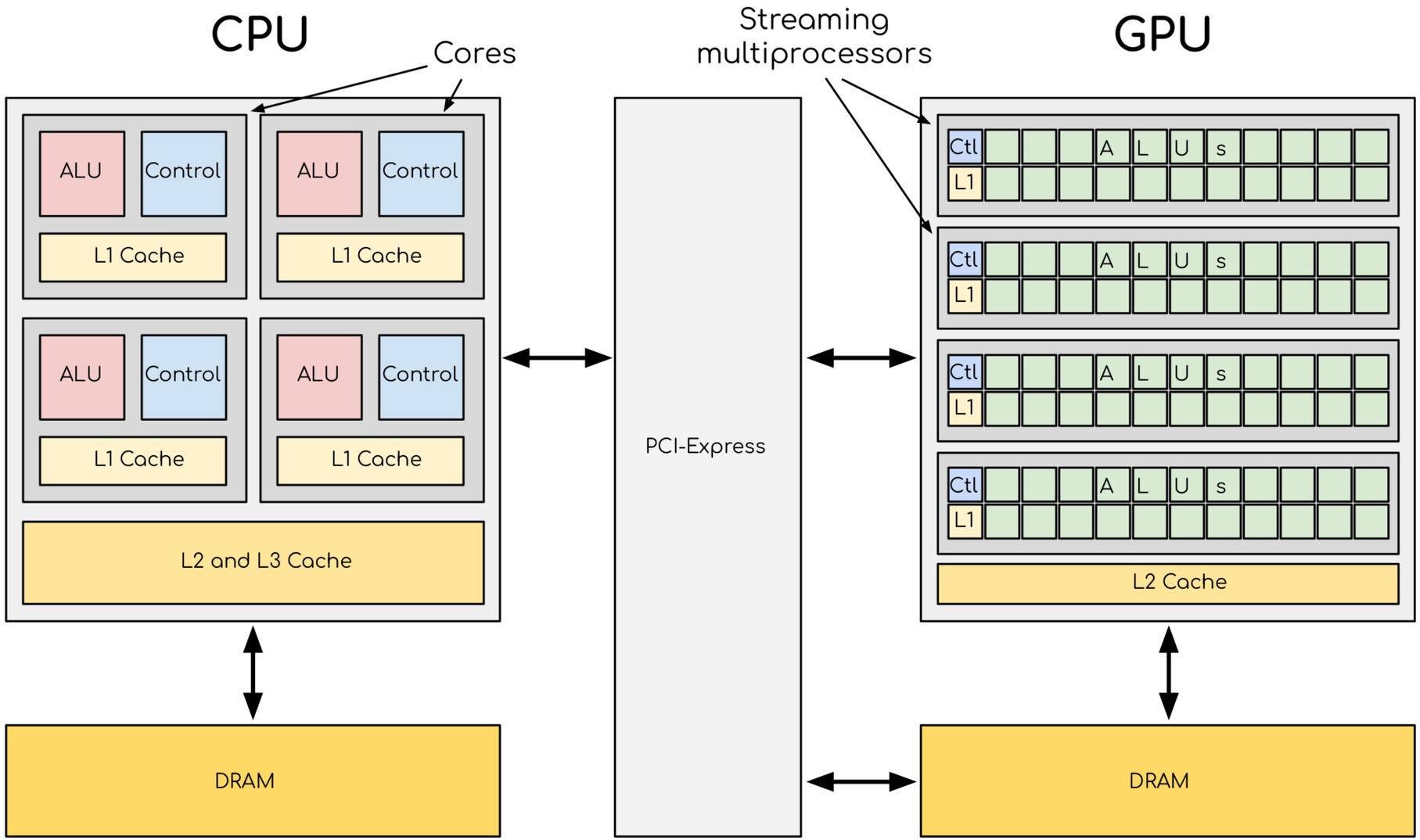
Machine Learning Systems

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

Acknowledgement

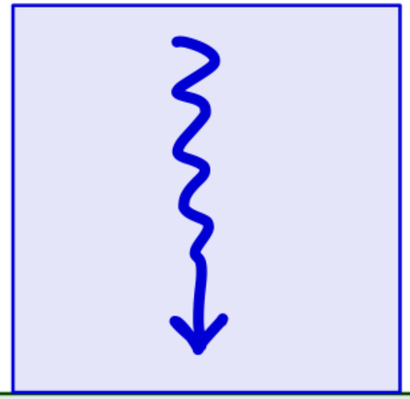
Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community.

How to program GPGPU?



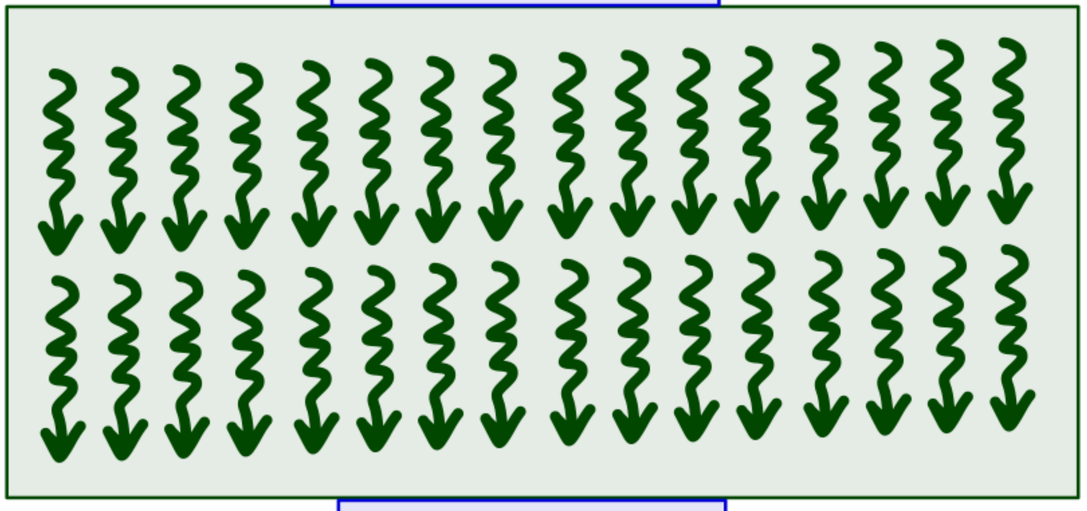
GPGPU as a really powerful accelerator

CPU thread

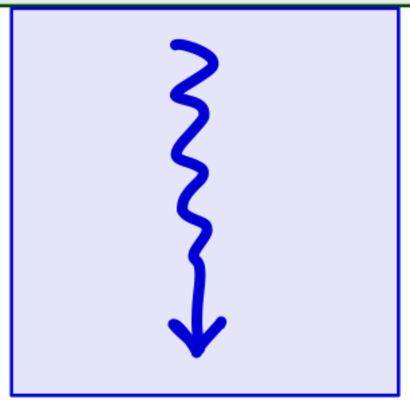


- Prepare data (matrices A, B, C).
- Copy data from RAM to VRAM.

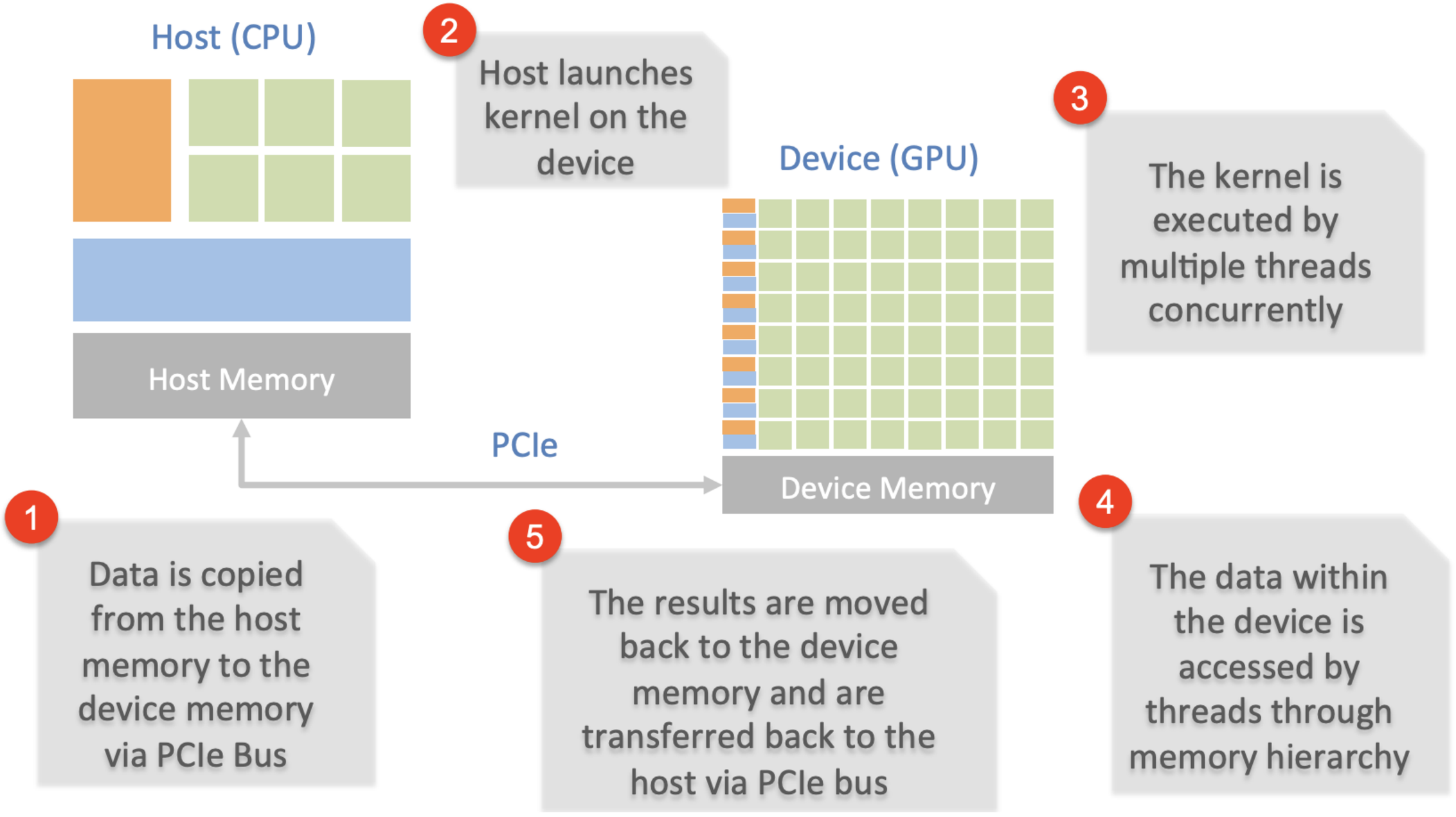
GPU threads



- 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

Kernel

A **kernel** is a function written in CUDA C/C++ (or another GPU programming language) that runs on the GPU and is executed **in parallel by many threads**.

- Declared with the `__global__` qualifier.
- Launched from the host (CPU) using the special syntax:

```
myKernel<<<numBlocks, threadsPerBlock>>>(arguments);
```

- When launched, CUDA creates a **grid** of threads (organized into blocks), and *each thread executes the same kernel function*, but on different data.

```cpp
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Kernel

```
// HOST code to queue kernel
simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

__global__ specifies kernel

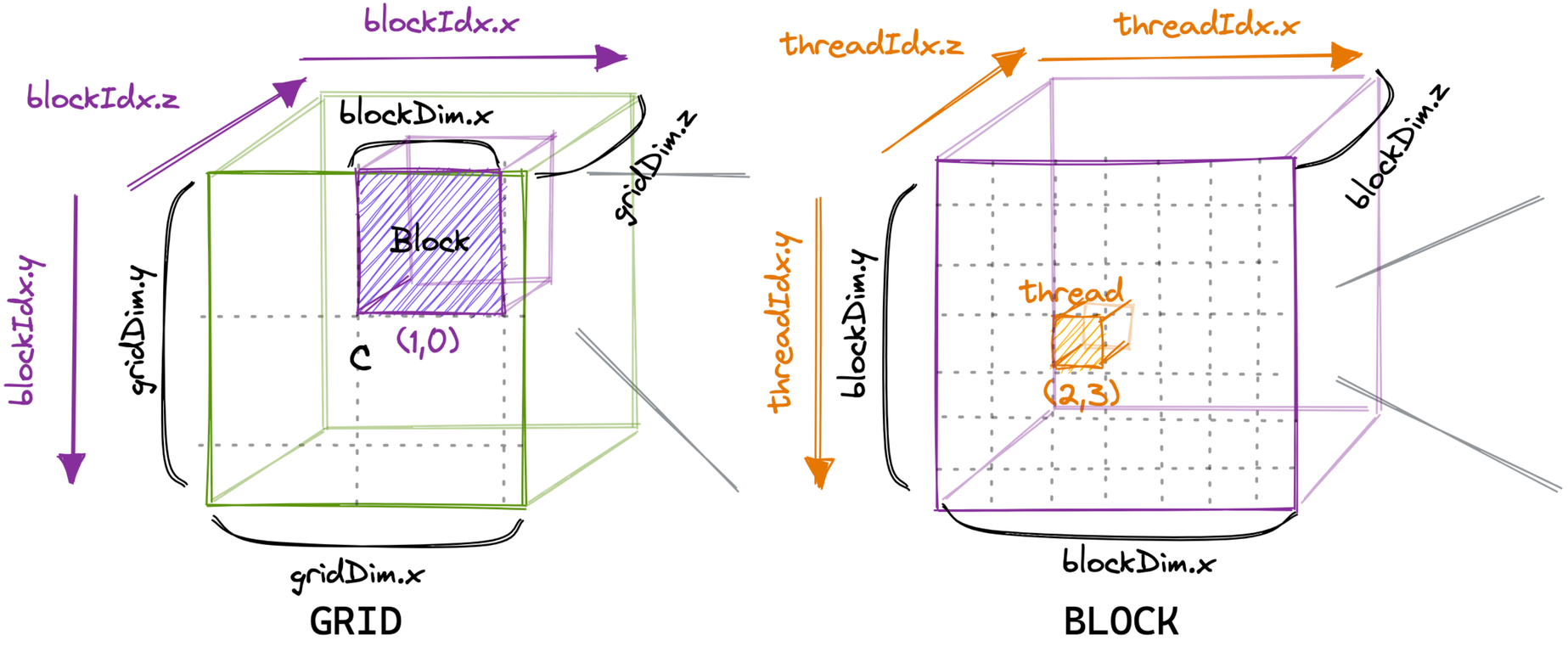
ThreadIdx & blockIdx determine thread rank that is mapped to array index

```
__global__ void simpleKernel(int N, float *d_a){

    // Convert thread and thread-block indices into array index
    const int n  = threadIdx.x + blockDim.x*blockIdx.x;

    // If index is in [0,N-1] add entries
    if(n<N)
      d_a[n] = n;
}
```

Action performed by each thread

Key observation: the loops are implicitly executed by thread parallelism and *do not* appear in the CUDA kernel code.

Thread Organization



a single thread of computation, minding its own business

THREAD

Now, let’s start writing CUDA code

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.

NVCC

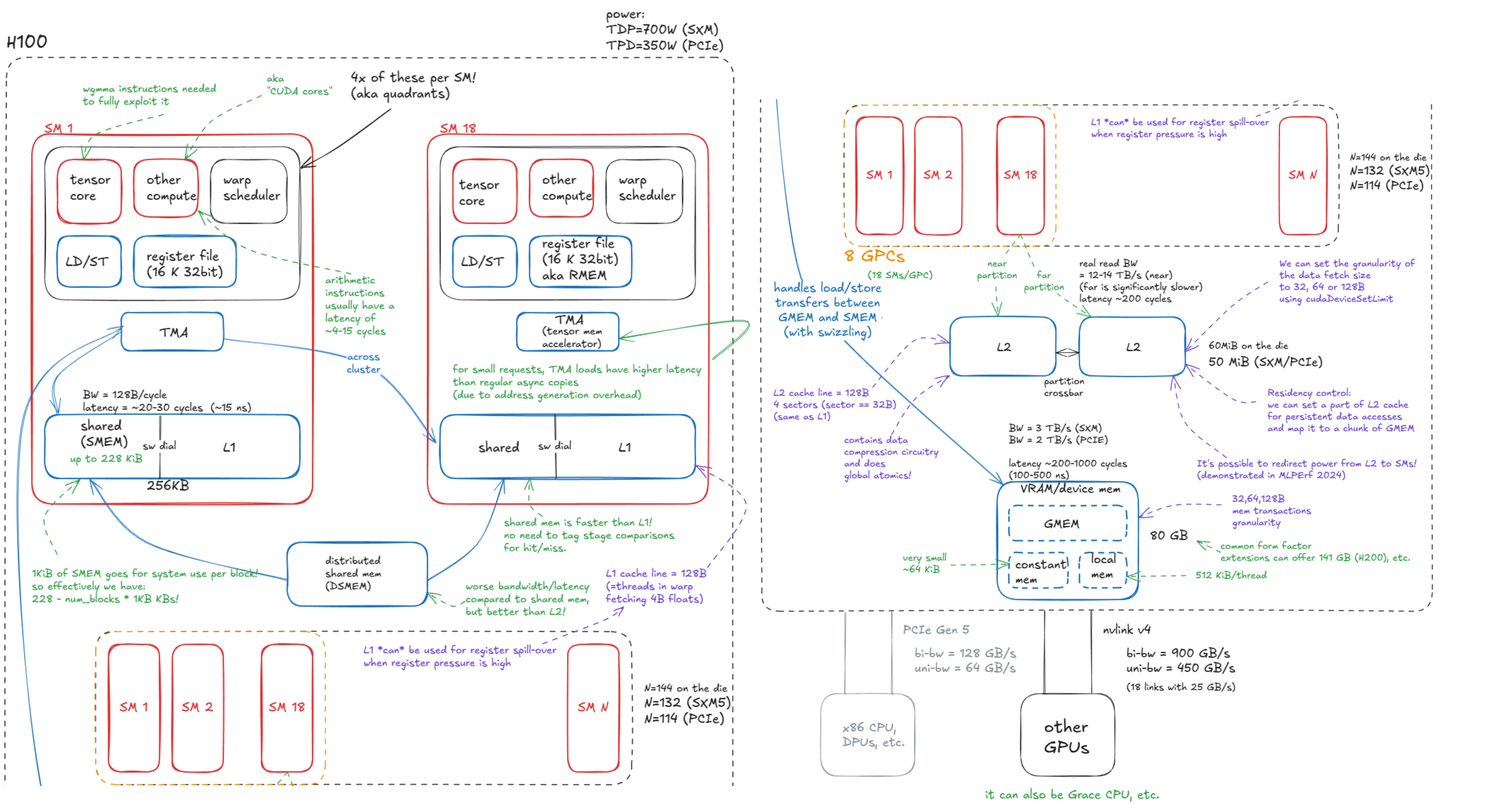
Tool	Function
cudafe	Separates host/device code, emits stubs & IR
ptxas	Assembles PTX → SASS GPU binary
fatbinary	Bundles multiple GPU binaries into one container
nvlink	Links host and device binaries
gcc / clang	Compiles host C/C++ source code

NVCC

Option	Description
`-o <file>`	Specify output filename
`-c`	Compile only, do not link
`-arch=sm_XX`	Target GPU architecture (e.g., `sm_89`, `sm_90`)
`--ptx`	Generate PTX assembly
`--cubin`	Generate GPU binary file
`-Xptxas -v`	Show register/shared-memory usage
`-G -g`	Debug mode (keep symbols)
`-O2 / -O3`	Enable optimizations

Stage	File	Produced By	Description
1 Source	.cu	User	Original CUDA source file containing host (CPU) and device (GPU) code. nvcc starts here and splits the file into separate compilation paths.
2 Preprocessing	.cpp1.i, .cpp4.ii	Preprocessor	Preprocessed C++ host code, macros and headers expanded, fed to a normal C++ compiler. Preprocessed device-side C++ code, for device compilation.
3 Front-end	.cudafe1.c, .stub.c, .gpu	cudafe	Generates host stubs + device IR.
4 Host Compile	.o	gcc / clang	Standard host object file compiled by gcc or clang. Contains CPU code and references to GPU stubs.
4′ Device Compile	.ptx	nvcc	PTX (*Parallel Thread Executiona*) virtual GPU ISA generated by the CUDA front end, human-readable assembly later be assembled into hardware-specific binary.
5 GPU Assembler	.sm_XX.cubin	ptxas	GPU binary for a specific compute capability (e.g., sm_52, sm_90). Assembles PTX into SASS (real GPU machine code).
6 Bundle	.fatbin, .fatbin.c	fatbinary	“Fat binary” — a bundle containing one or more .cubin binaries and possibly PTX. It enables running the same executable on multiple GPU architectures.
7 Device Link	.dlink.o	nvlink	Intermediate files created during device linking (-rdc=true). They merge multiple .cu translation units that call each other’s kernels before the final link step.
8 Executable	none / .exe	nvlink + gcc	Final ELF/PE binary with embedded GPU code.

CUDA C/C++ → PTX (virtual ISA) → SASS (machine ISA) → GPU execution


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.

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 (Compiler planned)

PTX (Parallel Thread Execution) is NVIDIA's virtual GPU instruction set architecture (ISA) — an intermediate layer between CUDA C/C++ source code and the hardware-level SASS machine code. It provides a human-readable, typed, and portable representation of GPU programs, defining how threads, memory spaces, and arithmetic operations behave in a device-independent way. PTX expresses what the compiler intends the hardware to do—including data movement, parallel execution, and synchronization—while abstracting away hardware-specific details like pipelines, cache policies, and timing.

SASS (GPU reality)

SASS (Streaming Assembler) is NVIDIA's hardware-level GPU instruction set architecture, representing the actual machine code executed by the Streaming Multiprocessors (SMs). Each SASS instruction is a 128-bit binary operation that encodes the precise arithmetic, memory, and control actions the GPU performs, including fused operations, cache policies, and register usage. Unlike PTX, which is virtual and portable, SASS is architecture-specific (e.g., sm_80, sm_90) and reveals how the hardware executes a kernel—making it essential for low-level performance tuning, scheduling analysis, and understanding real GPU behavior.

PTX and SASS

Analogy

- **PTX ≈ Java bytecode** — portable, virtual.
- **SASS ≈ x86 assembly** — concrete, hardware-specific.

Key insight:
PTX expresses *intent*, SASS exposes *reality*. Profiling and performance analysis must look at SASS to see the true instruction mix, cache policy, and resource usage.

```
CUDA C/C++  →  PTX (virtual ISA)  →  SASS (machine ISA)  →  GPU execution
```

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 ISA at a glance

Category	Examples	Purpose
Data Move	`ld`, `st`, `mov`, `cvt`	Move / convert data across registers & memory.
Arithmetic	`add`, `sub`, `mul`, `mad`, `fma`	Integer & floating-point math operations.
Logic / Bit	`and`, `or`, `xor`, `not`, `shl`, `shr`	Bitwise & shift operations.
Control Flow	`bra`, `@p bra`, `call`, `ret`, `exit`	Branching & predicated execution.
Comparison	`setp.eq`, `setp.lt`, `setp.ge`	Set predicate registers for conditions.
Memory Spaces	`.global`, `.shared`, `.local`, `.const`	Specify where data is loaded/stored.
Sync & Parallel	`bar.sync`, `membar`, `shfl.sync`	Coordinate threads & memory ordering.
Tensor / Async	`mma.sync`, `cp.async`	Tensor-core and async copy operations.

SASS ISA at a glance

Category	Examples	Purpose
Arithmetic	`FADD`, `FFMA`, `IMAD`, `IADD3`	Real GPU math ops — fused, hardware-optimized.
Logic / Bit	`LOP3`, `SHF`, `BFE`, `BMSK`	Bitwise logic and shift operations.
Memory Access	`LDG.E`, `STG.E`, `LDS`, `STS`, `LDC`	Load/store from global, shared, or constant memory (with cache policies).
Control Flow	`BRA`, `@P0 EXIT`, `SSY`, `SYNC`	Branching, predication, warp reconvergence.
Predicates	`ISETP`, `FSETP`, `@P`	Conditional execution on hardware predicate flags.
Special Regs	`S2R`, `S2UR`, `ULDC`	Move from thread/block registers or uniform regs.
Tensor / Async	`HMMA`, `LDGSTS`, `CPAsync`	Tensor-core matrix ops & async copy pipelines.
Sync / Barrier	`BAR.SYNC`, `MEMBAR`	Thread/block synchronization and memory fences.

Parameter Loads

PTX	SASS	Meaning
`ld.param.u64 %rd1, [_param_A];`	`LDC.64 R2, c[0x0][0x210];`	Load kernel parameter A pointer from constant space.
`ld.param.u64 %rd2, [_param_B];`	`LDC.64 R4, c[0x0][0x218];`	Load B pointer.
`ld.param.u64 %rd3, [_param_C];`	`LDC.64 R6, c[0x0][0x220];`	Load C pointer.
`ld.param.u32 %r2, [_param_N];`	`ULDC UR4, c[0x0][0x228];`	Load N (vector length) into uniform register.

Thread/Block Indexing

PTX	SASS	Explanation
`mov.u32 %r3, %ctaid.x;`	`S2UR UR4, SR_CTAID.X;`	BlockIdx.x → uniform register.
`mov.u32 %r4, %ntid.x;`	`LDC R9, c[0x0][RZ];`	BlockDim.x from launch constant (c[0]).
`mov.u32 %r5, %tid.x;`	`S2R R0, SR_TID.X;`	ThreadIdx.x → R0.
`mad.lo.s32 %r1, %r3, %r4, %r5;`	`IMAD R9, R9, UR4, R0;`	Global index i = blockIdx.x * blockDim.x + threadIdx.x.

Registers

Type	Purpose
General (R0–R254)	Standard per-thread registers used for integers, floats, and addresses (32-bit each; paired for 64-bit). They form the main computational workspace for every thread.
Predicate (P0–P7)	Boolean flags per thread, used for **predicated execution** (e.g., conditional instructions via `@P`, set by `ISETP`/`FSETP`).
Uniform (UR0–UR15)	Shared across a warp; store **warp-uniform values** (e.g., kernel parameters, descriptors, block indices). Save register space and broadcast same data to all threads.
Special (SR_*)	**Read-only hardware registers** containing built-in thread/block state (`SR_TID`, `SR_CTAID`, `SR_NTID`, `SR_LANEID`, `clock`).
Tensor (implicit)	Internal per-thread-group fragments used by **Tensor Core MMA instructions** (`HMMA`, `WMMA`), not directly programmer-visible.

Bounds Check

PTX	SASS	Explanation
`setp.ge.s32 %p1, %r1, %r2;`	`ISETP.GE.AND P0, PT, R9, UR4, PT;`	Predicate P0 = (i ≥ N).
`@%p1 bra DONE;`	`@P0 EXIT;`	Predicated early exit for out-of-range threads.

Address Computation and Loads

PTX	SASS	Explanation
`cvta.to.global.u64 %rd4, %rd1;`	*(implicit)*	PTX's address-space cast is redundant in SASS.
`mul.wide.s32 %rd5, %r1, 4;`	`IMAD.WIDE R2, R9, 0x4, R2;`	Compute byte offset i×4 for A.
`add.s64 %rd6, %rd4, %rd5;`	(fused in same `IMAD.WIDE`)	Fused multiply-add to get &A[i].
`ld.global.f32 %f2, [%rd6];`	`LDG.E R3, desc[UR4][R2.64];`	Load A[i] (cache policy .E = evict-normal).

Computation and Store

PTX	SASS	Explanation
`add.f32 %f3, %f2, %f1;`	`FADD R9, R4, R3;`	Floating-point add: result = A[i] + B[i].
`cvta.to.global.u64 %rd9, %rd3;`	*(implicit)*	C pointer conversion.
`add.s64 %rd10, %rd9, %rd5;`	`IMAD.WIDE R6, R9, 0x4, R6;`	Compute &C[i].
`st.global.f32 [%rd10], %f3;`	`STG.E desc[UR4][R6.64], R9;`	Store C[i] with .E cache policy.
`ret;`	`EXIT;`	End of kernel.

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.

What we can learn from PTX

Category	What PTX Reveals	Why It Matters
Algorithmic intent	Shows each kernel's logical operations (`add`, `mul`, `mad`, `ld.global`, `st.shared`).	Lets you verify what the compiler generated from your CUDA C++ code.
Dataflow & memory spaces	Explicit `.global`, `.shared`, `.local`, `.const` address spaces.	Helps analyze where data lives and how it moves—key for optimizing memory hierarchy use.
Thread & block semantics	Access to `%tid`, `%ctaid`, `%ntid`, `%nctaid`.	Shows how per-thread and per-block indexing is computed.
Register usage (virtual)	Declared with `.reg` (e.g., `.reg .f32 %f<8>`).	Indicates compiler's variable allocation and potential register pressure.
Control flow	Structured branches, predication (`setp`, `@%p bra`), `ret`, `exit`.	Lets you inspect compiler-generated divergence handling.
Precision & rounding modes	Modifiers `.f32`, `.f64`, `.rn/.rz/.rm/.rp`.	Clarifies arithmetic precision and rounding—important for numerical correctness.
Parallel synchronization	`bar.sync`, `membar.cta`, `membar.gl`.	Shows explicit thread coordination points.
Compiler optimizations	Instruction fusions (`mad.lo.s32`, `fma.rn.f32`), loop unrolling, constant folding.	Lets you see what optimizations were applied or missed.
Portability boundary	PTX abstracts away architecture specifics (SM version).	Ensures same PTX can be JIT-compiled for new GPUs.

DeepSeek’s lession

CUDA = productivity.

PTX = performance control.

DeepSeek used PTX to bridge the gap for extreme-scale model efficiency.

DeepSeek’s lession

Aspect	Summary
Goal	Push GPU efficiency beyond CUDA’s default compiler by writing or modifying **PTX (Parallel Thread Execution)** code directly.
Why	CUDA and `ptxas` hide low-level control (cache behavior, scheduling, memory hints). DeepSeek needed **fine-grained tuning** for bandwidth-critical LLM kernels on H100/H800 GPUs.
Approach	Selectively replaced compiler-generated PTX in **hot kernels** with custom instructions controlling **cache policy** and **memory streaming behavior**.
Example	Used a custom PTX load instruction:`ptx ld.global.nc.L1::no_allocate.L2::256B %r, [%rd]; `→ Non-coherent global load, **bypasses L1 cache**, **fetches 256B via L2**.Reduces L1 pollution and improves large-tensor streaming performance.
Result	Higher effective memory bandwidth and improved throughput (reported ~5–10% gain in key kernels).
Takeaway	PTX gives access to **hardware-level cache & memory control** normally hidden by CUDA — useful for expert optimization, but fragile and hardware-specific.

Takeaway:
PTX is the **virtual GPU ISA** that exposes *how the compiler views your algorithm—*
use it to inspect **logic, memory usage, precision, and compiler transformations**,
but remember it hides **hardware scheduling, cache policies, and timing**, which only appear in **SASS**.

What cannot be deduced from PTX?

Category	What's Hidden	Why It Matters
Hardware Scheduling	Instruction timing, dual-issue behavior, pipeline mapping	Needed to predict latency & throughput — only visible in SASS.
Cache Policies	L1/L2 eviction mode, prefetching, descriptor usage	PTX omits cache hints that strongly affect memory performance.
Instruction Fusion	Whether `mul+add` became `FFMA` or `IMAD.WIDE`	Real instruction mix and FLOP count unknown until SASS.
Register Allocation	Physical register count, spills, and bank conflicts	Impacts occupancy and performance but assigned later by `ptxas`.
Tensor / Async Details	Exact tensor-core shape, async copy overlap	PTX shows intent (`mma.sync`, `cp.async`) but not micro-architecture behavior.
Control Timing	Warp scheduling, divergence cost, barrier placement	PTX abstracts away real warp execution flow.

What cannot be deduced from PTX?

PTX shows **what the compiler planned**, not **how the GPU will execute it**.

→ Use **SASS and profilers** (like Nsight Compute) to uncover the true performance behavior.

What we can learn from SASS (that PTX cannot)

Insight	What You Learn
True Instruction Mix	Fused ops (`FFMA`, `IMAD.WIDE`), real FLOP count.
Cache Behavior	Actual memory ops and policies (`LDG.E`, `STG.CS`, etc.).
Register Usage	Physical registers, spills, and occupancy impact.
Pipeline Mapping	Which hardware units (FP, INT, MEM, Tensor) are active.
Scheduling & Latency	Instruction order, predication, warp issue timing.
Tensor & Async Ops	Real tensor-core shapes (`HMMA.1688`) and async overlaps.

What we cannot fully deduce from SASS

Category	Hidden in Hardware
Instruction latency / throughput	Not encoded in SASS — depends on microarchitecture (e.g., FP32 FMA latency differs between Ampere and Hopper).
Pipeline mapping	You can't tell exactly which pipeline (INT, FP, MEM, SFU) executes each op without the official scheduling tables.
Warp scheduler behavior	Which of the four warp schedulers issued a given instruction, or the issue rate, is not visible.
Cache hierarchy details	Cache sizes, line sizes, replacement policies not shown.
Register bank conflicts	The hardware's internal register bank mapping is invisible.
Tensor core datapath widths	You see `HMMA.1688`, but the underlying MMA unit shape and fusion granularity are proprietary.
Clock domain, frequency, or power gating	Completely outside SASS view.

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)	✅

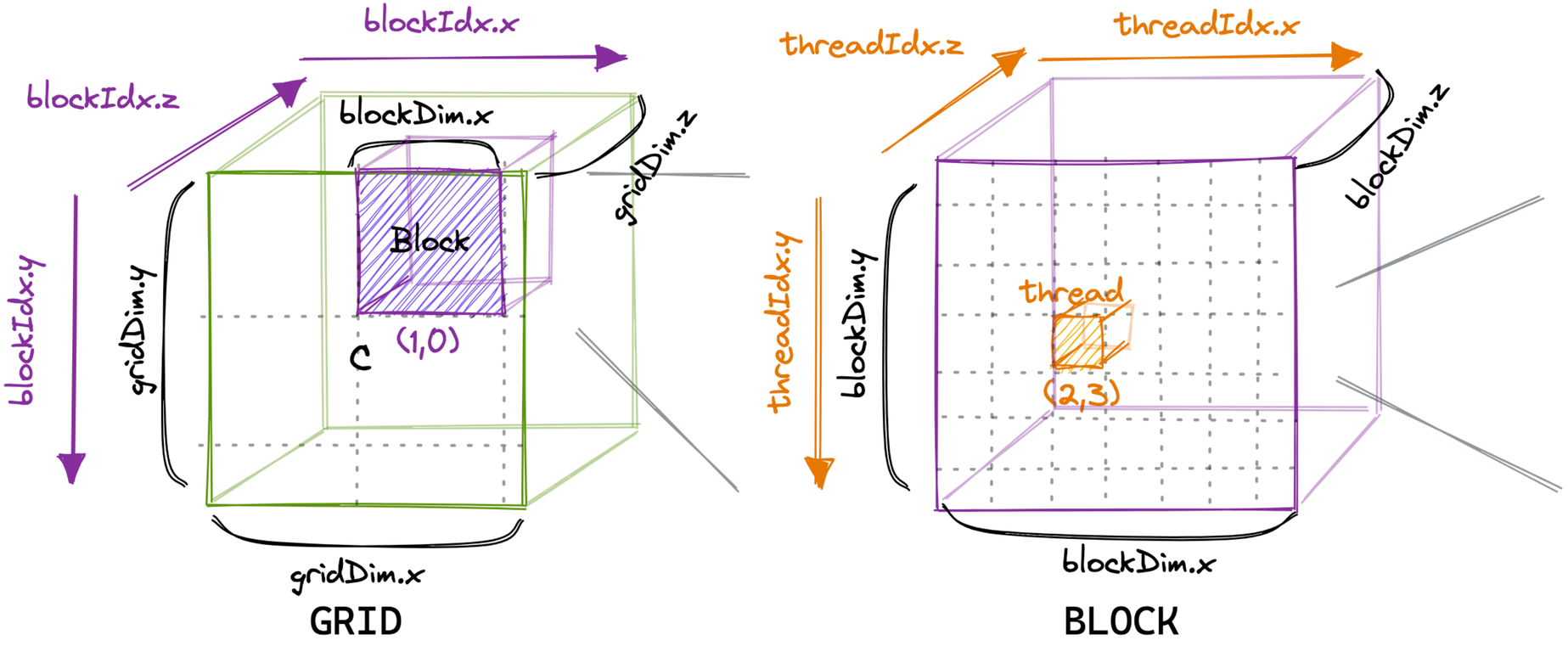
Matmul is computation heavy: computation complexity n^3 vs. communication cost n^2.

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.

Thread Organization



a single thread of computation, minding its own business

THREAD

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

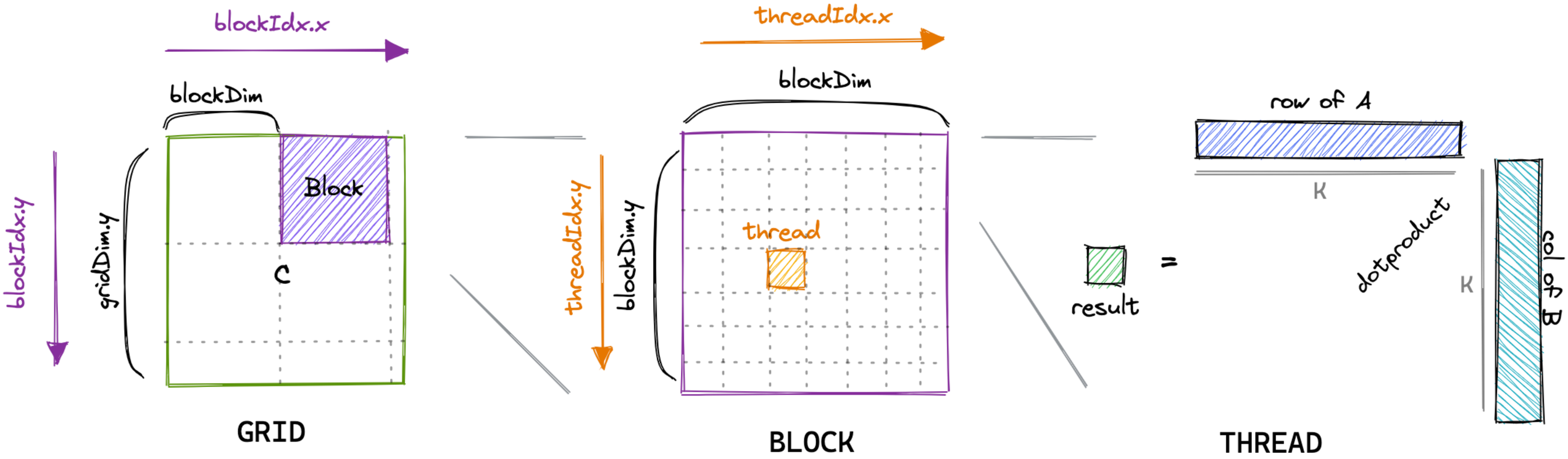
```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;
```

$$\begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,n-1} \end{bmatrix}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

```
}
```

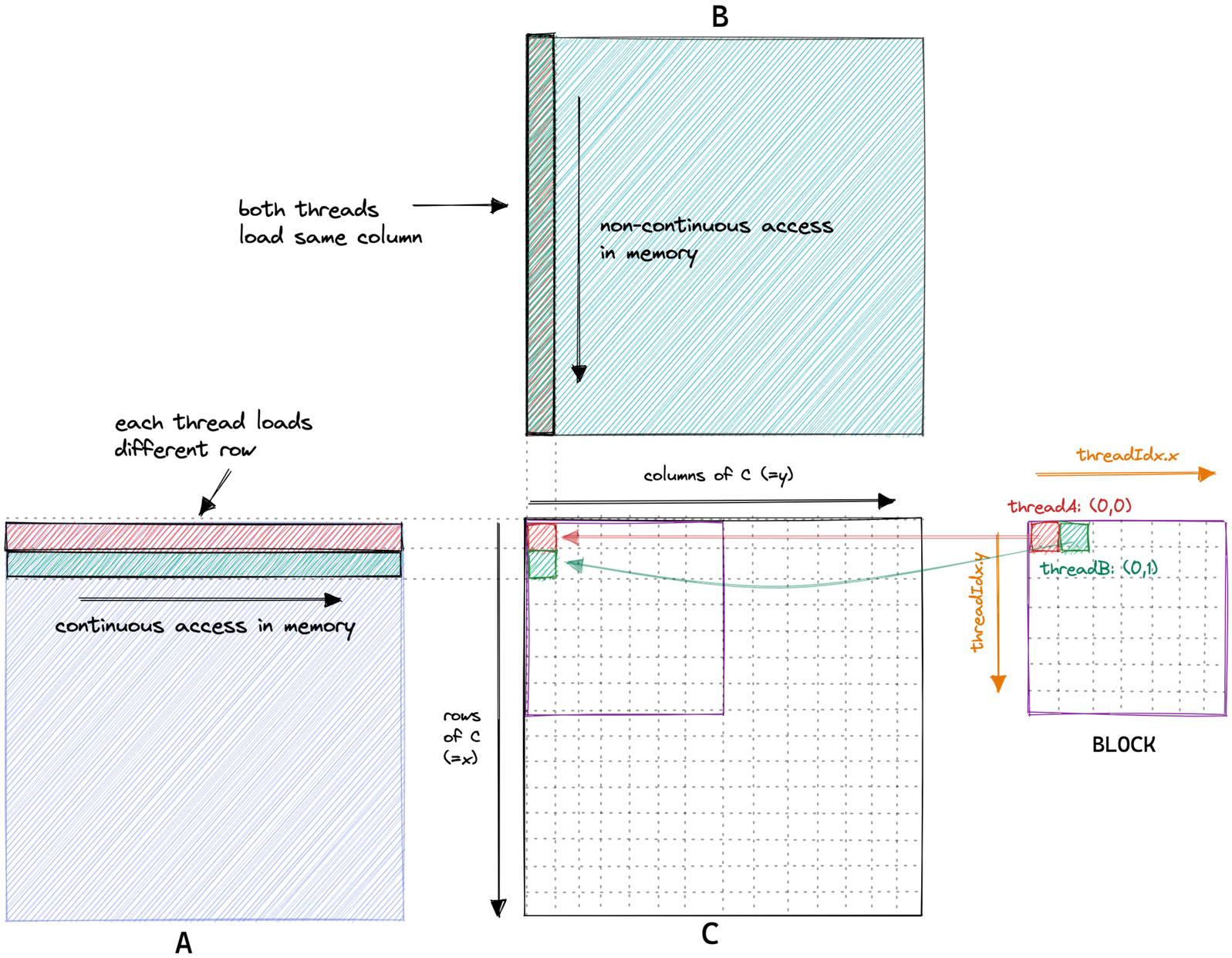
Thread Organization



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

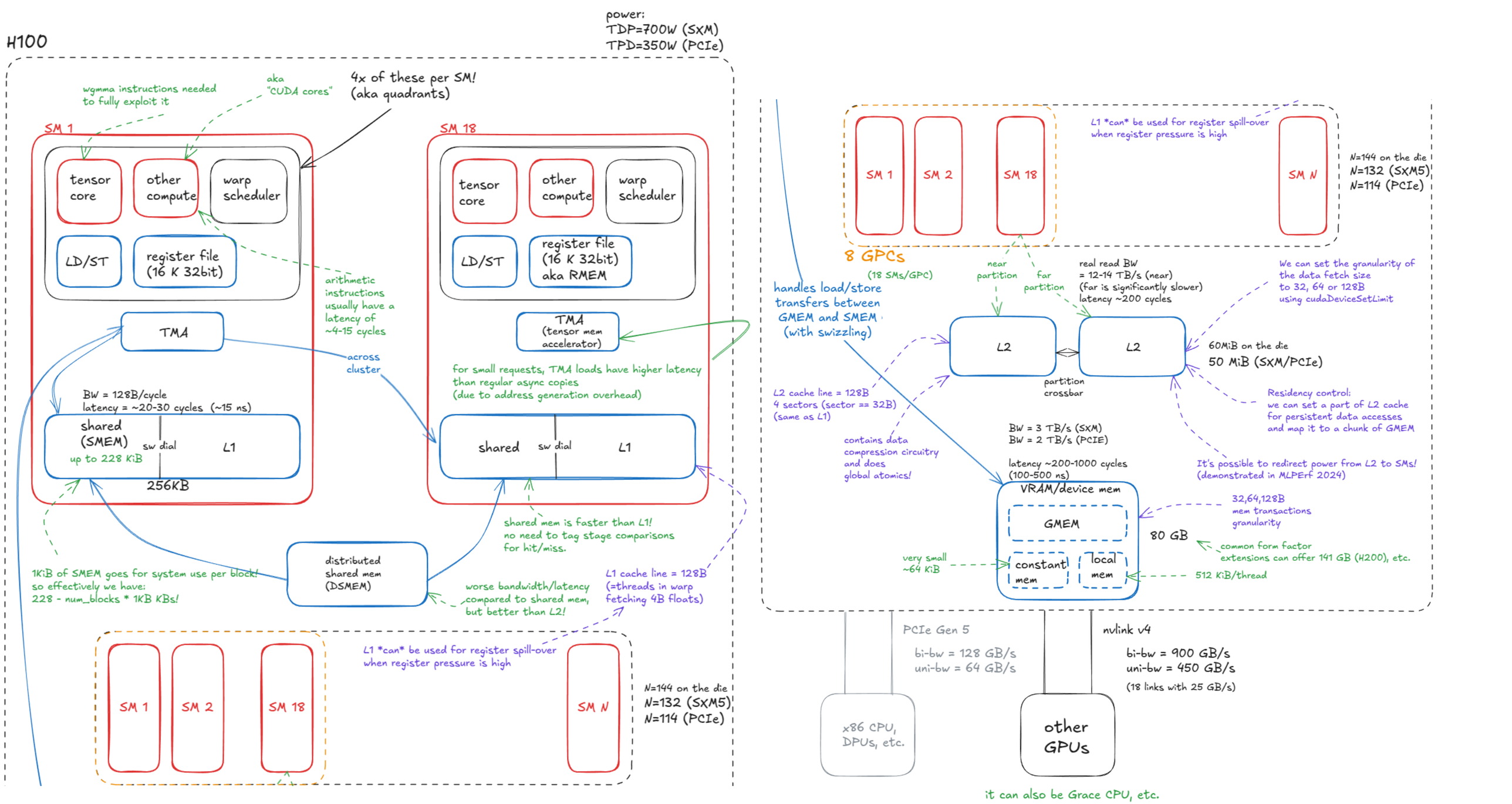
Each thread independently computes one entry of C



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
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.

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

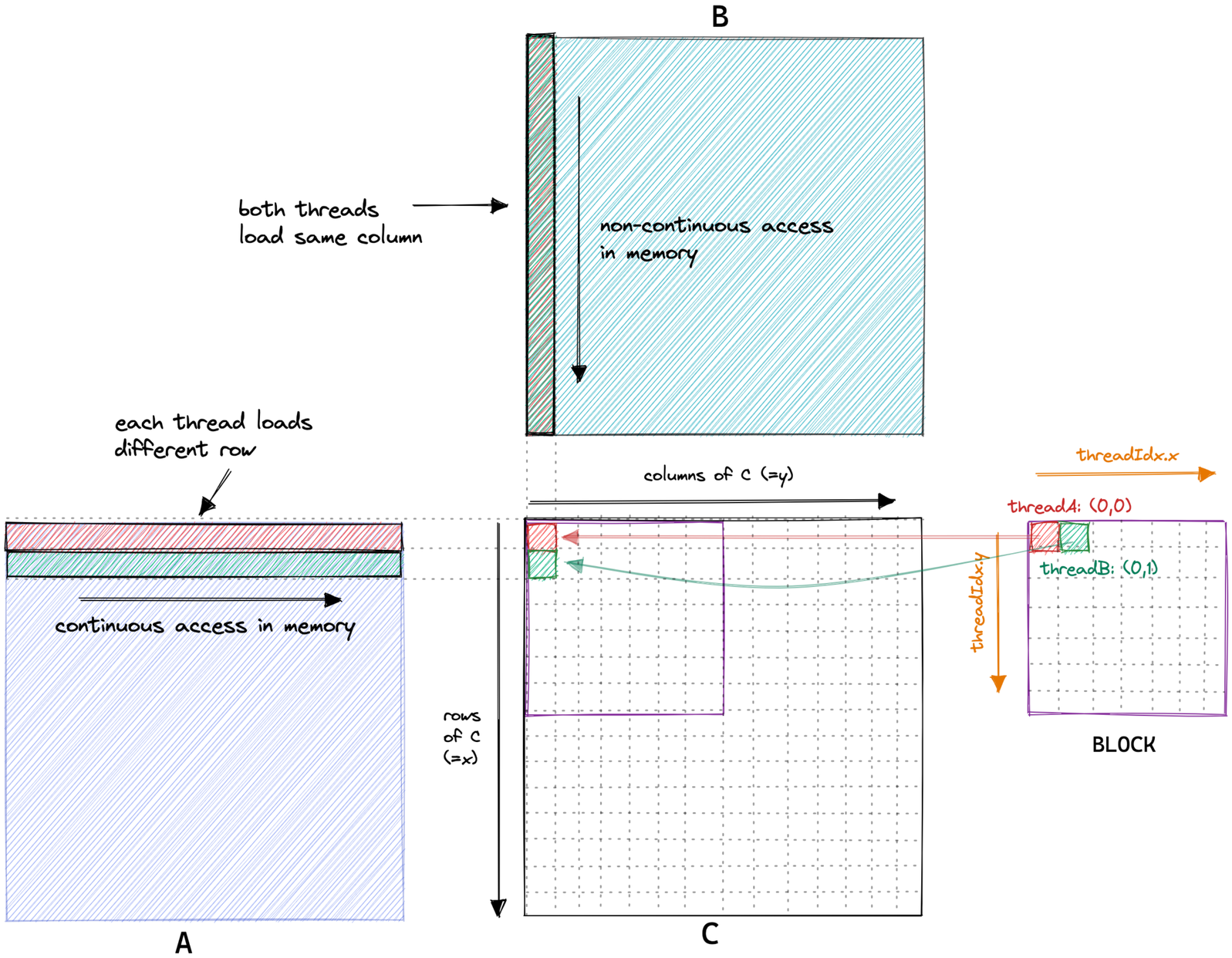
- The computation time (≈2.3 ms) is ~30× larger than the pure memory-transfer lower bound (≈0.07 ms).
- The operation is compute-bound, not bandwidth-bound — meaning the limiting factor is how fast the GPU's FP units can perform multiply-adds, not how fast data can be streamed from HBM3.
- In reality,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

```
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).
```

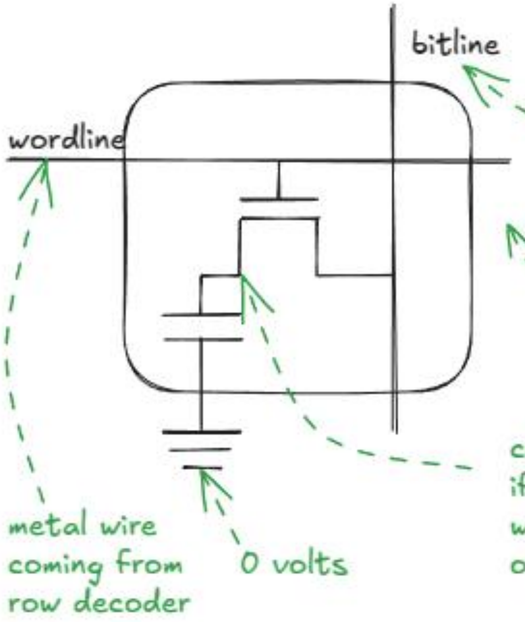


A, B, C are stored in row-major order.
This means that the last index (here y)
is the one that iterates continuous through
memory (=has stride 1).

Do we really understand GMEM?

GMEM is nothing but a matrix of DRAM cells.

and DRAM cells are glorified guarded capacitors. :)



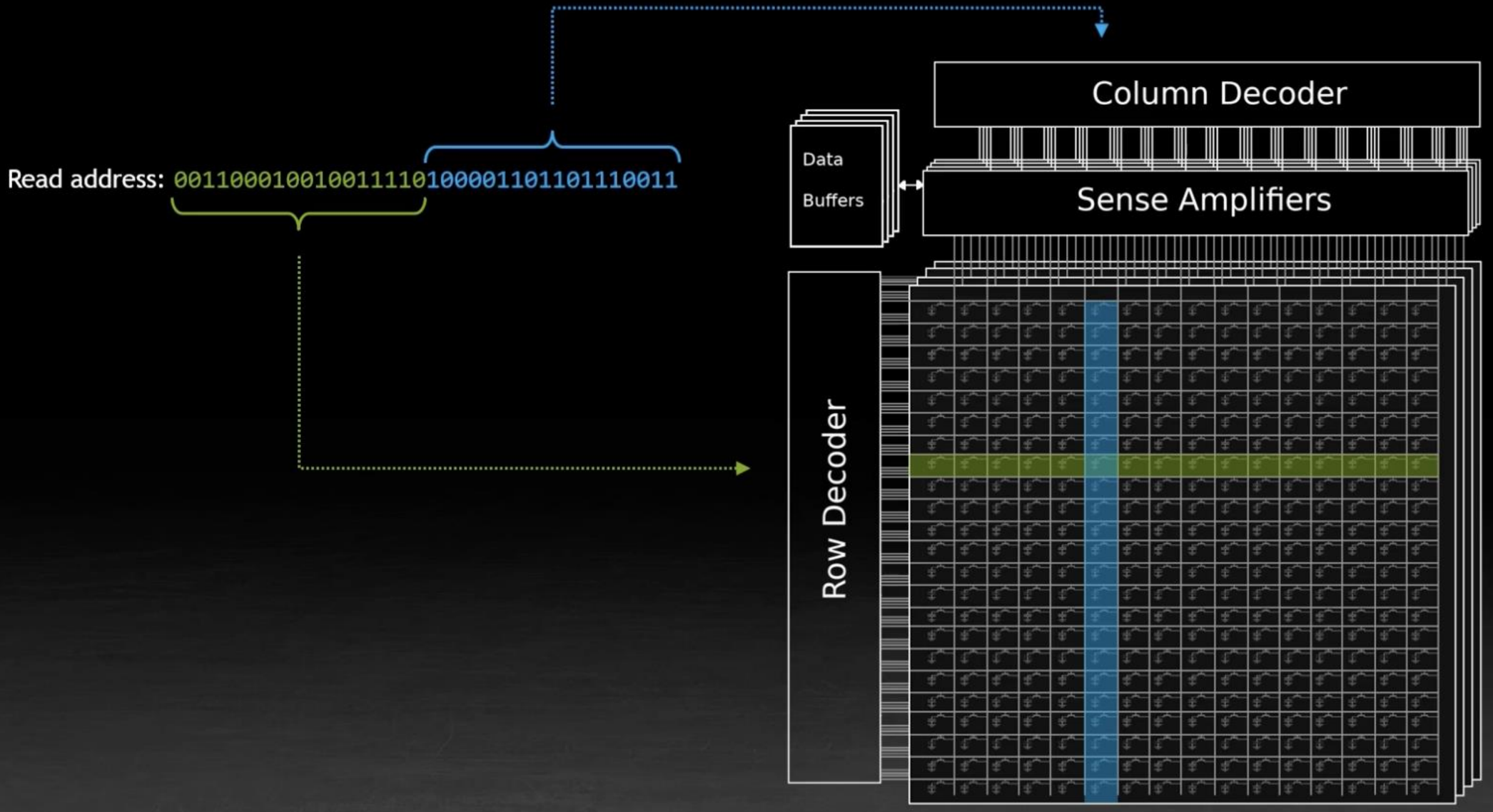
metal wire coming from sense amplifier

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

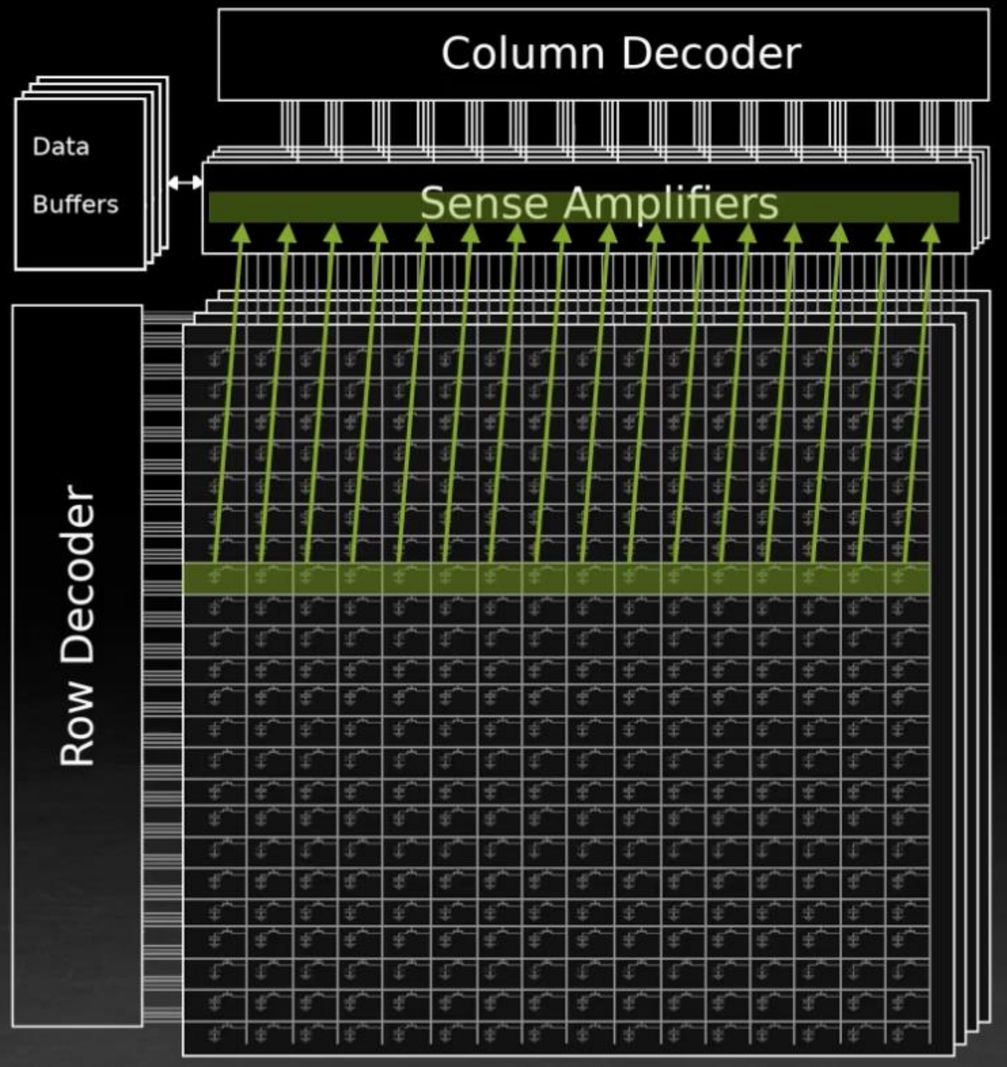
you can see how simple a DRAM cell is
and why we can build dense DRAM memories!

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.

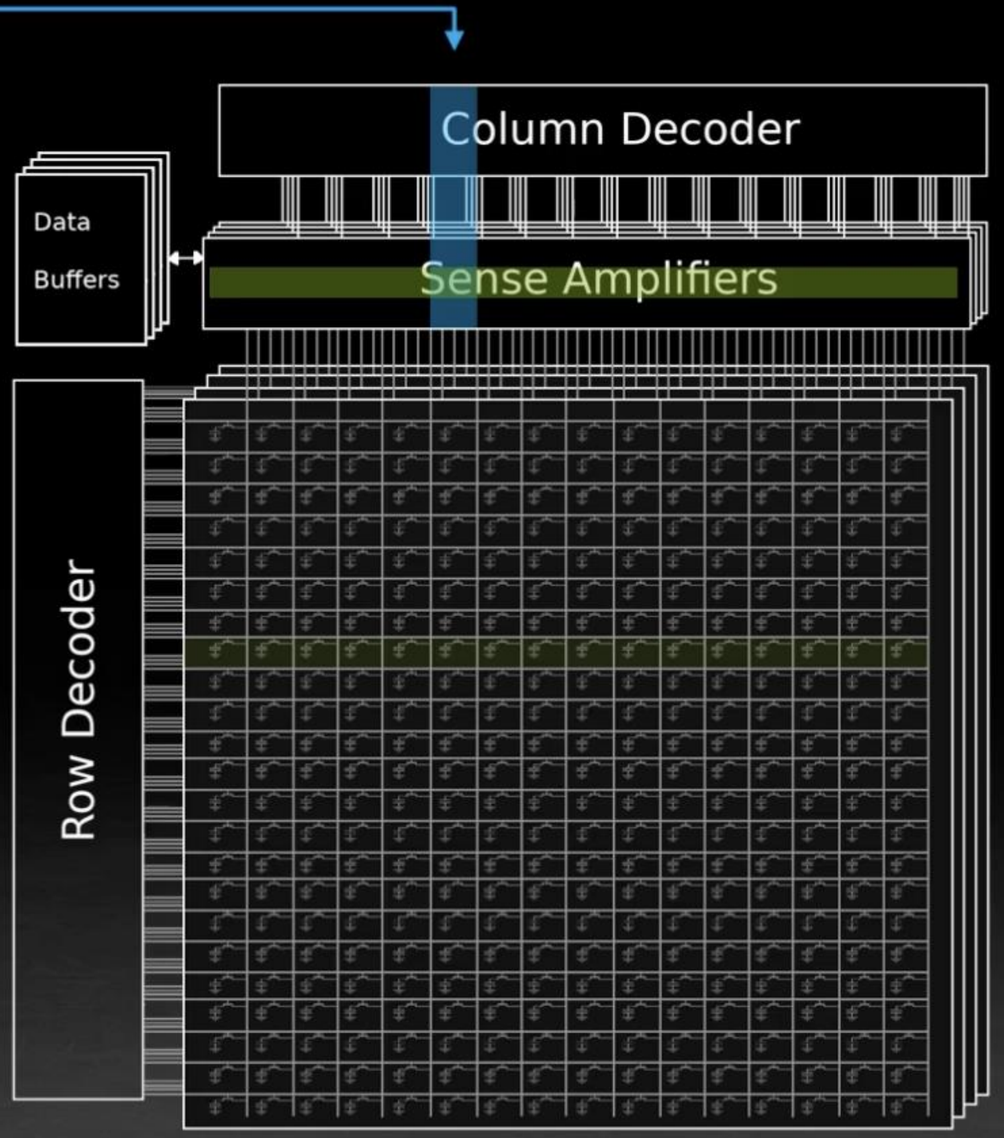

Read address: 00110001001001111010000110110111 0011
Column Decoder
Data
Buffers
Sense Amplifiers
Row Decoder

Read address: 0011000100100111101000011011011110011



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

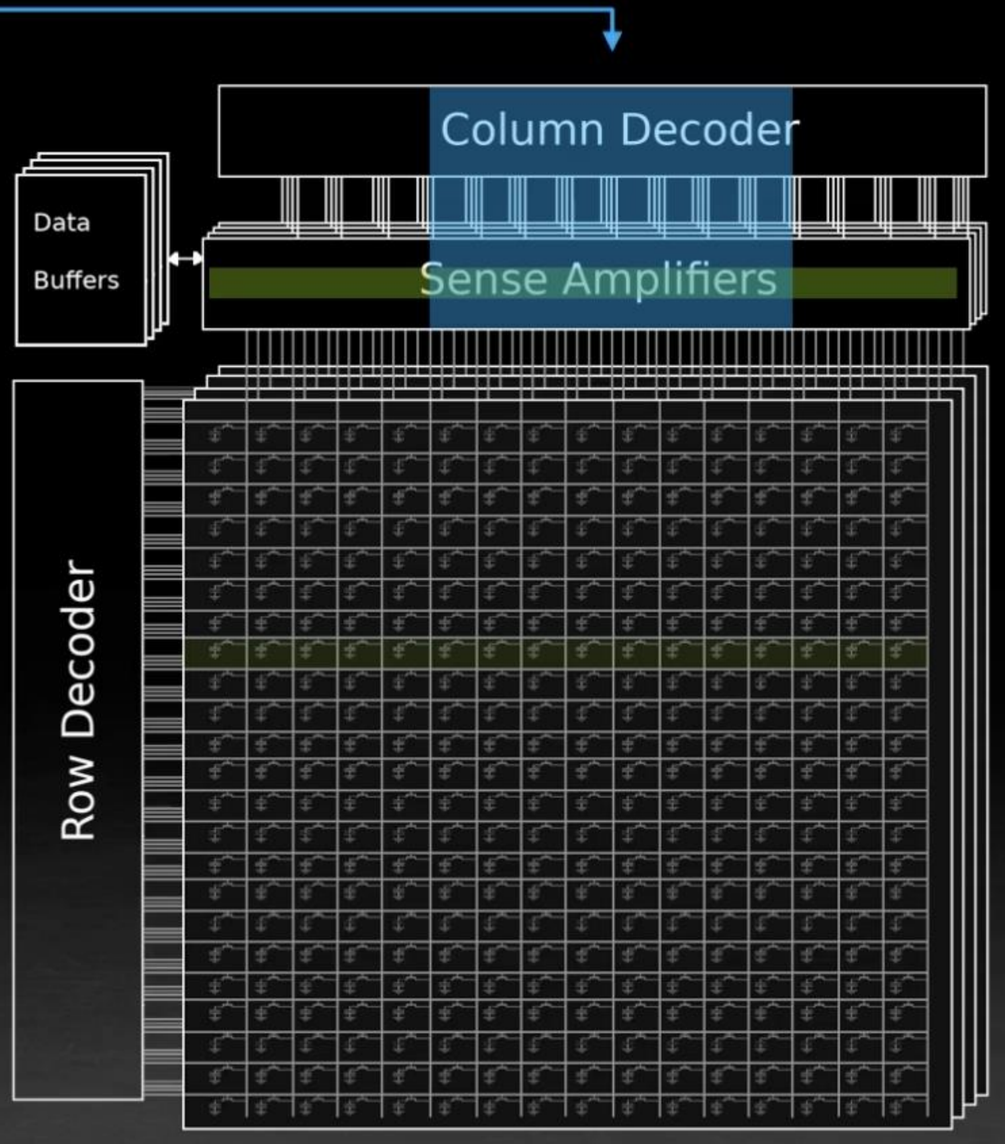
Read address: 00110001001001111010000110110111 0011



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

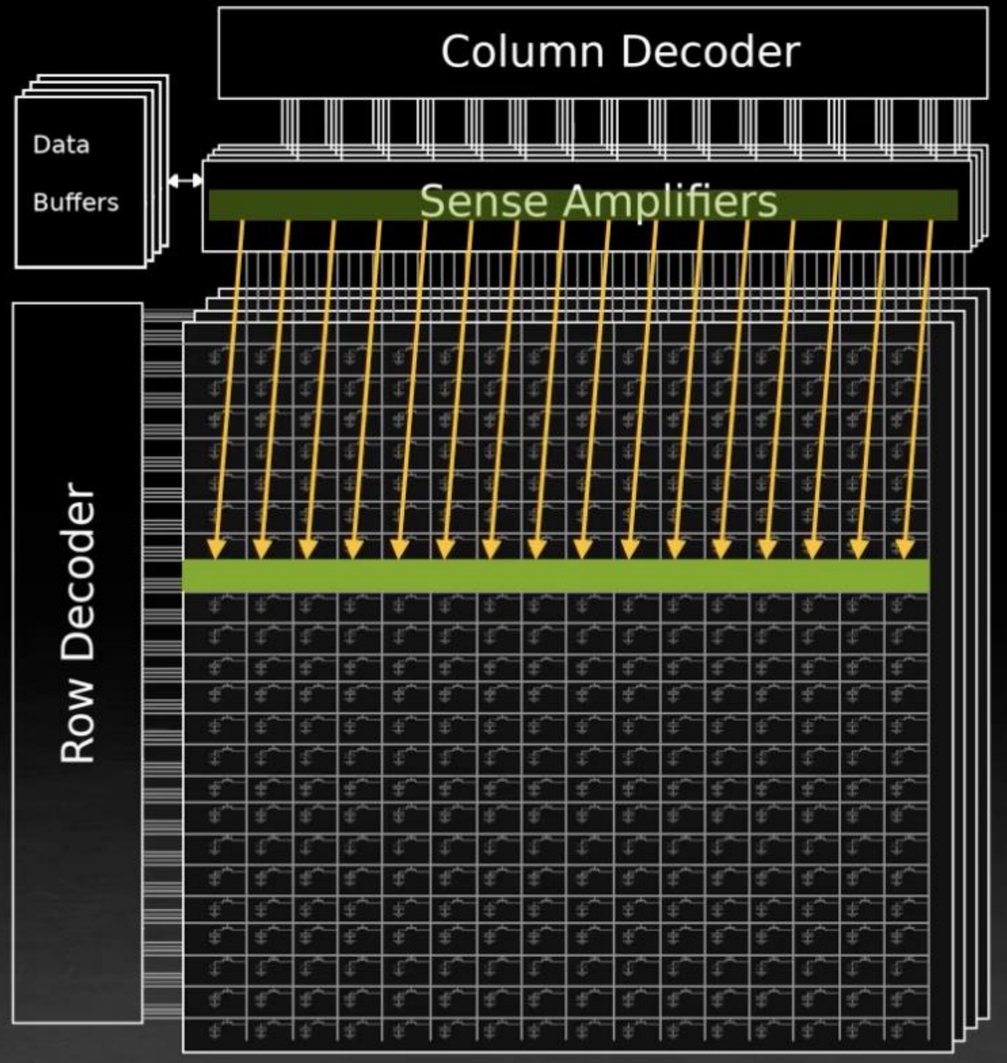
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers

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

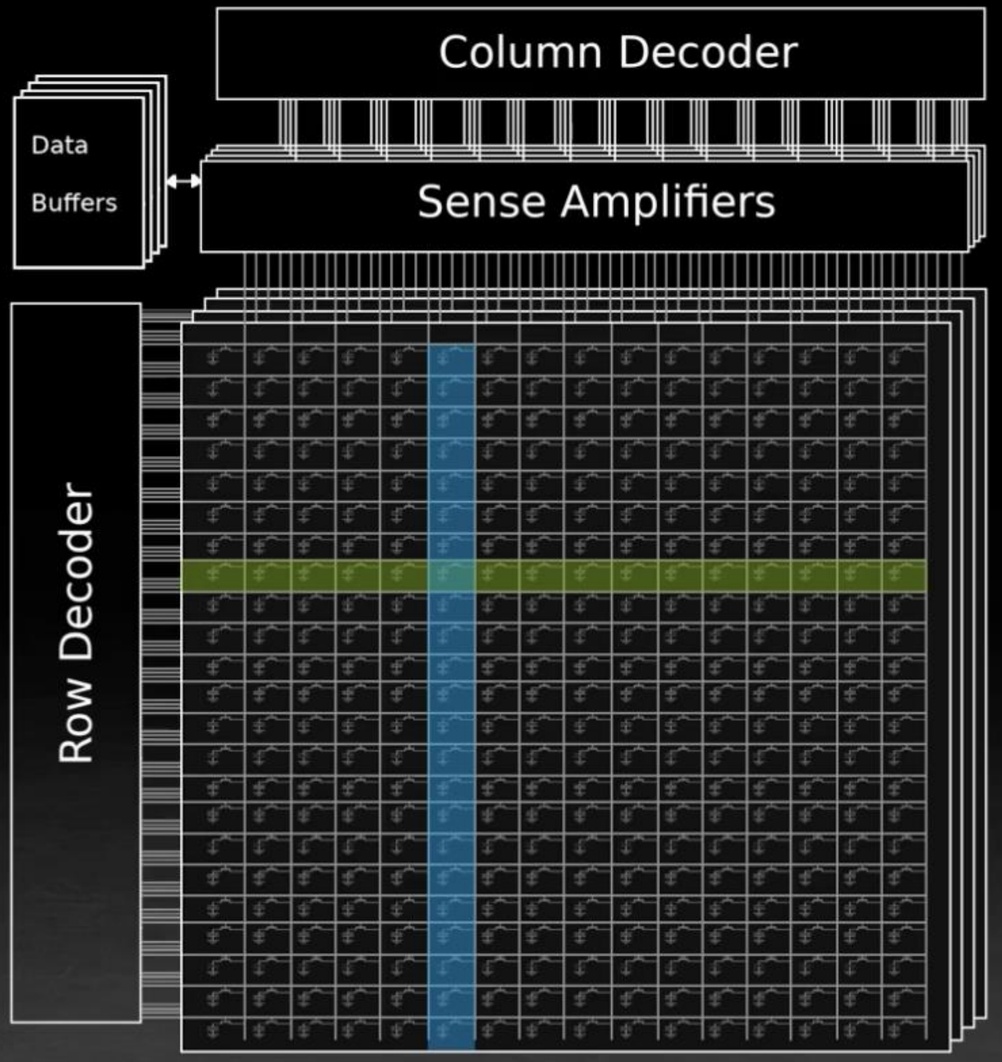
- Time to read new column: C_L = 16 cycles
- Time to load new page: T_{RCD} = 16 cycles
- Time to write back data: T_{RP} = 16 cycles
- Page (row) size = 1kB

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:

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



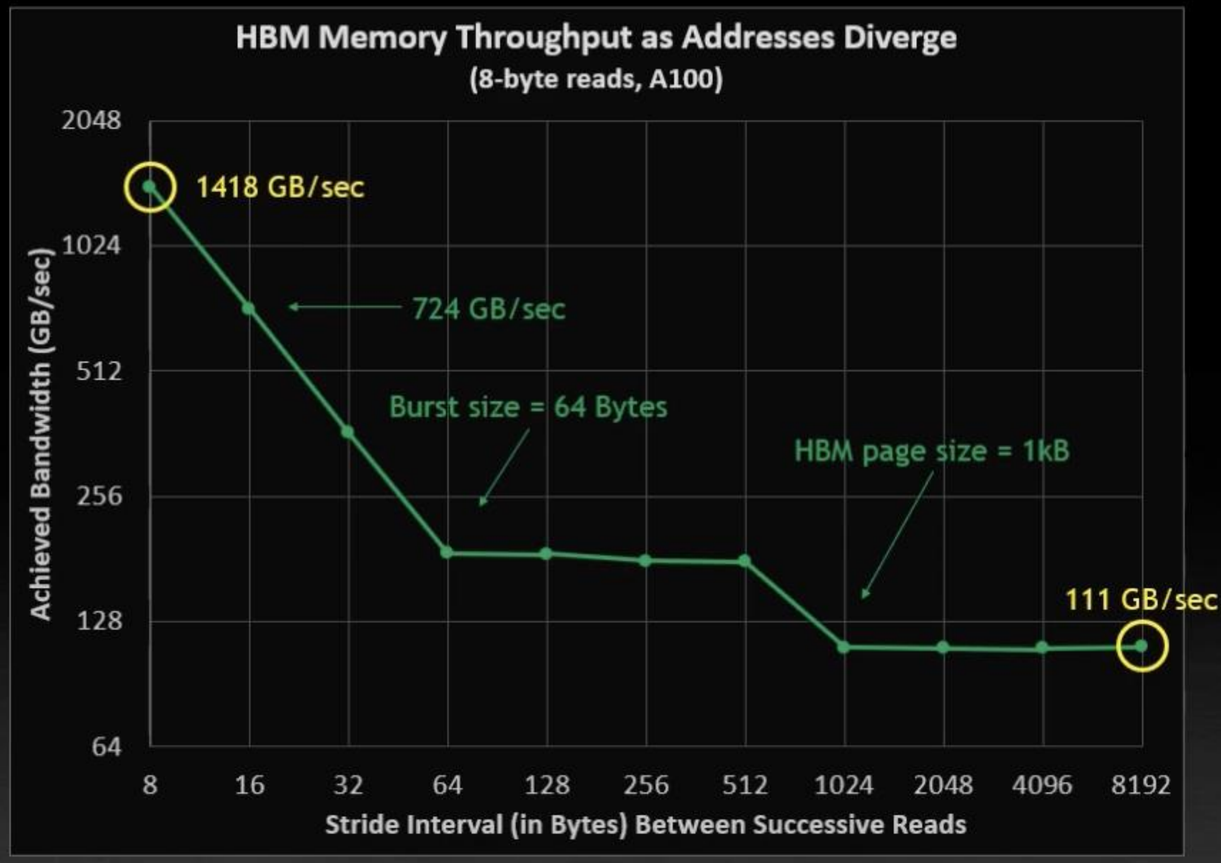
SO WHAT DOES THIS ALL MEAN?

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!



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

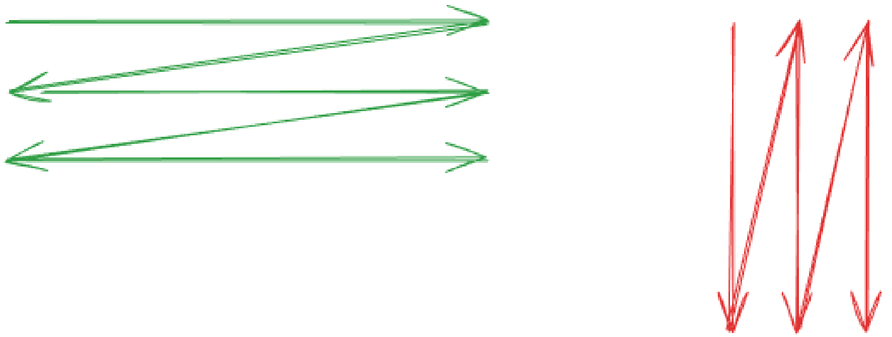
M=N=256

(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

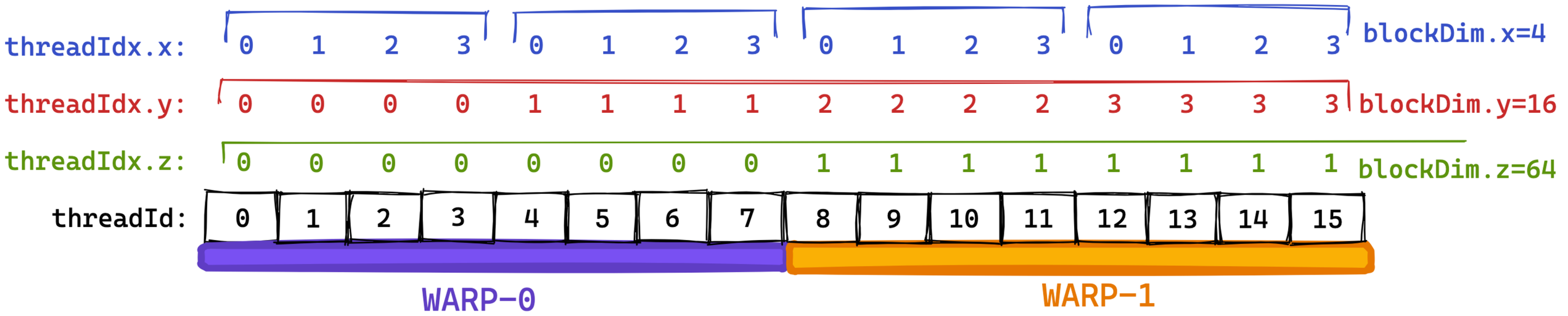


given what we just learned it's obvious that there is one right answer:
traversing columns should be in the inner/fast for loop as that would lead to a single row read
followed by 256 column reads for one iteration of the inner for loop (256 row reads in total).
the alternative would be 256 row reads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

Thread Organization

Wrap organization and its impact on memory access

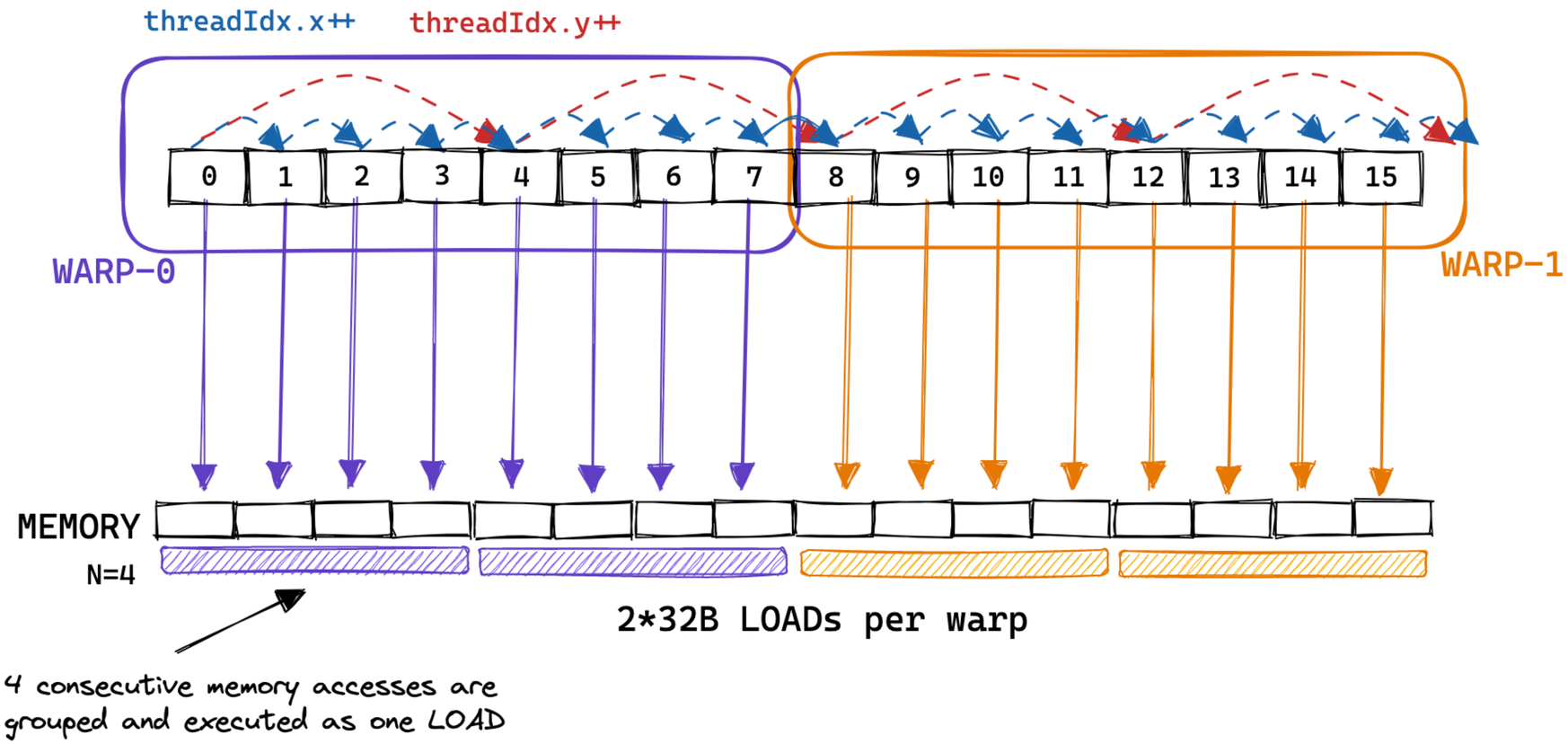
```
threadId = threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*threadIdx.z)
```




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

Thread Organization

This is what we want



Thread Organization

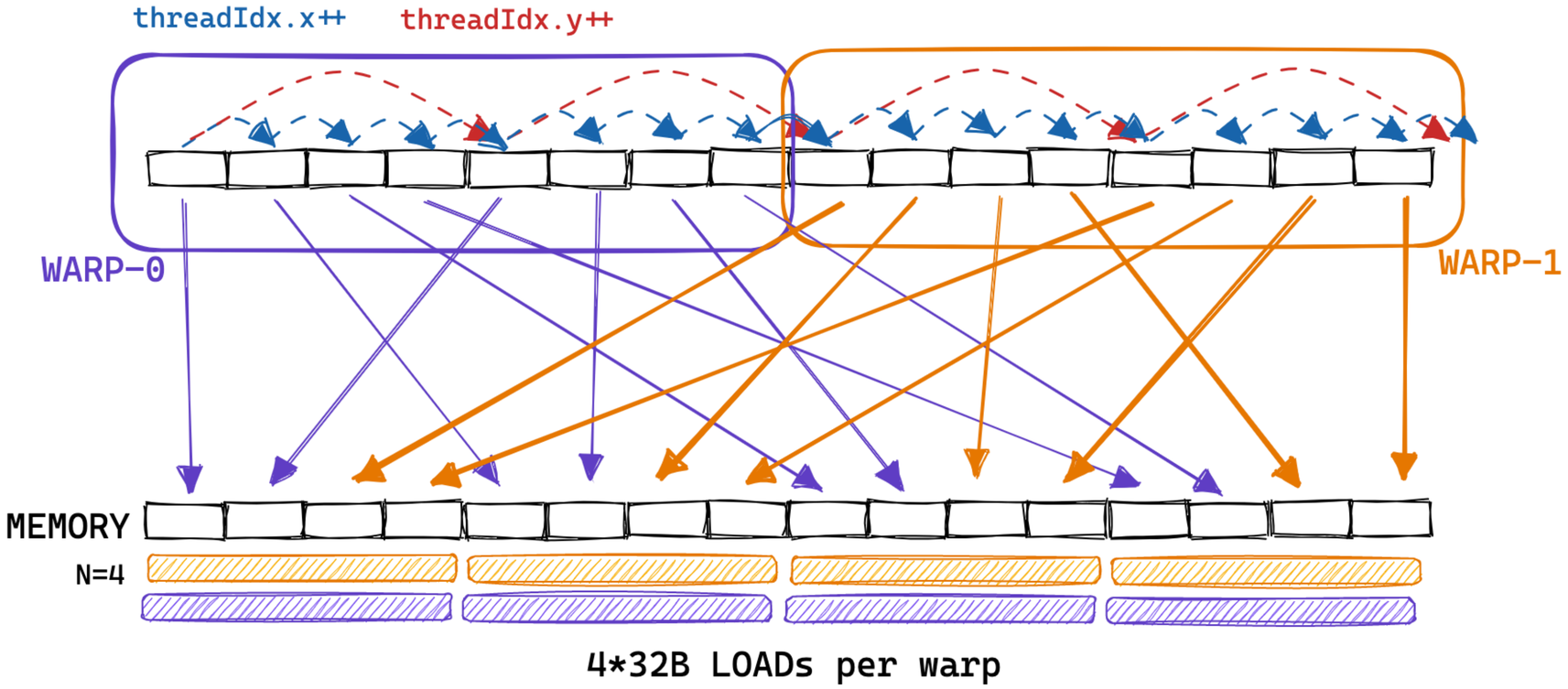
This is what we wrote

```
__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