Machine Learning Systems

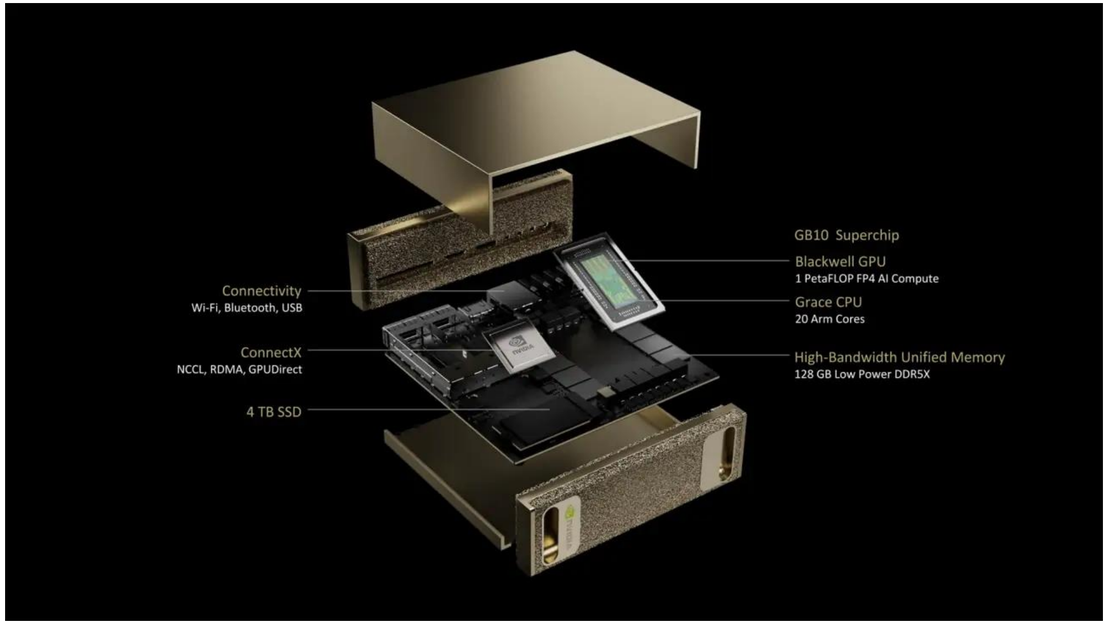
Build efficient and scalable ML services through the vertical integration of algorithms, system software, and hardware

Li Shang
lishang@slai.edu.cn

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.

Nvidia DGX Spark



Nvidia DGX Spark

	DGX Spark	Rubin CPX (chip)	Rubin Supernode (per R200 chip)	Rubin Supernode (rack)
Typical Use Case	Desktop development, fine-tuning, and inference for models up to 200 billion parameters.	Massive-context inference, specifically the compute-heavy "prefill" phase.	Large-scale AI training and bandwidth-intensive inference.	Large-scale training and inference in datacenter environments.
Memory Bandwidth	273 GB/s	~2 TB/s	20.5 TB/s	1.7 PB/s
Memory Type	128 GB LPDDR5x	128 GB GDDR7	288 GB HBM4	72 R200 chips with HBM4 and 144 CPX chips with GDDR7.
Architecture	Grace Blackwell GB10 Superchip (integrated CPU and GPU).	Monolithic die with GDDR7 memory.	Dual-die R200 chip with HBM4 memory.	Integrated MGX system that combines Rubin and Rubin CPX GPUs.

DGX Spark: With 273 GB/s, its memory bandwidth is a limiting factor for heavy training workloads compared to the datacenter-grade hardware. It is designed for development and fine-tuning where the high unified memory capacity is a key benefit, allowing developers to work on large models locally.

Rubin CPX (Chip): The CPX chip, by contrast, is a specialized accelerator designed with less memory bandwidth (~2 TB/s with GDDR7) and more computational FLOPS for the "prefill" stage of inference. It is not intended for the memory-bound training phase of AI and would perform poorly in this role.

Rubin Supernode (R200 Chip): The standard Rubin R200 chips use HBM4 memory, delivering a massive 20.5 TB/s of memory bandwidth per chip. This is engineered for demanding training applications that require extremely high throughput, such as training foundation models.

Rubin Supernode (Rack): A full Rubin supernode, like the Vera Rubin NVL144 CPX rack, combines multiple R200 and CPX chips to achieve a staggering 1.7 PB/s of total bandwidth across the rack. While a rack can perform training, its bandwidth is heavily influenced by the high-bandwidth R200 chips.

GDDR7 (Rubin CPX)

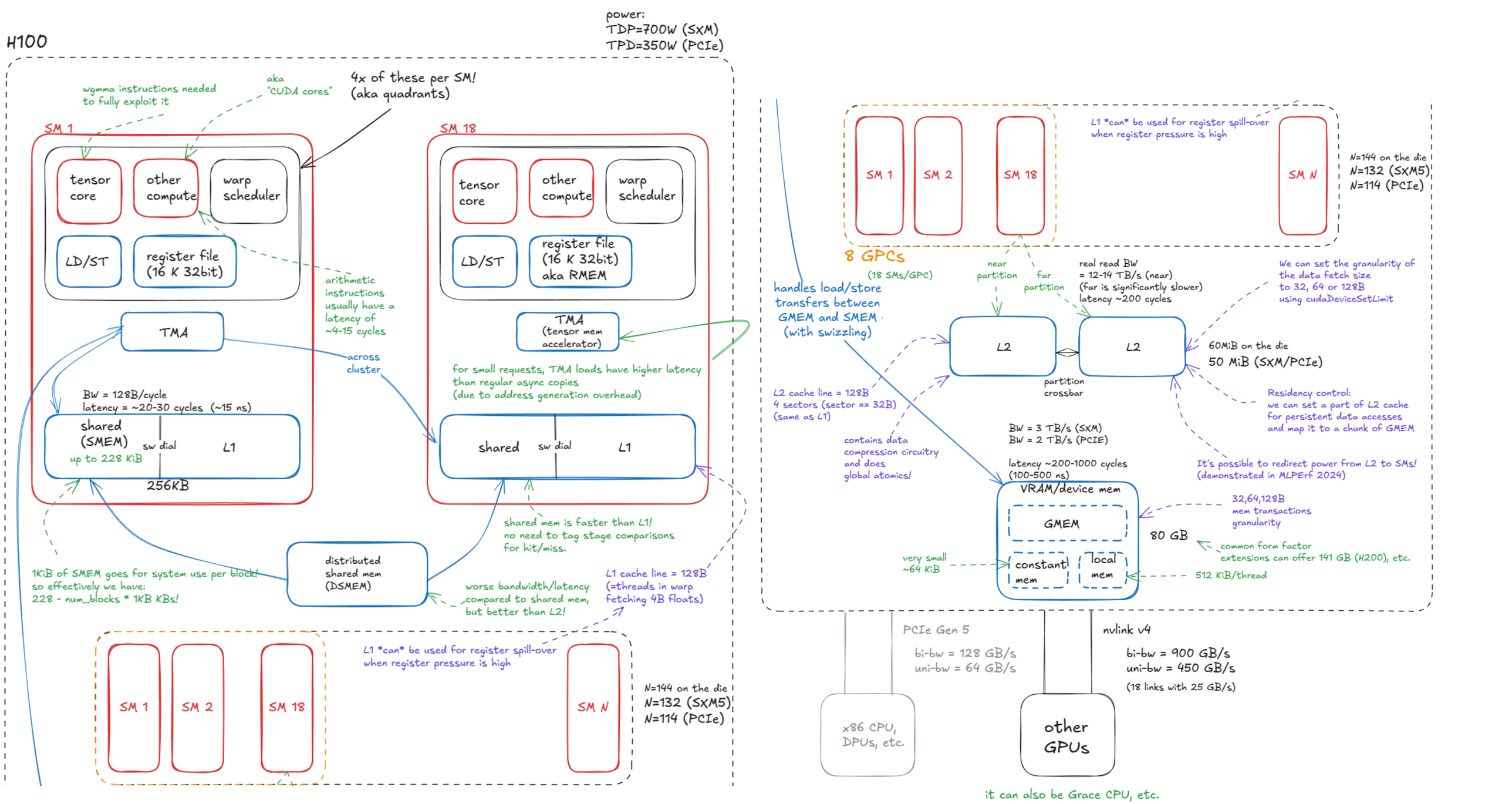
- **Formula:** $(Data\ Rate\ in\ Gbps) \times (Bus\ Width\ in\ bits)/8$
- **Calculation:** $(32\ \text{Gbps}) \times (512\ \text{bits})/8 = 2,048\ \text{GB/s}$
- **Total Bandwidth: ~2 TB/s**

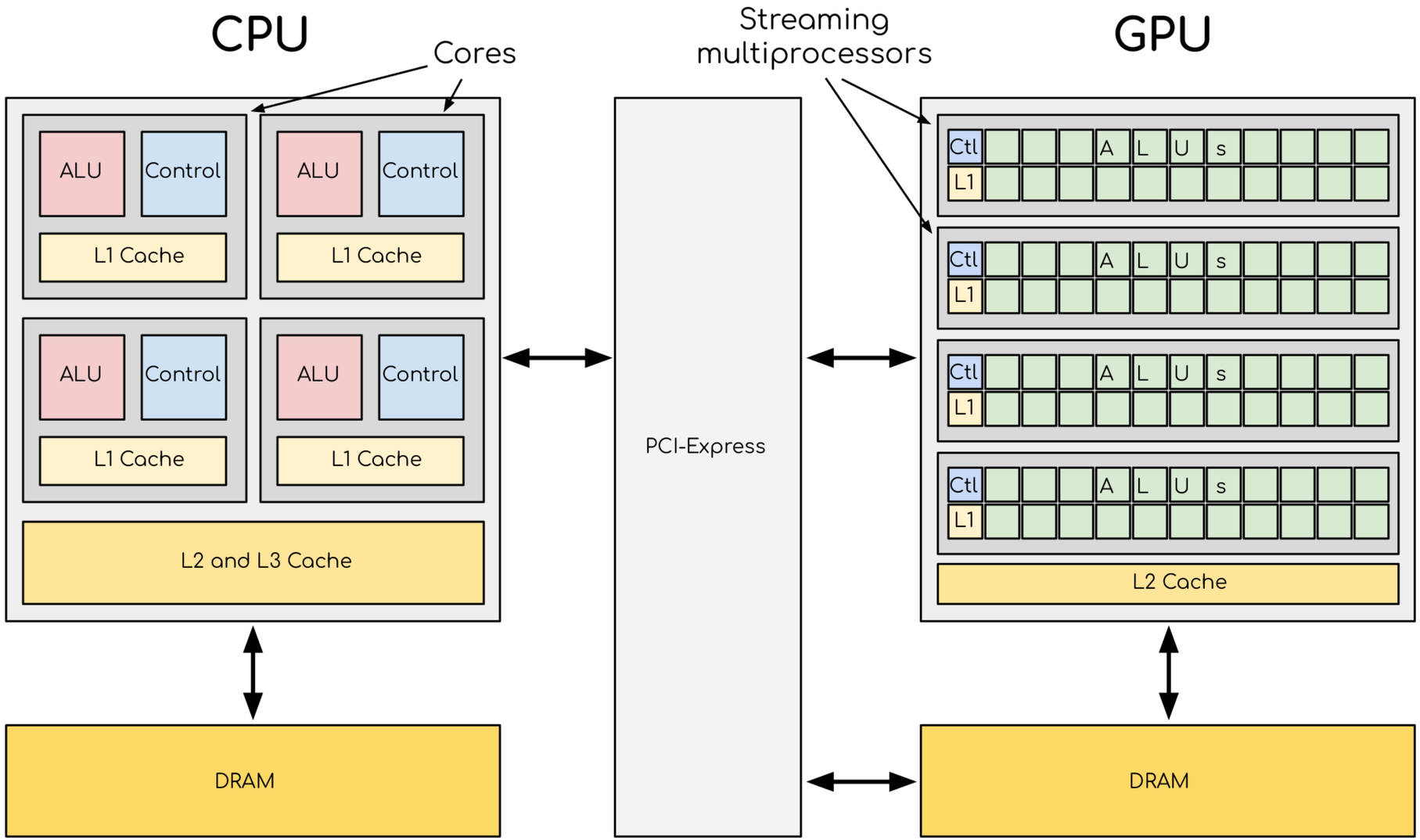
LPDDR5X (DGX Spark)

- **Formula:** $(Data\ Rate\ in\ MT/s) \times (Bus\ Width\ in\ bits)/8$
- **Calculation:** $(8533\ \text{MT/s}) \times (256\ \text{bits})/8 = 273,056\ \text{MB/}$
- **Total Bandwidth: ~273 GB/s**

- **Memory Type:** HBM4
- **Data Rate:** 8 Gbps (Gigabits per second) per pin, as defined by the JEDEC standard.
- **Bus Width:** 2048-bits per HBM4 stack, which is double the 1024-bit width of previous HBM generations.
- **Calculation:**
  - $Bandwidth = (Data\ Rate\ in\ Gbps) \times (Bus\ Width\ in\ bits)/8$
  - $Bandwidth = (8\ \text{Gbps}) \times (2048\ \text{bits})/8$
  - $Bandwidth = 8 \times 256\ \text{GB/s}$
  - $Bandwidth = 2048\ \text{GB/s}$
  - $Bandwidth = 2.048\ \text{TB/s}$ per stack.

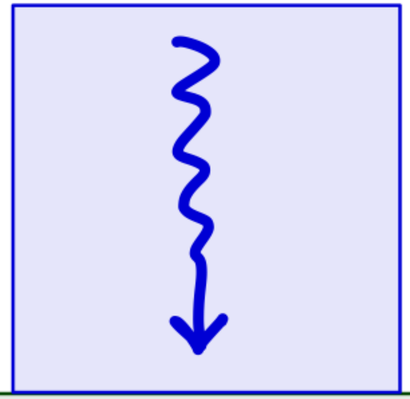
How to program GPGPU?


H100
power:
TDP=700W (SXM)
TPD=350W (PCIe)
wgmma instructions needed to fully exploit it
aka "CUDA cores"
4x of these per SM!
(aka quadrants)
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)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
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.



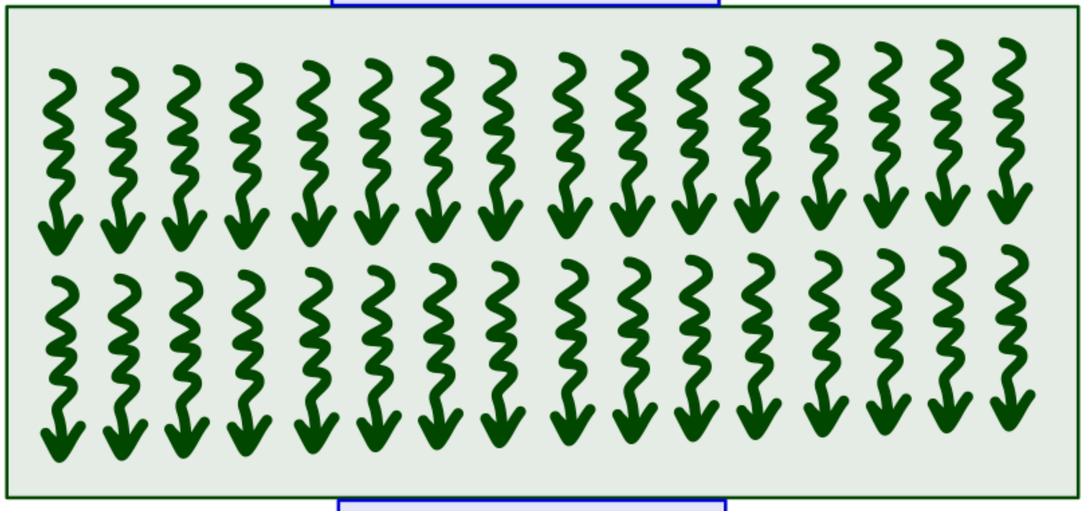
GPGPU as a really powerful accelerator

CPU thread

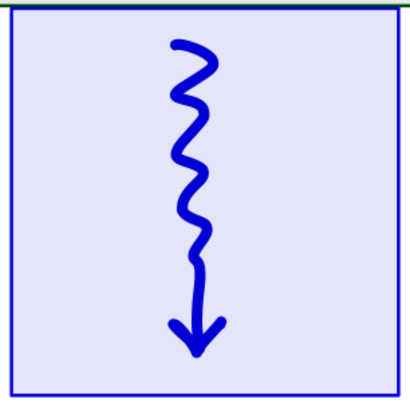


- Prepare data (matrices A, B, C).
- Copy data from RAM to VRAM.

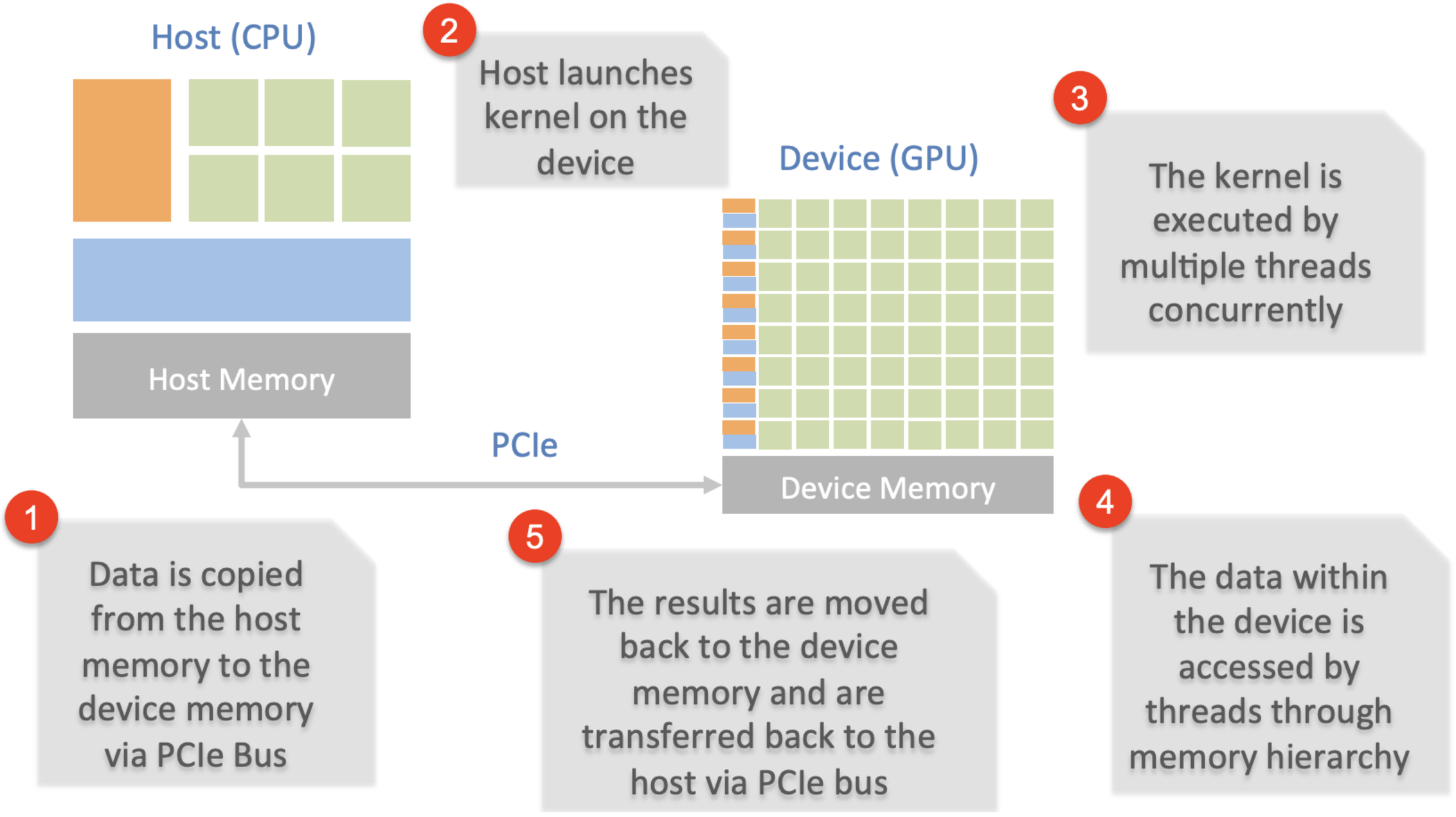
GPU threads



- Parallel Matrix Multiplication.



- Copy results from VRAM to RAM.


Host (CPU)
Host Memory
PCIe
Device (GPU)
Device Memory
1
Data is copied from the host memory to the device memory via PCIe Bus
2
Host launches kernel on the device
3
The kernel is executed by multiple threads concurrently
4
The data within the device is accessed by threads through memory hierarchy
5
The results are moved back to the device memory and are transferred back to the host via PCIe bus

Example 1

```cpp
void vectorAdd(const float* A, const float* B, float* C, int N) {
    for (int idx = 0; idx < N; idx++) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1 << 16; // 65536 elements
    size_t size = N * sizeof(float);

    // Allocate host memory using vectors (auto-managed)
    std::vector<float> A(N), B(N), C(N);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        A[i] = 1.0f;
        B[i] = 2.0f;
    }

    // Perform vector addition on CPU
    vectorAdd(A.data(), B.data(), C.data(), N);

    // Verify results (print first 5)
    for (int i = 0; i < 5; i++) {
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
    }

    return 0;
}
```

1. Define CUDA Kernel

```cpp
__global__ void vectorAdd(const float *A, const float *B, float *C, int N){...}
```

2. Allocate GPU memory

```cpp
cudaMalloc((void**)&d_A, size);
```

3. Copy data from CPU to GPU

```cpp
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

4. Launch CUDA kernel

```cpp
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

5. Copy data from GPU back to CPU

```cpp
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

6. Free resources

```cpp
cudaFree(d_A);
```

NVCC

The NVIDIA CUDA Compiler Driver

nvcc is the NVIDIA CUDA compiler driver, part of the CUDA toolkit. It compiles CUDA C/C++ programs containing both host (CPU) and device (GPU) code. Its job is to separate, compile, and link host and device code into one executable.

- Code Separation – Splits host and device code in each .cu file.
- Dual Compilation Paths
- Host code → compiled by gcc or cl.
- Device code → compiled by ptxas into PTX or cubin binaries.
- Linking – nvlink merges GPU objects with CPU objects into a final executable.

A host compiler (gcc / cl) for CPU code,
A device compiler (ptxas) for GPU code,
A linker (nvlink) to combine everything into one binary.

PTX and SASS

Target: SM_90 (H100). Cache policies `LDG.E/STG.E` shown, uniform regs (URx) used for warp-uniform values.

```
/*0000*/ LDC  R1, c[0x0][0x28];                 // const load (launch metadata)
/*0010*/ S2R  R0, SR_TID.X;                     // R0 = threadIdx.x
/*0020*/ S2UR UR4, SR_CTAID.X;                  // UR4 = blockIdx.x (uniform)
/*0030*/ LDC  R9, c[0x0][RZ];                   // R9 = blockDim.x (from c[0])
/*0040*/ IMAD R9, R9, UR4, R0;                  // i = bdim*bidx + tid  R9

/*0050*/ ULDC     UR4,  c[0x0][0x228];          // UR4 = N (length)
/*0060*/ ISETP.GE.AND P0, PT, R9, UR4, PT;      // P0 = (i >= N)
/*0070*/ @P0 EXIT;                              // predicated early exit

/*0080*/ LDC.64  R2,  c[0x0][0x210];            // R2 = base A
/*0090*/ ULDC.64 UR4, c[0x0][0x208];            // UR4 = global mem descriptor
/*00a0*/ LDC.64  R4,  c[0x0][0x218];            // R4 = base B
/*00b0*/ LDC.64  R6,  c[0x0][0x220];            // R6 = base C

/*00c0*/ IMAD.WIDE R2, R9, 0x4, R2;             // &A[i] = A + i*4
/*00d0*/ LDG.E     R3, desc[UR4][R2.64];        // R3 = A[i] via descriptor

/*00e0*/ IMAD.WIDE R4, R9, 0x4, R4;             // &B[i]
/*00f0*/ LDG.E     R4, desc[UR4][R4.64];        // R4 = B[i]

/*0100*/ IMAD.WIDE R6, R9, 0x4, R6;             // &C[i]
/*0110*/ FADD      R9, R4, R3;                  // R9 = B[i] + A[i]
/*0120*/ STG.E desc[UR4][R6.64], R9;            // C[i] = R9 (store result)

/*0130*/ EXIT;                                  // normal return
/*0140*/ BRA `(.L_pad);  NOP; NOP;              // padding/alignment
```

Kernel: `__global__ void vectorAdd(const float* A, const float* B, float* C, int N);`

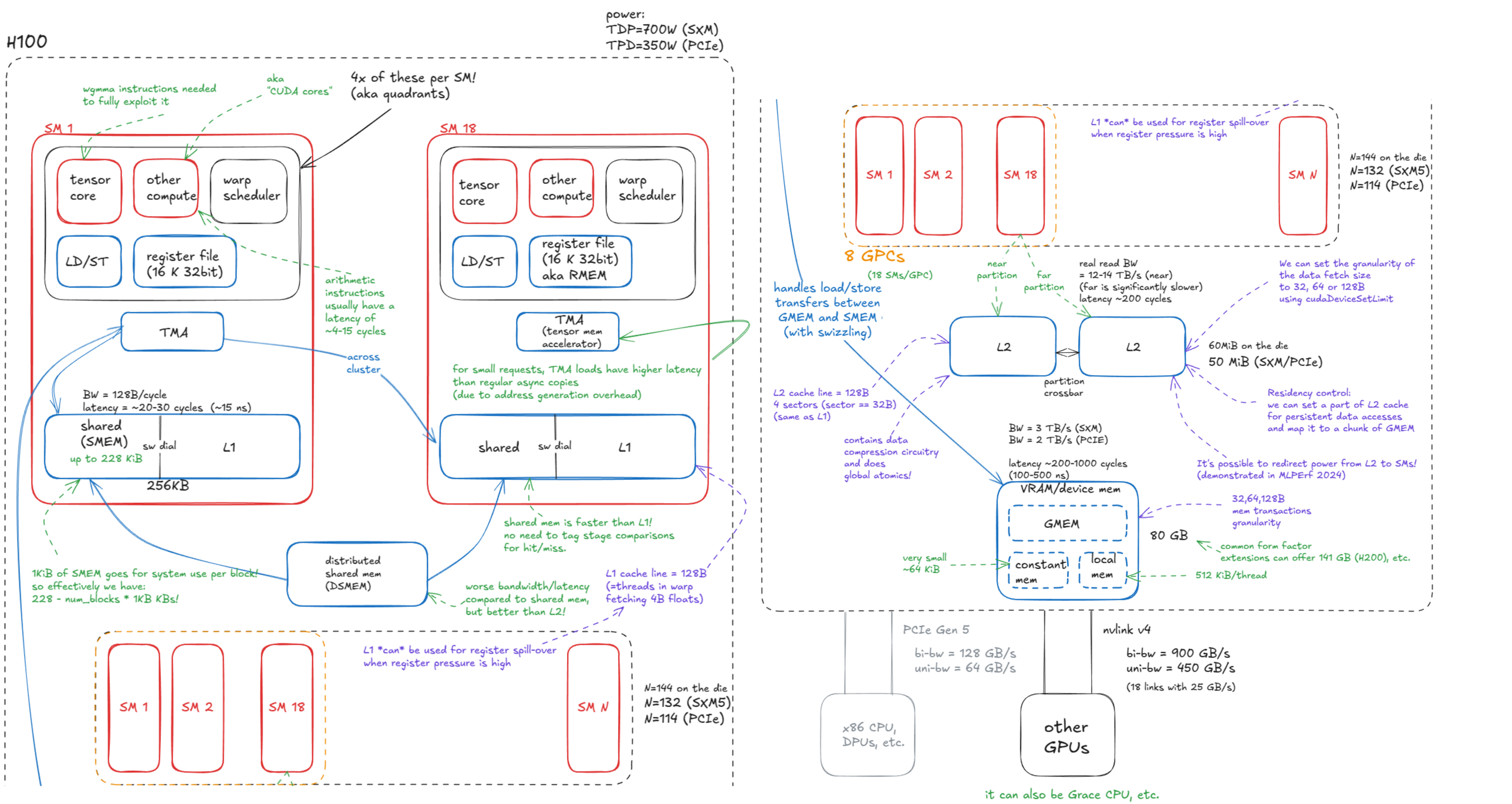
```
.visible .entry _Z9vectorAddPKfS0_Pfi(      // visible kernel entry (mangled name)
  .param .u64 _param_A,                     // A pointer (u64)
  .param .u64 _param_B,                     // B pointer (u64)
  .param .u64 _param_C,                     // C pointer (u64)
  .param .u32 _param_N                      // N length (u32)
){
  .reg .pred %p<2>;                         // predicate regs %p0-%p1
  .reg .f32  %f<4>;                         // float regs %f0-%f3
  .reg .b32  %r<6>;                         // 32-bit int regs %r0-%r5
  .reg .b64  %rd<11>;                       // 64-bit regs %rd0-%rd10

  ld.param.u64 %rd1, [_param_A];            // %rd1 = A (generic ptr)
  ld.param.u64 %rd2, [_param_B];            // %rd2 = B
  ld.param.u64 %rd3, [_param_C];            // %rd3 = C
  ld.param.u32 %r2,  [_param_N];            // %r2  = N

  mov.u32 %r3, %ctaid.x;                    // r3 = blockIdx.x
  mov.u32 %r4, %ntid.x;                     // r4 = blockDim.x
  mov.u32 %r5, %tid.x;                      // r5 = threadIdx.x
  mad.lo.s32 %r1, %r3, %r4, %r5;            // r1 = bidx*bdim + tid

  setp.ge.s32 %p1, %r1, %r2;                // p1 = (i >= N)
  @%p1 bra DONE;                            // out-of-range  jump to DONE

  cvta.to.global.u64 %rd4, %rd1;            // A: genericglobal
  mul.wide.s32 %rd5, %r1, 4;                // rd5 = (long)i*4 bytes
  add.s64 %rd6, %rd4, %rd5;                 // rd6 = &A[i]

  cvta.to.global.u64 %rd7, %rd2;            // B: genericglobal
  add.s64 %rd8, %rd7, %rd5;                 // rd8 = &B[i]
  ld.global.f32 %f1, [%rd8];                // f1 = B[i]
  ld.global.f32 %f2, [%rd6];                // f2 = A[i]
  add.f32 %f3, %f2, %f1;                    // f3 = A[i] + B[i]

  cvta.to.global.u64 %rd9, %rd3;            // C: genericglobal
  add.s64 %rd10, %rd9, %rd5;                // rd10 = &C[i]
  st.global.f32 [%rd10], %f3;               // C[i] = f3

DONE:
  ret;                                      // return
}
```

PTX and SASS

Aspect	PTX (Parallel Thread Execution)	SASS (Streaming Assembler)
Definition	*Virtual GPU instruction set* — compiler IR between CUDA C++ and hardware.	*Hardware machine code* actually executed by the GPU's SMs.
Purpose	Portability and optimization target for `nvcc`.	Final low-level binary tuned for a specific architecture (e.g., `sm_90`).
Generated by	CUDA frontend compiler.	`ptxas` or driver JIT compiler.
Portability	Same PTX can run on multiple generations.	Only valid for its specific GPU architecture.
Registers	Virtual, symbolic (`%r1`, `%f1`, `%p1`).	Physical (`R0`–`R255`, `P0`–`P7`, `UR0`–`UR15`).
Instruction form	Textual IR (e.g., `add.f32 %f3, %f1, %f2;`).	128-bit binary ops (e.g., `FADD R3, R1, R2;`).
Address spaces	Explicit `.global`, `.shared`, `.local`, `.const`.	Implicit in opcodes and descriptors (`LDG`, `STS`, `LDS`).
Abstraction level	Algorithmic — what to do.	Hardware — how it's done.
Use case	Compiler optimization, portability, IR analysis.	Performance tuning, hardware profiling, reverse engineering.

PTX and SASS

Insight Type	PTX	SASS	Need Hardware Docs?
Algorithmic logic	✅	✅	❌
Compiler optimization	✅	✅	❌
Memory/cache policy	⚠️ (abstract)	✅ (explicit)	❌
Tensor-core usage	⚠️	✅	❌
Register pressure	✅	✅	❌
Exact pipeline timing	❌	⚠️ (partial)	✅
Microarchitectural hazards	❌	❌	✅
Physical datapath layout	❌	⚠️ (implied)	✅


H100
power:
TDP=700W (SXM)
TPD=350W (PCIe)
wgmma instructions needed to fully exploit it
aka "CUDA cores"
4x of these per SM!
(aka quadrants)
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)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
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.

Example 2

Matmul is computation heavy: computation complexity n^3 vs. communication cost n^2.

Transformers are matmul-heavy: most FLOPs (MLP layers, attention QKV, output projections) in both training and inference.

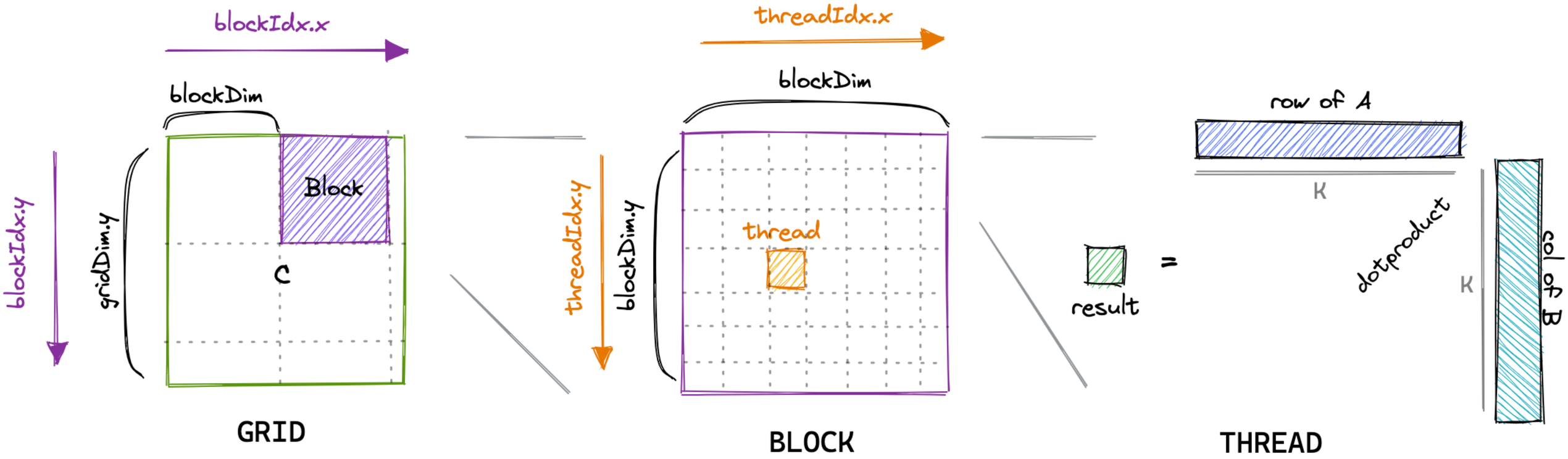
Embarrassingly parallel: matmuls map naturally onto GPUs.

Foundational: understanding matmul kernels equips you to design nearly any other high-performance GPU kernel.

CUDA code is written from a single-thread perspective: In the code of the kernel, we access the blockIdx and threadIdx built-in variables. These will return different values based on the thread that's accessing them.

```
// create as many blocks as necessary to map all of C
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
// 32 * 32 = 1024 thread per block
dim3 blockDim(32, 32, 1);
// launch the asynchronous execution of the kernel on the device
// The function call returns immediately on the host
sgemm_naive<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

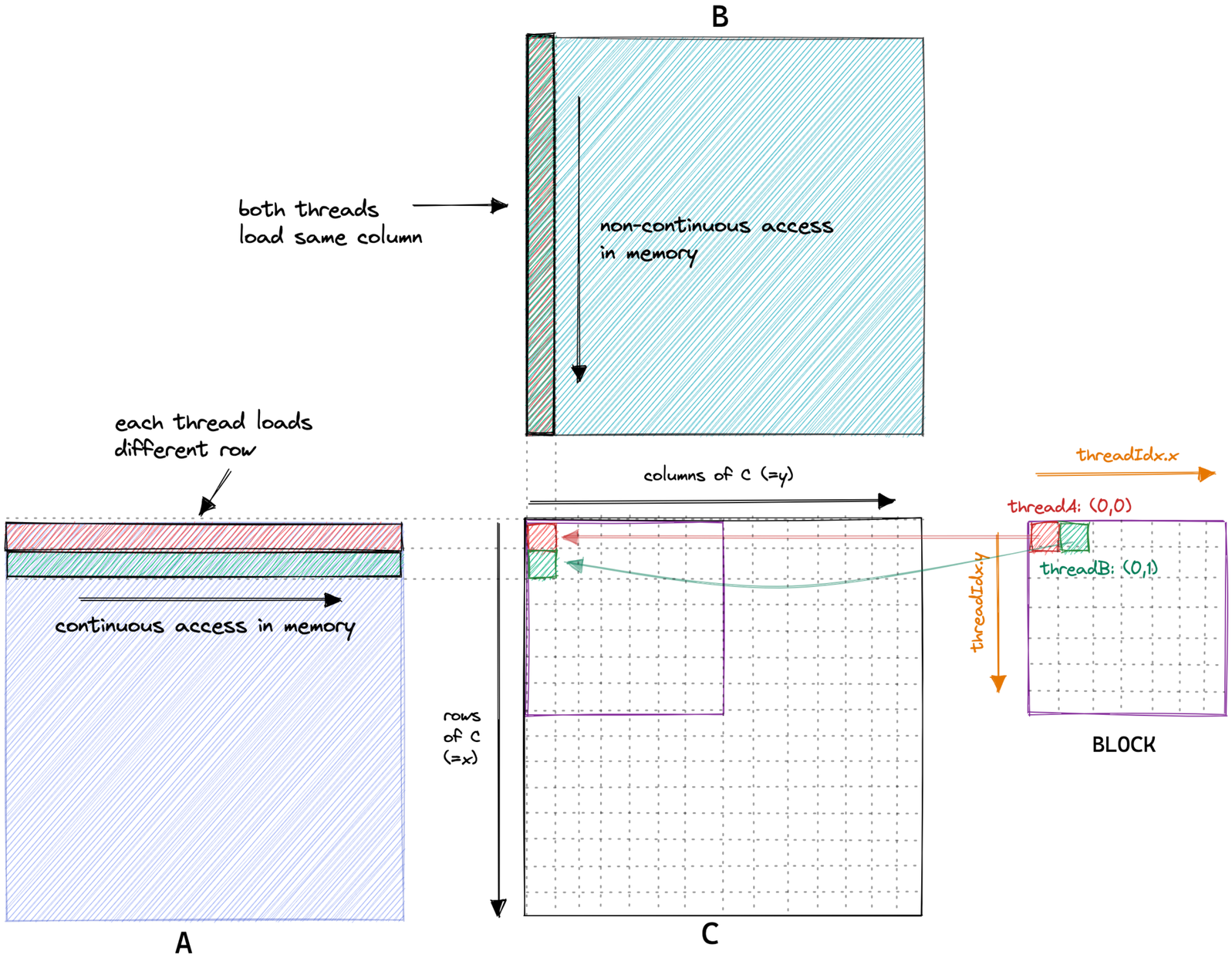
Thread Organization



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



A, B, C are stored in row-major order.
This means that the last index (here y)
is the one that iterates continuous through
memory (=has stride 1).

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];
  }
}
```

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 4092^2 entries of C, we have to perform a dot product of two vectors of size 4092, 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.

$2*4092^3 + 4092^2 = 137$ GFLOPS

Total data to read (minimum): $3 * 4092^2 * 4B = 201MB$

Total data to store: $4092^2 * 4B = 67MB$

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

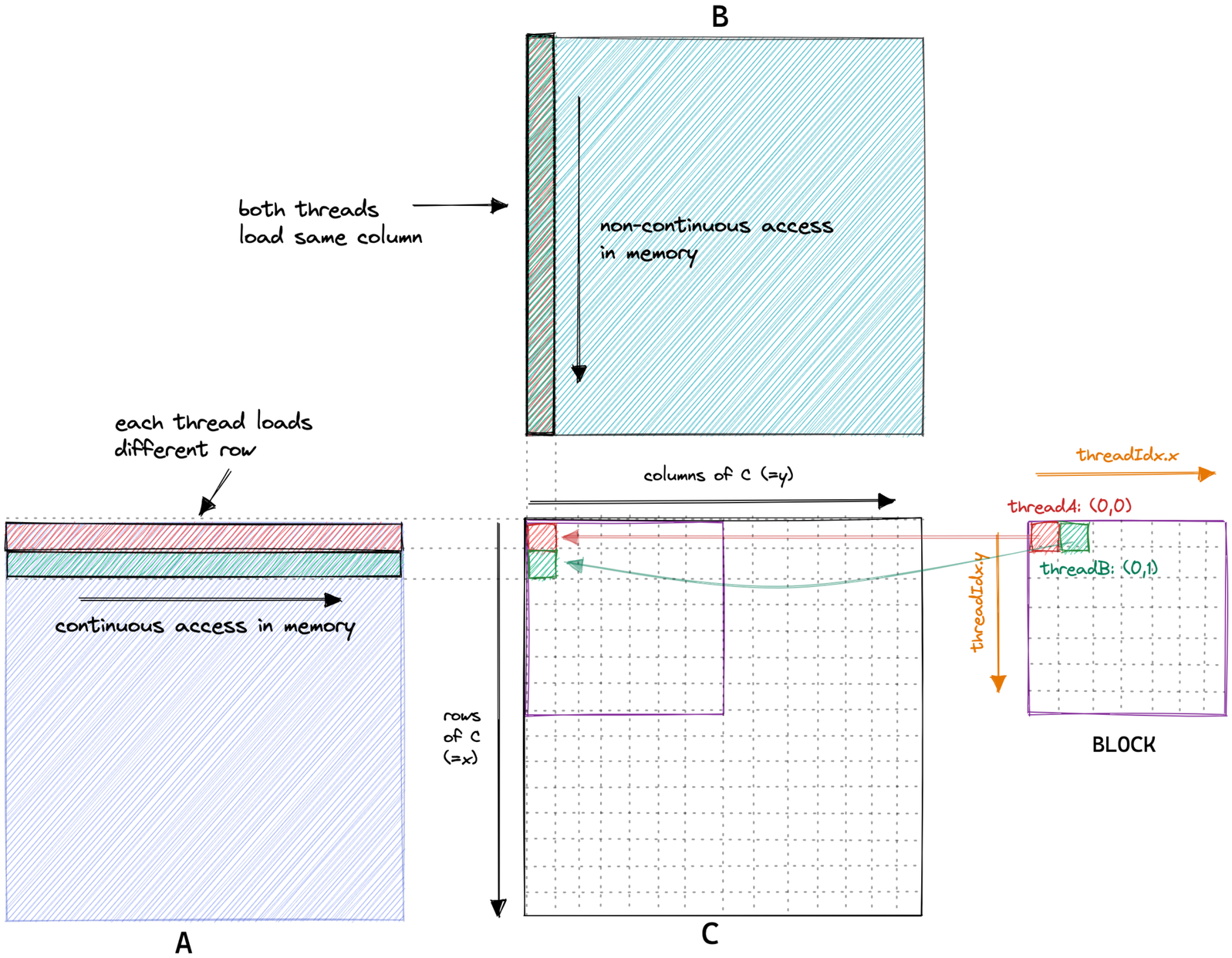
- The computation time (≈2.3 ms) is ~30× larger than the pure memory-transfer lower bound (≈0.07 ms).
- The operation is compute-bound, not bandwidth-bound — meaning the limiting factor is how fast the GPU's FP units can perform multiply-adds, not how fast data can be streamed from HBM3.
- In reality, 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 486.7 GFLOPS, and the run time is 273.527ms.

```
Max size: 4096
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000073) s, performance: (   57.4) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000142) s, performance: (  236.2) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 3
Average elapsed time: (0.000557) s, performance: (  482.2) GFLOPS. size: (512).
dimensions(m=n=k) 1024, alpha: 0.5, beta: 3
Average elapsed time: (0.004423) s, performance: (  485.5) GFLOPS. size: (1024).
dimensions(m=n=k) 2048, alpha: 0.5, beta: 3
Average elapsed time: (0.035296) s, performance: (  486.7) GFLOPS. size: (2048).
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3

Average elapsed time: (0.275612) s, performance: (  498.7) GFLOPS. size: (4096).
```

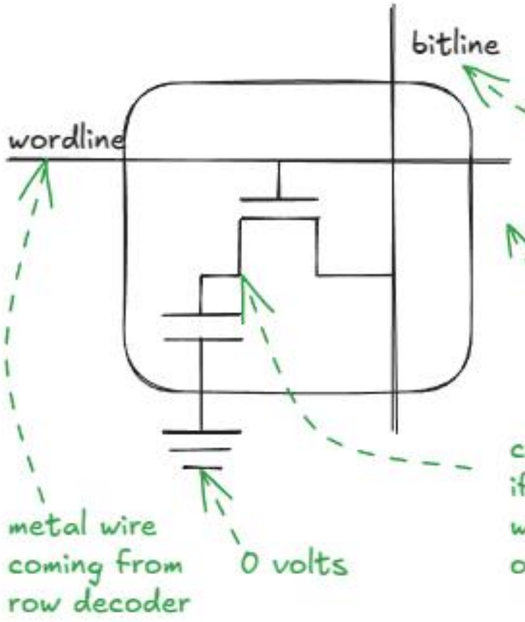


A, B, C are stored in row-major order.
This means that the last index (here y)
is the one that iterates continuous through
memory (=has stride 1).

Do we really understand GMEM?

GMEM is nothing but a matrix of DRAM cells.

and DRAM cells are glorified guarded capacitors. :)



metal wire coming from sense amplifier

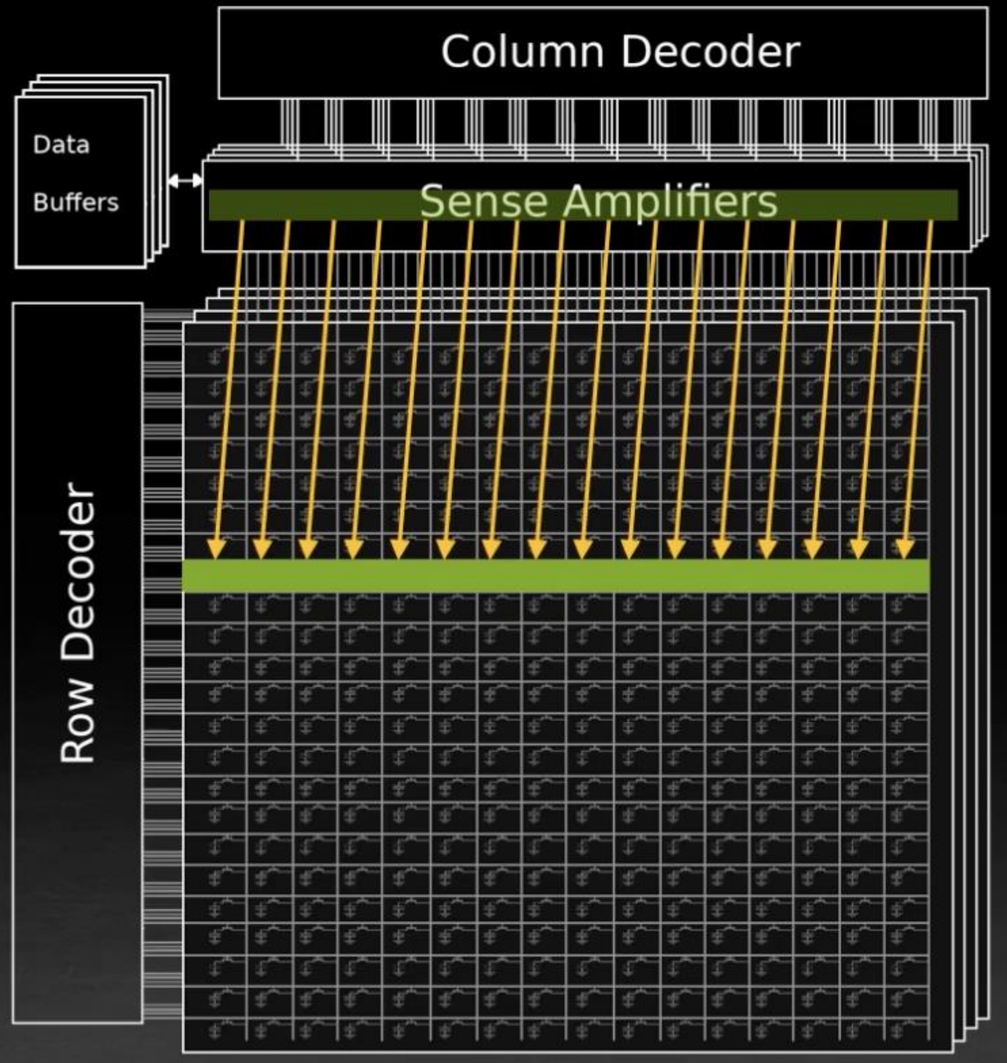
capacitor stores a charge;
if V > V_t (threshold)
we interpret it as a logical 1,
otherwise as a logical 0.

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: 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

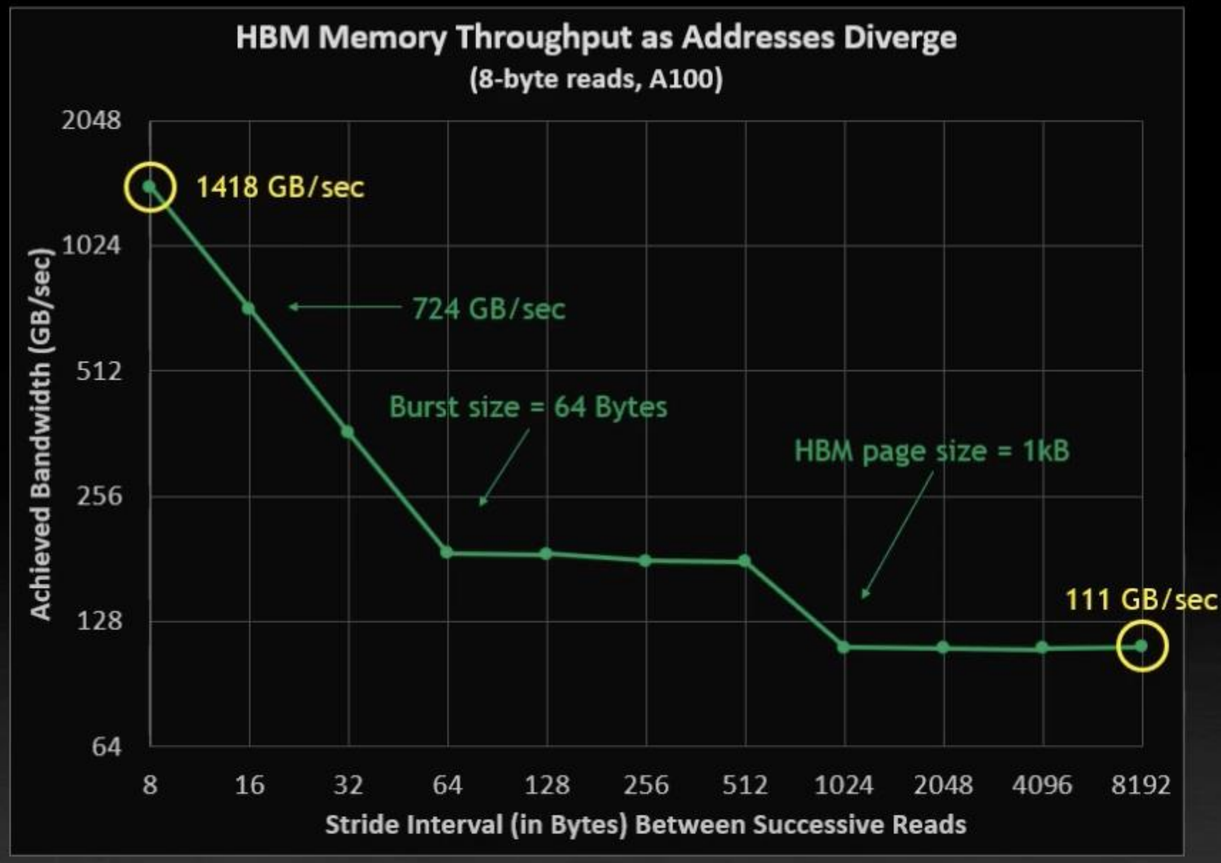
SO WHAT DOES THIS ALL MEAN?

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!



GMEM as a three-level pipeline

The devil is in the details

- Each DRAM read = **activate + burst transfer**.
- Bandwidth drops when stride > burst size (no coalescing).
- Drops further when stride > page size (no row locality).

Stage	Hardware object	Typical size	What happens
① **Page / Row**	Data stored in one DRAM bank row	**1 KB – 2 KB**	Opened by an `ACTIVATE` command into sense amplifiers
② **Column burst**	Portion read per `READ` or `WRITE`	**32–64 B**	Sent over the internal bus per command
③ **I/O bus**	External HBM/DDR link	**8–16 B per transfer**	The burst is serialized over multiple cycles

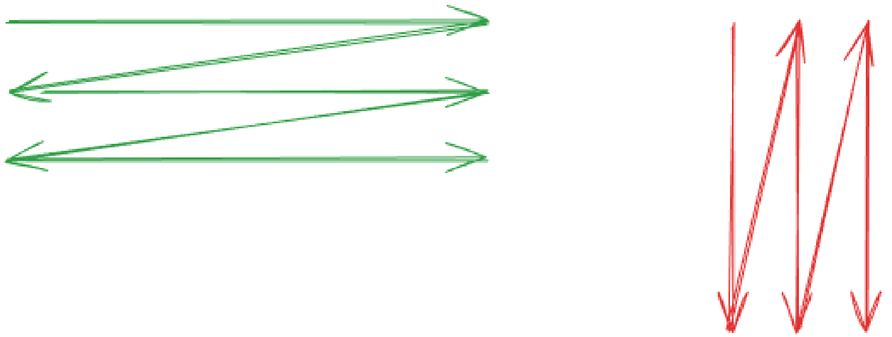
M=N=256

(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

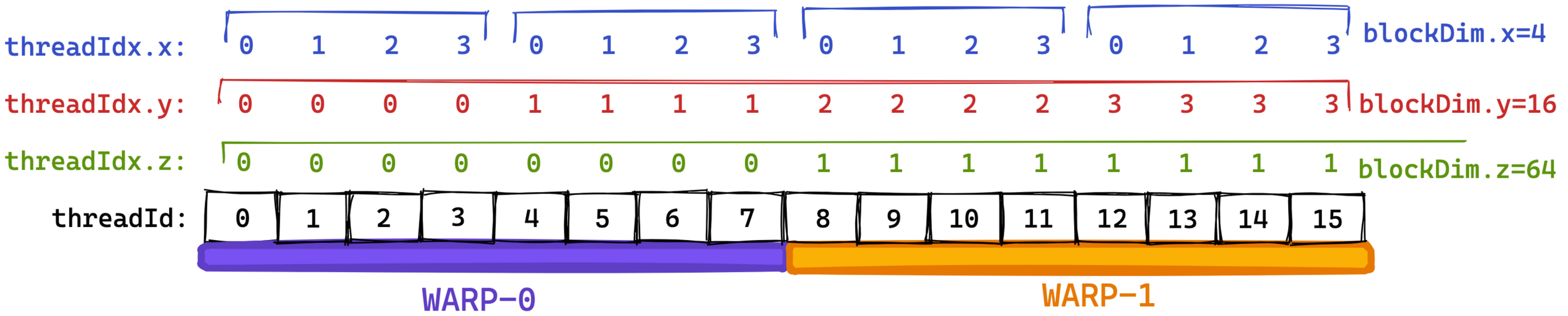


given what we just learned it's obvious that there is one right answer:
traversing columns should be in the inner/fast for loop as that would lead to a single row read
followed by 256 column reads for one iteration of the inner for loop (256 row reads in total).
the alternative would be 256 row reads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

Thread Organization

Wrap organization and its impact on memory access

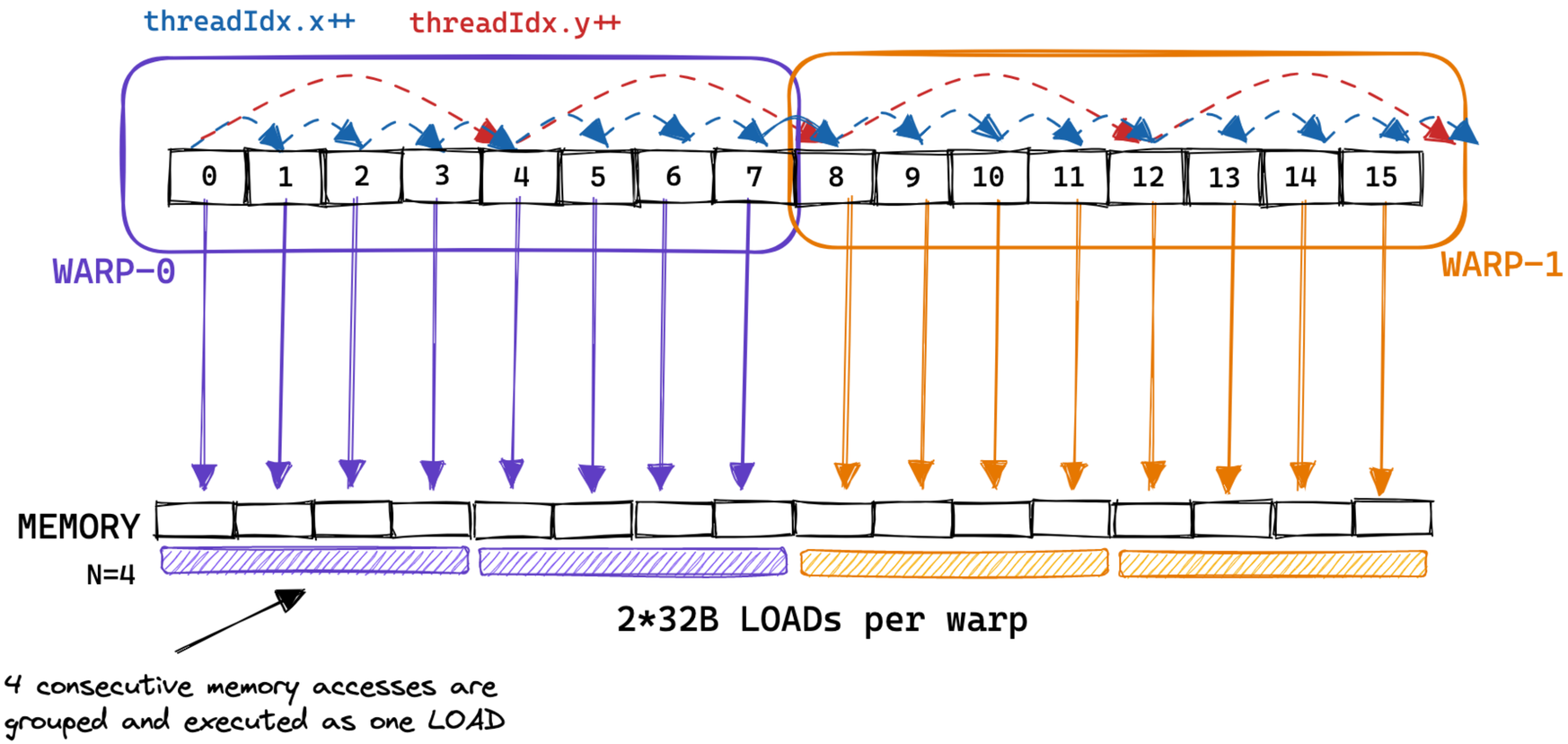
```
threadId = threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*threadIdx.z)
```




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

Thread Organization

This is what we want



Thread Organization

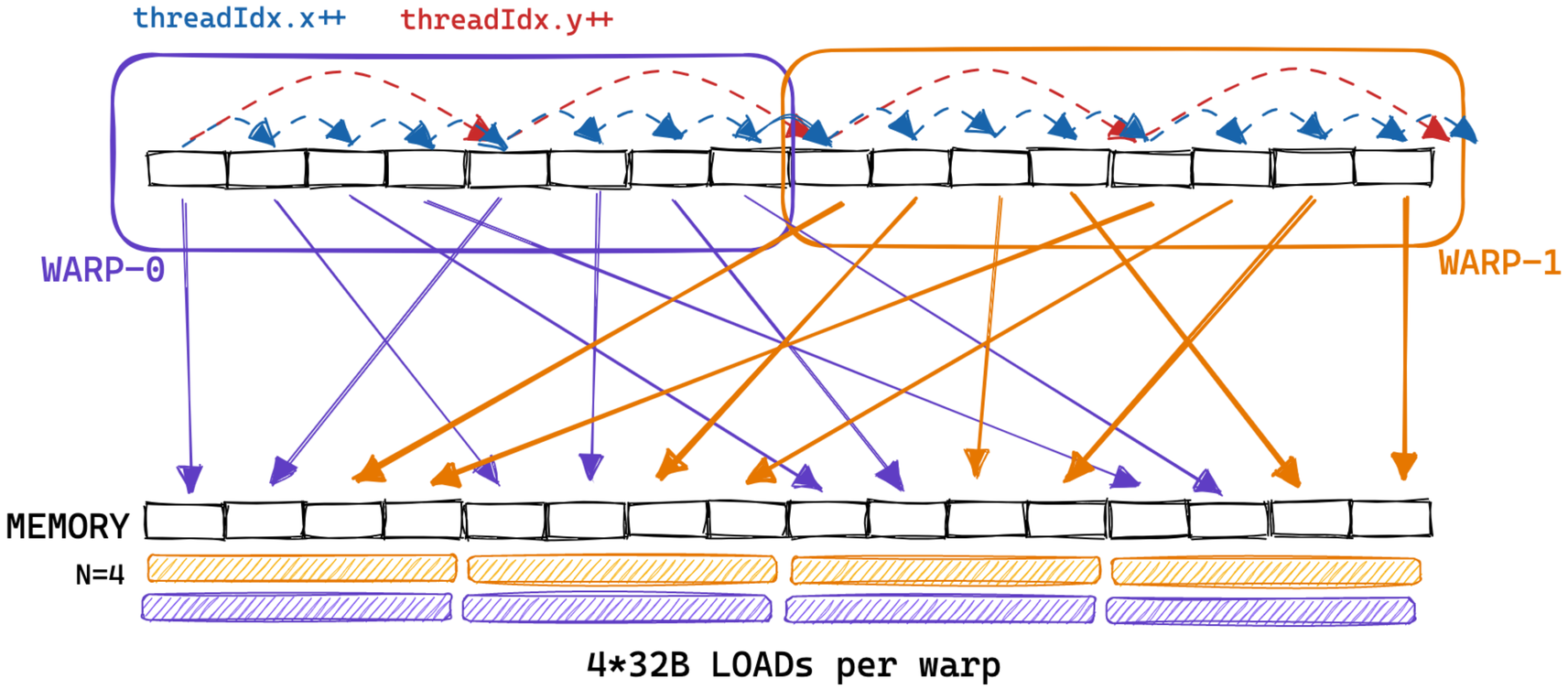
This is what we wrote

```
__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

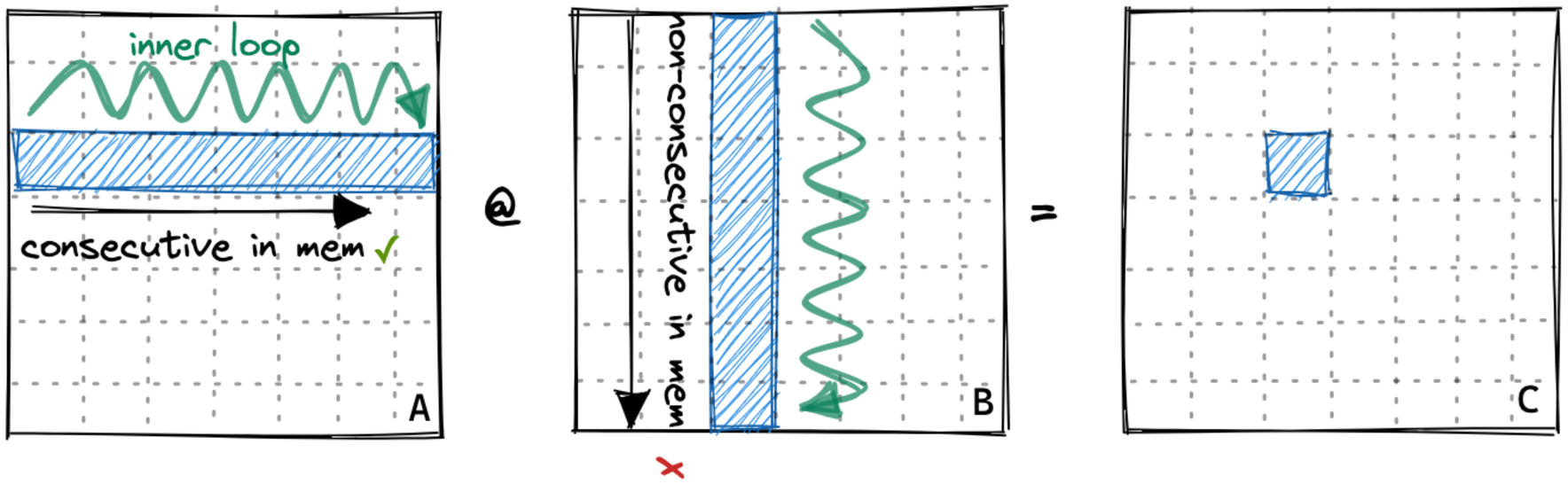
This is what we get

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



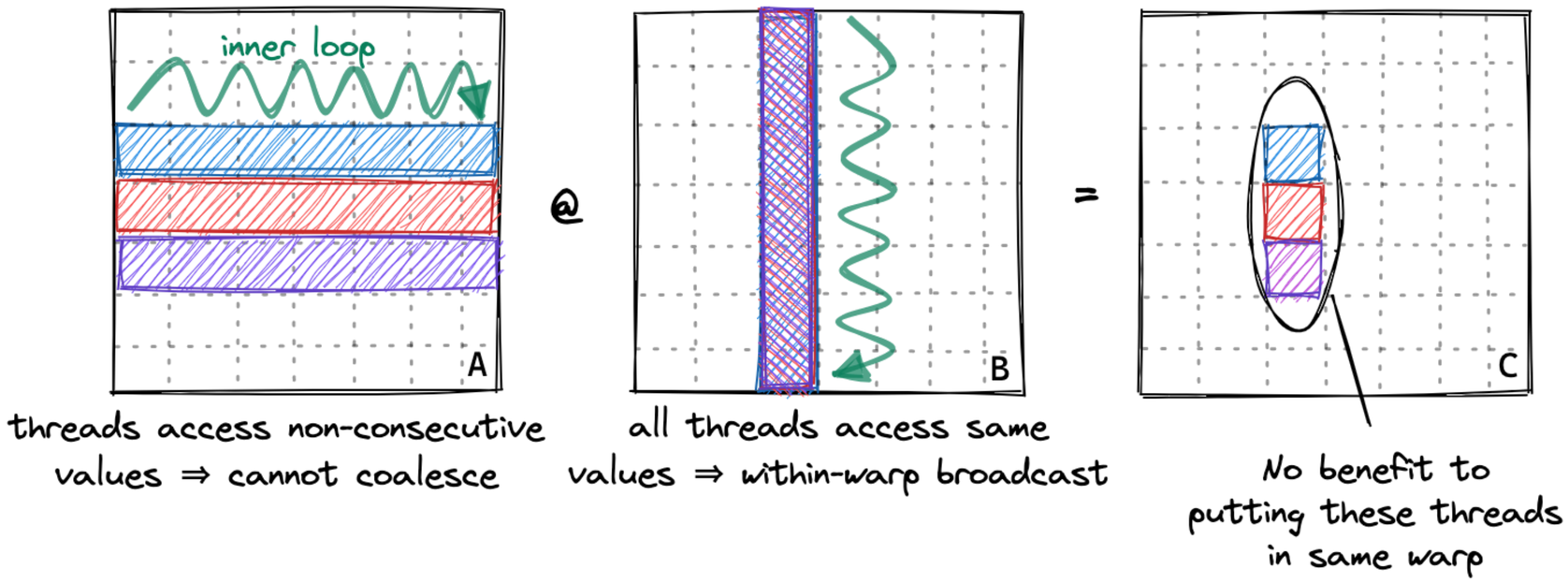
Thread Organization

A single thread



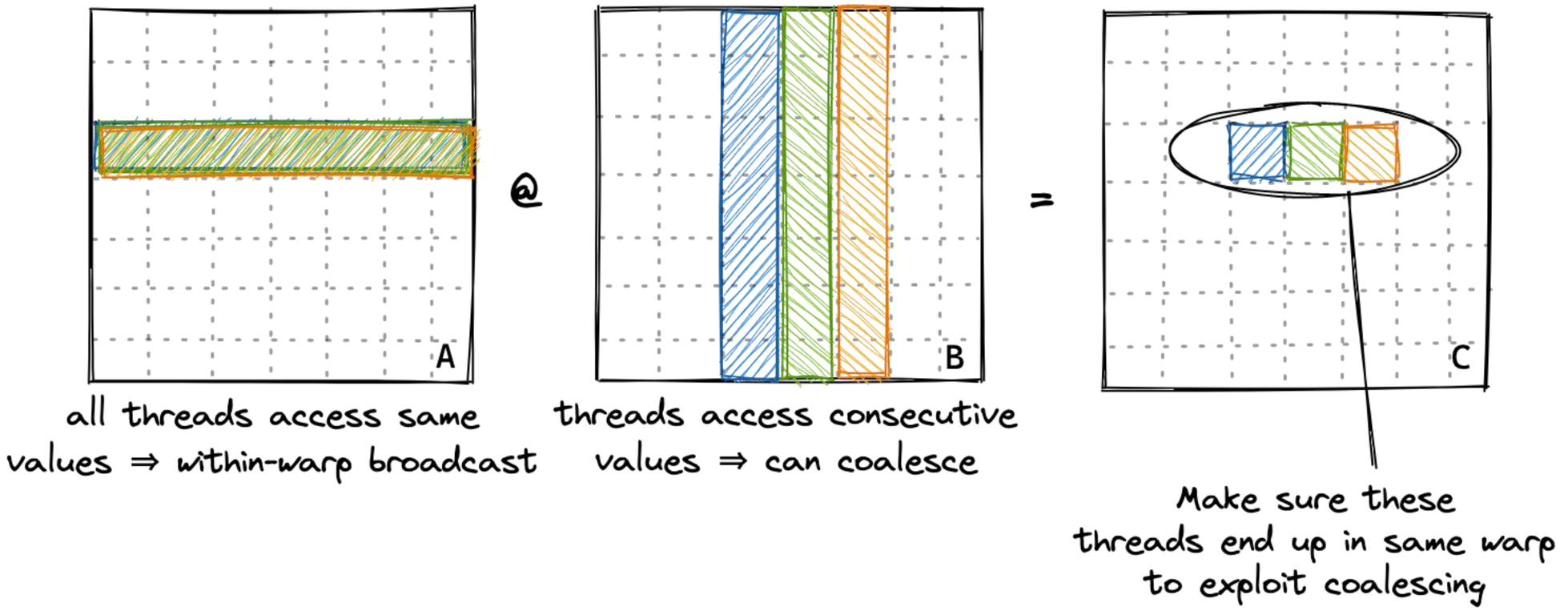
Thread Organization

A wrap of thread (Why is this bad?)



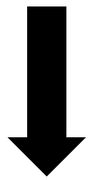
Thread Organization

A wrap of thread (what we want)



Thread Organization

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



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

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);









}
```

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.

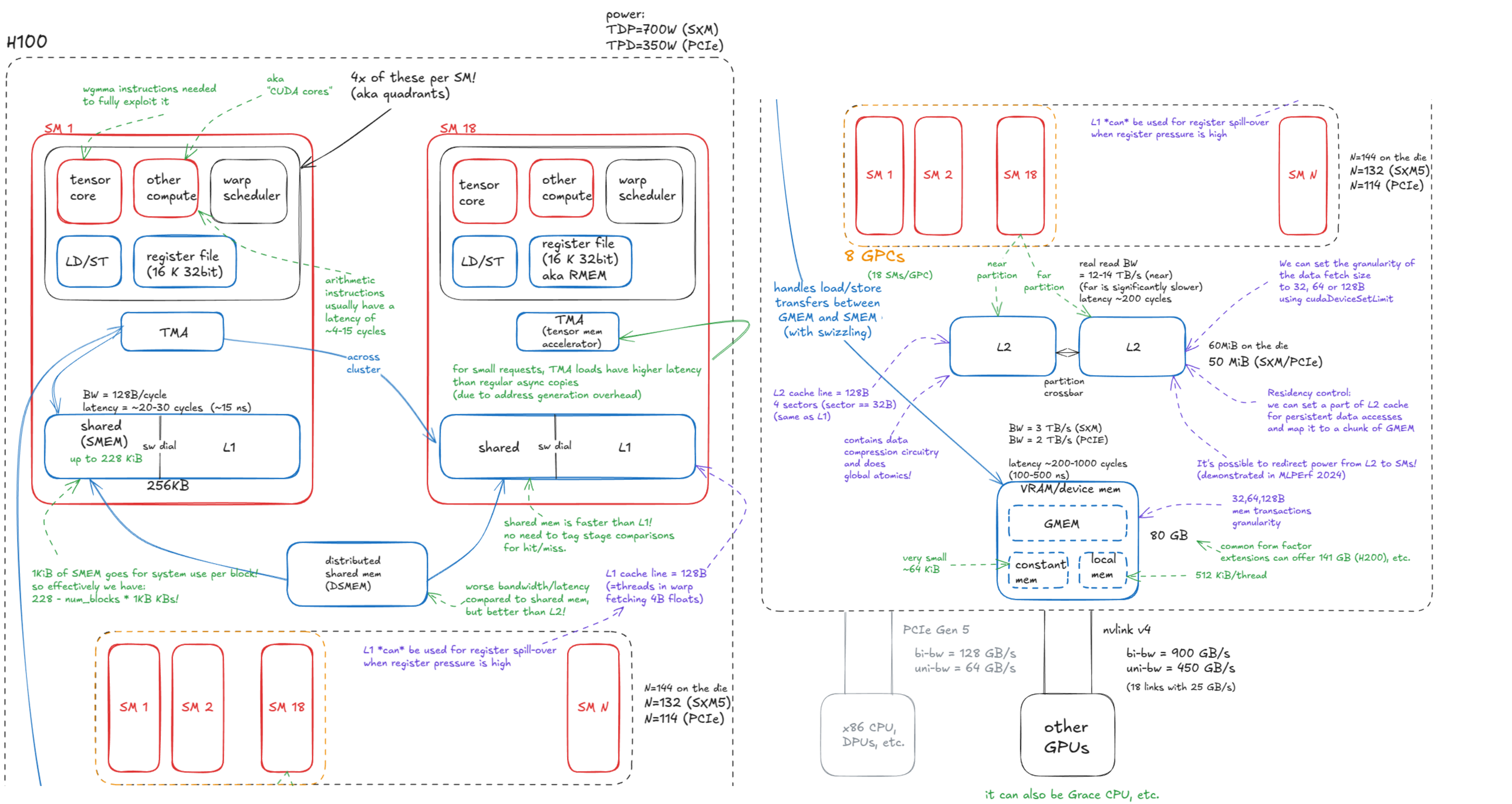
```
Max size: 4096
dimensions(m=n=k) 128, alpha: 0.5, beta: 3
Average elapsed time: (0.000008) s, performance: (  513.5) GFLOPS. size: (128).
dimensions(m=n=k) 256, alpha: 0.5, beta: 3
Average elapsed time: (0.000014) s, performance: ( 2401.4) GFLOPS. size: (256).
dimensions(m=n=k) 512, alpha: 0.5, beta: 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)
wgmma instructions needed to fully exploit it
aka "CUDA cores"
4x of these per SM!
(aka quadrants)
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)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
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.

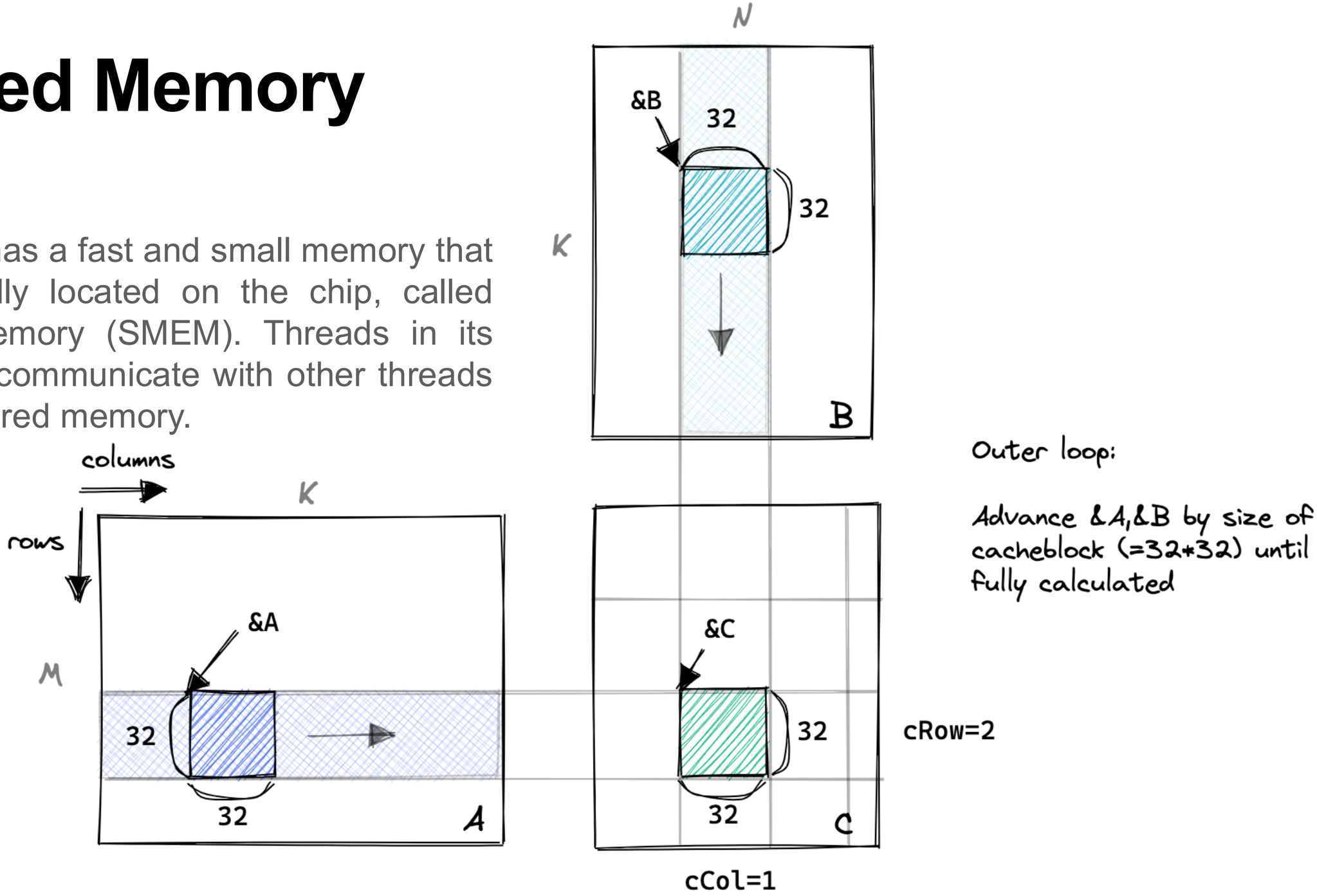
Transformation Equivalence

```
for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n++) {
        float tmp = 0.0f;  // accumulator
        for (int k = 0; k < K; k++) {
            tmp += A[m][k] * B[k][n];
        }
        C[m][n] = tmp;
    }
}
```

```
for (int k = 0; k < K; k++) {
    for (int m = 0; m < M; m++) {
        float a = A[m][k];  // reuse
        for (int n = 0; n < N; n++) {
            C[m][n] += a * B[k][n];
        }
    }
}
```

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 from 0.25 FLOPS/Byte to 8 FLOPS/BYTE.

```
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).
```

Matrix Multiplication

Welcome to the real world!

FP32 CUDA cores offer approximately 60 TFLOP/s computing power. However, the actual performance is 486.7 GFLOPS, and the run time is 273.527ms.

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

–Tom Jerry

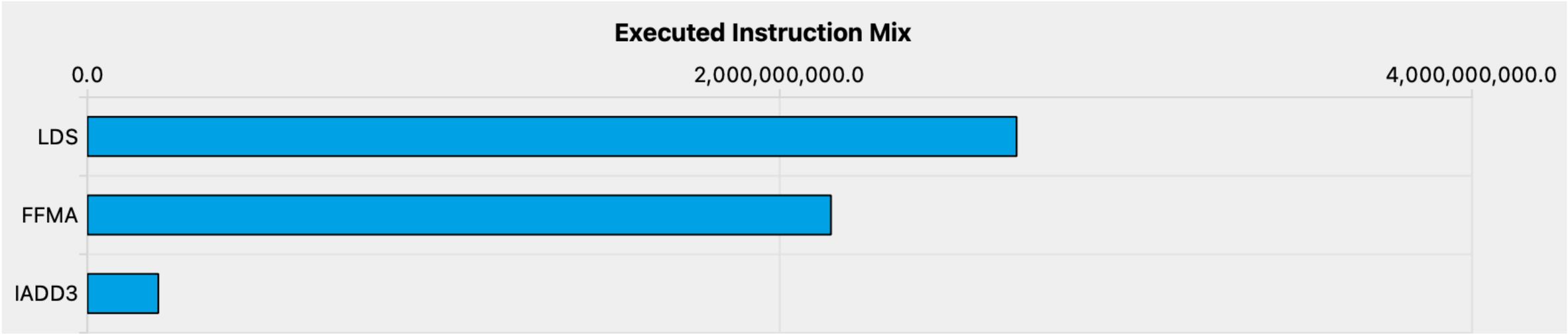
Where is the bottleneck now?

–Tom Jerry

Lots of memory accesses

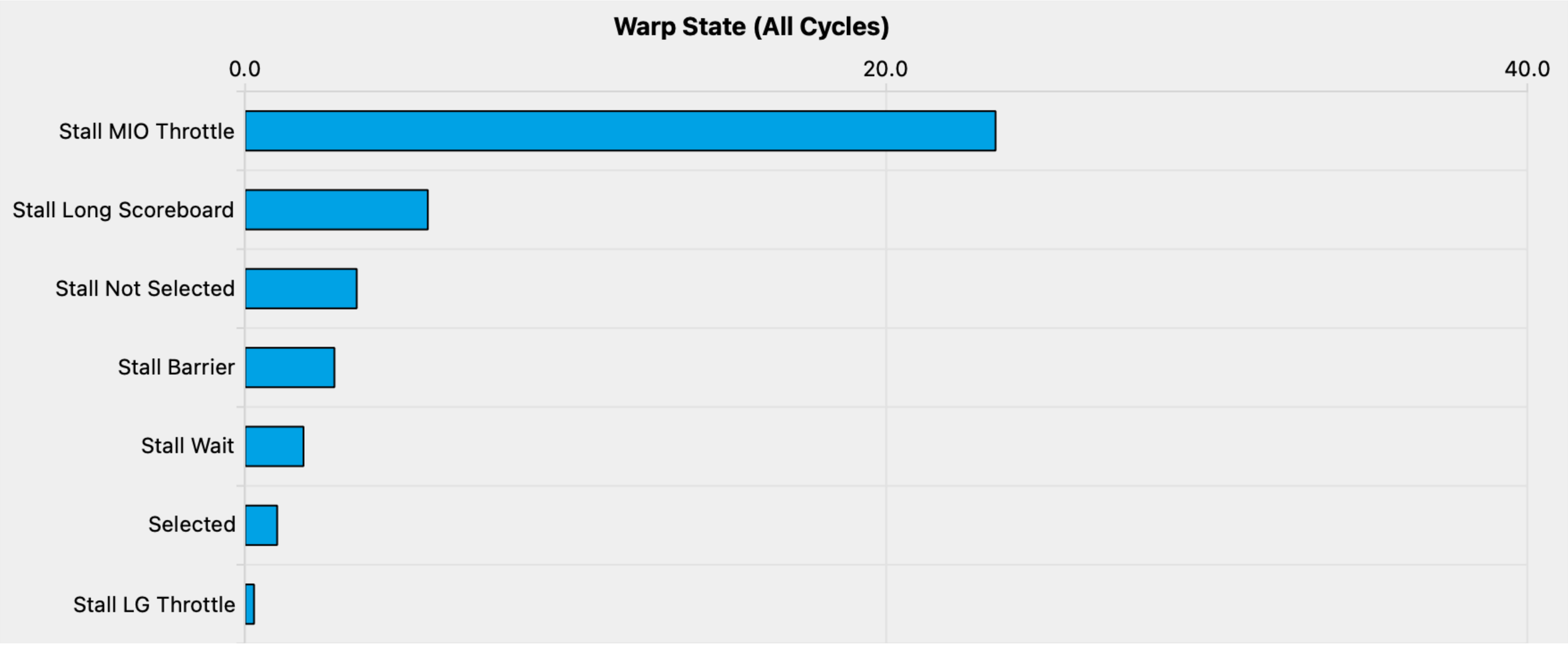
Still memory bound

```
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

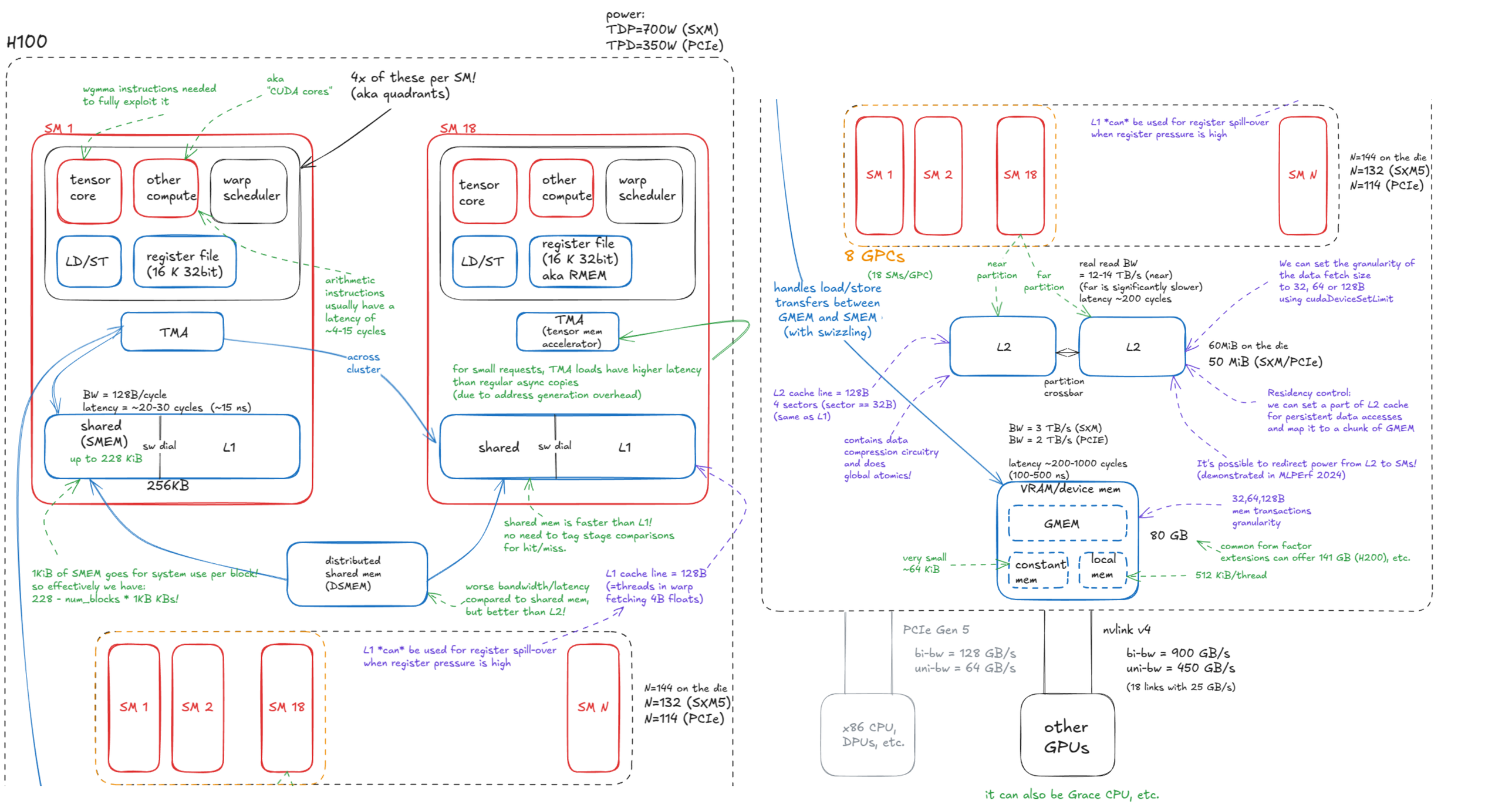
Still memory bound

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.

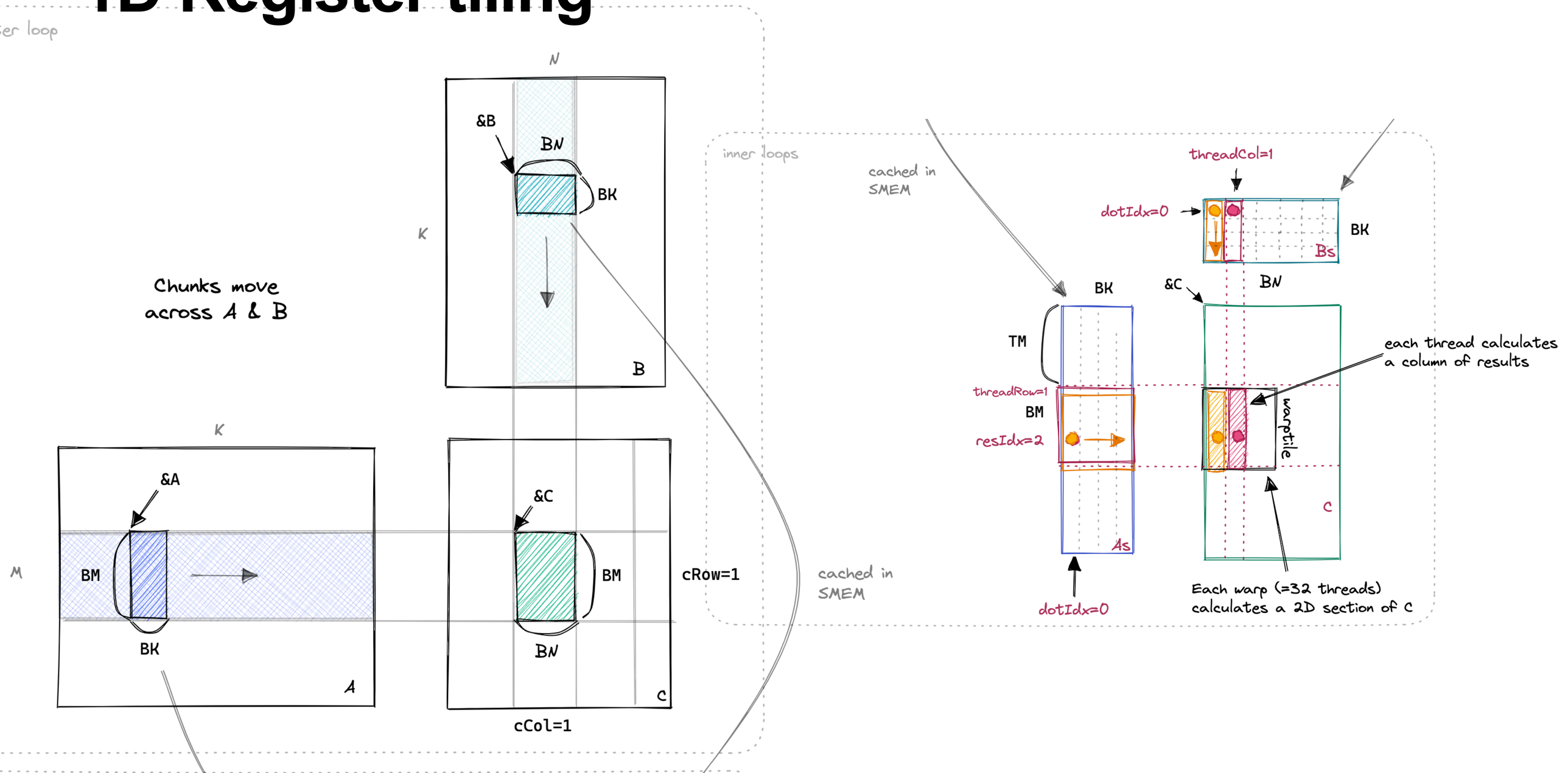
Lots of memory accesses

Still memory bound

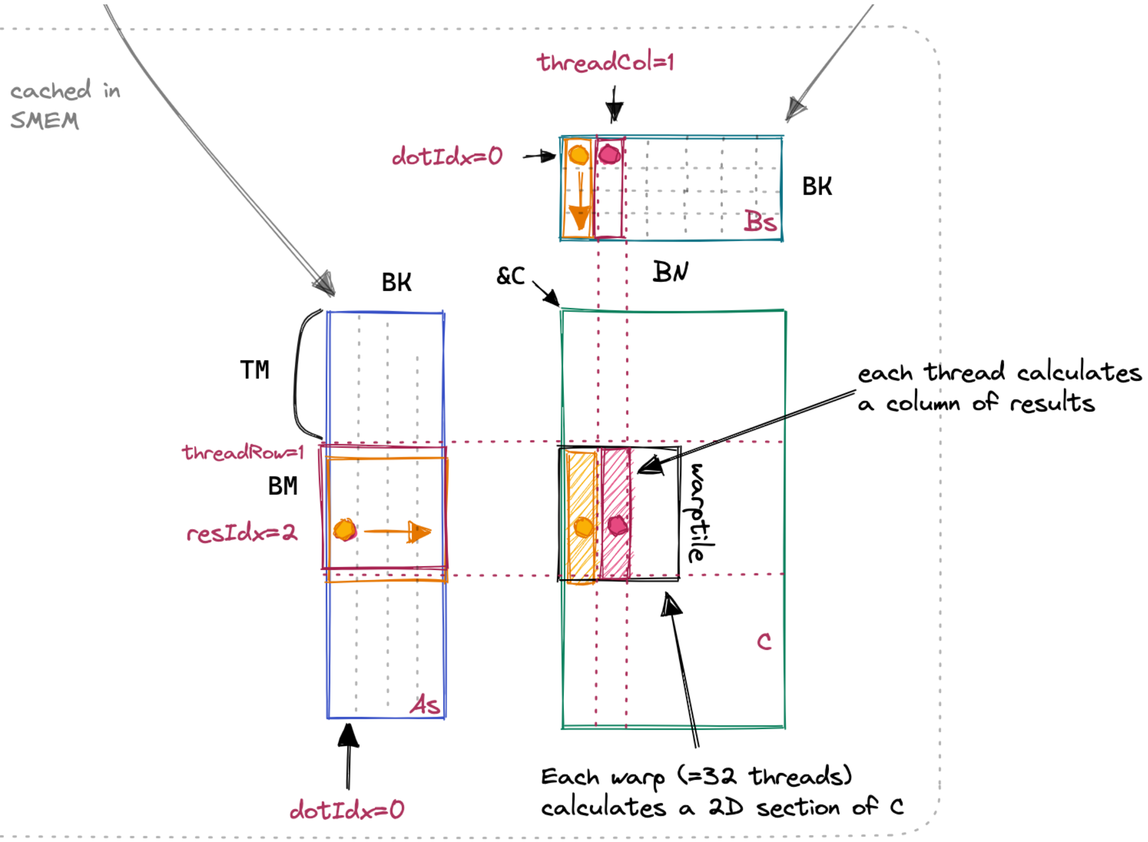
To reduce the number of SMEM instructions issued, we can increase the amount of work performed per thread—for example, by having each thread compute multiple output elements. This allows more computation to be done in registers and reduces reliance on shared memory.


H100
power:
TDP=700W (SXM)
TPD=350W (PCIe)
wgmma instructions needed to fully exploit it
aka "CUDA cores"
4x of these per SM!
(aka quadrants)
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)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
N=144 on the die
N=132 (SXM5)
N=114 (PCIe)
SM 1
SM 2
SM 18
L1 *can* be used for register spill-over when register pressure is high
SM N
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();
}
```

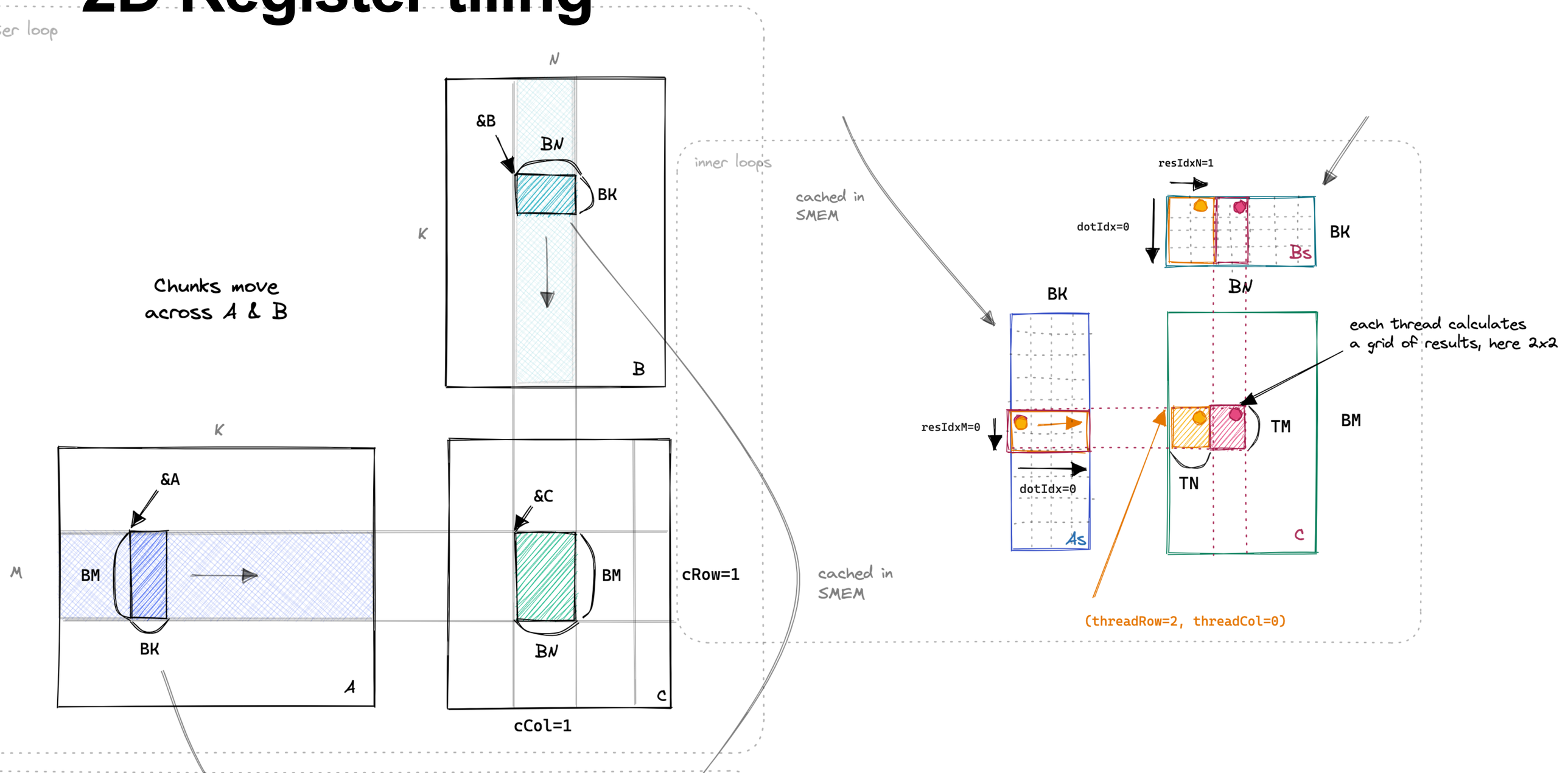
Matrix Multiplication

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



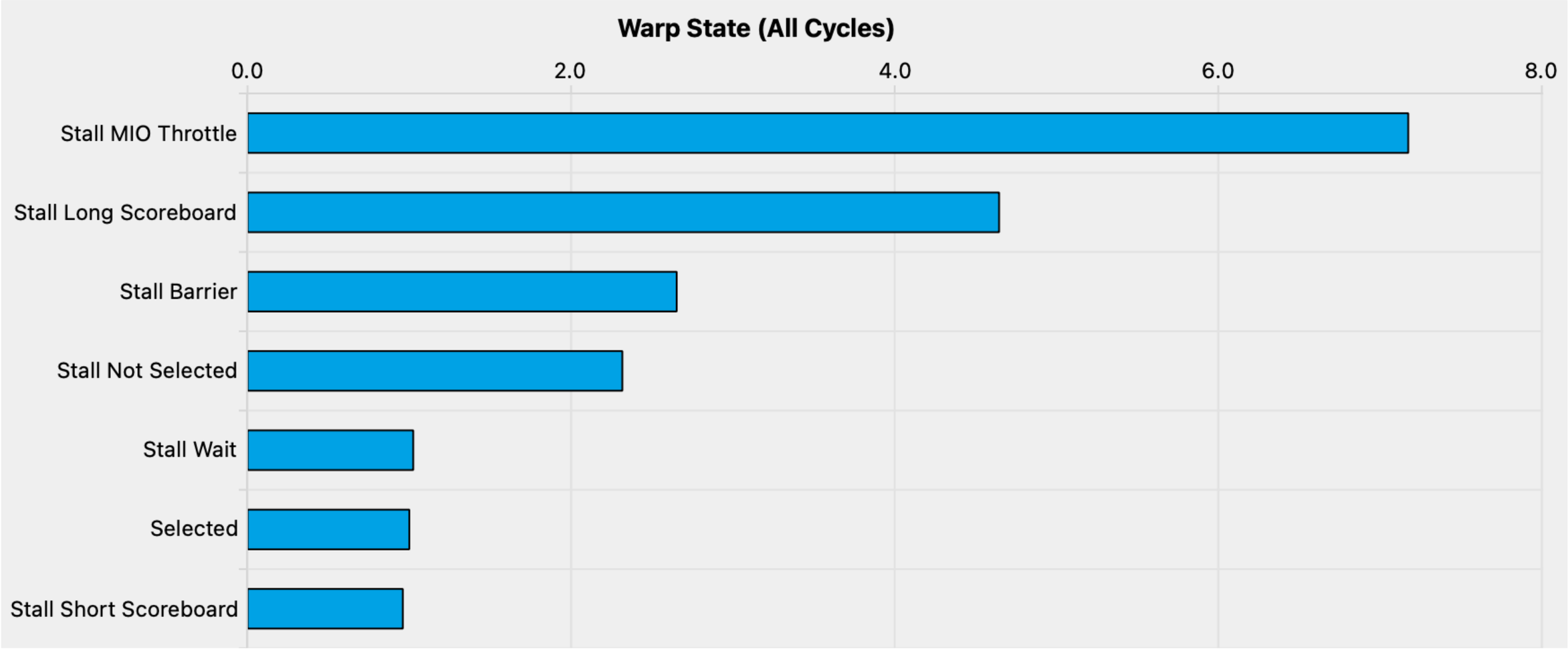
Matrix Multiplication

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).
```

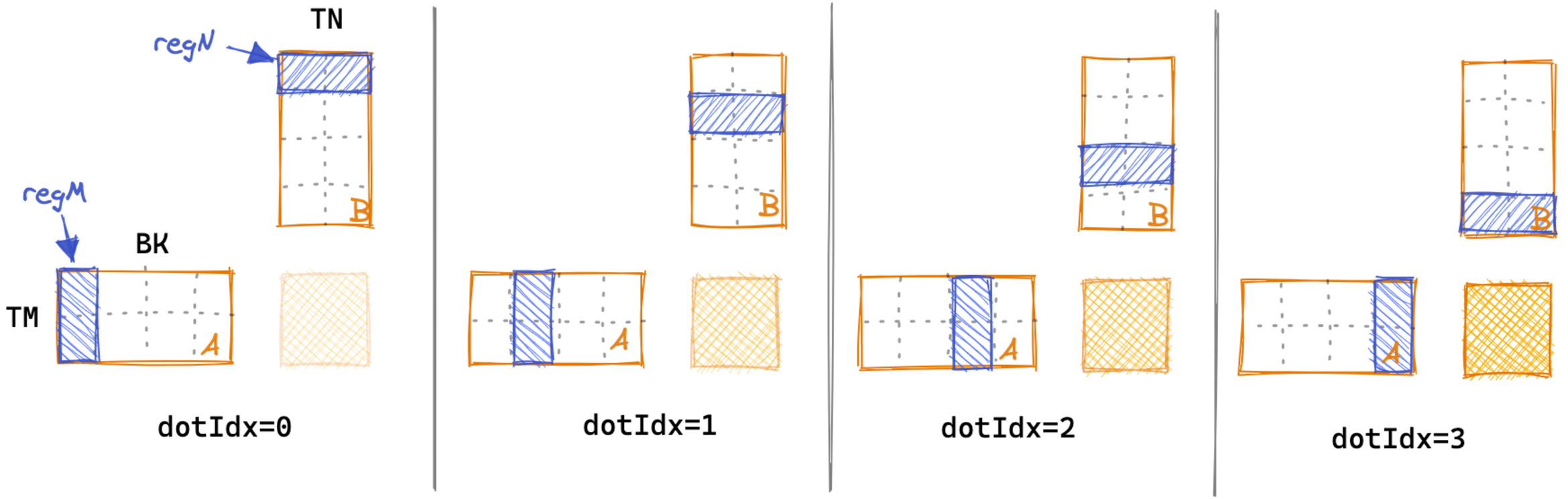
Memory access savings



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

–Tom Jerry

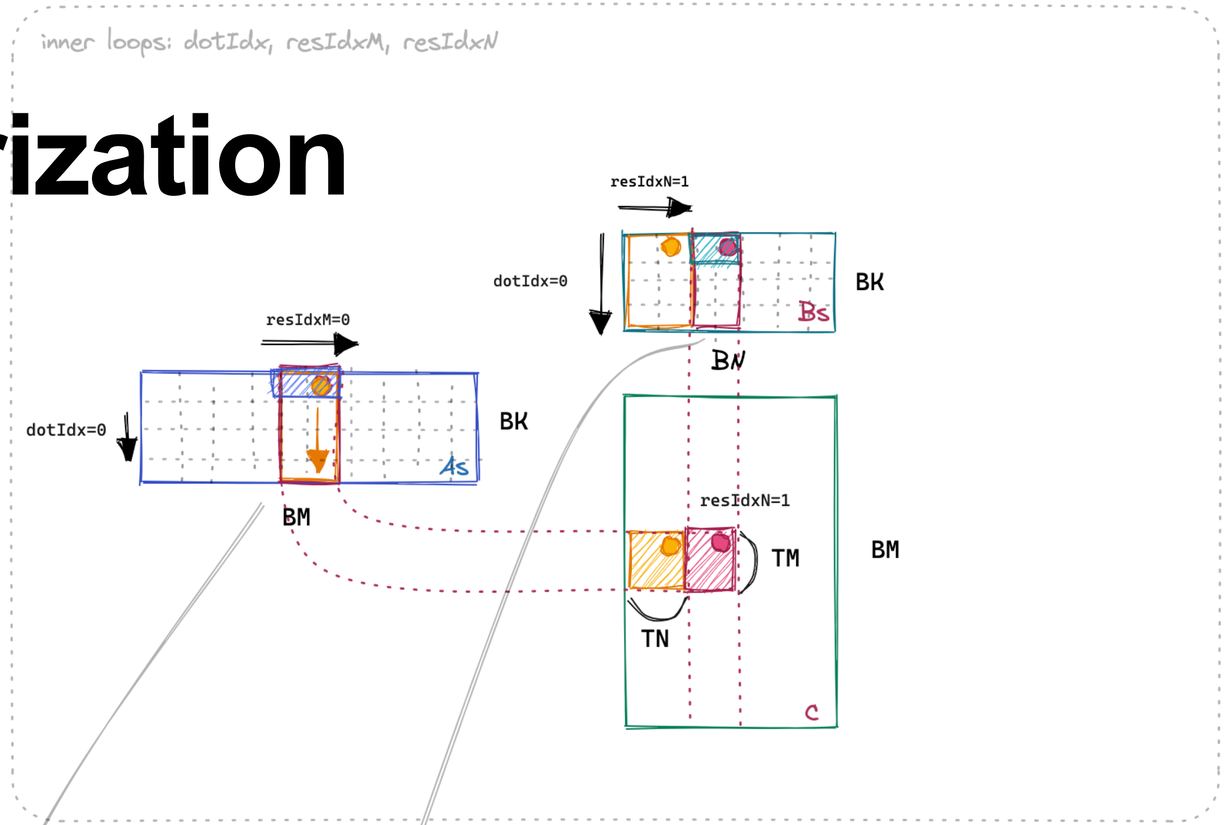
Unrolled dotIdx loop:



at each timestep, load the 4 relevant As&Bs entries into regM and regN registers, and accumulate outer product into threadResults.

Benefit: We only issue 16 SMEM loads in total

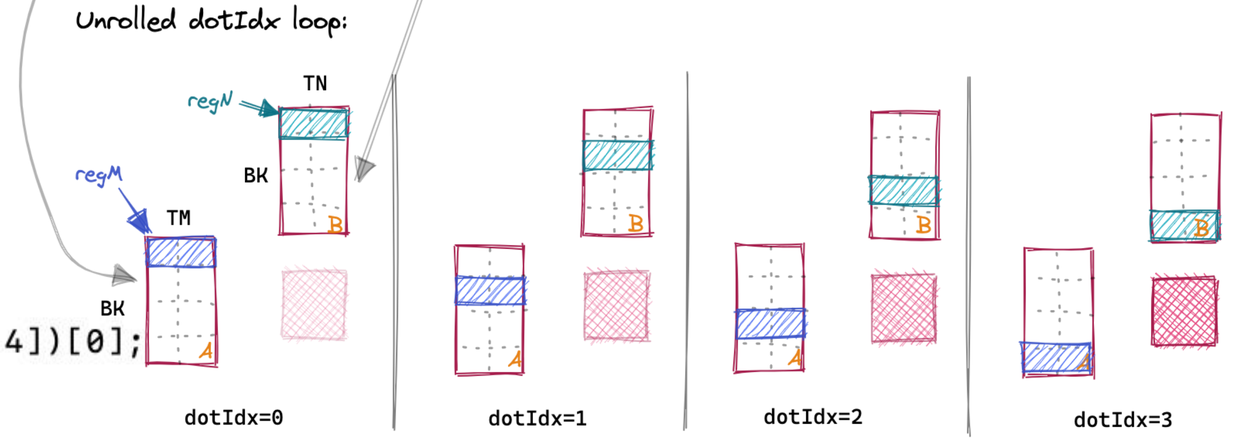
Memory Access Vectorization



🔷 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

Lesson 4: It’s hard to beat cuBLAS on its own turf. So, choice your battles wisely.

–Tom Jerry