
How GPUs differ from CPUs

Key takeaways

1. A CPU spends its chip on control and cache that make one thread finish sooner; a GPU spends it on many simple lanes that share instructions.
2. A CPU hides memory latency with caches and out-of-order execution. A GPU hides it by switching to another ready warp, so it needs spare resident work.
3. Programmers see the GPU's structure as SIMT: 32-thread warps share each instruction and memory request, so divergence and scattered addresses waste lanes.

Common misunderstandings of GPU performance

Which claim would you defend right now? Choose before the evidence.

“A GPU is a CPU with thousands of smaller cores, each running its own program.”

“If the work splits into enough independent pieces, a GPU makes it fast automatically.”

“Keeping more threads resident on the GPU always makes a kernel faster.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Same formula, same GPU, over 300× apart

Two kernels compute the same $C = A \times B$ on one NVIDIA B200

Naive kernel | 5.6 TFLOP/s

Scheduled kernel

1,772.9 TFLOP/s

Naive kernel: each thread computes one $C[i][j]$

$2 \cdot 4096^3 \approx 137$ billion FLOPs

```
acc = 0.0
for k in range(K):
    acc += A[i][k] * B[k][j]
C[i][j] = acc
```

Naive: 24.5 ms

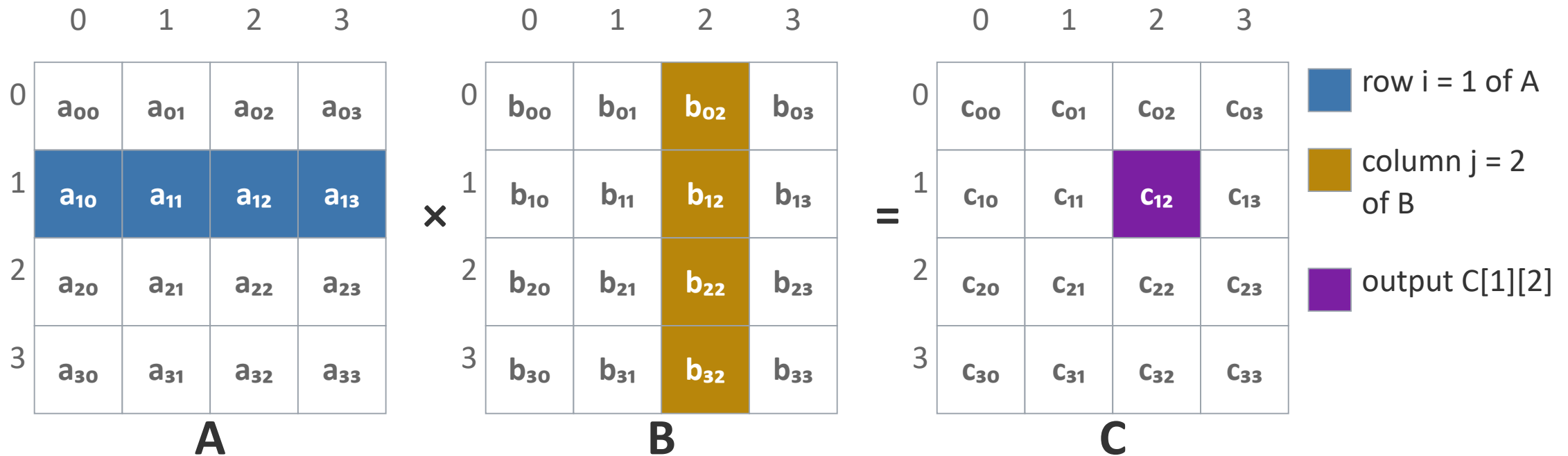
Scheduled: 78 μ s

Both are correct. This week: what the fast one knows about the machine.

derived: FLOPs $\div$ reported rate

One output is one dot product

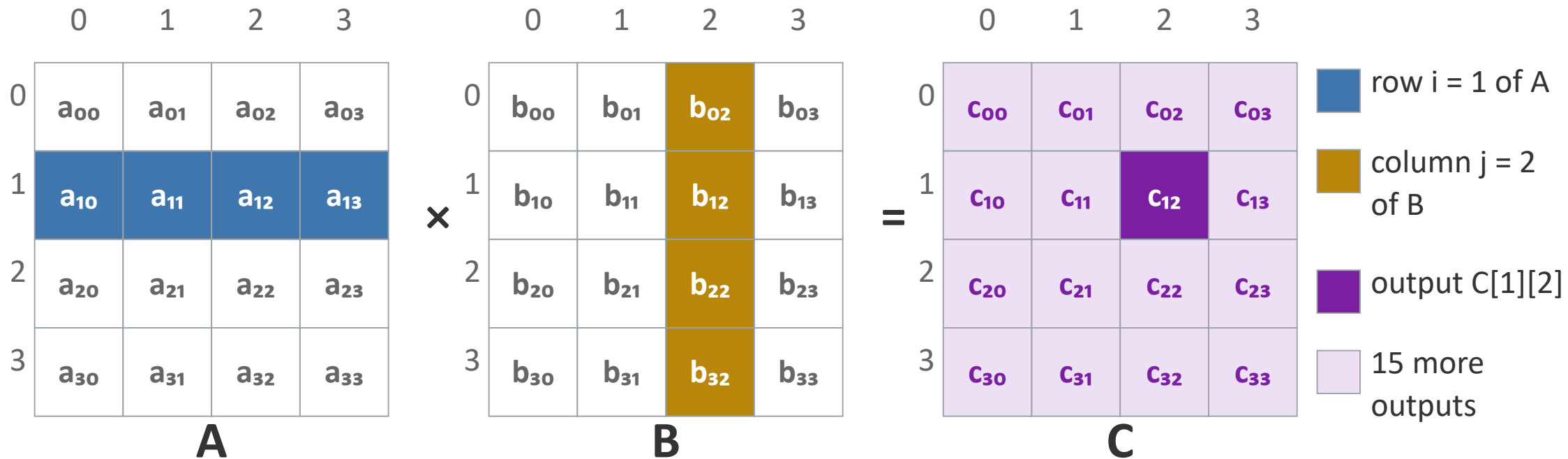
$C[i][j]$ reads row i of A and column j of B , and nothing another output computes.



$$c_{12} = a_{10} \cdot b_{02} + a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32} \text{ (4 multiplies, 4 adds)}$$

All 16 outputs are independent

They can run at once, in any order. None can skip fetching its 4 + 4 inputs.



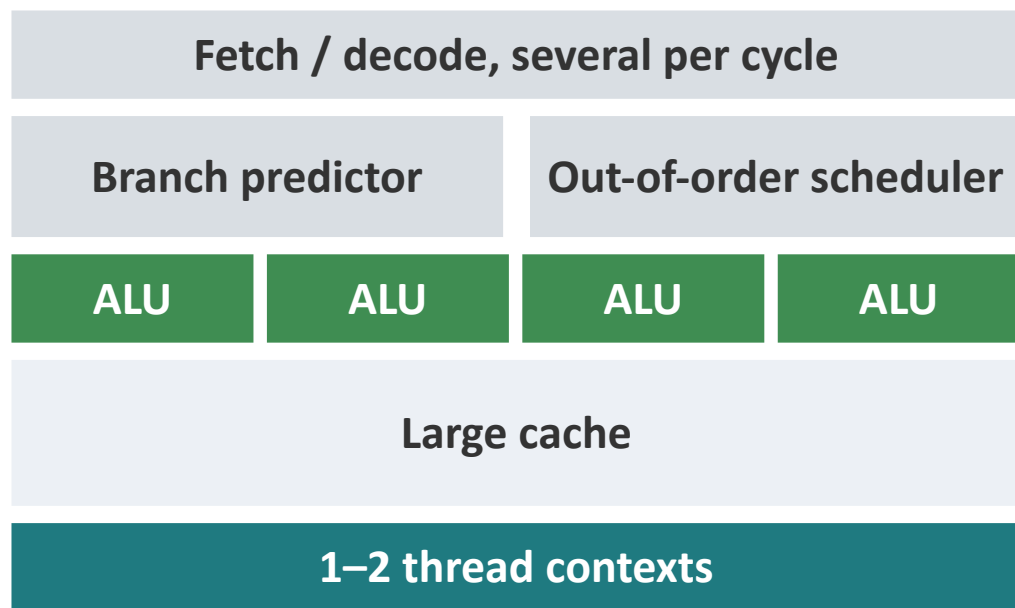
At 4096×4096 : 16.7 million independent outputs, each a dot product that reads 8,192 values.

Parallel work is abundant. Supplying its inputs is the hard part.

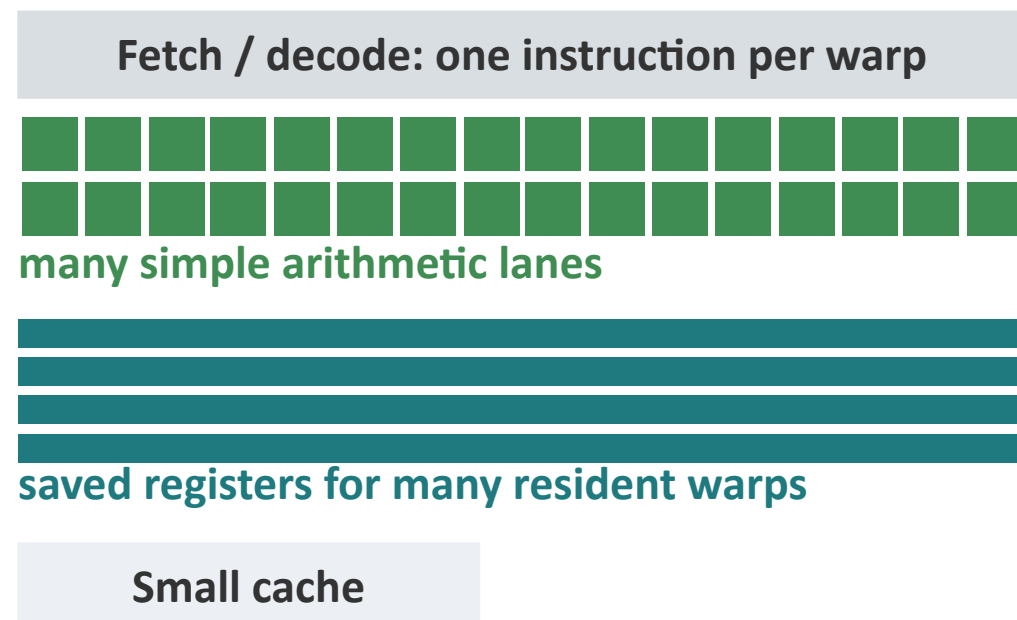
Two ways to spend a chip

Same area, opposite bets

CPU core (simplified)



GPU streaming multiprocessor (SM)

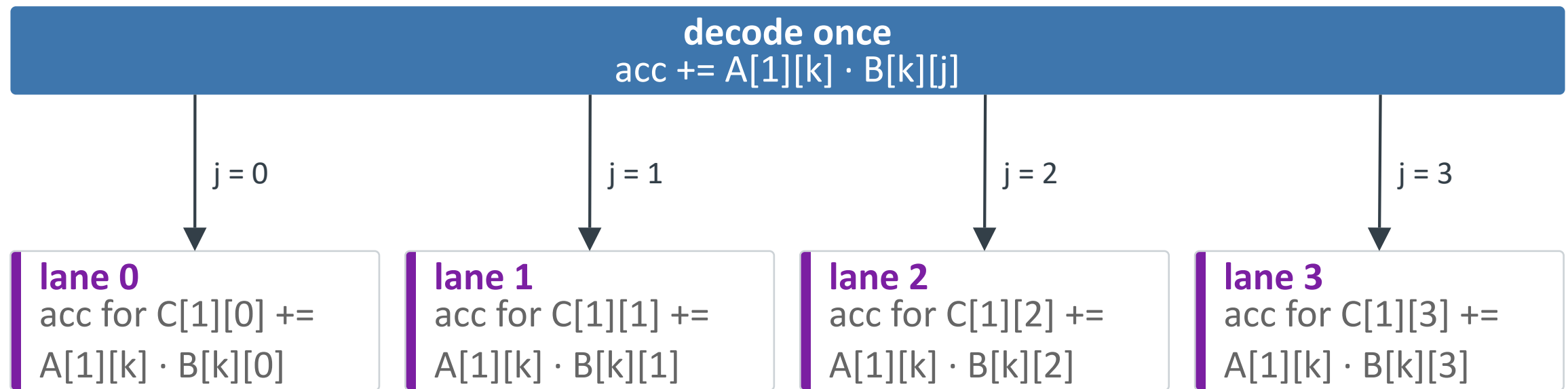


A CPU spends area to make one instruction stream finish sooner; a GPU spends it on lanes, and on saved state for many streams waiting their turn.

SIMD: decode once, compute on many lanes

One step of four outputs in row 1 of C: same instruction, different j

A lane is one arithmetic unit that applies the shared instruction to its own data. Decoding once for many lanes is how a GPU affords so many lanes.



The instruction and $A[1][k]$ are shared; only j differs. That is the regular work a GPU wants.

SIMT: write one thread, run 32 together

The naive kernel in CUDA: scalar code, one output per thread

One thread's work: no loop over outputs

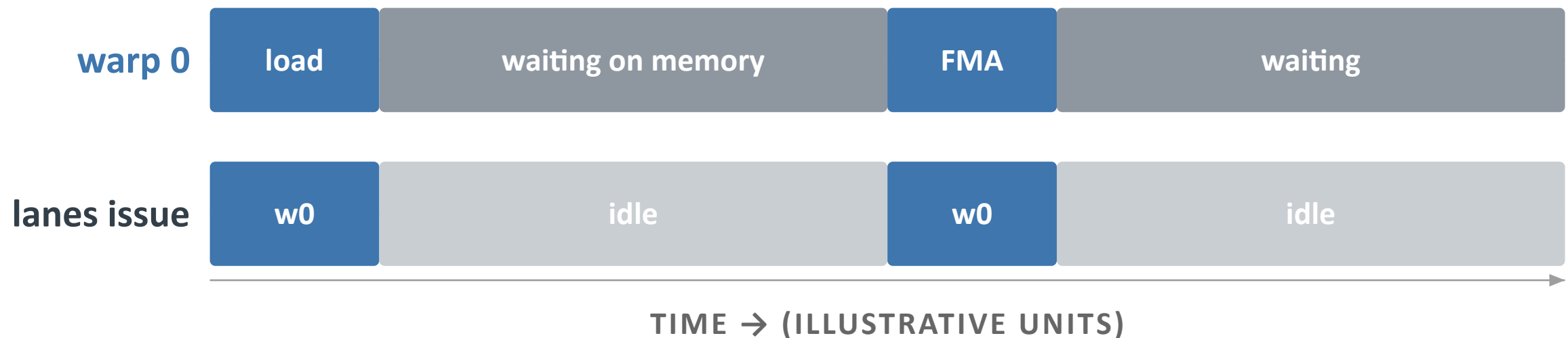
```
__global__ void matmul(const float *A,
    const float *B, float *C, int N) {
    // this thread's output cell C[i][j]
    int i = blockIdx.y*blockDim.y + threadIdx.y;
    int j = blockIdx.x*blockDim.x + threadIdx.x;
    float acc = 0.0f;
    for (int k = 0; k < N; k++)
        acc += A[i*N + k] * B[k*N + j];
    C[i*N + j] = acc;
}
```

thread	one run of the kernel, with its own index and registers
warp	32 threads that issue each instruction together, one per lane
SIMT	single instruction, multiple threads: you write the scalar thread, the hardware groups threads into warps

A memory load stalls a warp

A load from GPU memory takes hundreds of cycles; the math takes a few

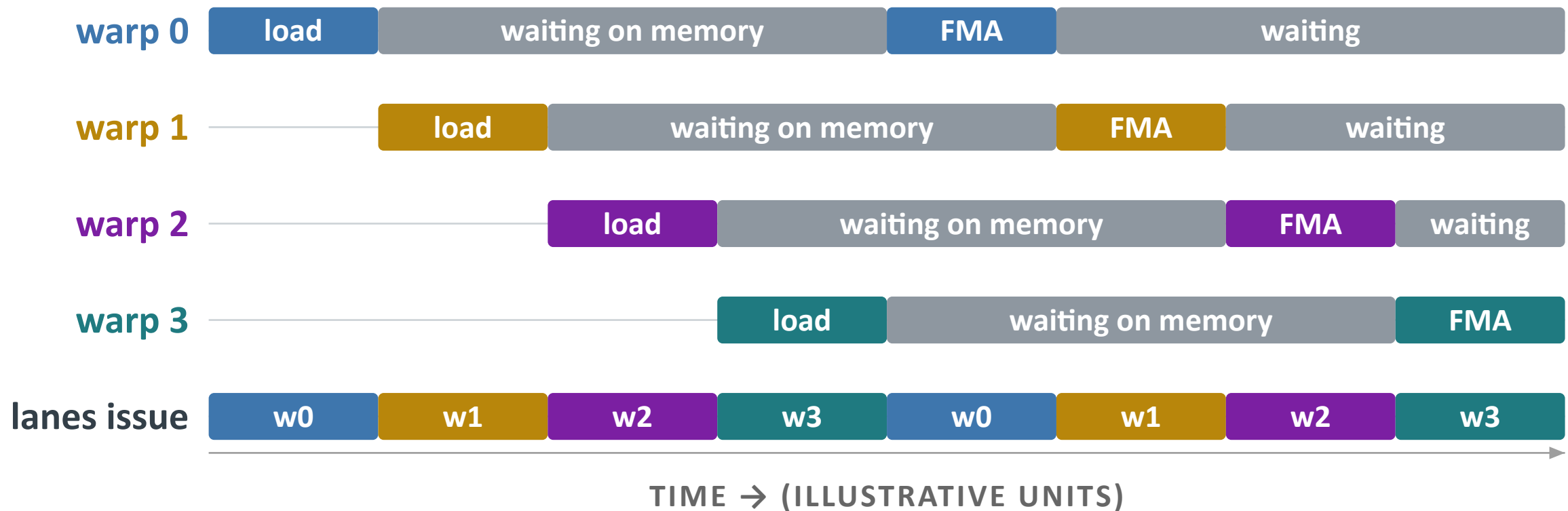
With one warp, the lanes finish the multiply and then wait. Here they issue 4 of 16 time units.



Faster arithmetic would not help this warp: it is waiting on memory, not math.

Other ready warps fill the stall

Same timeline, three more resident warps. Latency hiding: while warp 0 waits, the SM issues other ready warps. Warp 0's load is no faster.



Hiding needs spare resident work; real stalls are longer, so kernels need more.

Two ways to hide memory latency

Same problem, opposite uses of silicon

Both wait hundreds of cycles for DRAM. A CPU shortens one thread's wait; a GPU gives the lanes other work during it.

CPU: avoid the wait

Large caches

keep this thread's data close

Out-of-order execution

run later independent instructions now

Prefetchers

fetch what the thread will ask for next

GPU: work through the wait

Many resident warps

issue another while one waits

Small caches per thread

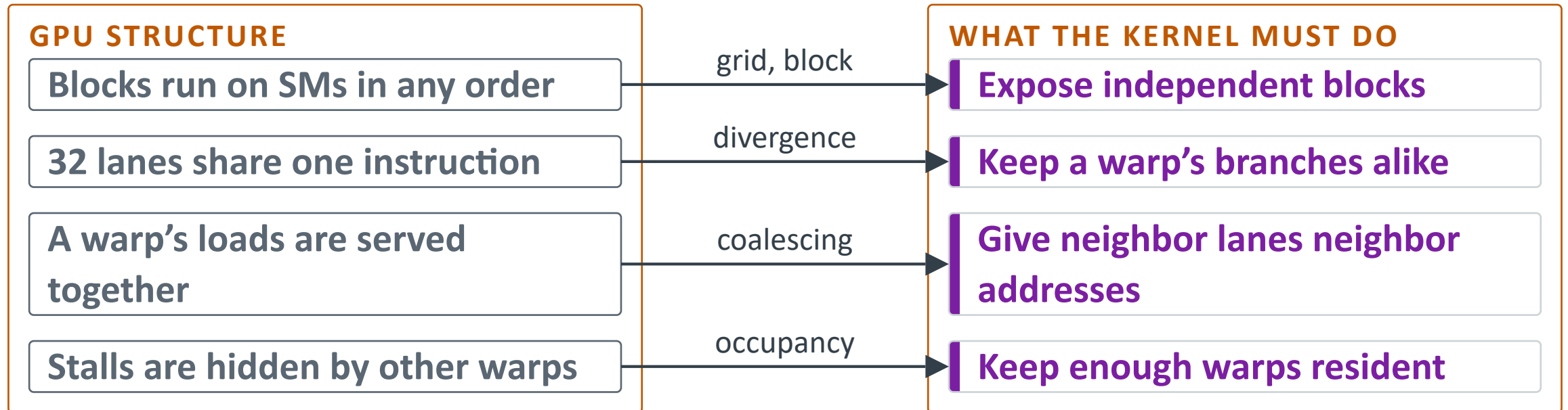
area went to lanes and registers

High memory bandwidth

7.7 TB/s on the B200; server CPUs,
hundreds of GB/s

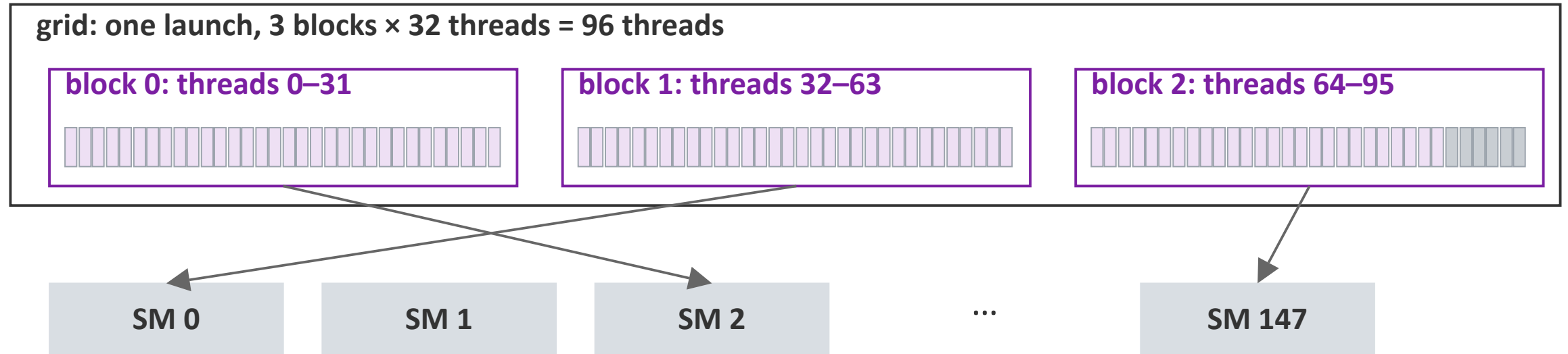
The structure, as the programmer sees it

Each hardware choice becomes an obligation on the kernel



One launch, four levels of grouping

Row i of C, N = 90: blocks go to SMs in no promised order



grid every thread of one launch

block threads that share one SM and can cooperate

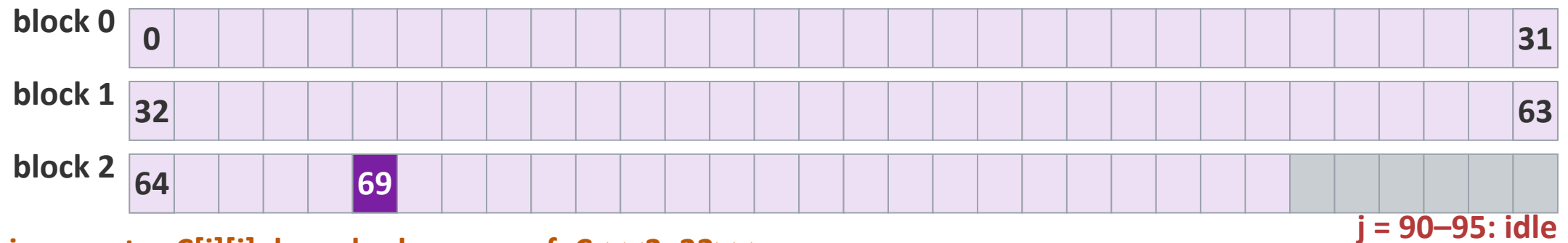
warp 32 threads that issue together

SM streaming multiprocessor; a B200 has 148

The global index, and a guard for the leftovers

96 threads for 90 outputs: which six must do nothing?

threadIdx.x →



Thread j computes $C[i][j]$; launched as `row_of_C<<<3, 32>>>`

```
int j = blockIdx.x * blockDim.x + threadIdx.x;
if (j < N) { // bounds guard
    float acc = 0.0f;
    for (int k = 0; k < N; k++)
        acc += A[i*N + k] * B[k*N + j];
    C[i*N + j] = acc;
}
```

Thread 5 of block 2:

$$j = 2 \cdot 32 + 5 = 69$$

No guard?

$j = 90 \dots 95$ write $C[i][90 \dots 95]$, which is $C[i+1][0 \dots 5]$: wrong answer, no crash.

Instapoll

Instapoll · Review · Oct 29 A

A GPU runs a group of 32 threads together. At an if/else, 16 of them take the if-branch and 16 take the else-branch, and each branch is 10 instructions long. Roughly how many instruction steps does the group spend on the if/else?

- A. About 10: each thread follows only its own branch, all at once
- B. About 20: the group runs both branches, and each thread waits out one
- C. About 5: each half of the group handles half of the work
- D. About 320: the 32 threads take turns, one thread at a time

Divergence masks lanes

An agent proposes: skip the multiply wherever B is zero

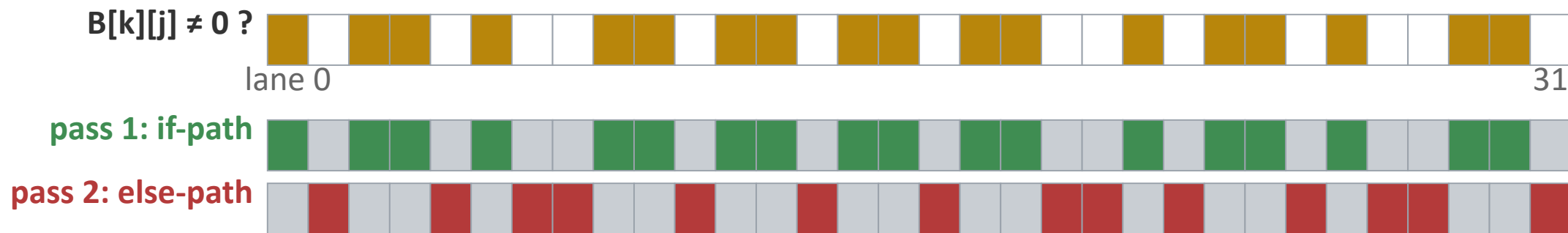
Where its lanes disagree, a warp runs each path in turn, other lanes masked.

Differs lane to lane: the warp splits

```
if (B[k*N + j] != 0.0f)
    acc += A[i*N + k] * B[k*N + j];
```

Same for every lane: the warp stays together

```
if (A[i*N + k] == 0.0f)
    continue; // every lane agrees
```



An if/else issues pass 1, then pass 2. This if has no else:
the zero lanes sit out pass 1, and pass 2 is empty.

Skipping zeros pays only when all 32 lanes skip; otherwise the multiply still issues.

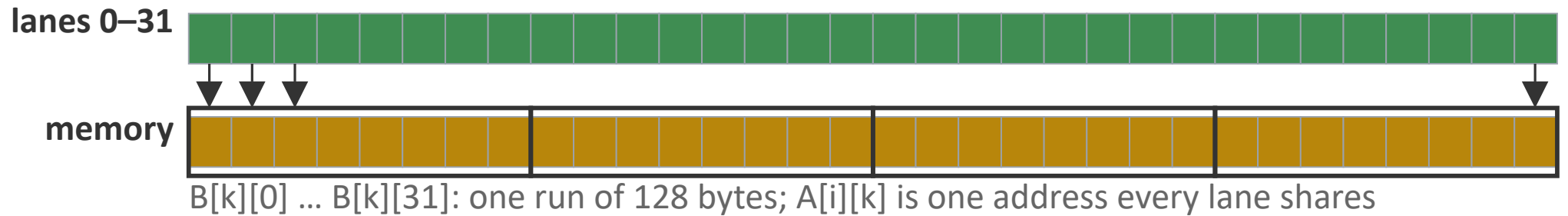
Neighbor lanes, neighbor addresses

Memory moves in 32-byte sectors; a warp's loads are served together

The index that follows threadIdx.x sets the traffic: 128 useful bytes cost 4 sectors or 32.

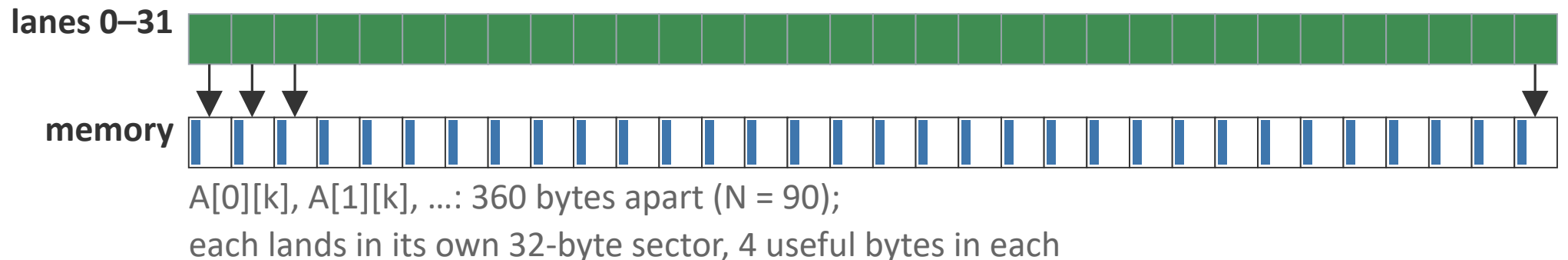
thread j computes $C[i][j]$: lane j loads $B[k][j]$

4 sectors · 128 B moved



thread j computes $C[j][i]$: lane j loads $A[j][k]$

32 sectors · 1,024 B moved



Occupancy: enough resident work, not a score

Occupancy: resident warps as a share of the most an SM can hold

More registers per thread leave fewer resident warps to cover a stall.

32 registers per thread → room for 2,048 threads = 64 warps



128 registers per thread → room for 512 threads = 16 warps



One SM's register file, 65,536 registers, split into resident warps. Other limits (shared memory, block slots) also apply.

- Too few ready warps: the stall slide again, lanes idle.
- Enough to cover the stalls: more warps buy nothing. Thursday's fast kernels spend registers and fast memory on reuse instead.

Key Takeaways

1. A CPU spends its chip on control and cache that make one instruction stream finish sooner. A GPU spends it on many simple lanes that share instructions, plus saved state for many warps, so it wins only on thousands of regular, independent pieces, like the outputs of $C = A \times B$.
2. Both wait hundreds of cycles for memory. A CPU shortens the wait with caches and out-of-order execution; a GPU issues other resident warps during it, so a kernel needs enough spare resident work. Beyond enough, occupancy buys nothing.
3. The programmer sees that structure as SIMT. A warp's 32 lanes share each branch and memory request: divergence costs the sum of the paths, and scattered addresses move many sectors for few useful bytes. Thursday: reuse and scheduling close the rest of the 300× gap.

Trace one warp

10 min

- In groups of three, trace warp 0 (lanes 0–31) through each kernel into one annotated table. Floats are 4 bytes; give byte offsets from each array's start.
- Masks: for each branch path, which lanes are active?
- Addresses: for each load at $k = 0$, list the offsets for lanes 0–7. Contiguous, strided, or one shared address?
- Rewrite: one change per kernel that improves lane use or address locality without changing the output.
- Compare with a neighbor: which rewrite might cost resident warps or extra work, so it needs a measurement?

Kernel 1: $y = M \cdot x$, M is 28×64 , one warp

```
int r = threadIdx.x;           // row
if (r < 28) {
    float acc = 0.0f;
    for (int k = 0; k < 64; k++)
        acc += M[r*64 + k] * x[k];
    y[r] = acc;
}
```

Kernel 2: one element per lane, one warp

```
int t = threadIdx.x;
float v = in[t];
if (t % 2 == 0)
    out[t] = slow(v);          // ~40 instr.
else
    out[t] = v * 0.5f;         // 1 instr.
```