Machine Learning Systems

Build efficient and scalable ML services through the vertical integration of algorithms, system software, and hardware

Acknowledgement

Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community.

CUDA Programming

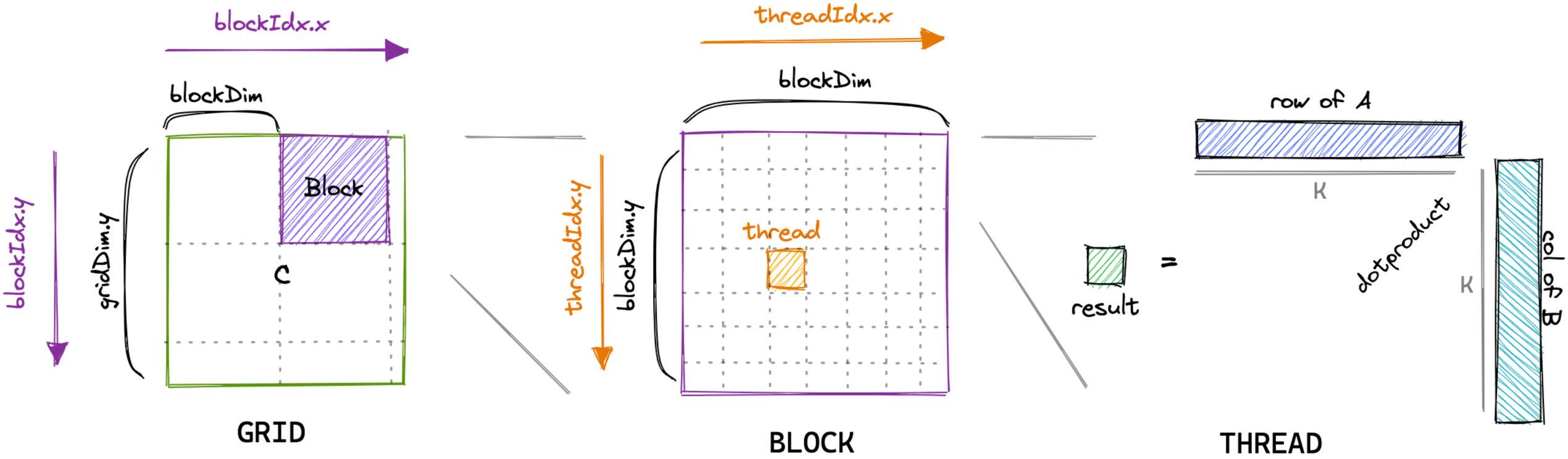
Mapping High-Dimensional Workloads onto Linear & Hierarchical Hardware

Data layout mismatch: Multi-dimensional tensors must map to linear memory → performance depends on coalescing and locality.

Execution mismatch: Multi-dimensional threads are executed as 1D warps → thread mapping determines memory access efficiency.

Communication placement: Same computation (e.g., O(N^3)) has very different cost depending on whether data movement happens in HBM, shared memory, or registers.

Thread Organization



We put as many blocks into the grid as necessary to span all of C

Each block is responsible for calculating a 32x32 chunk of C

Each thread independently computes one entry of C

Thread Organization

```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // if statement is necessary to make things work under tile quantization
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

Is matrix multiplication computation bound or memory bound?

H100 SXM

Communication vs. Computation

Communication:

- 80 GB of HBM3 memory, 6.4 Gb/s per pin, 1024 pin per stack:

$$\frac{1024 \text{ bits} \times 6.4 \times 10^9 \text{ bits/s}}{8} = 819.2 \times 10^9 \text{ bytes/s} = 0.8192 \text{ TB/s (i.e. } 819.2 \text{ GB/s).}$$

- HBM3 5 stacks, peak memory bandwidth $5 \times 0.8192 = 4.096 \text{ TB/s}$.

Computation: (>1 PFLOP/s theoretical)

- FP16/FP8 Tensor Core peak ≈ 1979 TFLOP/s
- FP32 Tensor Core peak ≈ 989 TFLOP/s
- FP32 CUDA cores ≈ 60 TFLOP/s

Matrix Multiplication

C = A x B + C, with size of 4096 x 4096

Total FLOPS: For each of the 4096^2 entries of C, we have to perform a dot product of two vectors of size 4096, involving a multiply and an add at each step. “Multiply then add” is often mapped to a single assembly instruction called FMA (fused multiply-add), but still counts as two FLOPs.

$2*4096^3 + 4096^2 = 137$ GFLOPS

Total data to read (minimum): $3 * 4096^2 * 4B = 201MB$

Total data to store: $4096^2 * 4B = 67MB$

Matrix Multiplication

C = A x B + C, with size of 4096 x 4096

1️⃣ Communication time (memory transfer)

You estimated that the kernel needs to **read ≈201 MB** (A + B + C) and **write ≈67 MB** (C).

Let's take the total **≈268 MB = 0.268 GB = 0.000268 TB**.

Peak H100 SXM global-memory bandwidth ≈ 3.35–4.09 **TB/s** (depending on clock/variant).

$$t_{\text{comm}} = \frac{0.268\ \text{TB}}{4.09\ \text{TB/s}} \approx 6.6 \times 10^{-5}\ \text{s} = 0.066\ \text{ms}$$

Matrix Multiplication

C = A x B + C, with size of 4096 x 4096

2️⃣ Compute time

The total work is 137 GFLOPs = 0.137 TFLOPs.

Peak H100 FP32 throughput ≈ 60 TFLOP/s.

$$t_{\text{compute}} = \frac{0.137\ \text{TFLOPs}}{60\ \text{TFLOP/s}} \approx 2.28 \times 10^{-3}\ \text{s} = 2.3\ \text{ms}$$

Matrix Multiplication

C = A x B + C, with size of 4096 x 4096

- The computation time (≈2.3 ms) is ~30× larger than the pure memory-transfer lower bound (≈0.07 ms).
- The operation is compute-bound, not bandwidth-bound.
- In reality, achieved bandwidth and FLOPs are a fraction of peak, but GEMM is very efficient, so you'll still be dominated by compute rather than memory.

Matrix Multiplication

Welcome to the real world!

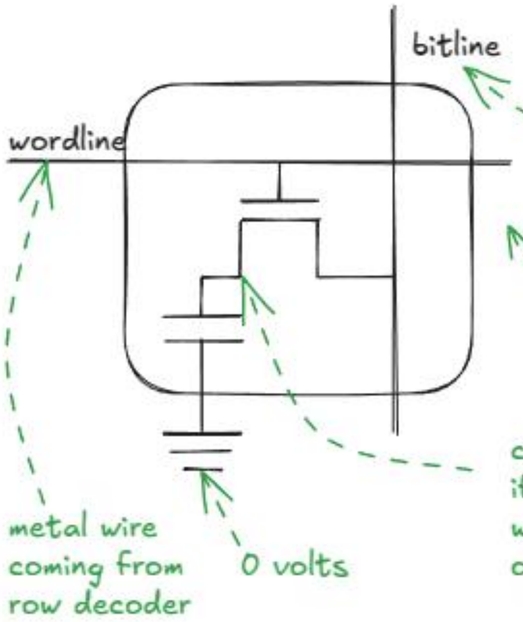
FP32 CUDA cores offer approximately 60 TFLOP/s computing power. However, the actual performance is approximately 500 GFLOPS, and the run time is over 200ms.

```
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000073) s, performance: (   57.7) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000141) s, performance: (  238.0) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 3
Average elapsed time: (0.000552) s, performance: (  485.9) GFLOPS. size: (512).
dimensions(m=n=k) 1024, alpha: 0.5, beta: 3
Average elapsed time: (0.004388) s, performance: (  489.4) GFLOPS. size: (1024).
dimensions(m=n=k) 2048, alpha: 0.5, beta: 3
Average elapsed time: (0.035030) s, performance: (  490.4) GFLOPS. size: (2048).
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.273527) s, performance: (  502.5) GFLOPS. size: (4096).
```

Do we really understand memory?

GMEM is nothing but a matrix of DRAM cells.

and DRAM cells are glorified guarded capacitors. :)



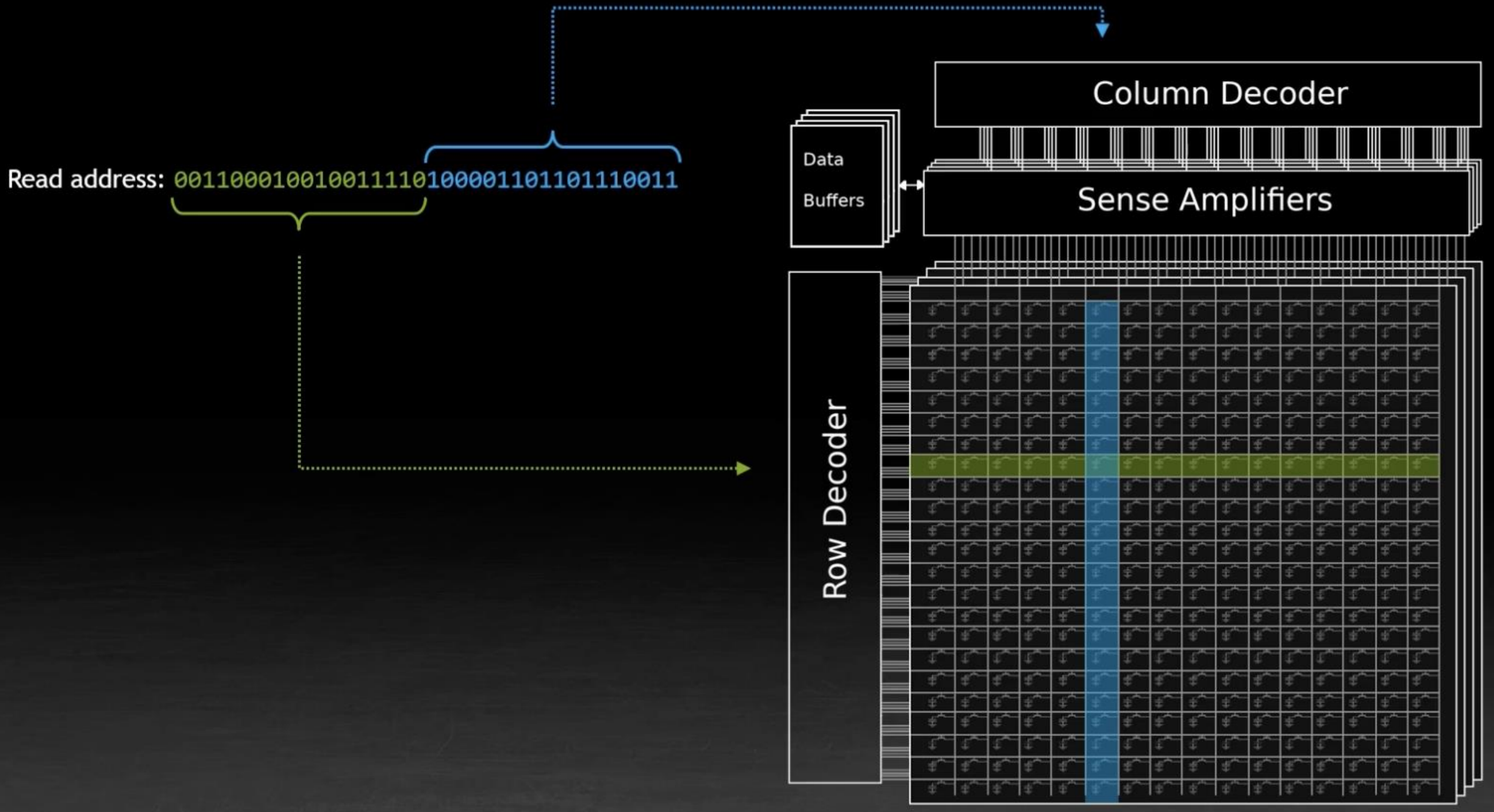
metal wire coming from sense amplifier

capacitor stores a charge;
if V > V_t (threshold)
we interpret it as a logical 1,
otherwise as a logical 0.

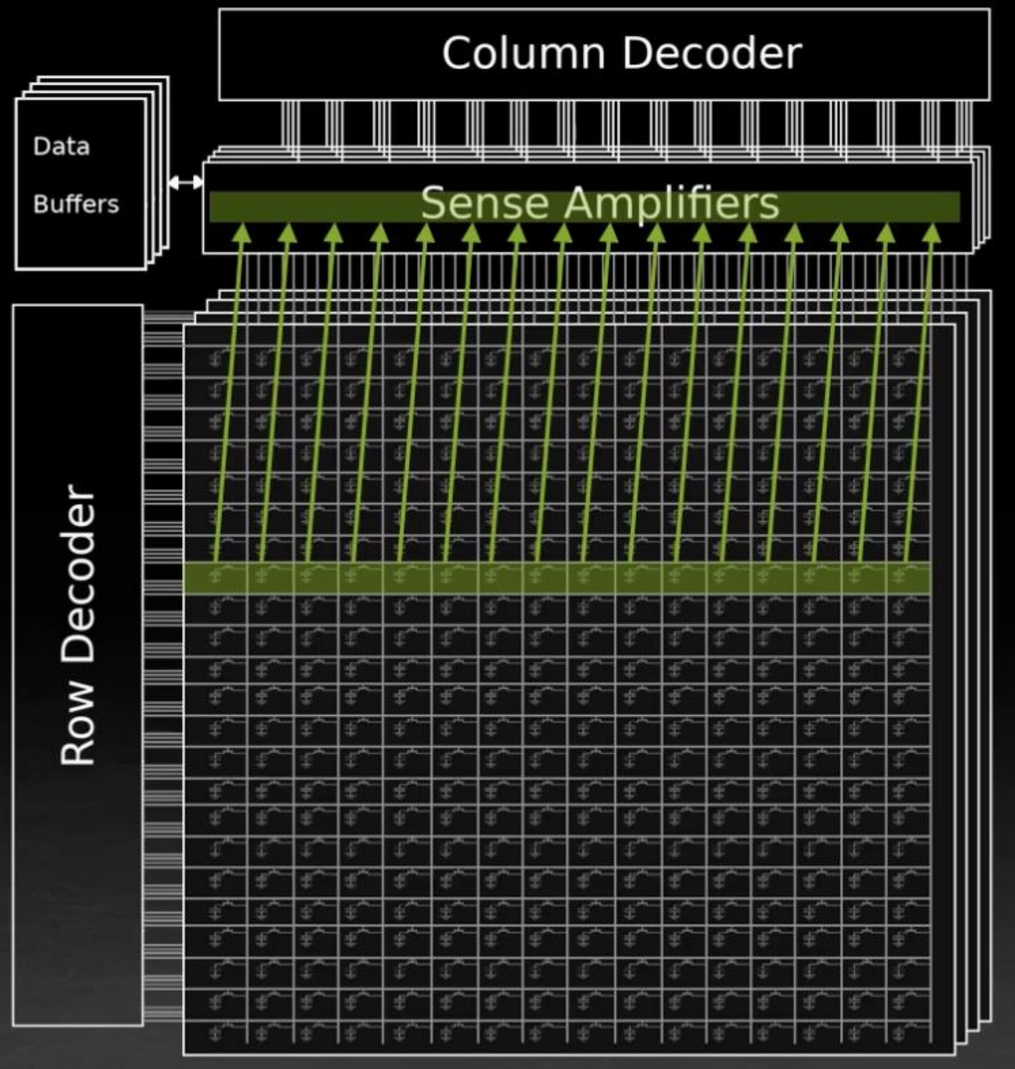
you can see how simple a DRAM cell is
and why we can build dense DRAM memories!

when the wordline "goes hot" (i.e. is activated by setting its voltage to V)
transistor's gate opens and the stored charge flows to the bitline, where it is sensed.

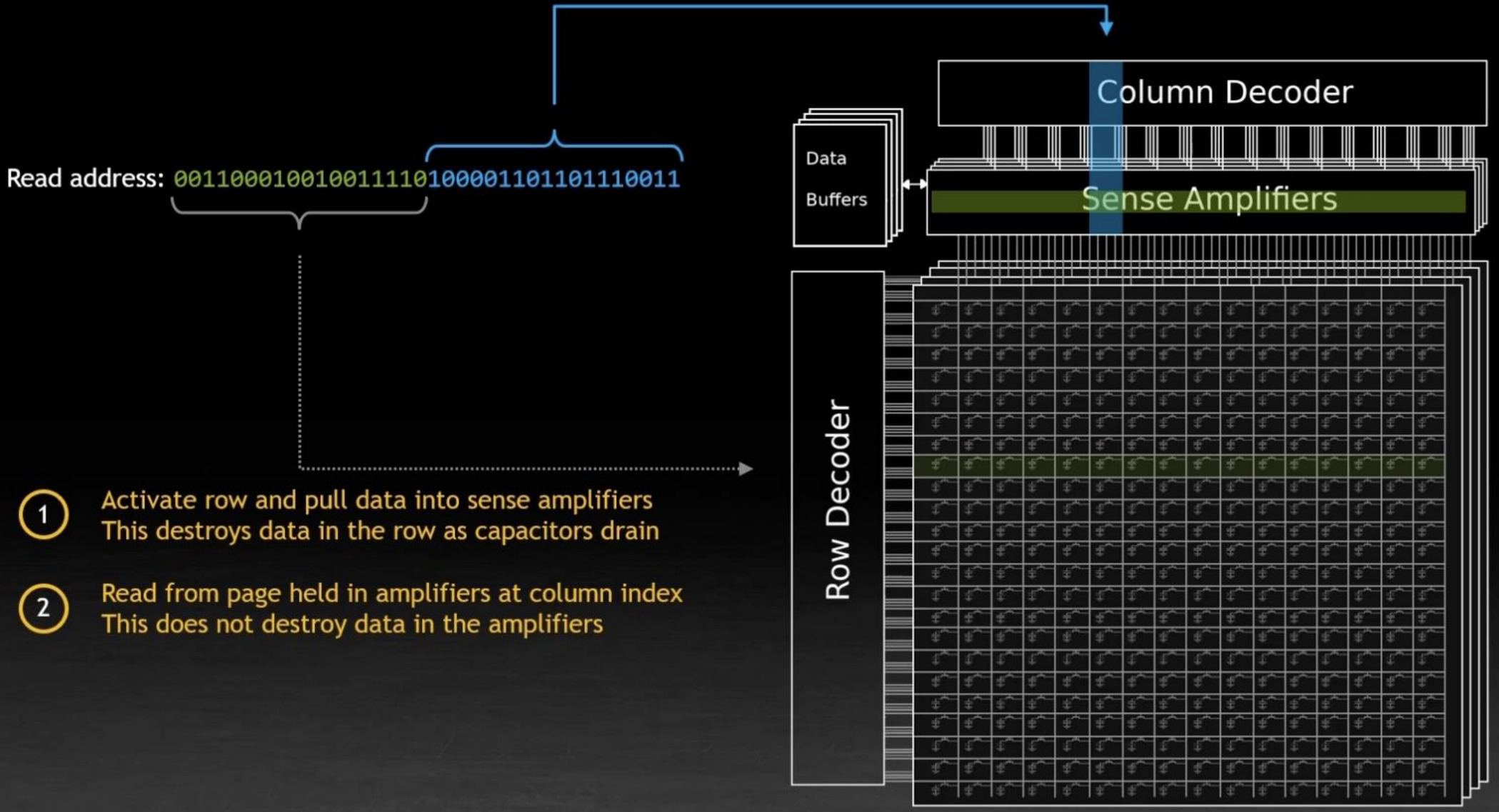
importantly: reading destroys the cell's stored information!
so the value must be "refreshed" (written back) before
the wordline is turned off.


Read address: 00110001001001111010000110110111001 1
Column Decoder
Data
Buffers
Sense Amplifiers
Row Decoder

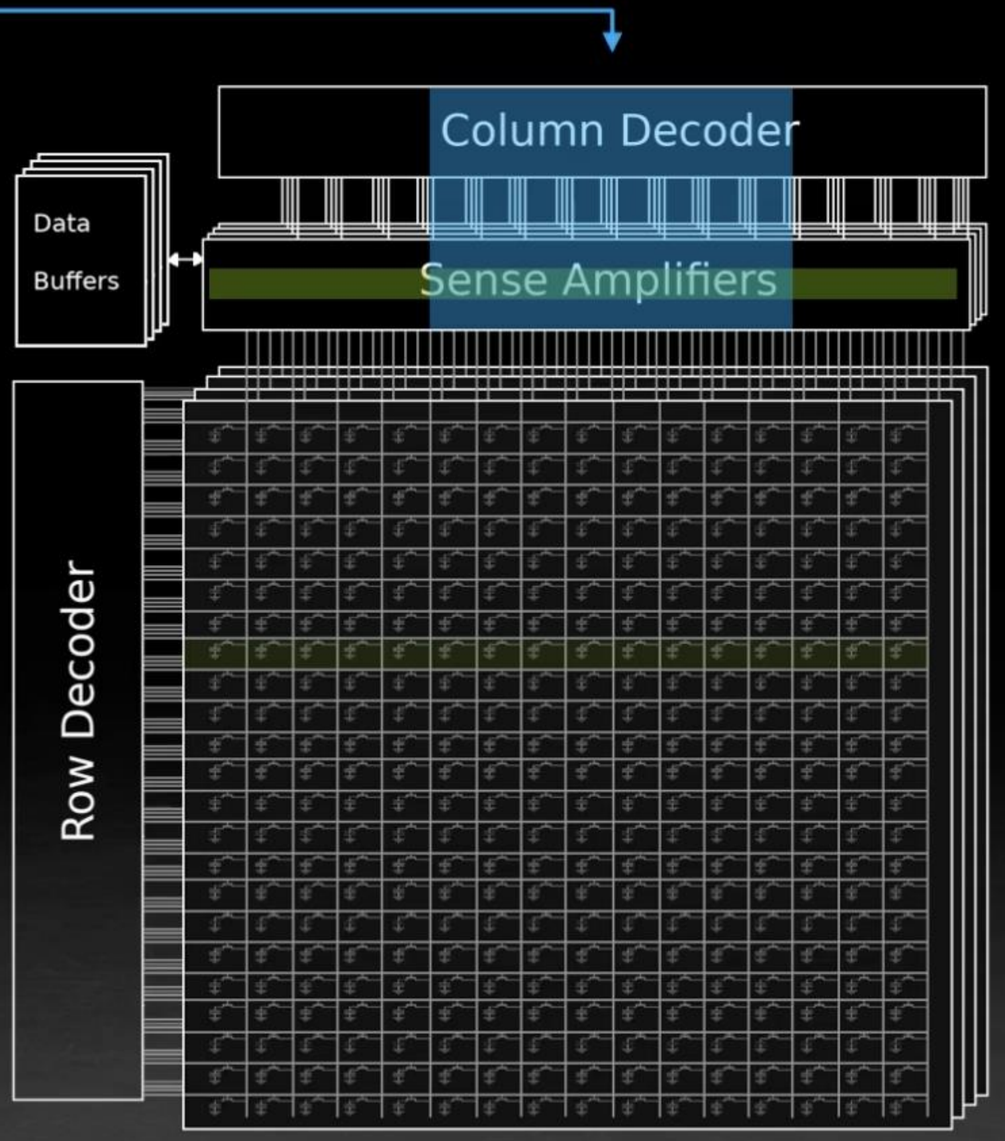
Read address: 00110001001001111010000110110111 0011



1 Activate row and pull data into sense amplifiers
This destroys data in the row as capacitors drain

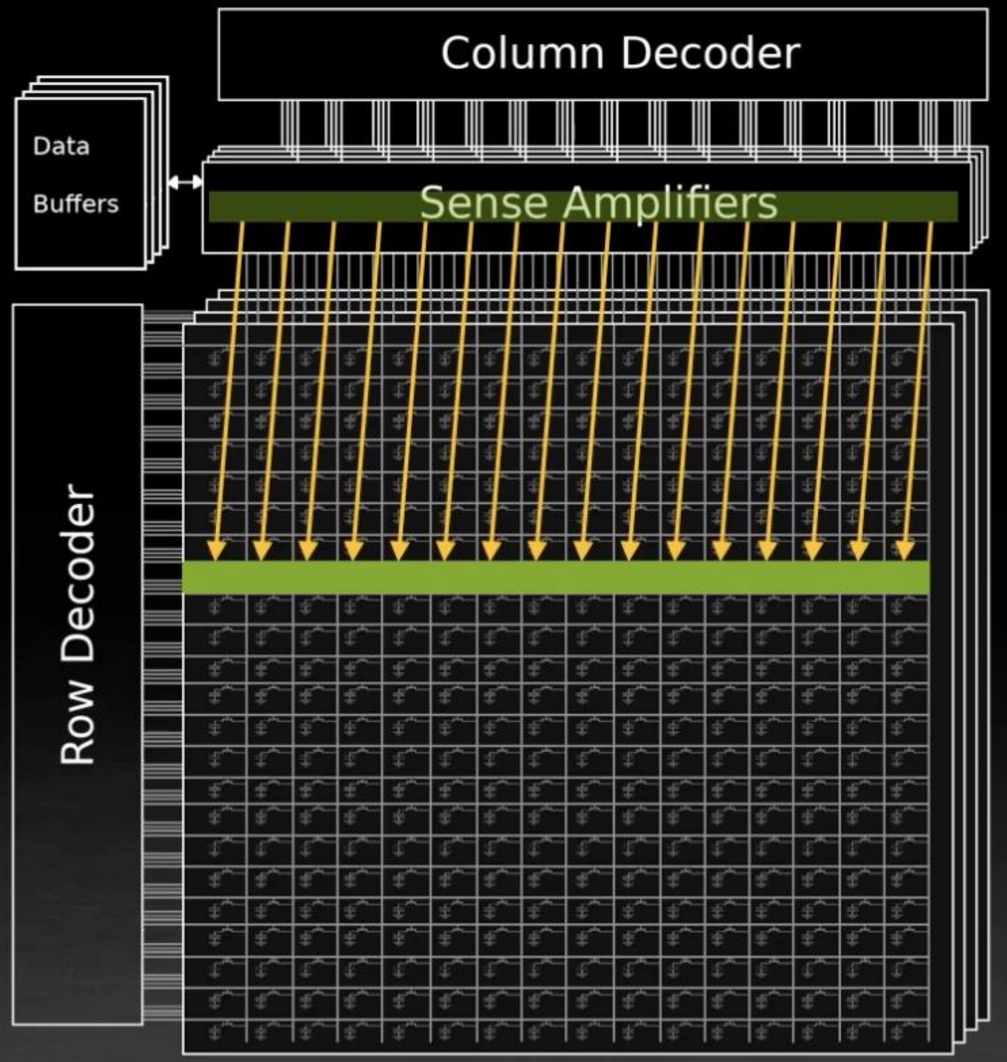

Read address: 00110001001001111010000110110111 0011
Column Decoder
Data
Buffers
Sense Amplifiers
Row Decoder
1
Activate row and pull data into sense amplifiers
This destroys data in the row as capacitors drain
2
Read from page held in amplifiers at column index
This does not destroy data in the amplifiers

Read address: 00110001001001111010000110110111010**0**



1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers
3. May make repeated reads from the same page
   “Burst” reads load multiple columns at a time

Read address: 00110001001001110110000110110110011



1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers
3. May make repeated reads from the same page
   “Burst” reads load multiple columns at a time
4. Before a new page is fetched, old row must be
   written back because data was destroyed

Example HBM values

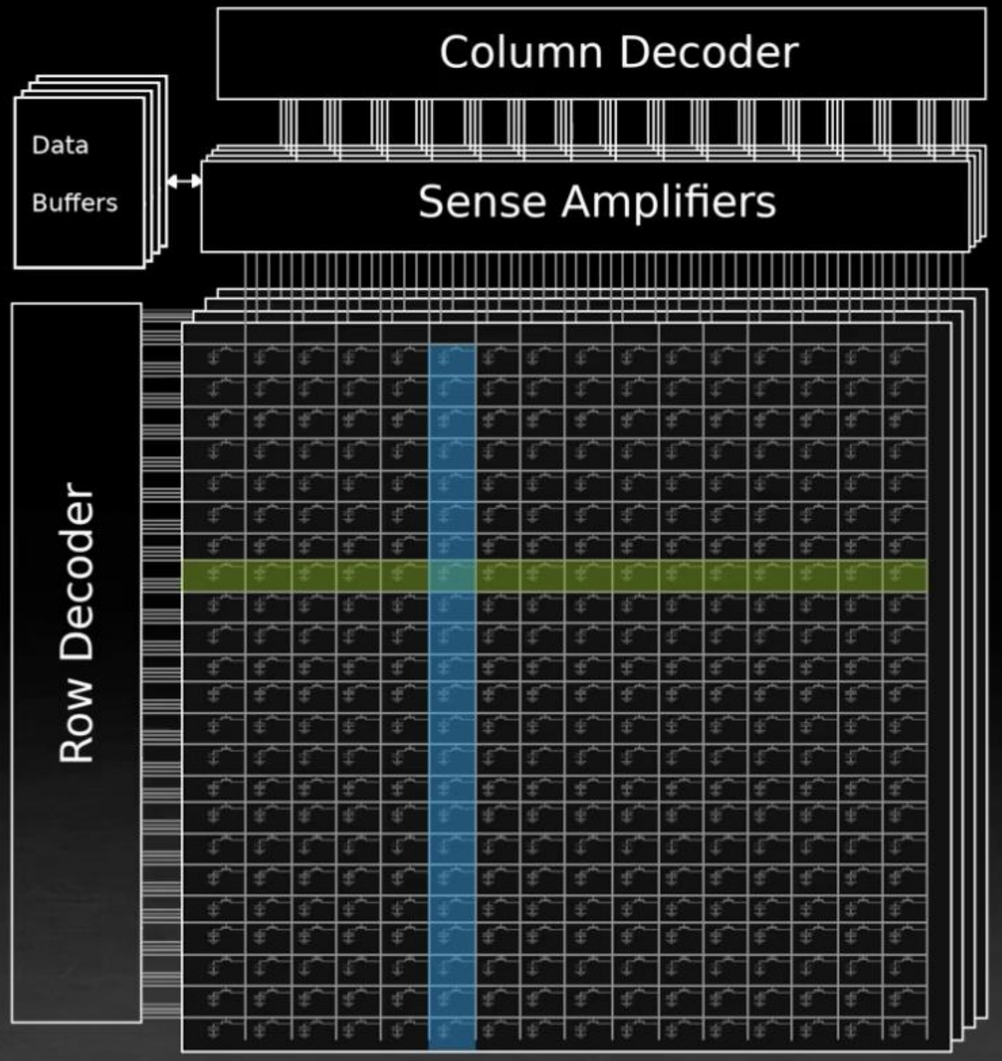
- Time to read new column: C_L = 16 cycles
- Time to load new page: T_{RCD} = 16 cycles
- Time to write back data: T_{RP} = 16 cycles
- Page (row) size = 1kB

Each read has a cost (C_L = 16 cycles)

Switching page has 3x larger cost ($T_{RP} + T_{RCD} + C_L$ = 48 cycles)

This is because switching page requires charging/discharging capacitors with a physical RC time constant:

$$V_C = V_S(1 - e^{(-t/RC)})$$



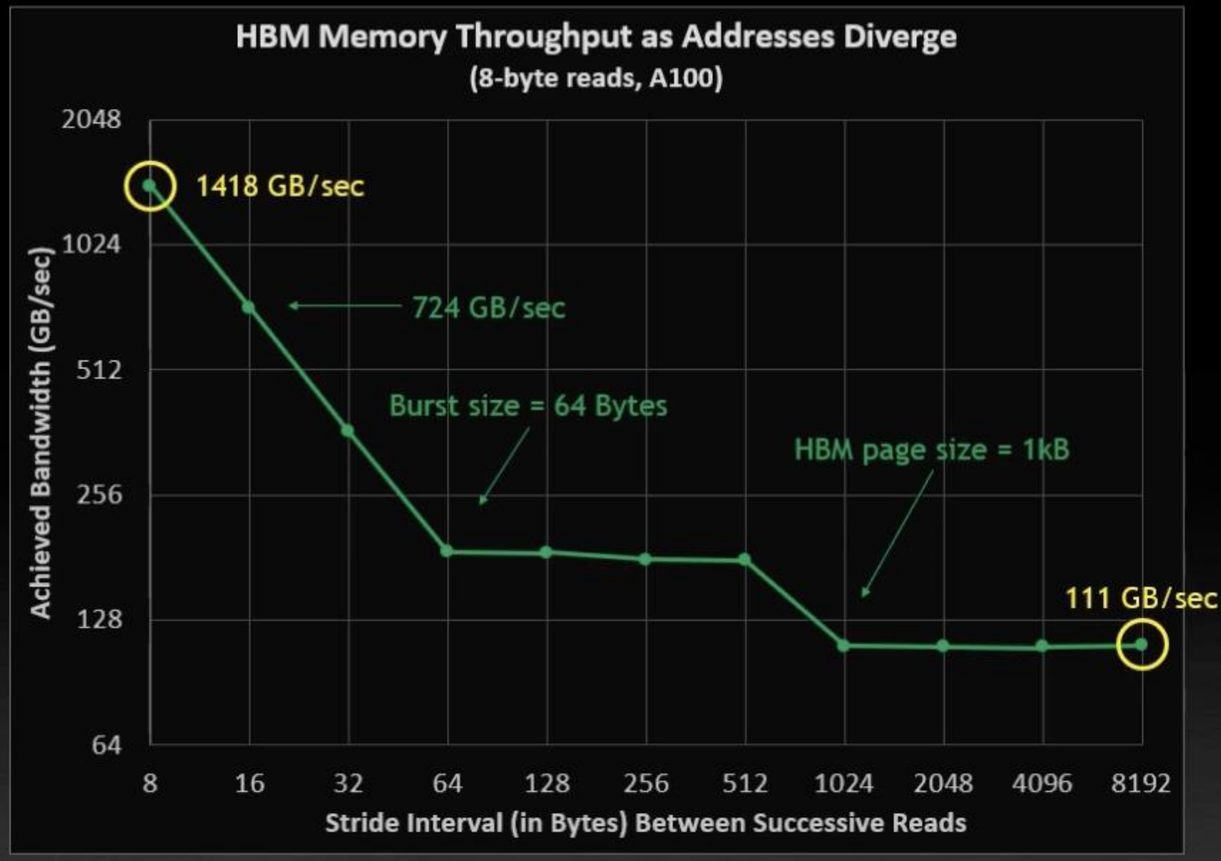
SO WHAT DOES THIS ALL MEAN?

We'd expect a significant performance difference for coalesced vs. scattered reads

On A100, memory bandwidth for widely-spaced reads is

$$\frac{111}{1418} = 8\% \text{ of peak bandwidth}$$

That's 1/13th of peak bandwidth!



HBM: The devil is in the details

- Bandwidth drops when **stride > packet size** (no coalescing)
- Drops further when **stride > row size** (no row locality)
- Peak bandwidth requires **parallel utilization across channels + banks + stacks**

HBM Memory Hierarchy

Stage	Hardware object	Typical size	What happens
① Row / Page	Data in one DRAM bank row (row buffer)	**~1 KB – 2 KB**	Activated into sense amplifiers via ACTIVATE (expensive)
② Column packet	Data returned per READ command (per channel)	**32 B**	Burst fetched from open row (cheap if row open)
③ I/O bus (channel)	External HBM channel interface	**64 bits = 8 B per beat**	Packet serialized over ~4 beats
④ Stack interface	16 channels per stack	**1024 bits total**	Parallel packet transfers across channels
⑤ System level	Multiple HBM stacks	**3–5 TB/s aggregate**	Massive parallel bandwidth across stacks

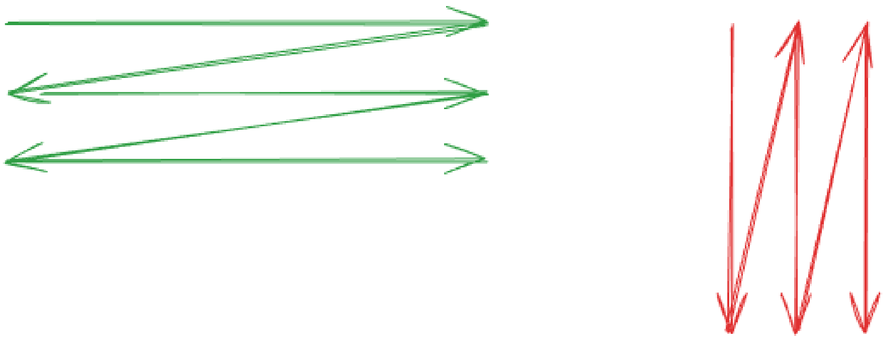
M=N=256

(insert which way western man meme here :P)

```
for (row=0; row < M; row++) {
    for (col=0; col < N; col++) {
        load(array[row][col]);
    }
}
```

```
for (col=0; col < N; col++) {
    for (row=0; row < M; row++) {
        load(array[row][col]);
    }
}
```

note: array[row][col] gets translated to
array + 4*col + 4*N*row pointer arithmetic

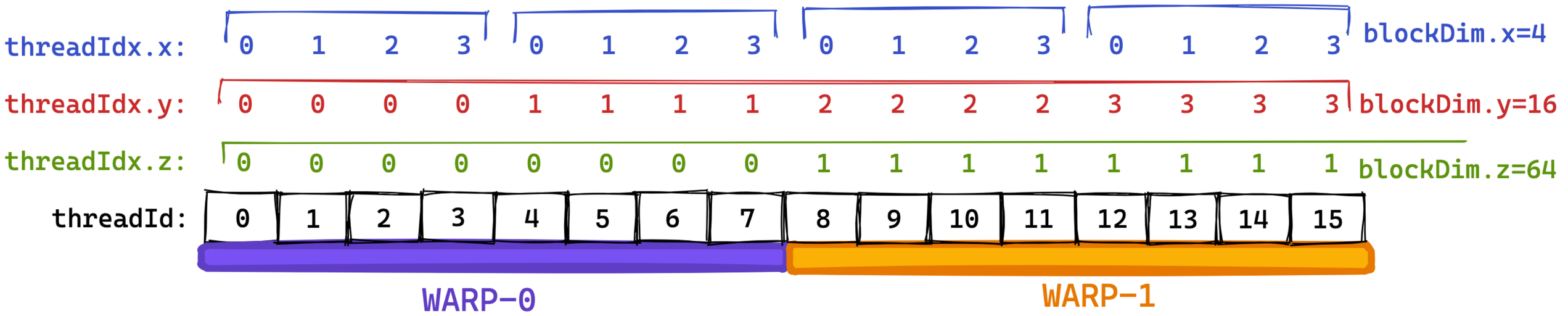


given what we just learned it's obvious that there is one right answer:
traversing columns should be in the inner/fast for loop as that would lead to a single row read
followed by 256 column reads for one iteration of the inner for loop (256 row reads in total).
the alternative would be 256 row reads for a single iteration of the inner for loop (256x256 row reads in total).

Thread Organization

Wrap organization and its impact on memory access

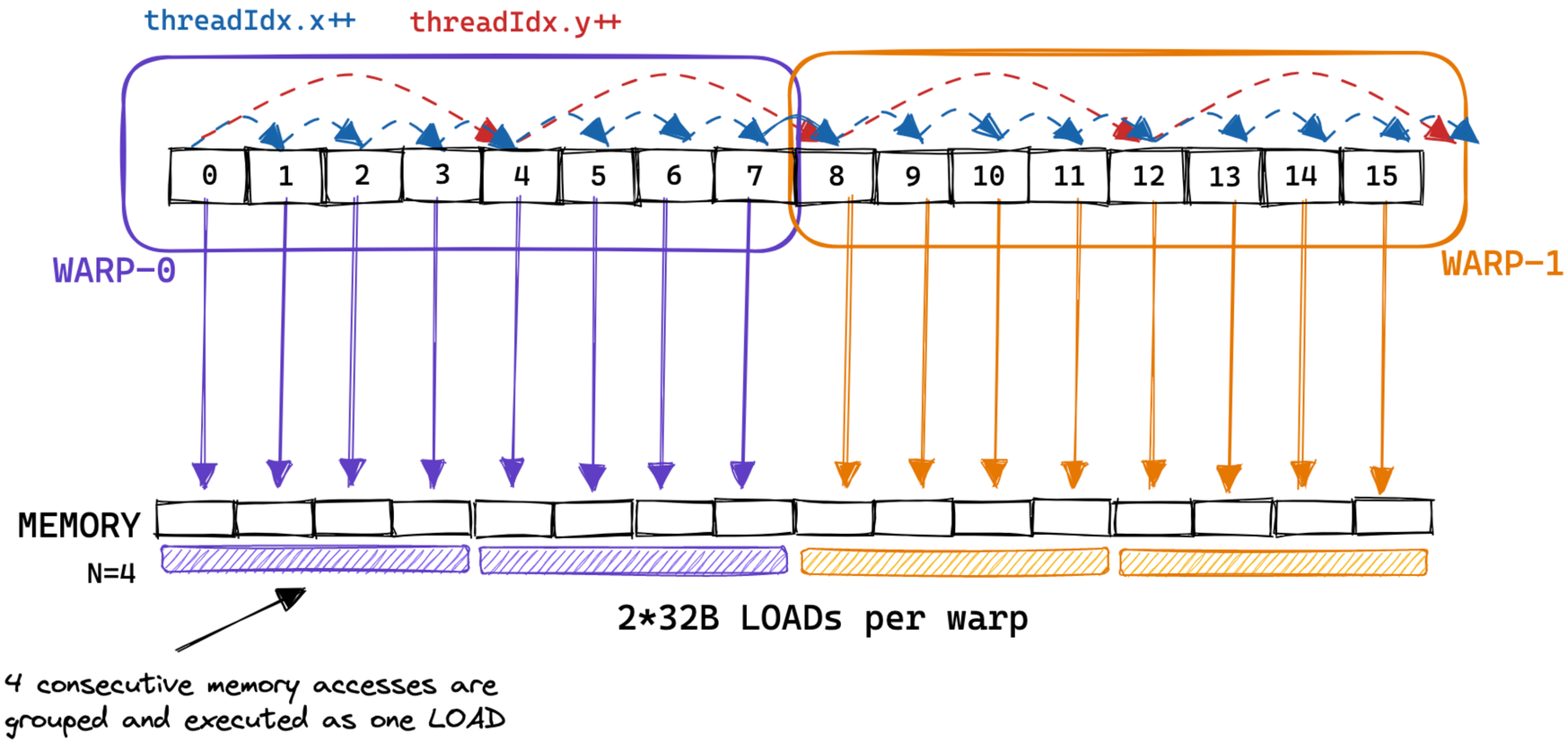
```
threadId = threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*threadIdx.z)
```




```
threadId = threadIdx.x + blockDim.x*threadIdx.y + blockDim.x* blockDim.y*threadIdx.z
```

Thread Organization

This is what we want



Thread Organization

This is what we wrote

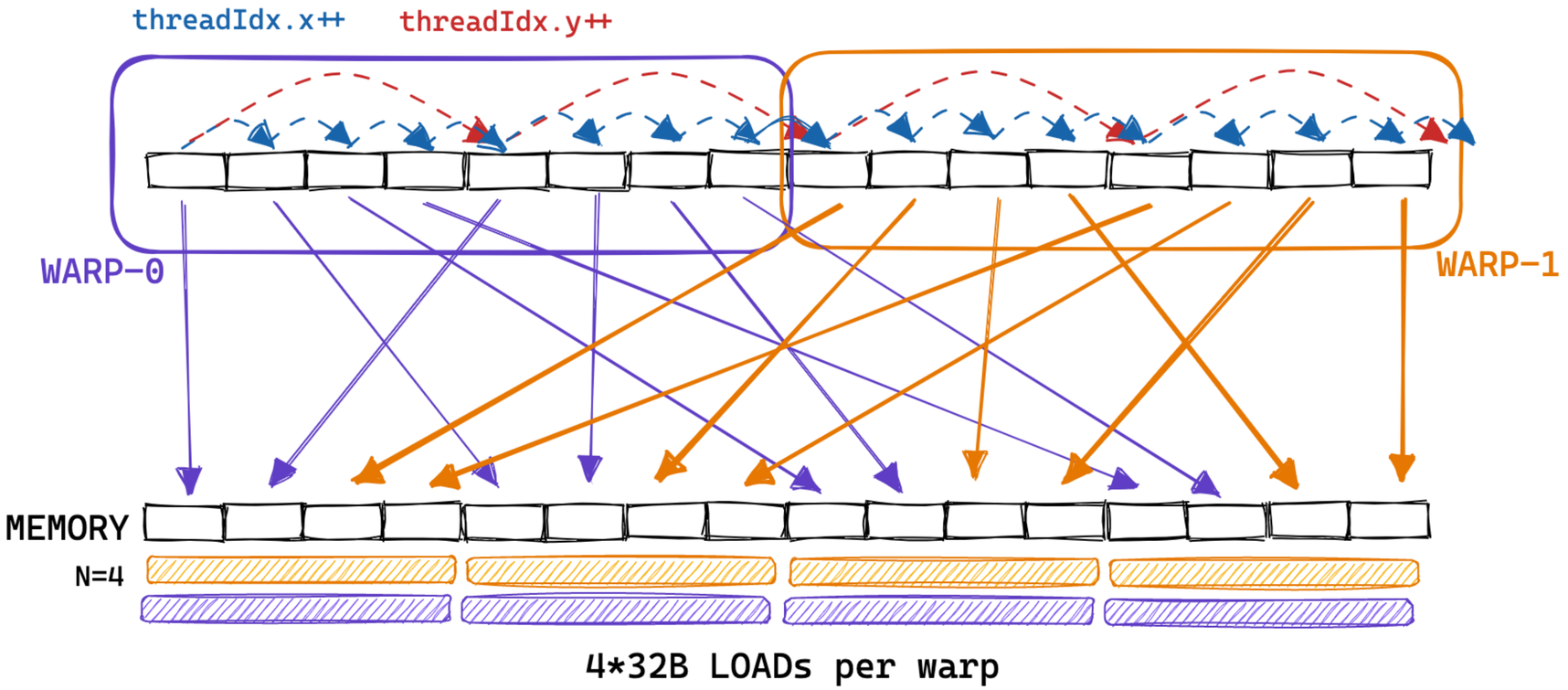
```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // if statement is necessary to make things work under tile quantization
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

Thread Organization

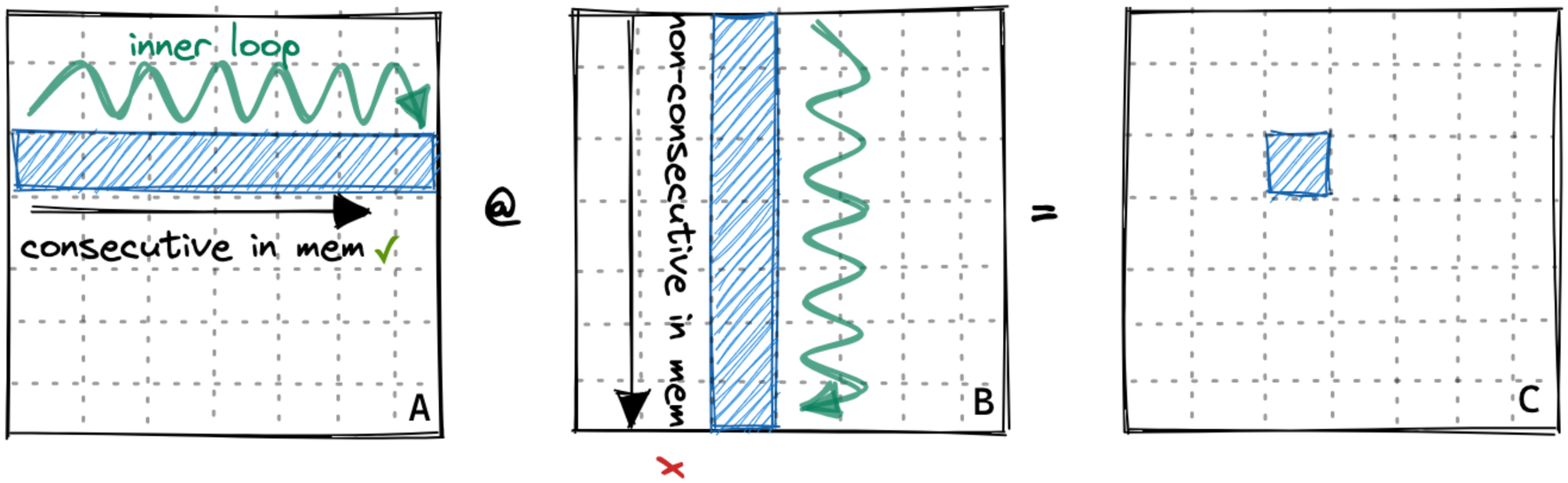
This is what we get

```
const uint x = blockIdx.x * blockDim.x + threadIdx.x;
const uint y = blockIdx.y * blockDim.y + threadIdx.y;
```



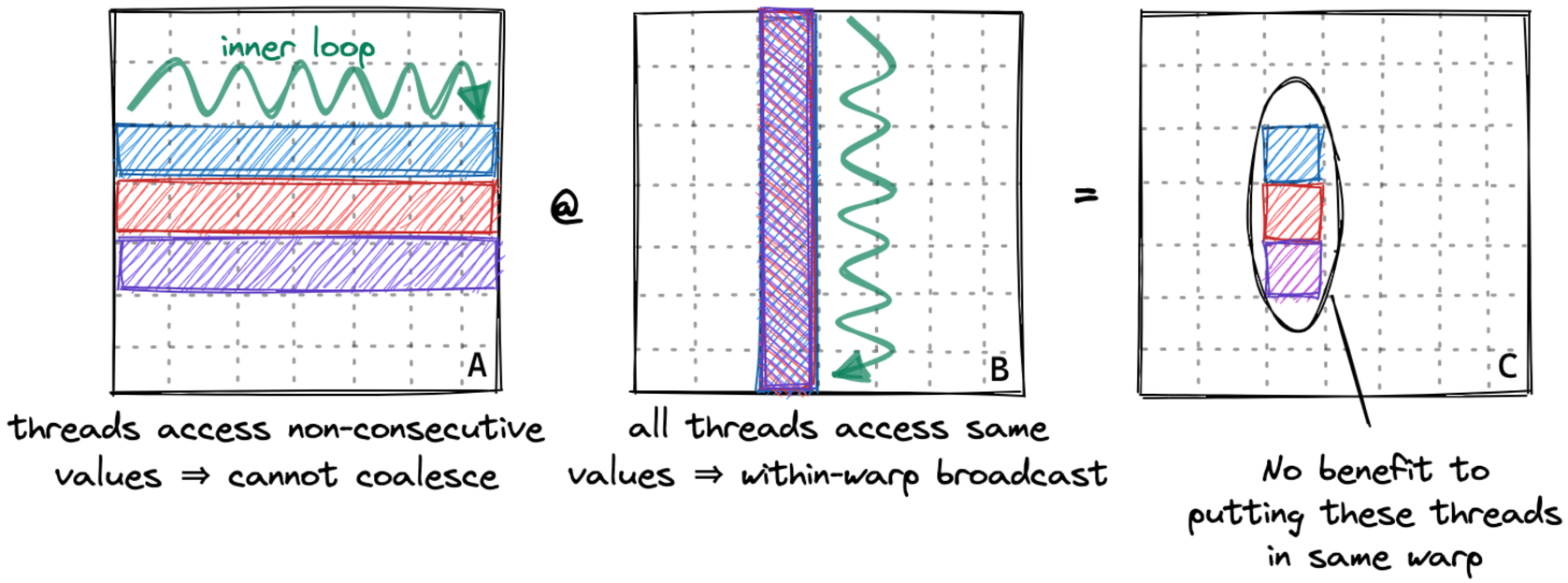
Thread Organization

A single thread



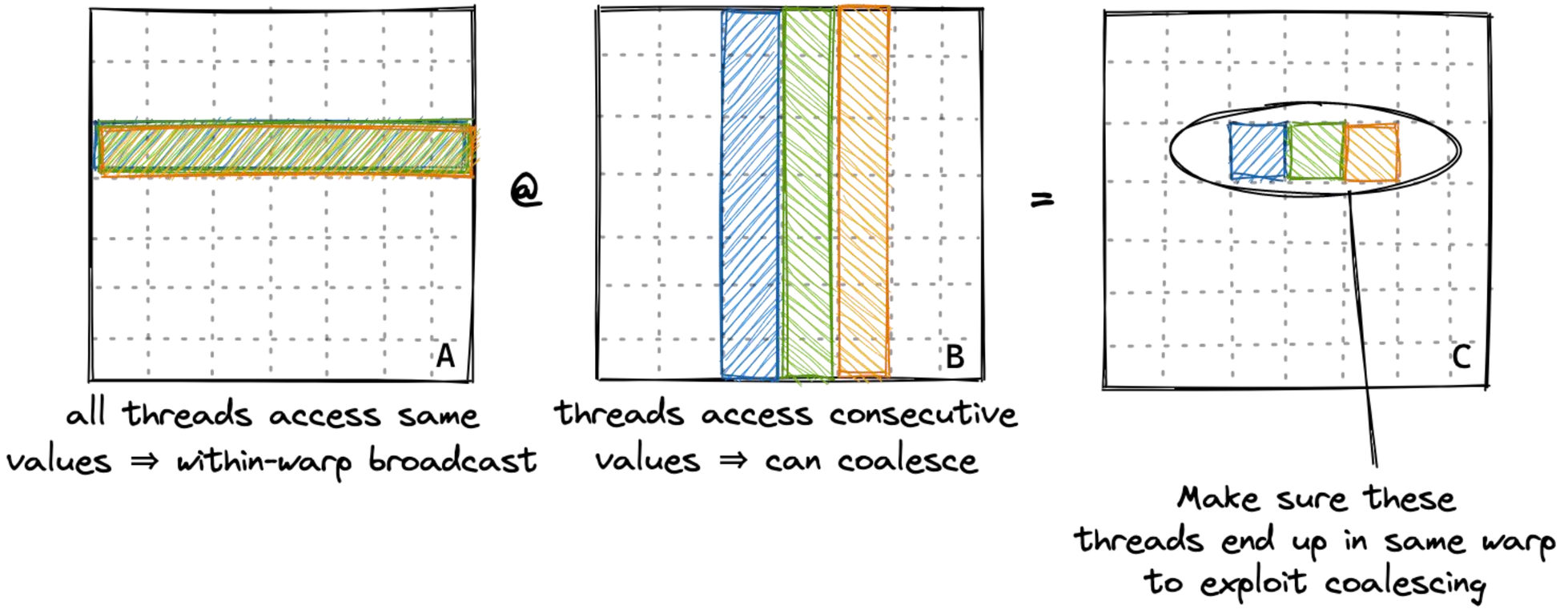
Thread Organization

A wrap of thread (Why is this bad?)



Thread Organization

A wrap of thread (what we want)



Thread Organization

```
template <const uint BLOCKSIZE>
__global__ void sgemm_global_mem_coalesce(int M, int N, int K, float alpha,
                                          const float *A, const float *B,
                                          float beta, float *C) {
  const int cRow = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
  const int cCol = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

  // if statement is necessary to make things work under tile quantization
  if (cRow < M && cCol < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[cRow * K + i] * B[i * N + cCol];
    }
    C[cRow * N + cCol] = alpha * tmp + beta * C[cRow * N + cCol];
  }
}
```

Matrix Multiplication

Welcome to the real world!

Memory coalesce improves, so does the arithmetic intensity.

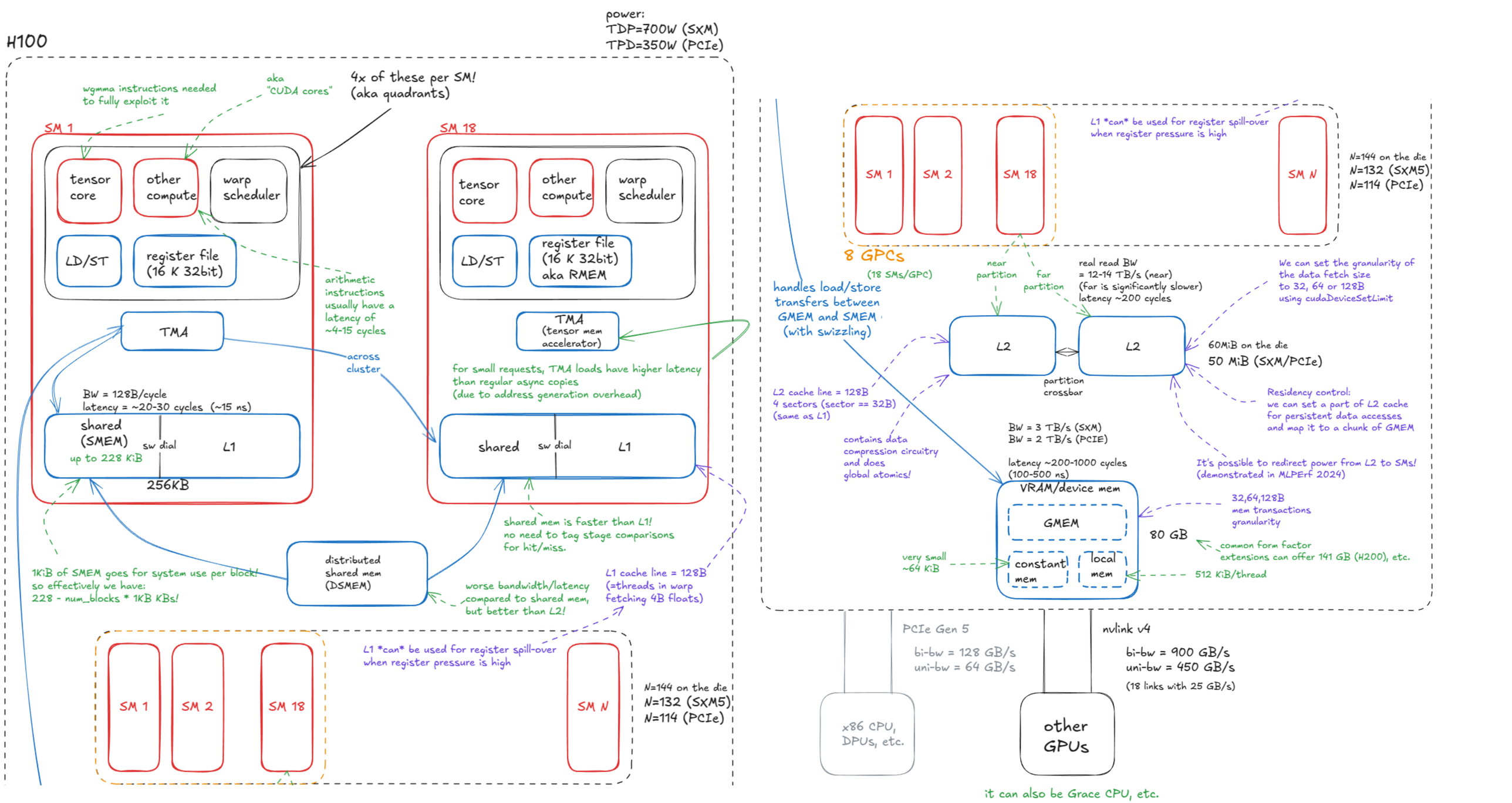
```
Max size: 4096
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000008) s, performance: (  513.5) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000014) s, performance: ( 2401.4) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 3
Average elapsed time: (0.000046) s, performance: ( 5777.9) GFLOPS. size: (512).
dimensions(m=n=k) 1024, alpha: 0.5, beta: 3
Average elapsed time: (0.000341) s, performance: ( 6301.7) GFLOPS. size: (1024).
dimensions(m=n=k) 2048, alpha: 0.5, beta: 3
Average elapsed time: (0.002690) s, performance: ( 6386.5) GFLOPS. size: (2048).
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.021614) s, performance: ( 6358.7) GFLOPS. size: (4096).
```

Lesson 1: Data stored in global memory (GMEM) should to be accessed contiguously for maximum bandwidth efficiency.

–Tom Jerry

Where is the bottleneck now?

Data fetched by threads are not shared.

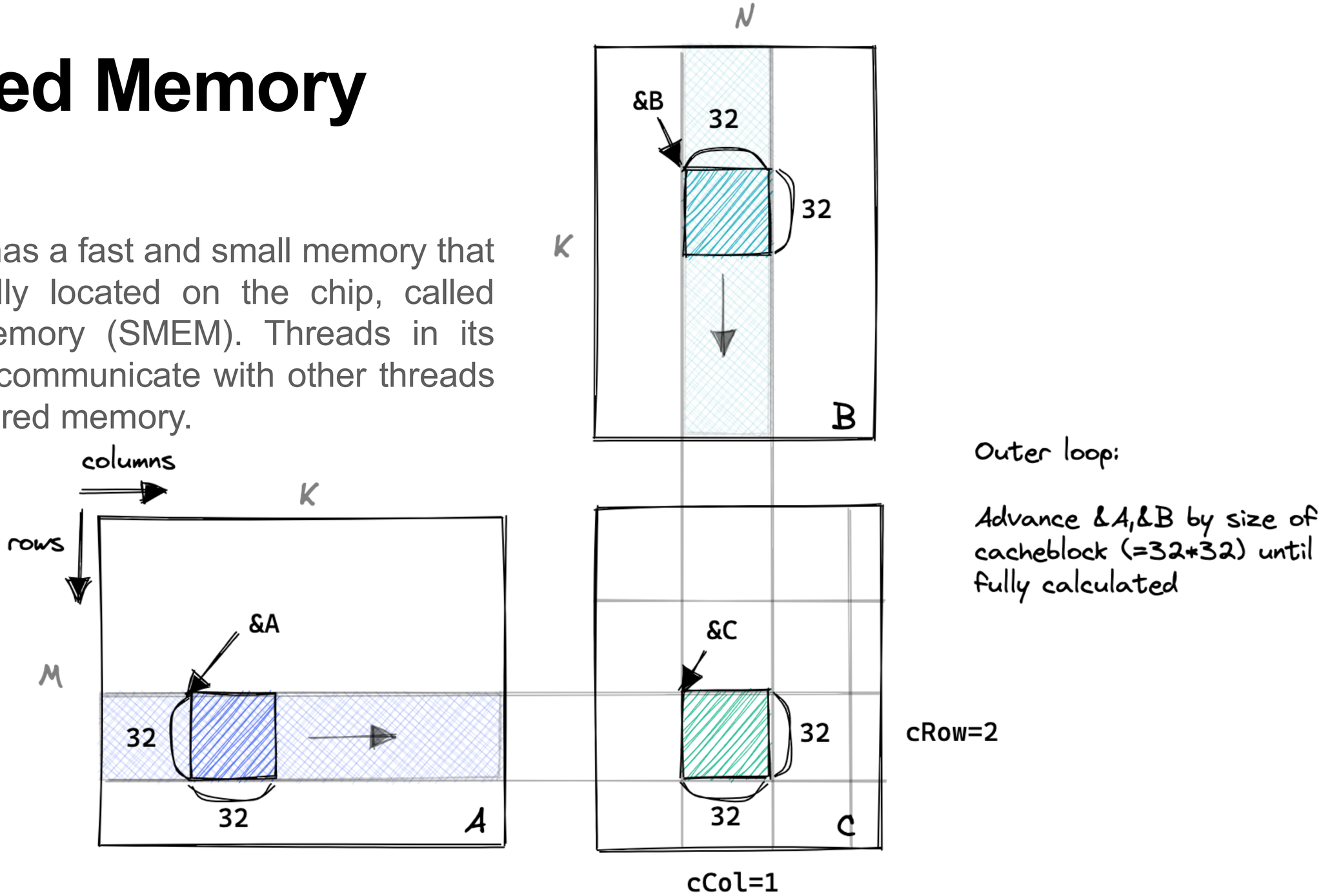

H100
power:
TDP=700W (SXM)
TPD=350W (PCIe)
4x of these per SM!
(aka quadrants)
wgmma instructions needed
to fully exploit it
aka
"CUDA cores"
SM 1
tensor core
other compute
warp scheduler
LD/ST
register file
(16 K 32bit)
arithmetic
instructions
usually have a
latency of
~4-15 cycles
TMA
across
cluster
BW = 128B/cycle
latency = ~20-30 cycles (~15 ns)
shared
(SMEM)
sw dial
L1
up to 228 KiB
256KB
SM 18
tensor core
other compute
warp scheduler
LD/ST
register file
(16 K 32bit)
aka RMEM
TMA
(tensor mem
accelerator)
for small requests, TMA loads have higher latency
than regular async copies
(due to address generation overhead)
shared
sw dial
L1
shared mem is faster than L1!
no need to tag stage comparisons
for hit/miss.
1KiB of SMEM goes for system use per block!
so effectively we have:
228 - num_blocks * 1KB KBs!
distributed
shared mem
(DSMEM)
worse bandwidth/latency
compared to shared mem,
but better than L2!
L1 cache line = 128B
(=threads in warp
fetching 4B floats)
L1 *can* be used for register spill-over
when register pressure is high
SM 1
SM 2
SM 18
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
SM 1
SM 2
SM 18
SM N
L1 *can* be used for register spill-over
when register pressure is high
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
8 GPCs
(18 SMs/GPC)
handles load/store
transfers between
GMEM and SMEM
(with swizzling)
near
partition
far
partition
real read BW
= 12-14 TB/s (near)
(far is significantly slower)
latency ~200 cycles
We can set the granularity of
the data fetch size
to 32, 64 or 128B
using cudaDeviceSetLimit
L2
L2
partition
crossbar
60MiB on the die
50 MiB (SXM/PCIe)
L2 cache line = 128B
4 sectors (sector == 32B)
(same as L1)
Residency control:
we can set a part of L2 cache
for persistent data accesses
and map it to a chunk of GMEM
contains data
compression circuitry
and does
global atomics!
BW = 3 TB/s (SXM)
BW = 2 TB/s (PCIE)
latency ~200-1000 cycles
(100-500 ns)
It's possible to redirect power from L2 to SMs!
(demonstrated in MLPErf 2024)
VRAM/device mem
GMEM
32,64,128B
mem transactions
granularity
80 GB
common form factor
extensions can offer 141 GB (H200), etc.
very small
~64 KiB
constant
mem
local
mem
512 KiB/thread
PCIe Gen 5
bi-bw = 128 GB/s
uni-bw = 64 GB/s
nvlink v4
bi-bw = 900 GB/s
uni-bw = 450 GB/s
(18 links with 25 GB/s)
x86 CPU,
DPUs, etc.
other
GPUs
it can also be Grace CPU, etc.

Shared Memory

Each SM has a fast and small memory that is physically located on the chip, called shared memory (SMEM). Threads in its block can communicate with other threads via the shared memory.

Shared Memory

Each SM has a fast and small memory that is physically located on the chip, called shared memory (SMEM). Threads in its block can communicate with other threads via the shared memory.



Outer loop:

Advance &A,&B by size of cacheblock (=32*32) until C is fully calculated

Shared Memory

Step by step

1. Determine the grid-block-thread arrangement
2. Determine the corresponding block-Matrix C matching
3. Determine the corresponding Matrix A rows and Matrix B columns
4. Determine the tiling arrangement in Matrix A rows and Matrix B columns
5. Copy data from global memory to shared memory
6. Compute the tile-wise partial results of Matrix C
7. Compute the final results of Matrix C

Shared Memory

Step by step

1. **Determine the grid-block-thread arrangement**

```
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
dim3 blockDim(32 * 32);
```

2. **Determine the corresponding block-Matrix C matching**

```
A += cRow * BLOCKSIZE * K;
B += cCol * BLOCKSIZE;
C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE;
```

3. **Determine the corresponding Matrix A rows and Matrix B columns**

4. **Determine the tiling arrangement in Matrix A rows and Matrix B columns**

```
const uint threadCol = threadId.x % BLOCKSIZE;
const uint threadRow = threadId.x / BLOCKSIZE;
```

Shared Memory

Step by step

5. Copy data from global memory to shared memory
6. Compute the tile-wise partial results of Matrix C
7. Compute the final results of Matrix C

```
float tmp = 0;
for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];

    __syncthreads();
    A += BLOCKSIZE;
    B += BLOCKSIZE * N;

    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
        tmp += As[ThreadRow * BLOCKSIZE + dotIdx] * Bs[dotIdx * BLOCKSIZE + threadCol];
    }

    __syncthreads();
}
C[threadRow * N + threadCol] = alpha * tmp + beta * CpthreadRow * N + threadCol];
```

Matrix Multiplication

Welcome to the real world!

Arithmetic intensity improves.

```
Max size: 4096
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000007) s, performance: (  631.4) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000011) s, performance: ( 3155.5) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 3
Average elapsed time: (0.000033) s, performance: ( 8147.6) GFLOPS. size: (512).
dimensions(m=n=k) 1024, alpha: 0.5, beta: 3
Average elapsed time: (0.000234) s, performance: ( 9173.7) GFLOPS. size: (1024).
dimensions(m=n=k) 2048, alpha: 0.5, beta: 3
Average elapsed time: (0.001837) s, performance: ( 9353.8) GFLOPS. size: (2048).
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.014954) s, performance: ( 9190.8) GFLOPS. size: (4096).
```

Lesson 2: **Tiling facilitates intra-block data reuse among threads by exploiting the shared memory of each streaming multiprocessor (SM).**

–Tom Jerry

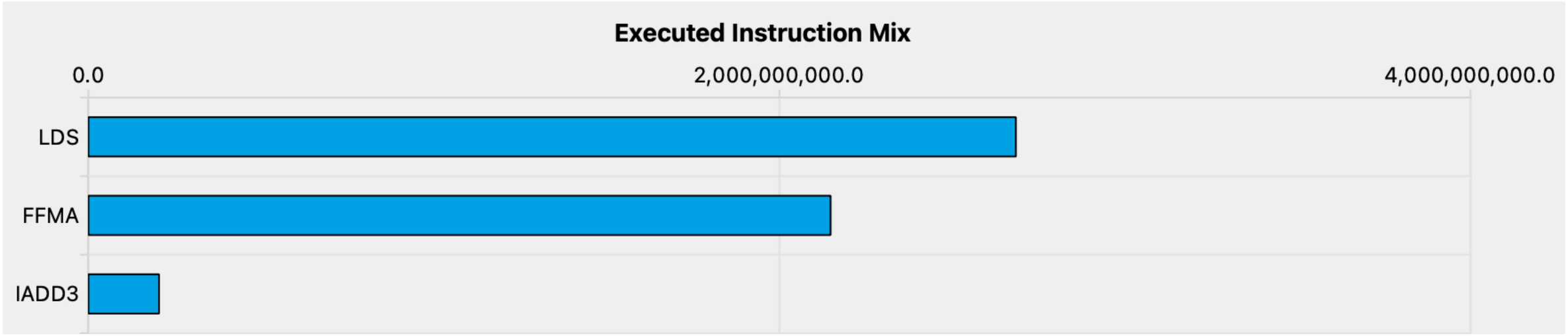
Where is the bottleneck now?

–Tom Jerry

Lots of memory accesses

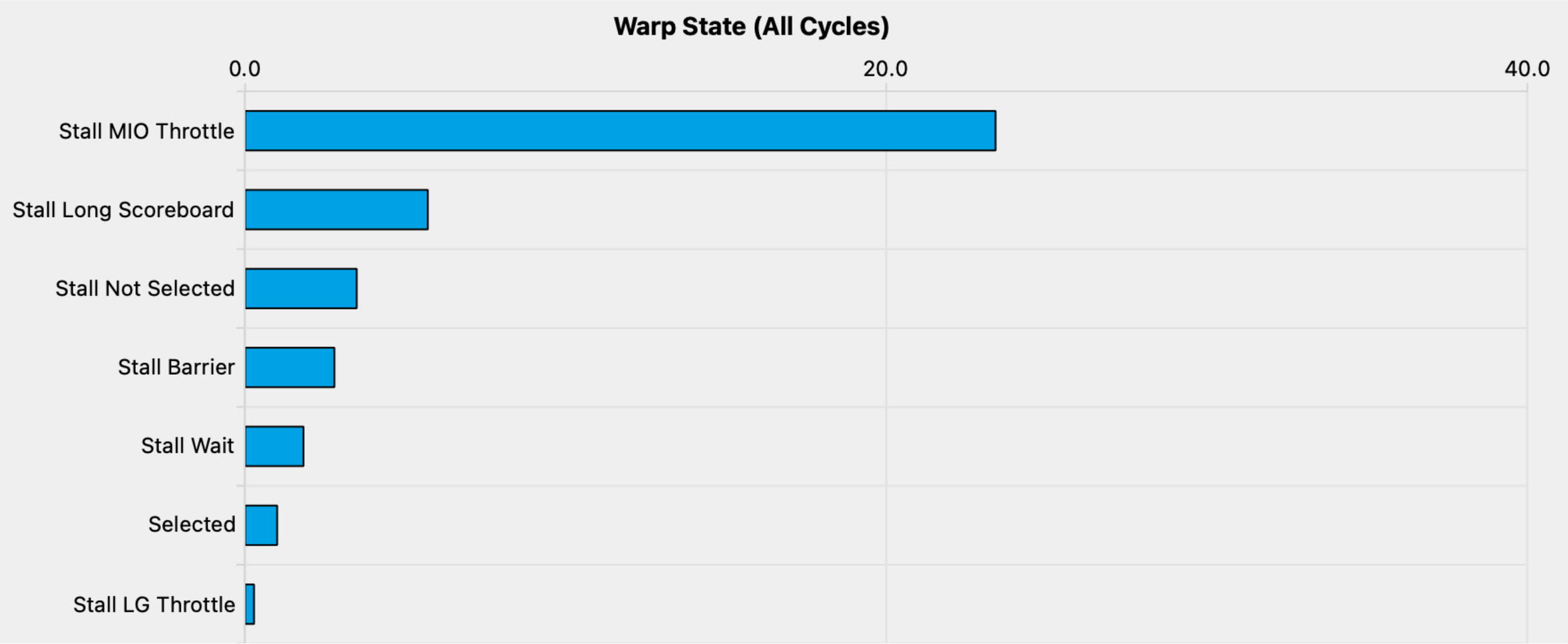
Still memory bound

```
ld.shared.f32   %f91, [%r8+3456];
ld.shared.f32   %f92, [%r7+108];
fma.rn.f32      %f93, %f92, %f91, %f90;
```



Lots of memory accesses

Still memory bound



Lots of memory accesses

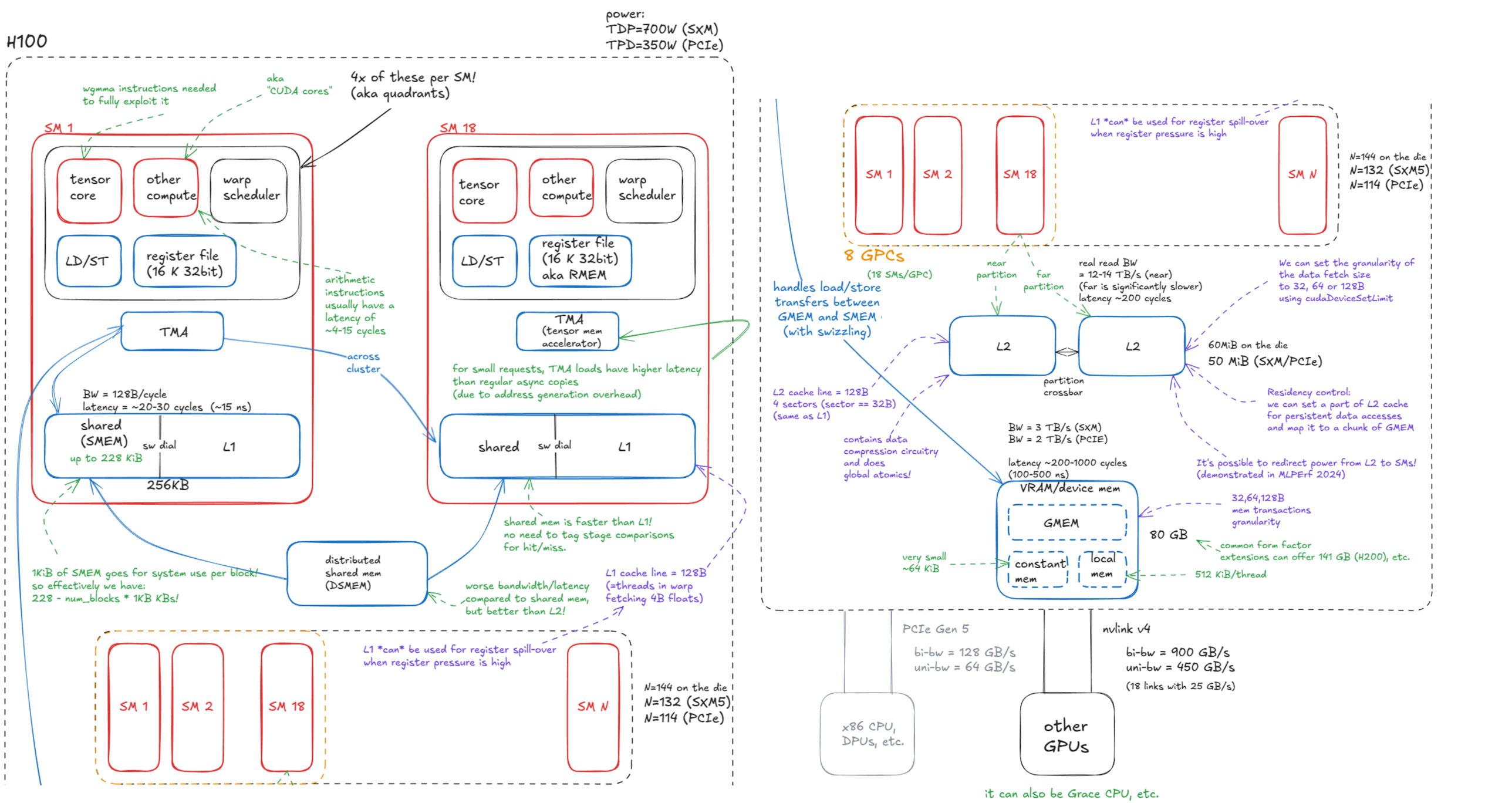
Still memory bound

The warp was stalled because the MIO (memory input/output) instruction queue was full. This stall reason occurs under heavy utilization of the MIO pipelines as threads in the same block issue lots of shared memory (SMEM) instructions.

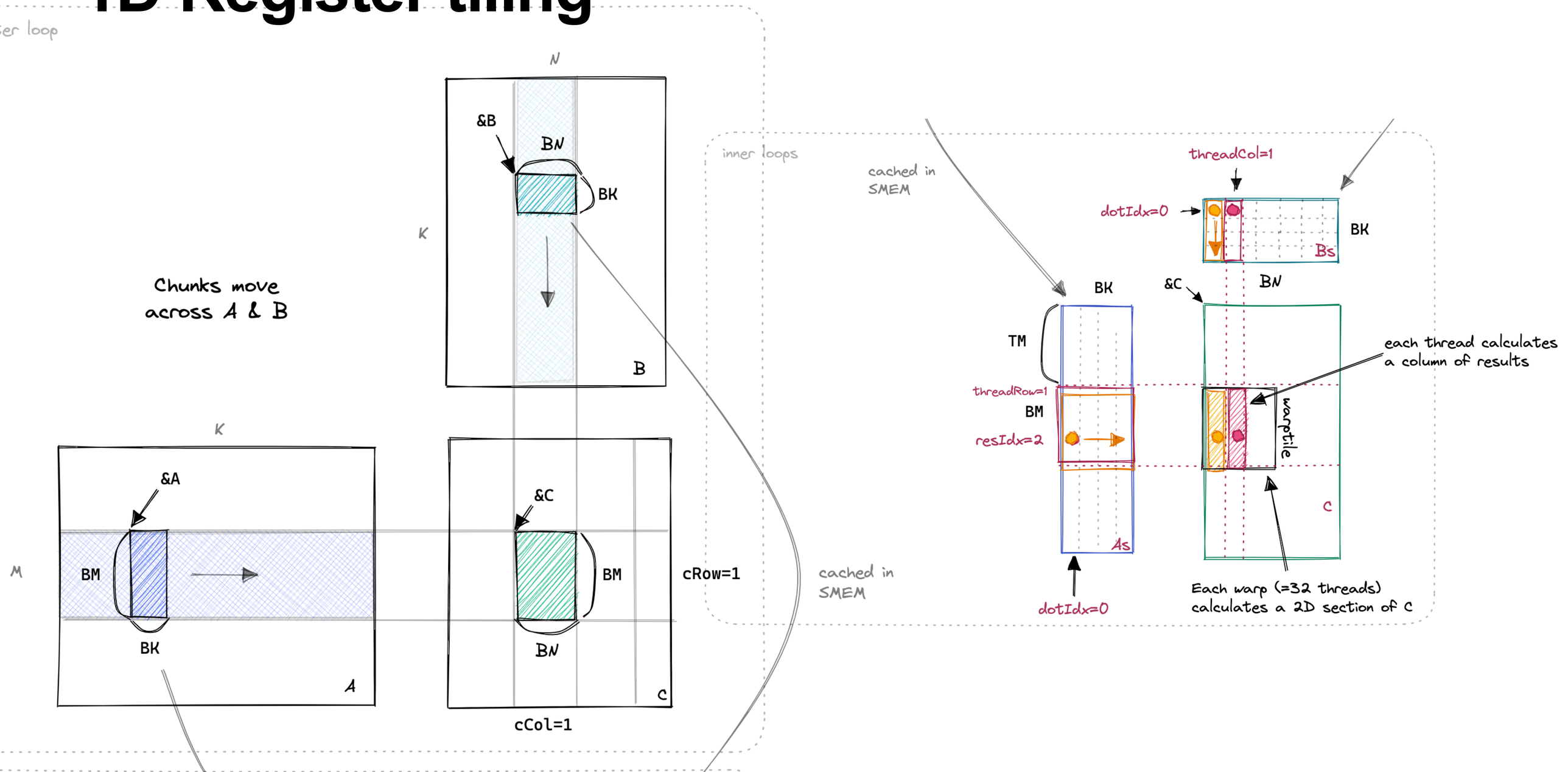
Lots of memory accesses

Still memory bound

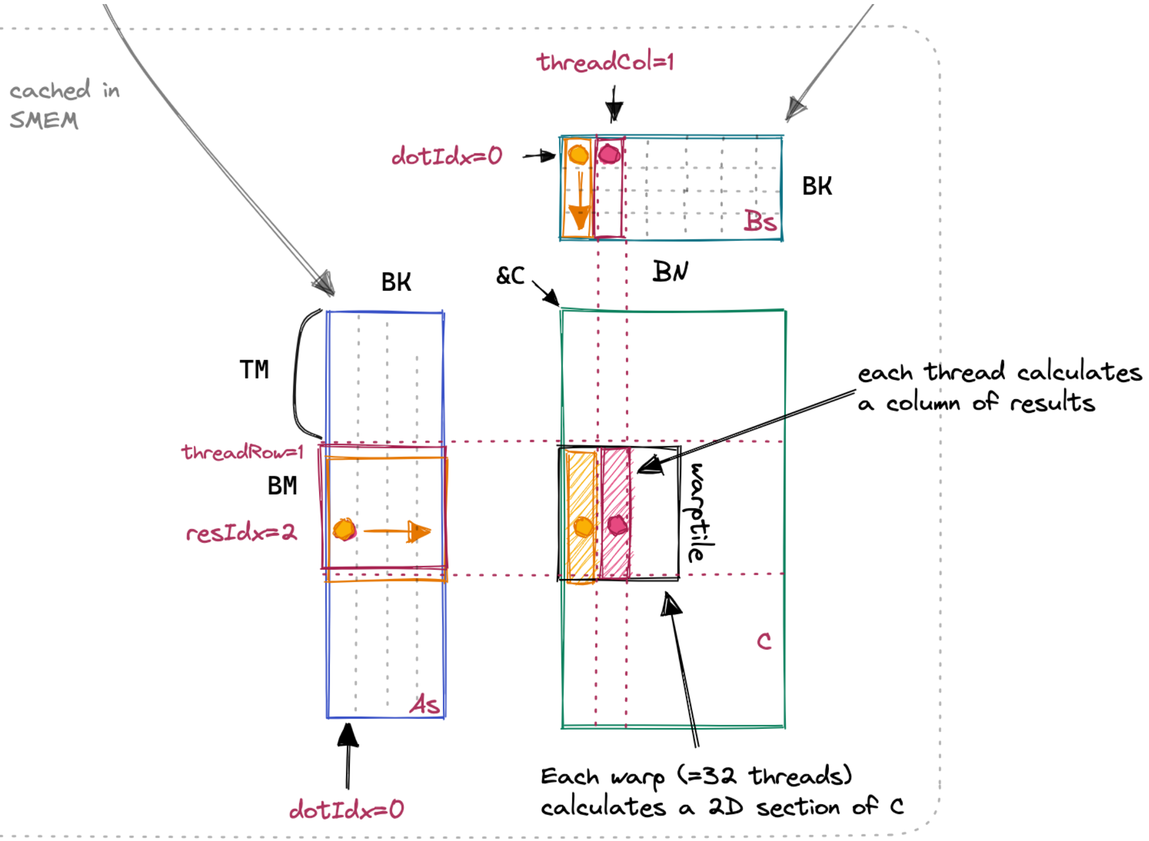
To reduce the number of SMEM instructions issued, we can increase the amount of work performed per thread—for example, by having each thread compute multiple output elements. This allows more computation to be done in registers and reduces reliance on shared memory.


H100
power:
TDP=700W (SXM)
TPD=350W (PCIe)
4x of these per SM!
(aka quadrants)
wgmma instructions needed
to fully exploit it
aka
"CUDA cores"
SM 1
tensor core
other compute
warp scheduler
LD/ST
register file
(16 K 32bit)
arithmetic
instructions
usually have a
latency of
~4-15 cycles
TMA
across
cluster
BW = 128B/cycle
latency = ~20-30 cycles (~15 ns)
shared
(SMEM)
sw dial
L1
up to 228 KiB
256KB
SM 18
tensor core
other compute
warp scheduler
LD/ST
register file
(16 K 32bit)
aka RMEM
TMA
(tensor mem
accelerator)
for small requests, TMA loads have higher latency
than regular async copies
(due to address generation overhead)
shared
sw dial
L1
shared mem is faster than L1!
no need to tag stage comparisons
for hit/miss.
1KiB of SMEM goes for system use per block!
so effectively we have:
228 - num_blocks * 1KB KBs!
distributed
shared mem
(DSMEM)
worse bandwidth/latency
compared to shared mem,
but better than L2!
L1 cache line = 128B
(=threads in warp
fetching 4B floats)
L1 *can* be used for register spill-over
when register pressure is high
SM 1
SM 2
SM 18
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
SM 1
SM 2
SM 18
SM N
L1 *can* be used for register spill-over
when register pressure is high
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
8 GPCs
(18 SMs/GPC)
handles load/store
transfers between
GMEM and SMEM
(with swizzling)
near
partition
far
partition
real read BW
= 12-14 TB/s (near)
(far is significantly slower)
latency ~200 cycles
We can set the granularity of
the data fetch size
to 32, 64 or 128B
using cudaDeviceSetLimit
L2
L2
partition
crossbar
60MiB on the die
50 MiB (SXM/PCIe)
L2 cache line = 128B
4 sectors (sector == 32B)
(same as L1)
Residency control:
we can set a part of L2 cache
for persistent data accesses
and map it to a chunk of GMEM
contains data
compression circuitry
and does
global atomics!
BW = 3 TB/s (SXM)
BW = 2 TB/s (PCIE)
latency ~200-1000 cycles
(100-500 ns)
It's possible to redirect power from L2 to SMs!
(demonstrated in MLPErf 2024)
VRAM/device mem
GMEM
32,64,128B
mem transactions
granularity
80 GB
common form factor
extensions can offer 141 GB (H200), etc.
very small
~64 KiB
constant
mem
local
mem
512 KiB/thread
PCIe Gen 5
bi-bw = 128 GB/s
uni-bw = 64 GB/s
nvlink v4
bi-bw = 900 GB/s
uni-bw = 450 GB/s
(18 links with 25 GB/s)
x86 CPU,
DPUs, etc.
other
GPUs
it can also be Grace CPU, etc.

1D Register tiling



1D Register tiling




```
// allocate thread-local cache for results in registerfile
float threadResults[TM] = {0.0};

// outer loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
  // populate the SMEM caches (same as before)
  As[innerRowA * BK + innerColA] = A[innerRowA * K + innerColA];
  Bs[innerRowB * BN + innerColB] = B[innerRowB * N + innerColB];
  __syncthreads();

  // advance blocktile for outer loop
  A += BK;
  B += BK * N;

  // calculate per-thread results
  for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // we make the dotproduct loop the outside loop, which facilitates
    // reuse of the Bs entry, which we can cache in a tmp var.
    float Btmp = Bs[dotIdx * BN + threadCol];
    for (uint resIdx = 0; resIdx < TM; ++resIdx) {
      threadResults[resIdx] +=
          As[(threadRow * TM + resIdx) * BK + dotIdx] * Btmp;
    }
  }
  __syncthreads();
}
```

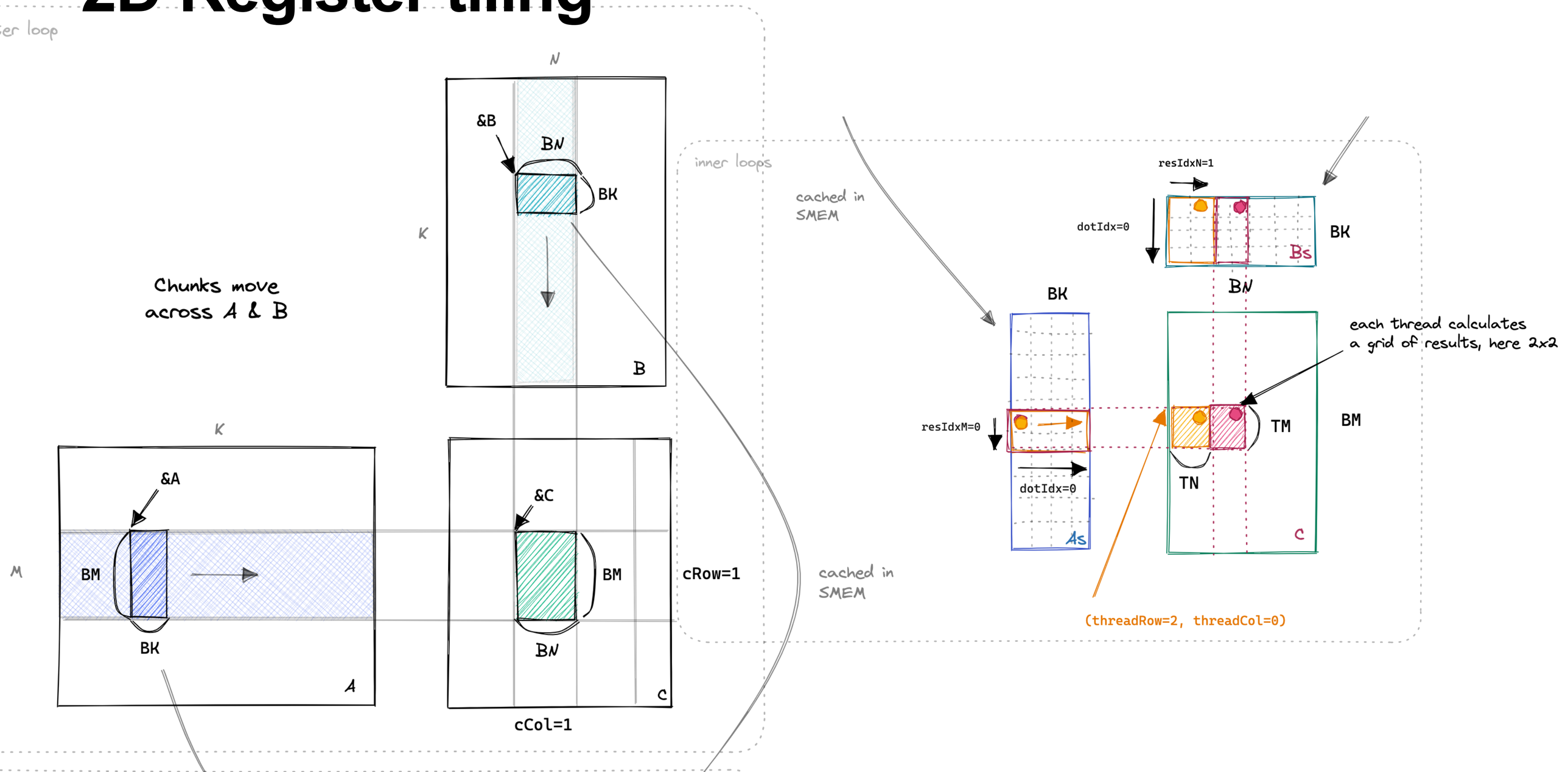
Matrix Multiplication

Welcome to the real world!

Share memory bandwidth usage has been significantly improved.

```
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000013) s, performance: (  320.4) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000023) s, performance: ( 1478.7) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 3
Average elapsed time: (0.000043) s, performance: ( 6314.3) GFLOPS. size: (512).
dimensions(m=n=k) 1024, alpha: 0.5, beta: 3
Average elapsed time: (0.000121) s, performance: (17761.8) GFLOPS. size: (1024).
dimensions(m=n=k) 2048, alpha: 0.5, beta: 3
Average elapsed time: (0.000888) s, performance: (19345.3) GFLOPS. size: (2048).
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.006933) s, performance: (19823.7) GFLOPS. size: (4096).
```

2D Register tiling



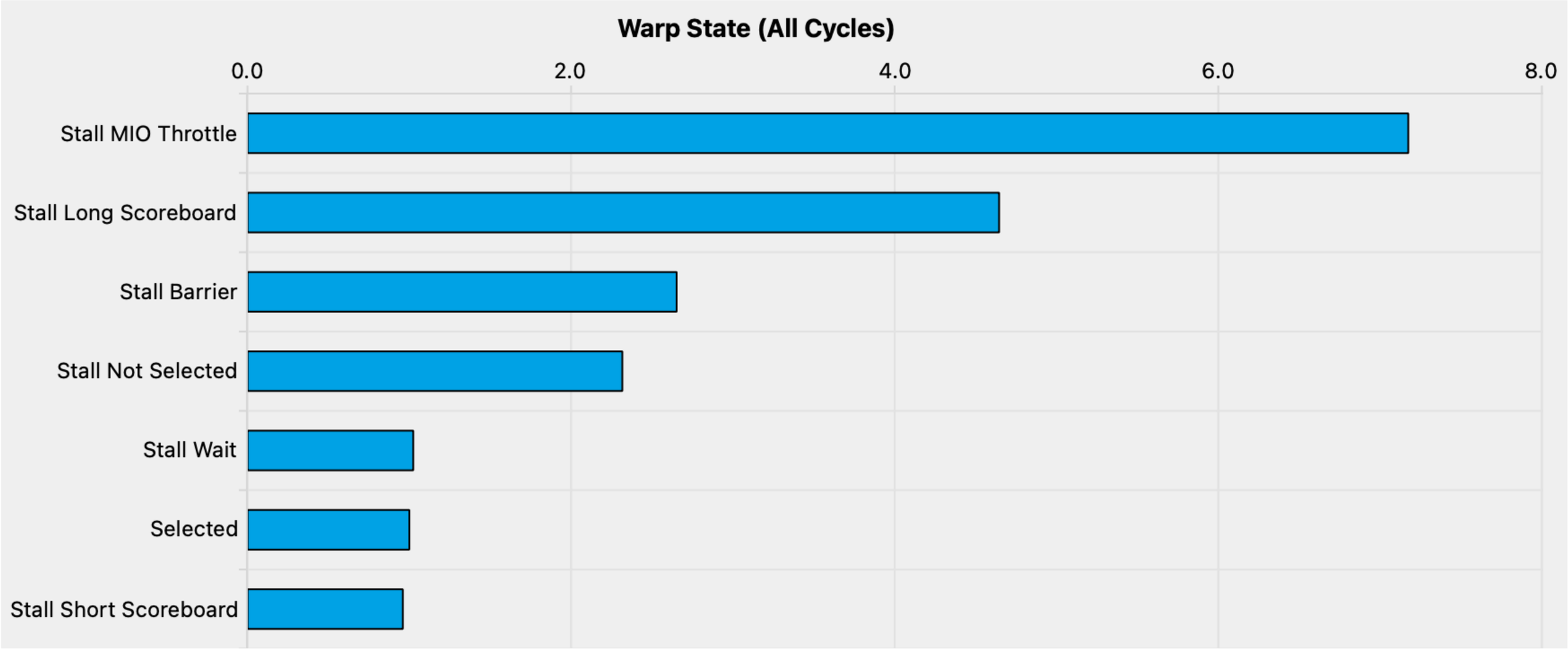
Matrix Multiplication

Welcome to the real world!

Share memory bandwidth usage has been improved further with 2D tiling.

```
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000034) s, performance: (  122.5) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000058) s, performance: (  576.0) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 3
Average elapsed time: (0.000109) s, performance: ( 2462.2) GFLOPS. size: (512).
dimensions(m=n=k) 1024, alpha: 0.5, beta: 3
Average elapsed time: (0.000204) s, performance: (10537.3) GFLOPS. size: (1024).
dimensions(m=n=k) 2048, alpha: 0.5, beta: 3
Average elapsed time: (0.000685) s, performance: (25089.7) GFLOPS. size: (2048).
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.005326) s, performance: (25806.8) GFLOPS. size: (4096).
```

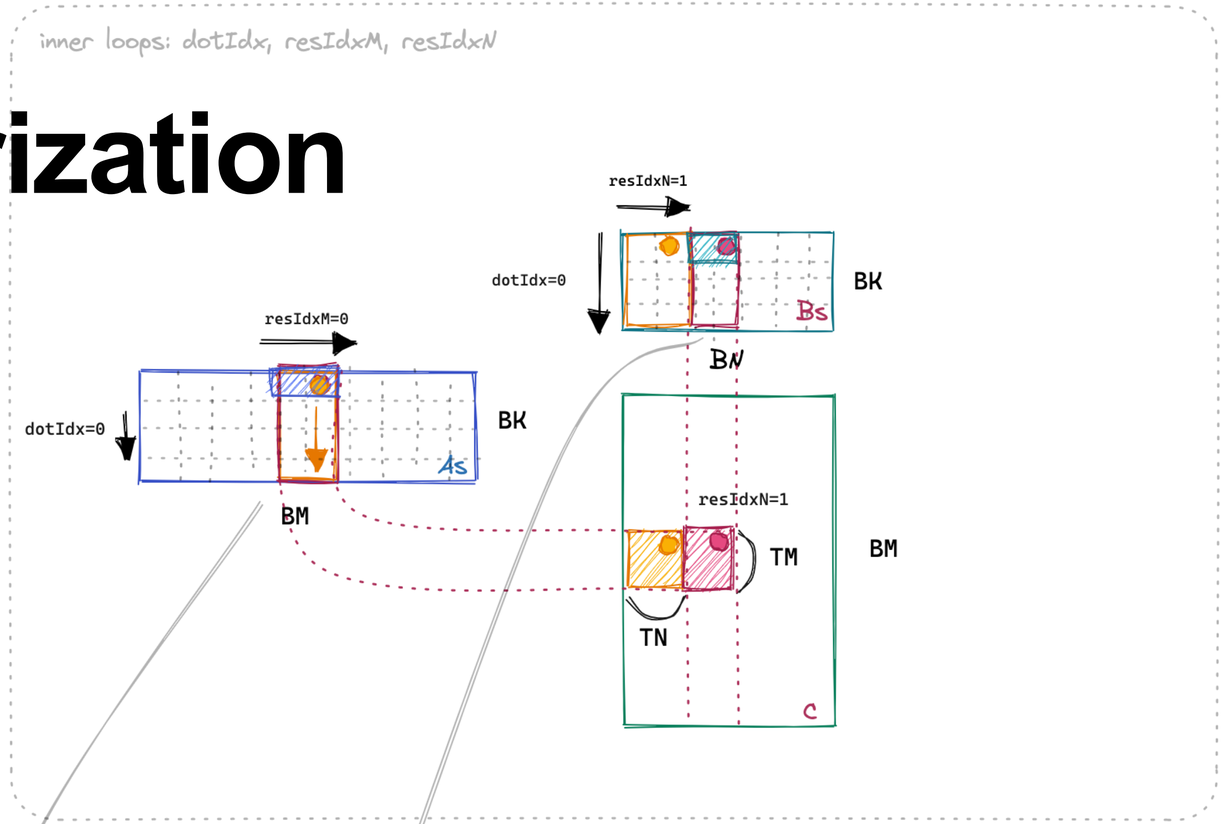
Memory access savings



Lesson 3: Let’s each thread do more work to reduce SMEM data access traffic.

–Tom Jerry

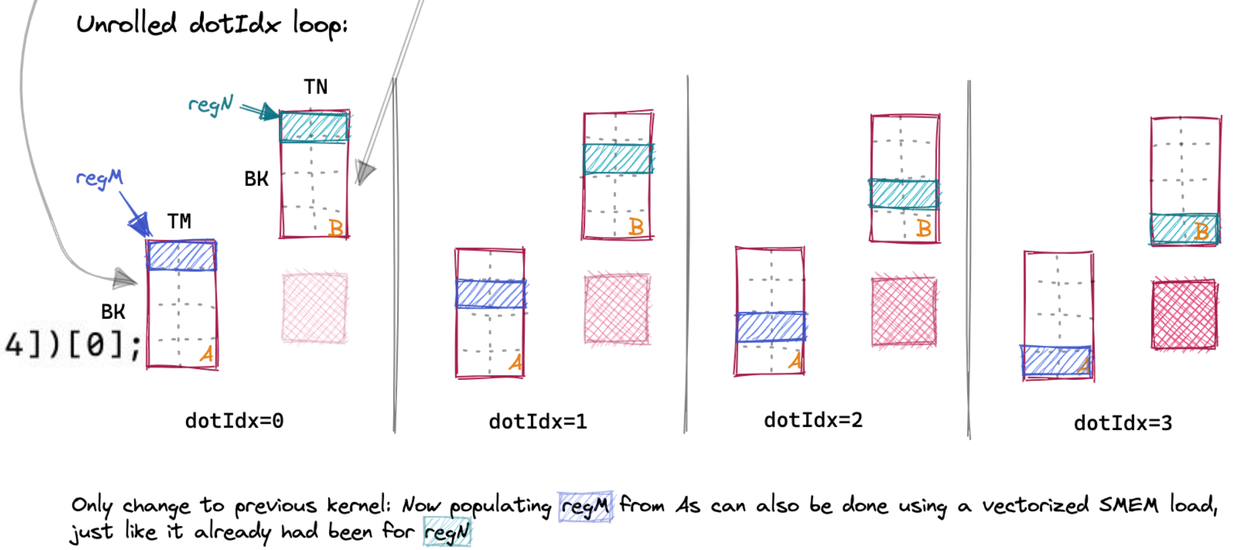
Memory Access Vectorization



🔷 Benefits

- **Fewer memory instructions** (1 vector load instead of 4 scalars).
- **Better coalescing and bus utilization** in GMEM.
- **Contiguous SMEM accesses** → vectorized `LDS.128` in inner loop.
- Overall: lower MIO pressure, fewer stalls, and ~2–3 % kernel speedup (≈ +500 GFLOPs).

```
float4 tmp =
    reinterpret_cast<float4 *>(&A[innerRowA * K + innerColA * 4])[0];
As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;
```



Only change to previous kernel: Now populating regM from As can also be done using a vectorized SMEM load, just like it already had been for regN

Warp Tiling
Warp level computation mapping

Definition: A warp (32 threads) collaboratively computes a sub-tile of the output matrix instead of each thread computing a single element.

Purpose: Align computation with the warp execution unit to maximize parallel efficiency and reduce synchronization overhead.

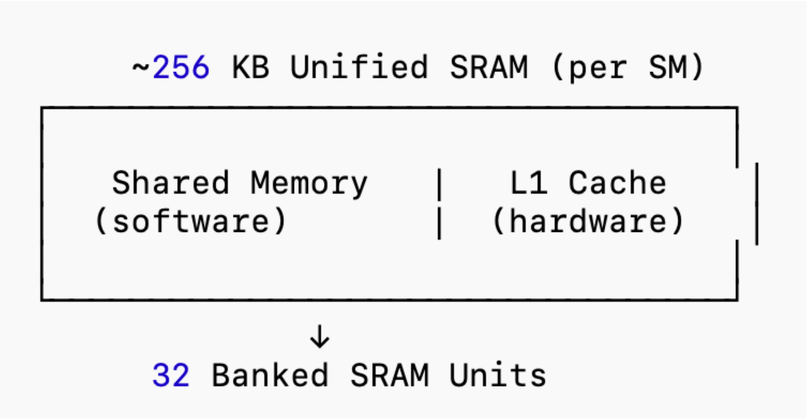
Benefit: Enables data reuse in registers, reducing shared memory bandwidth pressure and increasing arithmetic intensity.

Key challenge: Introduces strided access patterns, which can cause shared memory bank conflicts and must be carefully managed.

H100 Shared Memory + L1 Cache

The devil is in the details

- ~256 KB unified SRAM per SM (≈228 KB usable), dynamically split between Shared Memory and L1 Cache
- 32-bank design, 4 B per bank → **128 B/cycle per warp (ideal)**
- Bank mapping: **bank = (address / 4) mod 32**
- Access behavior: conflict-free = 1 cycle; broadcast = 1 cycle; k-way conflict = k cycles
- **Key takeaway:** performance is governed by bank mapping efficiency, not memory size




```
Conflict-free → 1 cycle
Broadcast     → 1 cycle
k-way conflict → k cycles
```

CUDA Programming

Mapping High-Dimensional Workloads onto Linear & Hierarchical Hardware

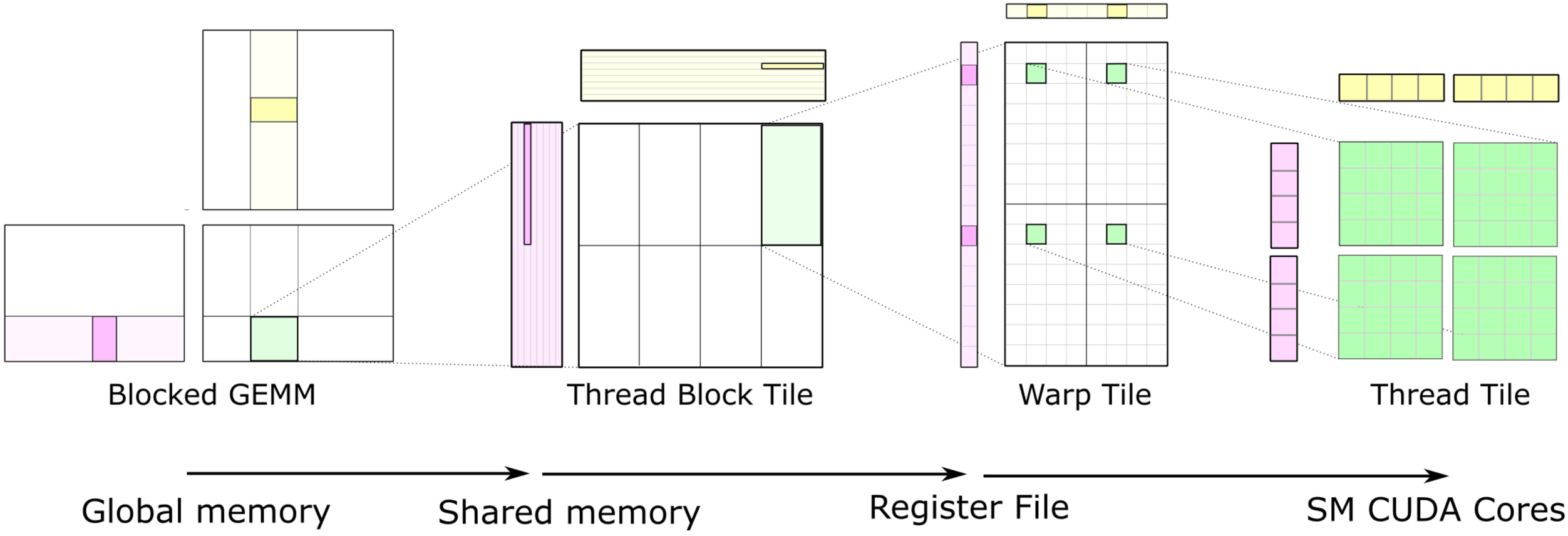
Data layout mismatch: Multi-dimensional tensors must map to linear memory → performance depends on coalescing and locality.

Execution mismatch: Multi-dimensional threads are executed as 1D warps → thread mapping determines memory access efficiency.

Communication placement: Same computation (e.g., O(N^3)) has very different cost depending on whether data movement happens in HBM, shared memory, or registers.

CUDA Programming

Mapping High-Dimensional Workloads onto Linear & Hierarchical Hardware



Lesson 4: It’s hard to beat cuBLAS on its own turf. Is this really true? To be continued…

–Tom Jerry