

---

# Matrix multiply is a reuse problem

## Key takeaways

1. The naive kernel fetches each value from memory once for every output that uses it. A tile stages each value once and reuses it across many outputs.
2. Tensor cores multiply whole tiles per instruction, so they pay off only when staged tiles reach them in a layout free of shared-memory bank conflicts.
3. Mixed precision is a contract you state: BF16 storage halves the traffic, FP32 accumulation keeps long sums accurate, and C is rounded once.

# Common misunderstandings of fast matrix multiply

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

“Tensor cores are why matrix multiply is fast; everything else is tuning.”

“One thread per output, reading straight from memory, is the most parallel design, so it is the fastest.”

“Storing the inputs as 16-bit floats means the whole computation runs at 16-bit precision.”

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

# Where we are

From How GPUs differ from CPUs (Oct 27)

## The problem

$C = A \times B$ . One output  $C[i][j]$  is one dot product: row  $i$  of  $A$  with column  $j$  of  $B$ . Every output is independent.

## The machine

A grid of blocks; each block's threads run as warps of 32 lanes on one SM. Ready warps hide memory latency.

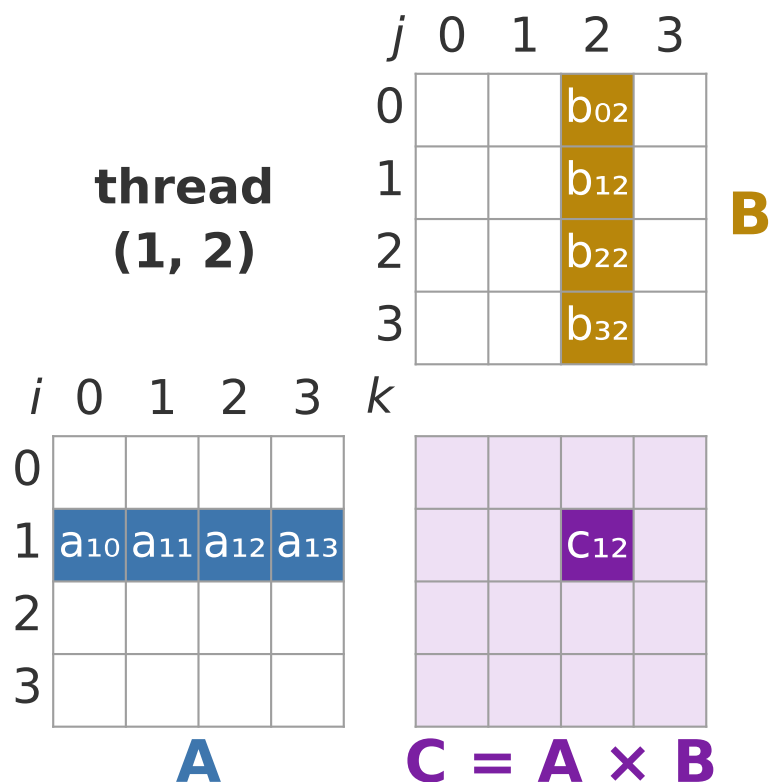
## The gap

Same GPU, same  $4096^3$  BF16 multiply: 5.6 TFLOP/s naive, 1,772.9 TFLOP/s tuned. Today climbs the first rungs.

Independent outputs gave us parallelism. Today's question: where do their inputs come from?

# The four-line kernel, again

One thread per output; every operand it needs comes from global memory.



The naive kernel from Oct 27:  $N \times N$ , one thread per output

```
// thread (i, j) computes C[i][j]
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;
```

- Row  $i$  of  $A$ :  $N$  loads. Column  $j$  of  $B$ :  $N$  loads.
- Two global-memory loads per multiply-add.

Thread (1, 2) and thread (1, 3) both read all of row 1 of  $A$ .

# Count the reuse: 128 reads of 32 values

Each A value is read once per output in its row; each B value once per output in its column.

Diagram illustrating the dot product of two 4x4 matrices  $A$  and  $B$  to produce a 4x4 matrix  $C$ .

Matrix  $A$  (reads per value):

| $i$ | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0   | 4 | 4 | 4 | 4 |
| 1   | 4 | 4 | 4 | 4 |
| 2   | 4 | 4 | 4 | 4 |
| 3   | 4 | 4 | 4 | 4 |

Matrix  $B$  (reads per value):

| $j$ | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0   | 4 | 4 | 4 | 4 |
| 1   | 4 | 4 | 4 | 4 |
| 2   | 4 | 4 | 4 | 4 |
| 3   | 4 | 4 | 4 | 4 |

Matrix  $C$  (result of  $A \times B$ ):

| $i$ | 0  | 1  | 2  | 3  |
|-----|----|----|----|----|
| 0   | 16 | 16 | 16 | 16 |
| 1   | 16 | 16 | 16 | 16 |
| 2   | 16 | 16 | 16 | 16 |
| 3   | 16 | 16 | 16 | 16 |

The calculation shown is the dot product of the first row of  $A$  and the first column of  $B$ :

$$[4, 4, 4, 4] \cdot [4, 4, 4, 4] = 16 + 16 + 16 + 16 = 64$$

However, the diagram shows the result of the dot product as 16, which is the value of a single element in the resulting matrix  $C$ .

- $A[1][0]$  is read by the four threads computing row 1 of C.
- Every A and B value: 4 reads, one per output that uses it.
- $16 \text{ threads} \times 8 \text{ reads} = 128 \text{ reads}$  of only 32 distinct values.
- For  $n \times n$  matrices, each value is read  $n$  times:  $4,096$  times at  $4096^3$ .

The same numbers cross from memory again and again, once per output that needs them.

# Instapoll

Instapoll · Review · Nov 3 A

A matrix-multiply kernel gives each output its own thread, and every multiply-add reads its two operands from the GPU's main memory. An agent keeps that structure but makes each multiply-add 50 times cheaper. What speedup should you expect?

- A. Close to 50×, because the multiply-adds are where the kernel does its work
- B. Well under 2×, because each multiply-add still waits on two memory reads
- C. About 10×, because the caches already serve some of the repeated reads
- D. None until the thread count also rises 50× to keep the faster units busy

# Memory, not arithmetic, caps the naive kernel

Arithmetic intensity: floating-point operations per byte fetched from memory. The naive kernel does 2 FLOPs per 4 bytes: 0.5.

**$\leq 3.8$  TFLOP/s**

Ceiling if every operand came from GPU memory:  $7.672 \text{ TB/s} \div 4 \text{ bytes per multiply-add} \times 2 \text{ FLOPs}$ .

**5.6 TFLOP/s**

Measured naive kernel. The caches catch some repeats, so it beats the ceiling, but not by much.

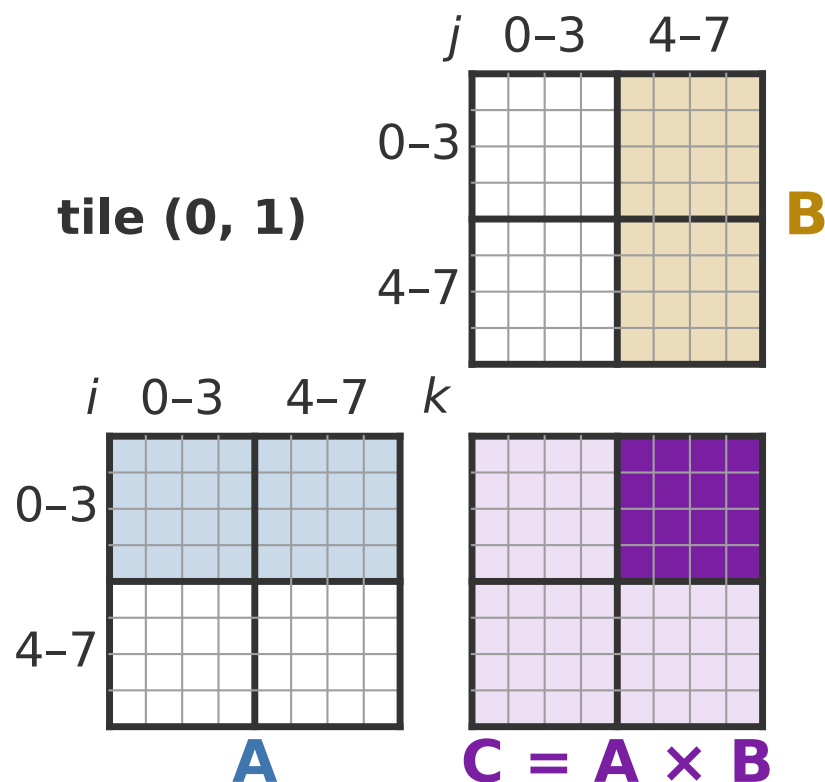
**2,250 TFLOP/s**

BF16 peak. Reaching it needs about 290 FLOPs per byte fetched, almost 600 times the naive kernel's 0.5.

Faster arithmetic cannot help until each value fetched feeds hundreds of multiply-adds.

# Split C into tiles; one block owns one tile

A tile is a block of neighboring outputs that one thread block computes together.

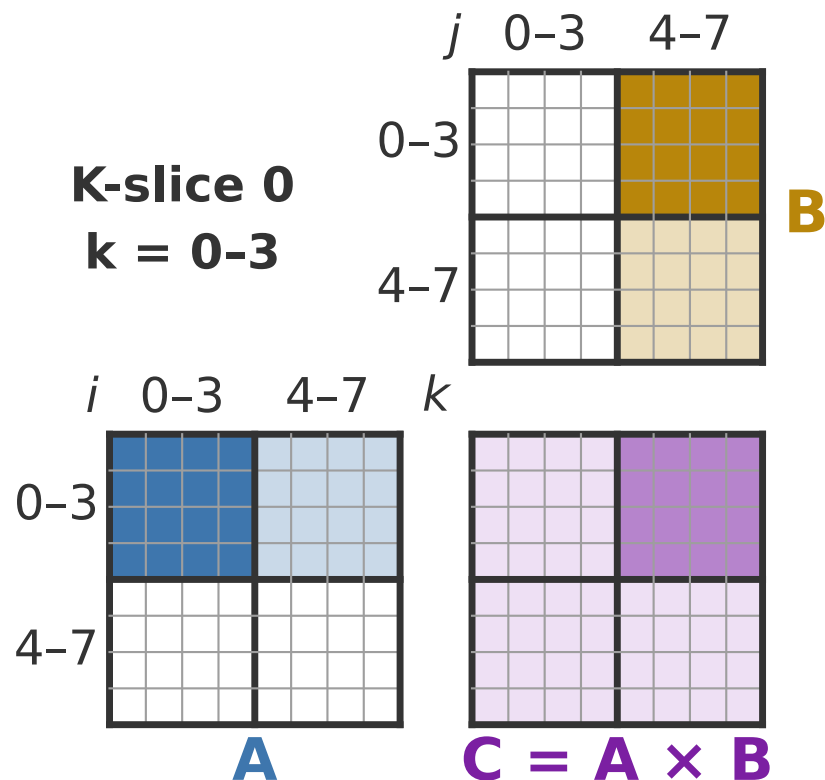


- $8 \times 8$  C, cut into four  $4 \times 4$  tiles: four independent blocks.
- Tile (0, 1) is rows 0–3, columns 4–7 of C.
- It needs rows 0–3 of A and columns 4–7 of B: all of K.
- Its 16 outputs share those rows and columns. Each A value feeds four of them.

Neighbors in C need the same inputs. A tile lets them share one copy.

# Walk K in slices: slice 0

A K-slice is a run of  $k$  values. The block loads the A and B pieces for that run, then computes.

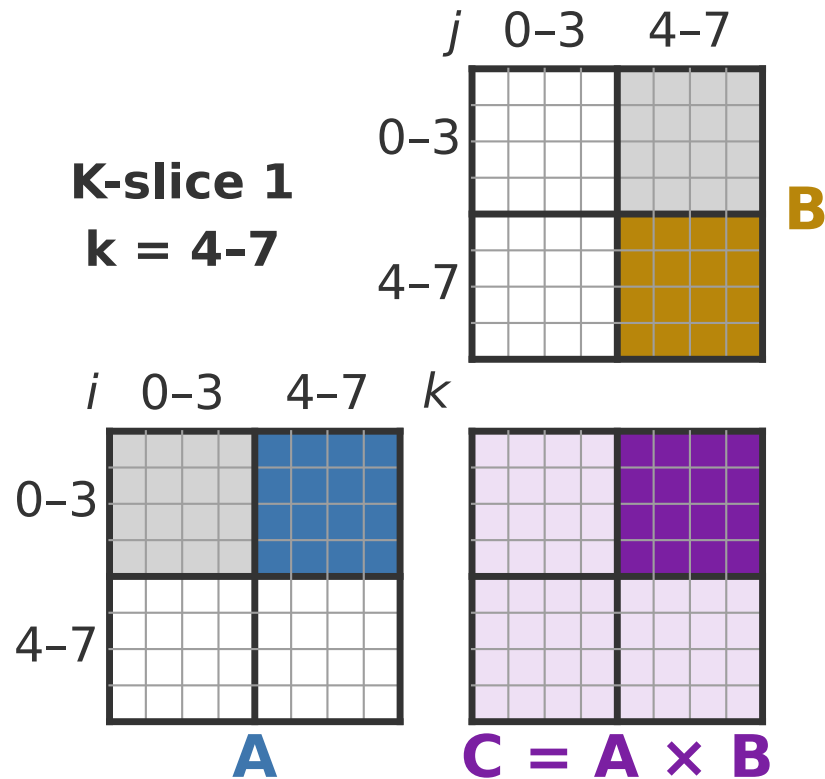


- Load  $A[0-3][0-3]$  and  $B[0-3][4-7]$ : 32 values.
- Then  $4 \times 4 \times 4 = 64$  multiply-adds, all from the loaded copy.
- Each loaded value is used 4 times.
- The 16 partial sums stay with their threads.

32 loads for 64 multiply-adds: half a load each, down from two.

# Slice 1 adds to the same partial sums

The block streams through K; the outputs stay put until they are final.



- Load  $k = 4 \dots 7$  into the same space, over slice 0.
- The same 16 sums keep accumulating.
- $K / 4 = 2$  slices, then the tile is final and written to C once.
- Whole tile: 64 loads for 128 multiply-adds. The naive threads made 256.

Same outputs, a quarter of the memory reads: the reuse was always there to collect.

# Shared memory and its two barriers

Shared memory: on-chip, filled by the block. Barrier: no thread passes until all arrive.

**Tiled, BF16 inputs:  $T \times T$  threads per block, one per output**

```
__shared__ bf16 As[T][T], Bs[T][T];
float acc = 0.0f;
for (int k0 = 0; k0 < N; k0 += T) {
    As[ty][tx] = A[i*N + k0 + tx];
    Bs[ty][tx] = B[(k0 + ty)*N + j];
    __syncthreads();                // (1)
    for (int k = 0; k < T; k++)
        acc += float(As[ty][k])
               * float(Bs[k][tx]);
    __syncthreads();                // (2)
}
C[i*N + j] = __float2bfloat16(acc);
```

## (1) After the loads

No thread reads until all have stored, or thread (0, 3) can read `As[0][2]` before thread (0, 2) writes it.

## (2) After the compute

No thread overwrites the slice while another is still reading it.

The slice changes hands twice per loop, and each handoff needs a barrier.

# Reuse factor equals tile width

Reuse factor: how many multiply-adds each loaded value feeds. For a  $T \times T$  tile it is  $T$ .

| Tile                                            | Loads per multiply-add | Uses per loaded value | Shared memory per slice (BF16) |
|-------------------------------------------------|------------------------|-----------------------|--------------------------------|
| Naive, one output                               | 2                      | 1                     | none                           |
| $4 \times 4$ , 4-wide slices                    | 0.5                    | 4                     | 64 bytes                       |
| $64 \times 64$ , 64-wide slices (source kernel) | 1/32                   | 64                    | 16 KB of the SM's 228 KB       |

Bigger tiles buy reuse with shared memory, and a bigger tile means fewer blocks to spread across the SMs.

# Tensor cores consume tiles, not scalars

A tensor core runs a matrix multiply-accumulate (MMA),  $D += A \times B$  on tiles, per instruction.

---

## ORDINARY LANES Scalar FMA

```
acc += a * b;    // one per lane
```

- One multiply-add per lane per instruction.
- A warp instruction does 32.
- Operands come from registers.

---

## CASE-STUDY GPU: BLACKWELL Tensor-core MMA

```
D[128×256] += A[128×16] × B[16×256]
```

- One instruction: 524,288 multiply-adds.
- A and B tiles are read from shared memory.
- Asynchronous: runs for many cycles across the SM's tensor cores.

Tensor cores multiply what tiling stages. Without tiles in shared memory there is nothing to feed them.

# Precision is part of the contract

Accumulation precision: the format of the running sum, chosen apart from the storage format.

## The contract

|                   |                                  |
|-------------------|----------------------------------|
| <b>Store</b>      | A, B in BF16: 2 bytes, ~3 digits |
| <b>Multiply</b>   | BF16 × BF16 is exact in FP32     |
| <b>Accumulate</b> | FP32 running sum across all of K |
| <b>Write C</b>    | round to BF16 once, at the end   |

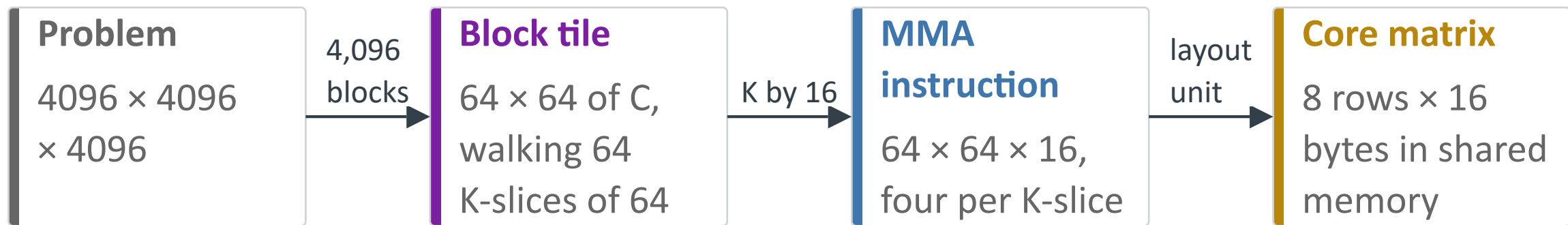
## Why the sum stays wide

| 4,096 products in [0, 1), simulated | Sum     |
|-------------------------------------|---------|
| Exact                               | 1,017.1 |
| FP32 sum, rounded once              | 1,016   |
| BF16 sum at every step              | 256     |
| (past 256, BF16 steps by 2)         |         |

Tell the agent the contract. A BF16 accumulator can be faster and silently wrong.

# Tiles nest down to the core matrix

One multiply, four tile sizes. Each level is the unit the level below consumes.



$4096 \div 64 = 64$  K-slices per block. Tensor cores and the copy engine read shared memory as core matrices, which is why the layout on the next slide matters.

Tile sizes are related but not interchangeable: each one answers to a different piece of hardware.

# Shared memory has 32 banks

A bank serves one 4-byte word per cycle. A bank conflict is two lanes wanting different words from one bank: they take turns.

one 128-byte row of the tile = all 32 banks, 4 bytes each

| row | 0-3       | 4-7 | 8-11 | 12-15 | 16-19 | 20-23 | 24-27 | 28-31 |
|-----|-----------|-----|------|-------|-------|-------|-------|-------|
| 0   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 1   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 2   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 3   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 4   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 5   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 6   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |
| 7   | <b>c0</b> | c1  | c2   | c3    | c4    | c5    | c6    | c7    |

- Consecutive 4-byte words sit in consecutive banks.
- A 64-wide BF16 row is 128 bytes: one pass over all 32 banks.
- So chunk c of every row lands in the same four banks.
- One core matrix is chunk 0 of rows 0–7: 8 passes, not 1.

The natural row-by-row tile makes every core-matrix access an 8-way bank conflict.

# Swizzle: XOR moves chunks, not the matrix

A swizzle stores logical chunk  $c$  of row  $r$  at physical chunk  $c \text{ XOR } (r \bmod 8)$ .

one 128-byte row of the tile = all 32 banks, 4 bytes each

| row | 0-3       | 4-7       | 8-11      | 12-15     | 16-19     | 20-23     | 24-27     | 28-31     |
|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 0   | <b>c0</b> | c1        | c2        | c3        | c4        | c5        | c6        | c7        |
| 1   | c1        | <b>c0</b> | c3        | c2        | c5        | c4        | c7        | c6        |
| 2   | c2        | c3        | <b>c0</b> | c1        | c6        | c7        | c4        | c5        |
| 3   | c3        | c2        | c1        | <b>c0</b> | c7        | c6        | c5        | c4        |
| 4   | c4        | c5        | c6        | c7        | <b>c0</b> | c1        | c2        | c3        |
| 5   | c5        | c4        | c7        | c6        | c1        | <b>c0</b> | c3        | c2        |
| 6   | c6        | c7        | c4        | c5        | c2        | c3        | <b>c0</b> | c1        |
| 7   | c7        | c6        | c5        | c4        | c3        | c2        | c1        | <b>c0</b> |

- Chunk 0 of rows 0–7 now sits in 8 different bank groups.
- Same core matrix, one pass.
- Values and logical coordinates are unchanged.
- The copy that writes the tile and the MMA that reads it use the same XOR.

A swizzle changes addresses, not values. The same read now costs one pass instead of eight.

# Each rung removes one named bottleneck

Source-reported TFLOP/s, 4096<sup>3</sup> BF16 on one B200; percent of the authors' cuBLAS run.

## Naive

5.6 0.3%

## Tiles + tensor cores

155.0 8.8%

## + swizzled layout

295.6 16.8%

## Final (next deck)

1,772.9 100.6%

- 5.6: every multiply-add waits on memory.
- 155.0: tiles cut the reads; MMAs replace scalar FMAs.
- 295.6: the swizzle ends 8-way bank conflicts.
- 1,772.9: a schedule of data movement. Next deck.

At 17% the tensor cores still wait. What remains is a schedule for feeding them.

# Key Takeaways

1. Matrix multiply is limited by memory before arithmetic: the naive kernel reads every operand from global memory each time an output uses it. A  $T \times T$  tile loads each value once per K-slice and uses it  $T$  times, so ask how many multiply-adds each fetched byte feeds.
2. Tensor cores execute a whole tile multiply-accumulate per instruction, which only helps once tiles are staged in shared memory. Layout matters too: a swizzle spreads each core matrix across the banks and nearly doubled the source kernel's speed.
3. Precision belongs in the specification. Store BF16 to halve the traffic, accumulate in FP32 so long sums keep their small terms, round once on store, and have the agent check results against a reference within a stated tolerance.

# Tile reuse worksheet: $8 \times 8$ , $K = 8$

10 min

In pairs.  $C = A \times B$ , all  $8 \times 8$ , BF16. Fill one row per plan: global reads, distinct values read, uses per loaded value, number of blocks.

## Plan A

One thread per output;  
every operand read from  
global memory.

## Plan B

Four  $4 \times 4$  tiles, one block  
each, staging 4-wide  
K-slices in shared memory.  
Mark where its barriers go.

## Plan C

An agent proposes one  
 $8 \times 8$  tile in one block, for  
twice the reuse.

- Pick a plan for a GPU with many SMs and defend it to the next pair.
- Then: what breaks if a tile outgrows shared memory? If every lane of a warp reads a different word from the same bank?