
Keeping the tensor cores fed

Key takeaways

1. Once tensor cores make the math fast, the kernel waits on data. Overlapping loads, math, and stores took it from 360.2 to 1,772.9 TFLOP/s.
2. Overlap needs a second buffer and two barriers per buffer: full says the data has landed, empty says the reader is done and the buffer may be reused.
3. A speed number belongs to one shape, datatype, GPU, baseline, and timing method: the kernel that matched cuBLAS at 4096^3 reached 90% of it at 8192^3 .

Common misunderstandings of fast GPU matrix multiply

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

“Once tensor-core math is fast, the data movement around it stops mattering.”

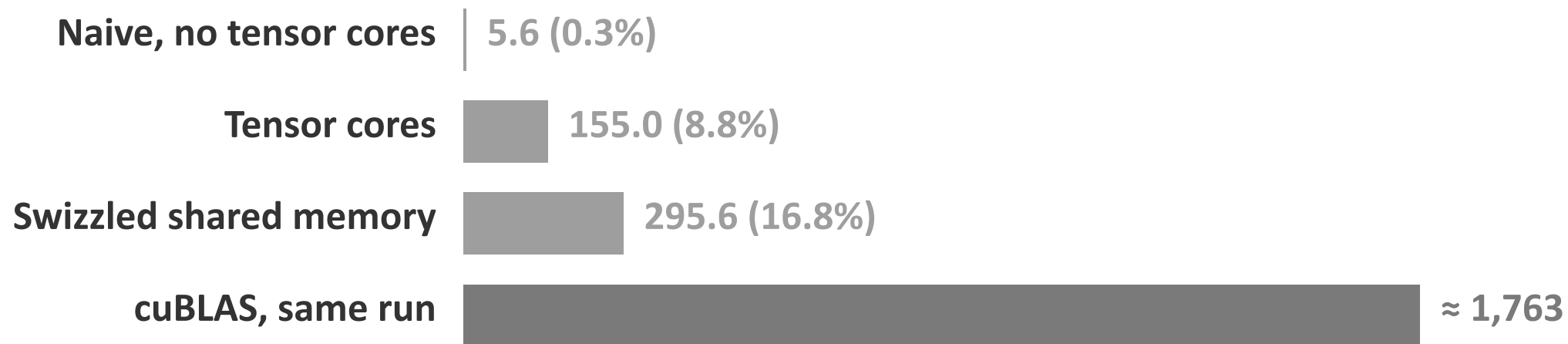
“Asynchronous copies make synchronization unnecessary.”

“A kernel that beats cuBLAS on one square matrix is the faster kernel in general.”

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

Where we are: 295.6 of about 1,763

Tiles, tensor cores, and swizzling bought 53×; the kernel is still at 16.8% of cuBLAS



TFLOP/s, source-reported: 4096×4096×4096, BF16 inputs, one B200. Percent is of that post's cuBLAS run, not of hardware peak.

Same words as the reuse lesson: a block walks K in slices, staging each A and B slice in shared memory for the tensor cores.

The loop at 295.6: load, wait, compute, wait

One block, one output tile, $K = 4096$ in 64 slices of 64 (a Python-shaped sketch)

```
acc = zeros(fp32)      # running sum for this output tile
for k in range(64):    # one trip per K-slice
    tma_load(buf, slice=k, signal=full) # returns at once
    full.wait()        # tensor cores idle until bytes land
    mma(acc, buf, signal=done) # returns at once
    done.wait()        # copy engine idle until MMA reads buf
store(C_tile, bf16(acc)) # after the last slice
```

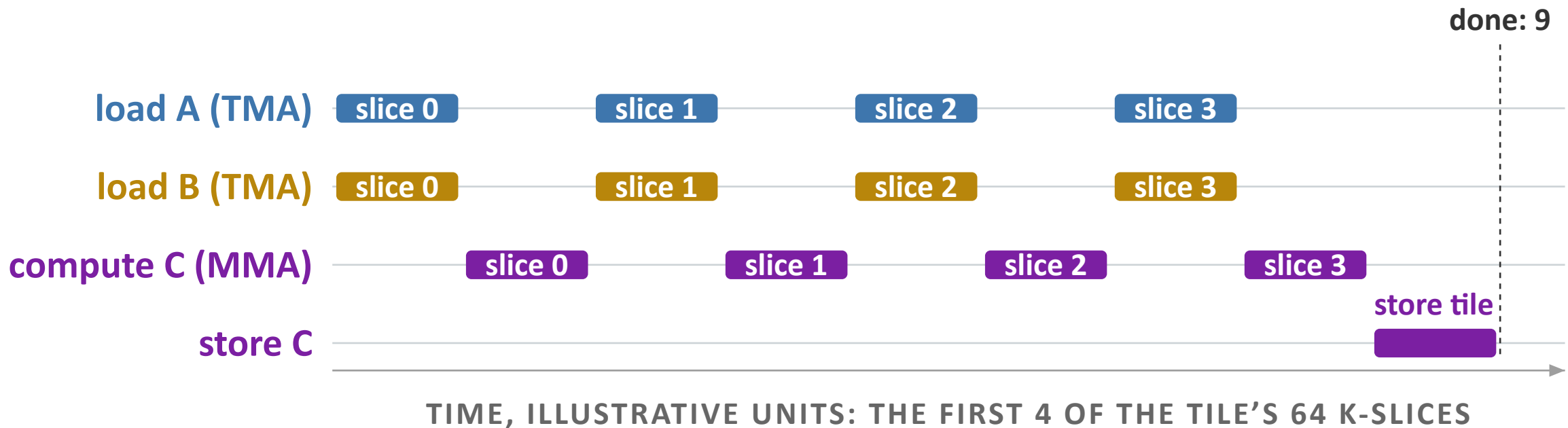
TMA Tensor Memory Accelerator: a copy engine that moves a whole tile into shared memory, then signals a barrier.

asynchronous The call returns before the work is done.

Both operations are asynchronous, and the loop waits for each one at once: nothing overlaps.

Four K-slices: each unit works half the time

Tracks are hardware units; a bar is that unit busy on one slice



At every instant either the tensor cores wait for data or the copy engine waits for them.

Two rungs that barely moved the number

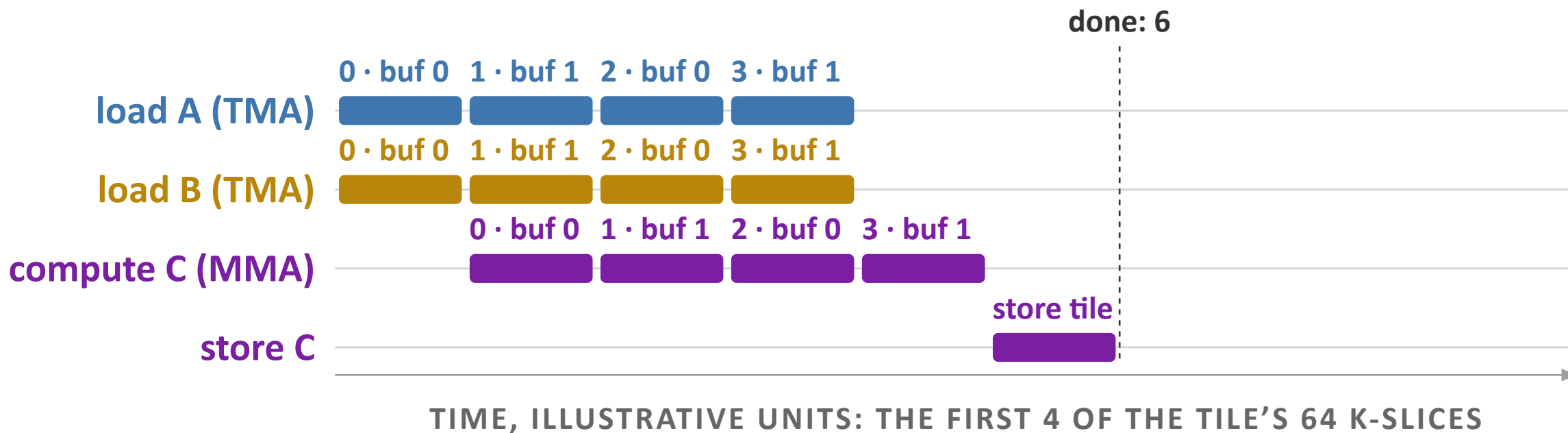
Each made a real operation cheaper; neither removed the wait in the loop

Next rung	TFLOP/s	What it made cheaper	Why the loop barely moved
TMA store for output	293.6 (-0.7%)	Writing the finished tile: one bulk asynchronous copy instead of many small writes	The store runs once per tile, after 64 slices; the stall is inside the slice loop
Two-SM MMA	360.2 (+22%)	Two SMs' tensor cores run one larger MMA and each loads half of the B tile, so less traffic per output	Every slice still waits for its own load before the math starts

A mechanism helps only when it removes the current bottleneck. Here the bottleneck is the wait.

Two buffers: load $k + 1$ while computing k

Same slices, same durations; slice k lands in buffer $k \bmod 2$



Pipeline stage: one step of the per-slice work (load, compute, store) run by its own unit. Double buffer: two shared-memory slots, one filled while the other is read.

The loads now hide under the math: 9 units become 6 for the same work.

Instapoll

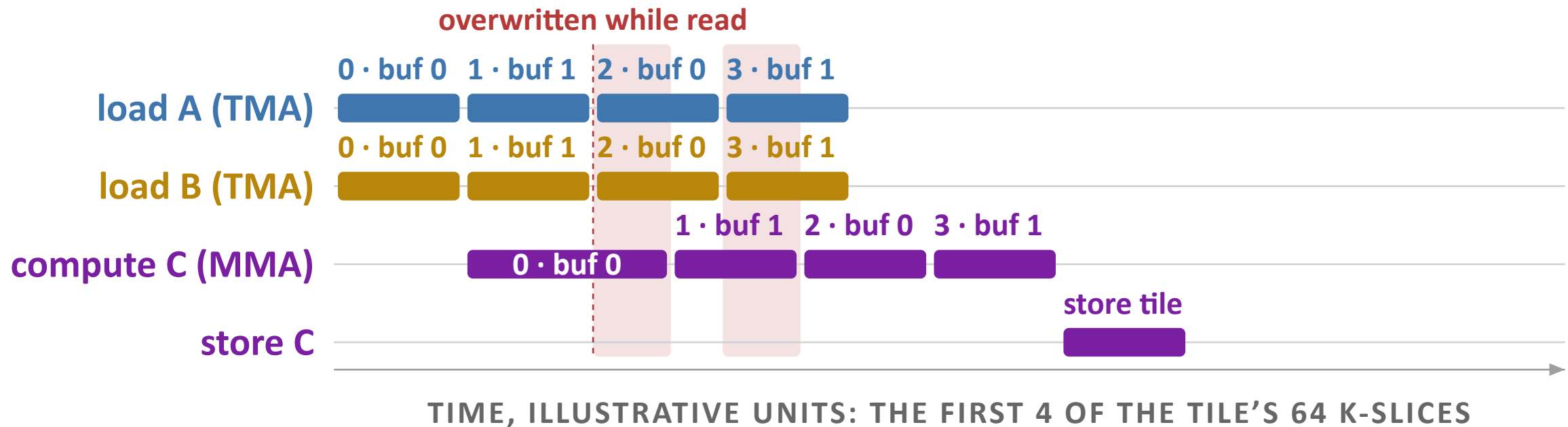
Instapoll · Review · Nov 3 B

A matrix-multiply kernel keeps two shared-memory buffers. The copy engine loads K-slice 0 into buffer 0 and slice 1 into buffer 1, while the tensor cores are still working on slice 0 from buffer 0. The copy engine is now free and the copy is asynchronous. Before it starts loading slice 2 into buffer 0, what must it wait for?

- A. Nothing — an asynchronous copy never has to wait
- B. The tensor cores to finish reading slice 0 out of buffer 0
- C. Slice 1 to finish arriving in buffer 1
- D. The finished output tile to be stored to global memory

One slow step overwrites live data

One change from the last frame: the MMA on slice 0 runs long

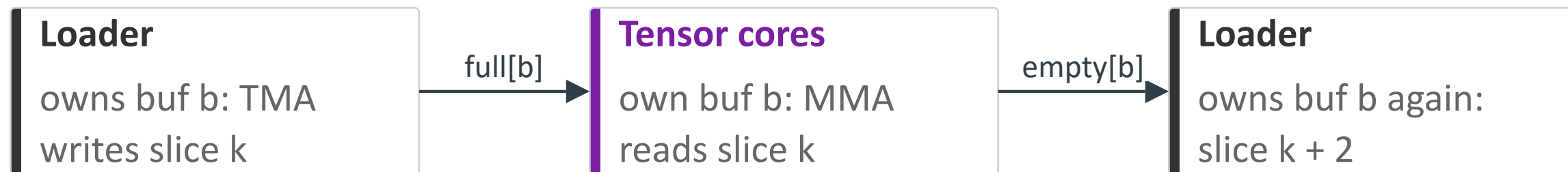


The loader finished slice 1 and moved on: slice 2 goes into buffer 0 while the tensor cores still read slice 0 there, and slice 3 does the same to slice 1. No error is raised; the output tile is silently wrong.

Asynchrony removed the wait, and with it the only thing that stopped the overwrite.

Two barriers pass ownership of each buffer

A barrier per hand-off: full[b] from loader to tensor cores, empty[b] back again



Both buffers start empty; each side waits only on the barrier it needs

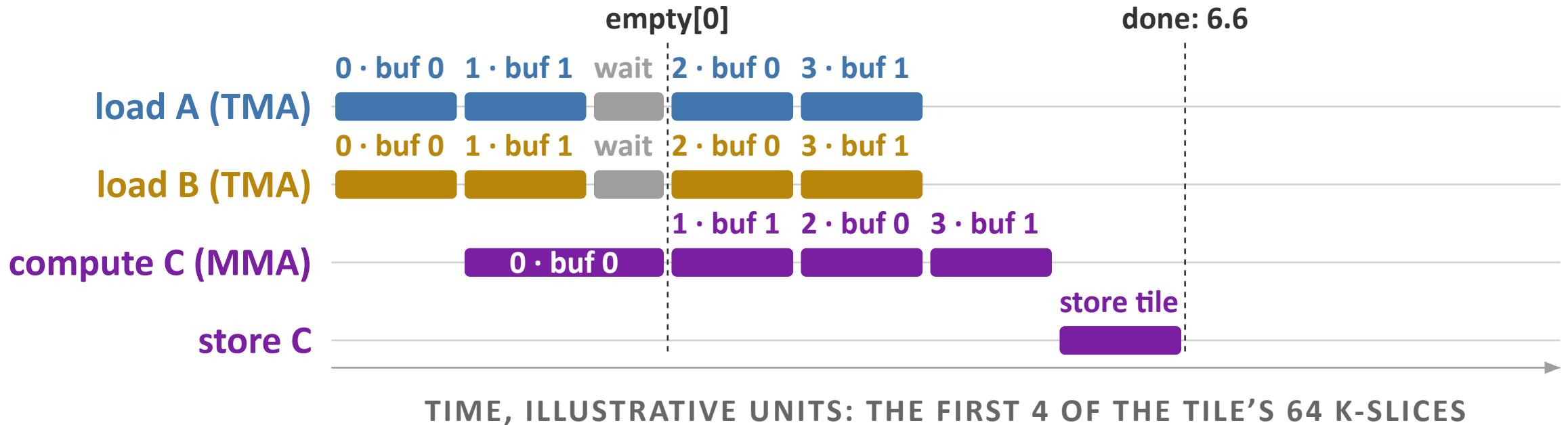
```
# loader
for k in range(64):
    b = k % 2
    empty[b].wait() # reader done?
    tma_load(buf[b], slice=k,
             signal=full[b])
```

```
# MMA issuer
for k in range(64):
    b = k % 2
    full[b].wait() # landed?
    mma(acc, buf[b],
         signal=empty[b])
```

full[b]: the bytes landed, MMA may read. empty[b]: MMA is done, the loader may overwrite.

The same slow step, with barriers

One change from the race: the loader waits for empty[0] before slice 2



The slow MMA now costs 0.6 units of delay instead of a corrupted tile. A deeper ring lets the loader run further ahead; the source kernel used five buffers.

Asynchrony takes the wait off the critical path only while every dependency still has an edge.

Warp specialization: one job per warp

Warps in one block take different jobs and meet only at the barriers

Load warp

Waits for empty[b], then asks TMA to copy slice k into buf b. The copy signals full[b] when its bytes land.

1 warp

MMA warp

Waits for full[b], issues the tensor-core MMA on buf b, and signals empty[b] once the MMA has read it.

1 warp

Epilogue warps

After the last slice: read the FP32 accumulator, convert to BF16, store the output tile.

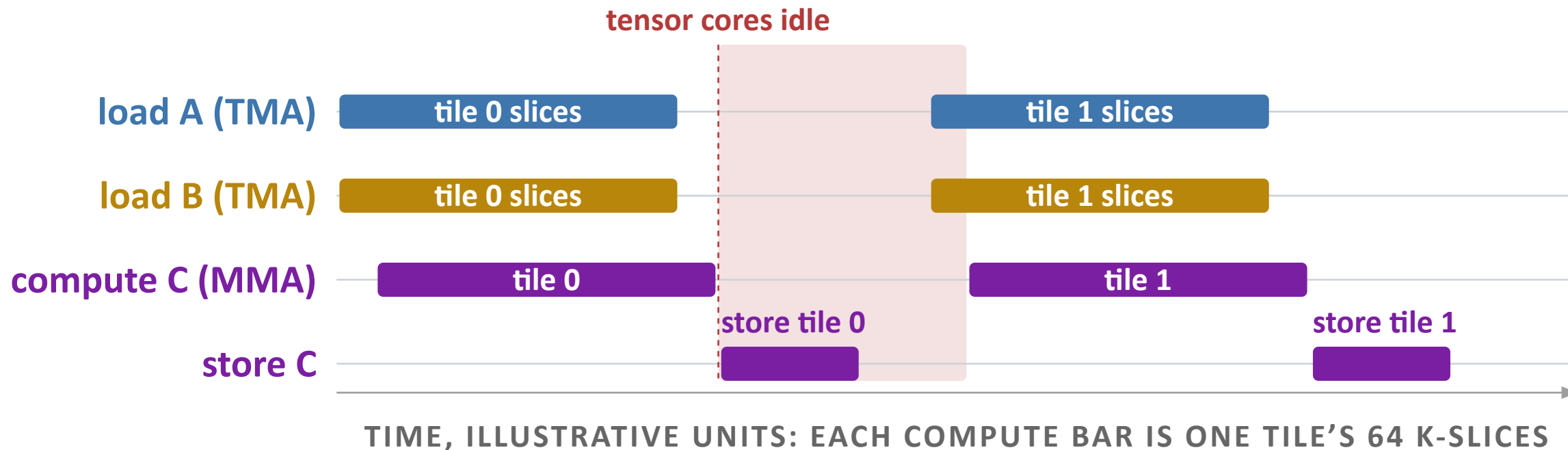
4 warps

Epilogue: the work after the last K-slice that moves the result out of the accumulator and into the output matrix. Same tensor-core instructions as the 360.2 kernel; what changed is that they stopped waiting.

Input pipeline with specialized warps: 360.2 → 1,429.6 TFLOP/s, about 4×.

Zoom out: between tiles the tensor cores idle

Same four tracks; each bar is now a whole output tile, one block per tile

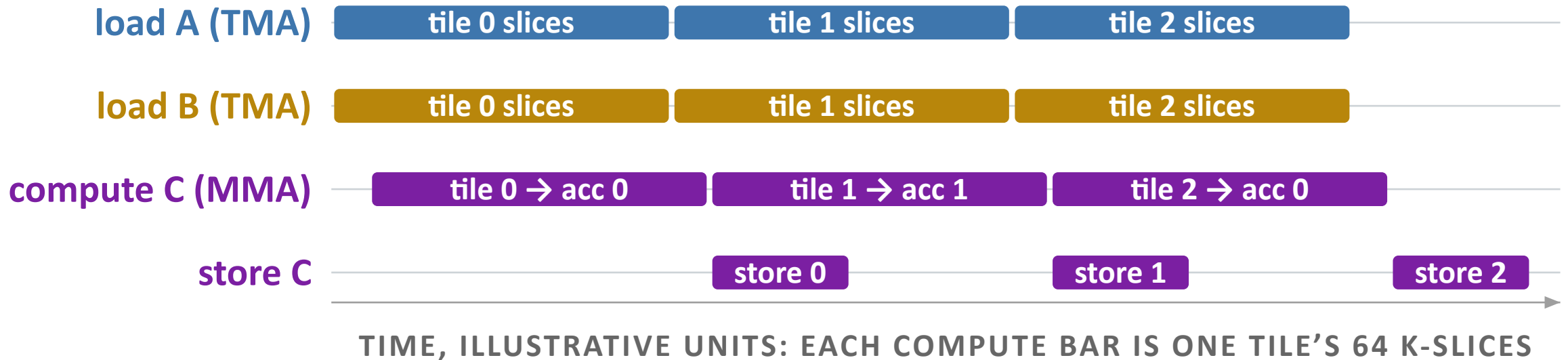


Each block computes one tile and exits: the next MMA waits for the epilogue to drain the accumulator and for a new block to set up. Pipelining the store (1,493.0) left this gap.

Between tiles, the tensor cores idle while the epilogue drains the accumulator.

Store tile t – 1 while computing tile t

One change: a persistent kernel with two accumulators

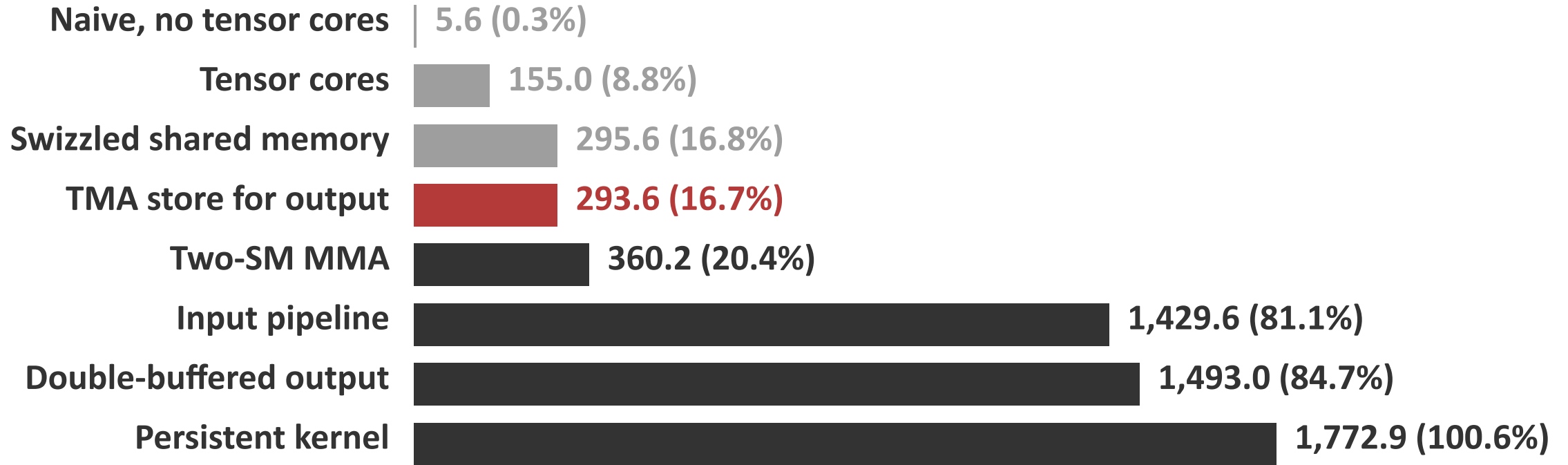


Persistent kernel: about one block per SM, looping over output tiles. Tensor memory (case-study GPU): FP32 accumulator storage beside the tensor cores; its halves are two accumulators.

The same full/empty protocol, now on the accumulator: 1,493.0 → 1,772.9 TFLOP/s.

The whole ladder, one bottleneck per rung

Reuse took the first 53x; scheduling took the last 6x



Light gray: rungs from the reuse lesson. Red: the rung that went backward. Source-reported TFLOP/s, 4096³ BF16, one B200; percent of its cuBLAS run.

The biggest single jump, 360.2 → 1,429.6, added no arithmetic: it removed a wait.

Tile order is a cache decision

Tile order: the sequence output tiles go to blocks; four blocks form one wave

WAVE 0 READS A0 + B0–B3
Row order: 5 panels

A PANEL	B PANEL			
	B0	B1	B2	B3
A0	0	0	0	0
A1	1	1	1	1
A2	2	2	2	2
A3	3	3	3	3

WAVE 0 READS A0, A1 + B0, B1
2×2 grouped order: 4 panels

A PANEL	B PANEL			
	B0	B1	B2	B3
A0	0	0	1	1
A1	0	0	1	1
A2	2	2	3	3
A3	2	2	3	3

Tiles that run together should share input panels, so a panel's second read hits in L2.

The shape chooses the schedule

A model's matmul is rarely square: Gemma 3 27B runs $512 \times 8192 \times 5376$

Output tile (two-SM MMA)	Blocks for 512×8192	On 148 SMs
256×256 , the 4096^3 choice	$4 \times 32 = 128$	20 SMs get no tile
256×224	$4 \times 37 = 148$	every SM gets one tile

- The same series reports 100.6% of cuBLAS at 4096^3 and 90% at 8192^3 .
- Its authors pick tile shape, pipeline depth, and tile order per shape with an autotuner: the sweep is mechanical, choosing the shapes to sweep is not.

A schedule tuned for one shape is a hypothesis about every other shape.

Audit the 0.6% claim: what it states

“Our kernel beats cuBLAS by 0.6%”, read as a card: every field is a condition

Field	What the source reports	What to ask before you believe 0.6%
Shape and layout	4096 × 4096 × 4096; A is M × K, B stored transposed	Is that the shape and layout your model runs?
Datatypes	BF16 inputs, FP32 accumulation, BF16 output	Does the baseline use the same precision contract?
GPU and software	One B200; kernels written in Mojo	Clocks, driver, CUDA, and compiler versions?
Baseline	cuBLAS at ≈ 1,763 TFLOP/s on the same shape	Which cuBLAS version and configuration, run how?

Stated fields bound the claim: one shape, one precision, one GPU, one baseline run.

Audit the 0.6% claim: what it leaves out

The missing fields are where a 0.6% claim fails

Field	What the source reports	What to ask before you believe 0.6%
Timing	Not in the post text	Timed region, warmup, repetitions, statistic, and spread
Correctness	Not in the post text	Checked against which reference, within what tolerance?
A second shape	Reported: 8192 ³ reaches 90% of cuBLAS	What about the shapes you actually run?
Spread	No run-to-run spread; one rung is printed as 288.3 in Part 2 and 295.6 in Part 4	Is 0.6% larger than the run-to-run variation on this machine?

Without repetitions and spread, 0.6% is indistinguishable from a tie.

What 1,772.9 does and does not prove

TFLOP/s on $4096 \times 4096 \times 4096$ BF16, one B200, against one cuBLAS run

It shows

- A hand-scheduled kernel can match the library on this GPU, for 4096^3 BF16
- Each rung's mechanism mattered once it removed the current bottleneck
- Overlap, not arithmetic, carried the last 6×

It does not show

- That it is faster than cuBLAS in general: 8192³ reached 90%
- That it is faster on a model's shapes without re-tuning
- That 0.6% exceeds run-to-run noise
- That its output matches a reference within tolerance

Treat a speed number as evidence about one setup; ask for the setup before you use it.

Key Takeaways

1. At 295.6 TFLOP/s the tensor cores sat idle half the time because each K-slice waited for its own load. Two buffers let slice $k + 1$ load while slice k computes; that overlap, not faster math, took the kernel from 360.2 to 1,429.6 TFLOP/s.
2. An asynchronous copy still needs ordering. A full barrier tells the tensor cores a buffer has landed; an empty barrier tells the loader they are done with it. Check both edges for every buffer, including a double-buffered accumulator.
3. 1,772.9 TFLOP/s is 100.6% of one cuBLAS run on 4096^3 BF16 on one B200; the same kernel reached 90% at 8192^3 . Before trusting a win, get the shape, datatypes, versions, baseline, timing, and correctness check, then test your own shapes.