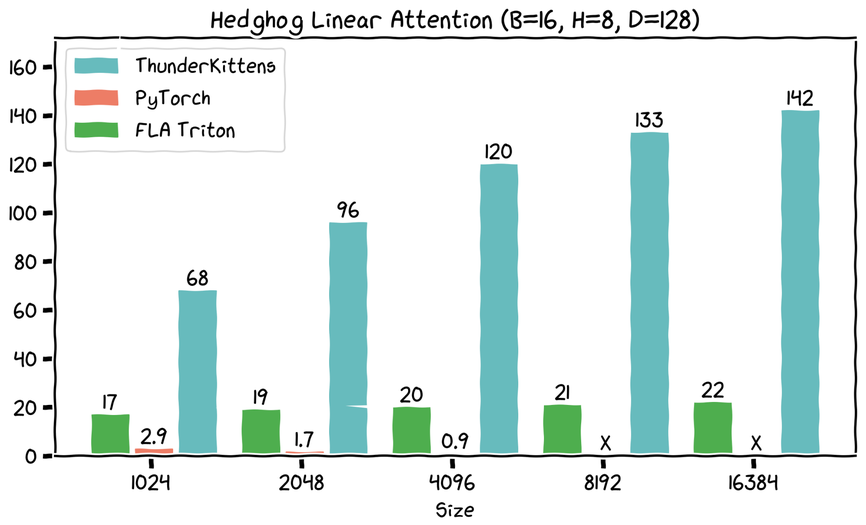
Dao et al., 2024.



Arora et al., 2023.



Zhang et al., 2024.

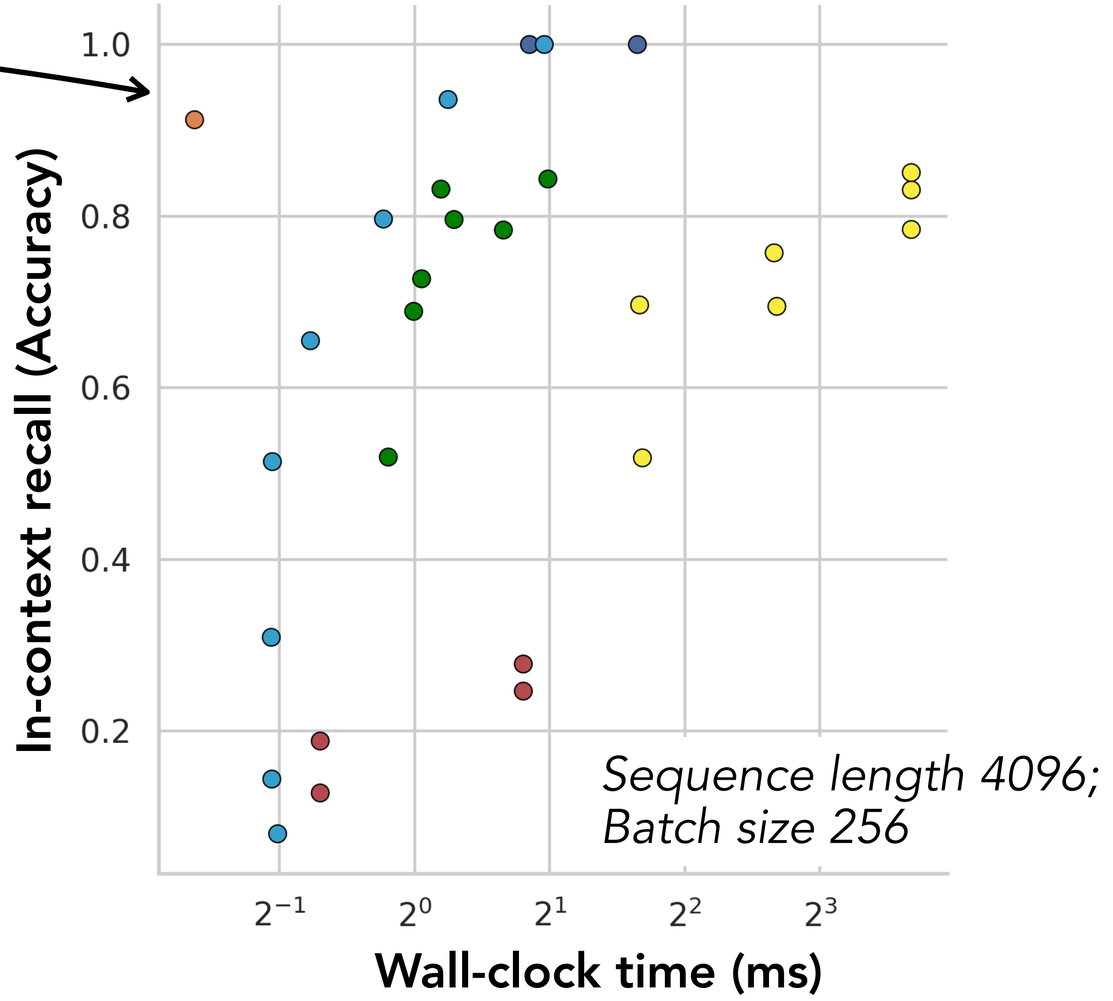


How should we choose the feature map $\phi()$ to balance quality and efficiency?

Based: efficient linear attention *for Hopper++ GPUs*

Based can use ***8x the memory*** of *Mamba*!

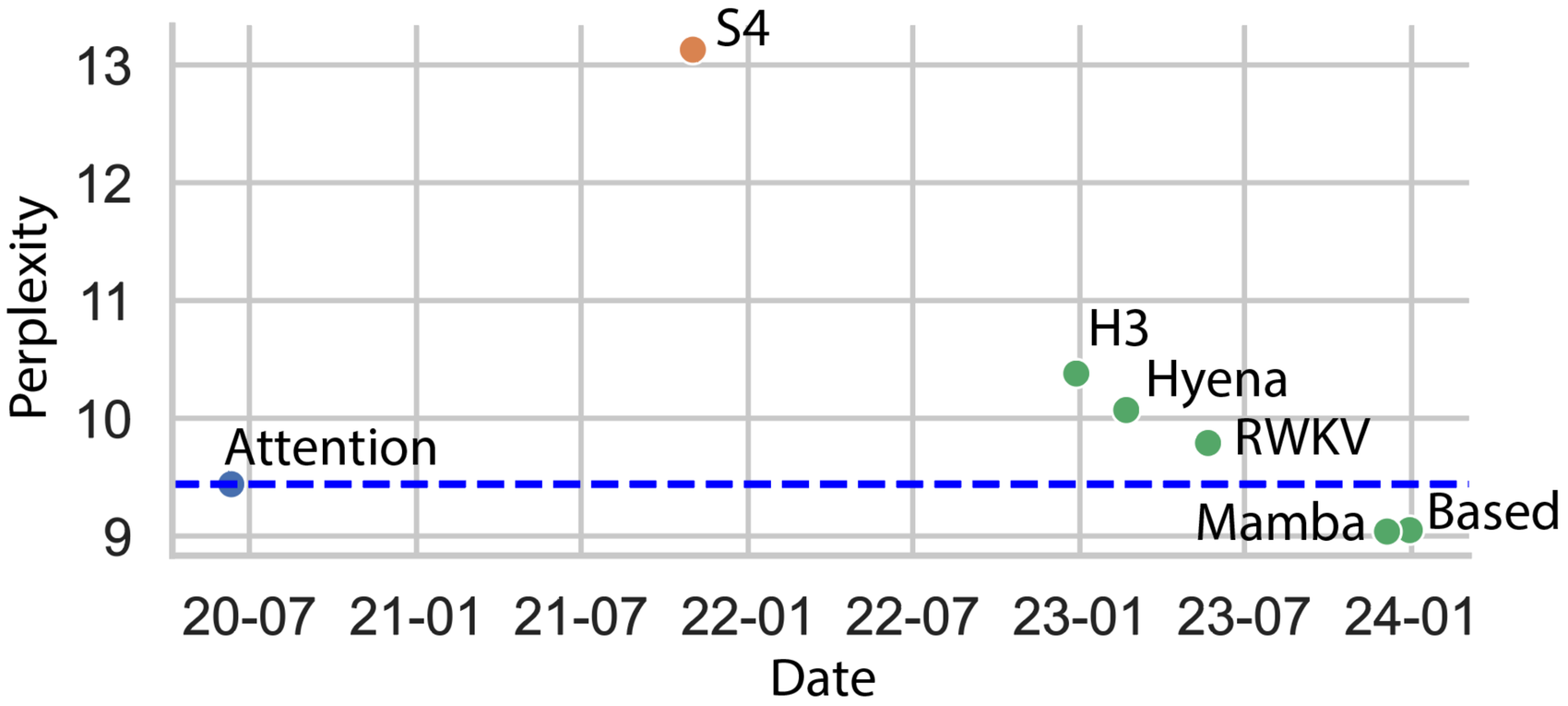
While running at ***5x hardware utilization on Hopper GPUs***!



Overview

1. Introduction to AI hardware
2. ThunderKittens: Tile-based programming for AI kernels
3. What architecture does *the hardware* prefer?
4. **Key directions**

1. Theoretical vs. hardware efficiency perspective



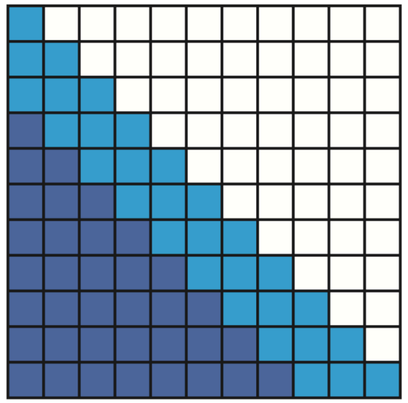
Pile 10B @ 350M parameters.
Trained on Eleuther AI (GPT-NeoX)

Based approaches the perplexity of attention.

Many models can get you there! … But which model does the hardware prefer?

Impact of Based on efficient models!

✨BASED✨
(December 2023)



Mamba-v2
(June 2024)

RWKV-v6
(Sept 2024)

MiniMax
(January 2025)

Nvidia Nemotron-H
Tencent Hunyuan-T1
Liger attention
(March 2025)

Many popular models use approaches from BASED.

Communication and memory

https://www.nvidia.com/en-eu/data-center/gb200-nvl72/

- Nvidia
  - NVL72 (72 super chips, each with a Blackwell and grace CPU) at 900 GB/s unidirectional speeds
  - 100 GB/s for NVL72 A <-> NVL72 B
- AMD
  - XGMI scale up 64 GB/s point-to-point mesh
  - Scale out at 50 GB/s node <-> node

Ben Spector (TK, MegaKernels)

Aaryan Singhal (TK, ThunderMittens)

Will Hu (AMD TK)

Sabri Eyuboglu (Based)

Jordan Juravsky (MegaKernels)

Conner Takehana (ThunderMittens)

Dan Fu (TK)

Stuart Sul (MegaKernels)

Dylan Lim (MegaKernels)

Chris Ré

Try it out and contribute!

https://github.com/HazyResearch/ThunderKittens

https://github.com/HazyResearch/ThunderMittens

https://github.com/HazyResearch/ThunderKittens-HIP (Coming soon!)