

Modern AI Systems: Agents and System Optimizations

LLM Serving Basics II: Intro to GPUs

Juncheng Yang



Harvard John A. Paulson
School of Engineering
and Applied Sciences



HARVARD UNIVERSITY

MaDSys

Measurement and Design of Systems Lab

Logistics

- Assignment 4 will go out today and due in two and half weeks
 - **Goal: reason about performance, bottlenecks and improve iteratively, how real-world workloads differ from synthetic benchmarks**
 - Write-up is important, you only need to write a small number of lines of code
 - NO LATE DAY for leaderboard submission. DO NOT grab more than one GPU
 - Hard if you have limited prior experience in systems programming
- Proposal due next Wednesday (Oct 28)
 - No limit on format or page
 - DO NOT upload AI-generated content

Plan for today

- Introduction to GPU for LLM serving^[1]
 - GPU basics
 - GPU architecture
 - GPU programming and recent kernel advances

[1] We will not go into details; the goal is to help you build intuition

GPU Basics



Everything you must know about GPUs

GPU form factors

Consumer · PCIe card in a desktop



Data center · 8-GPU board in a server



2 x Management top of rack
Ethernet switches

4 x Power shelves

10 x Compute Nodes

9 x NVLink switches

8 x Compute Nodes

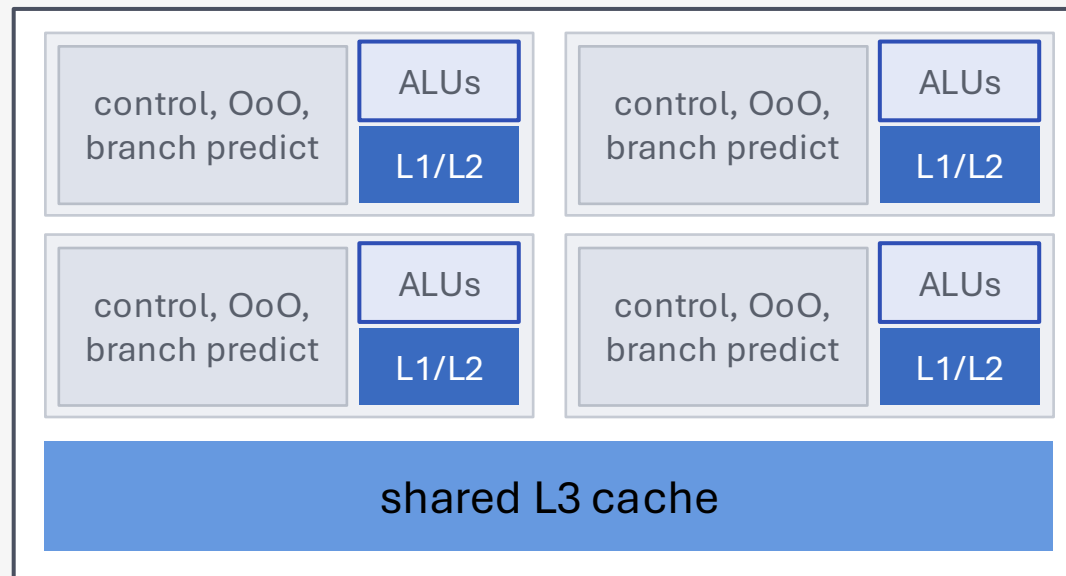
4 x Power shelves

Seismic bracing



A GPU is a throughput machine

CPU · optimized for one thread

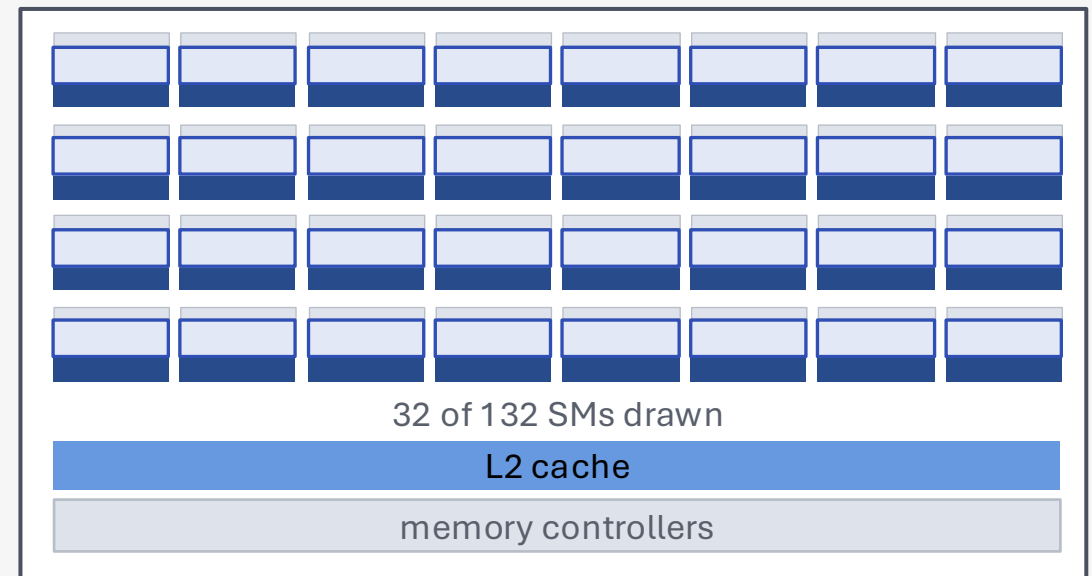


memory: a few DDR channels

control and speculation arithmetic units

 **Sports car:** 2 riders, fastest trip

GPU · optimized for the whole chip



memory: HBM stacks, thousands of bits wide

registers private caches shared cache

 **Bus fleet:** slower, moves thousands

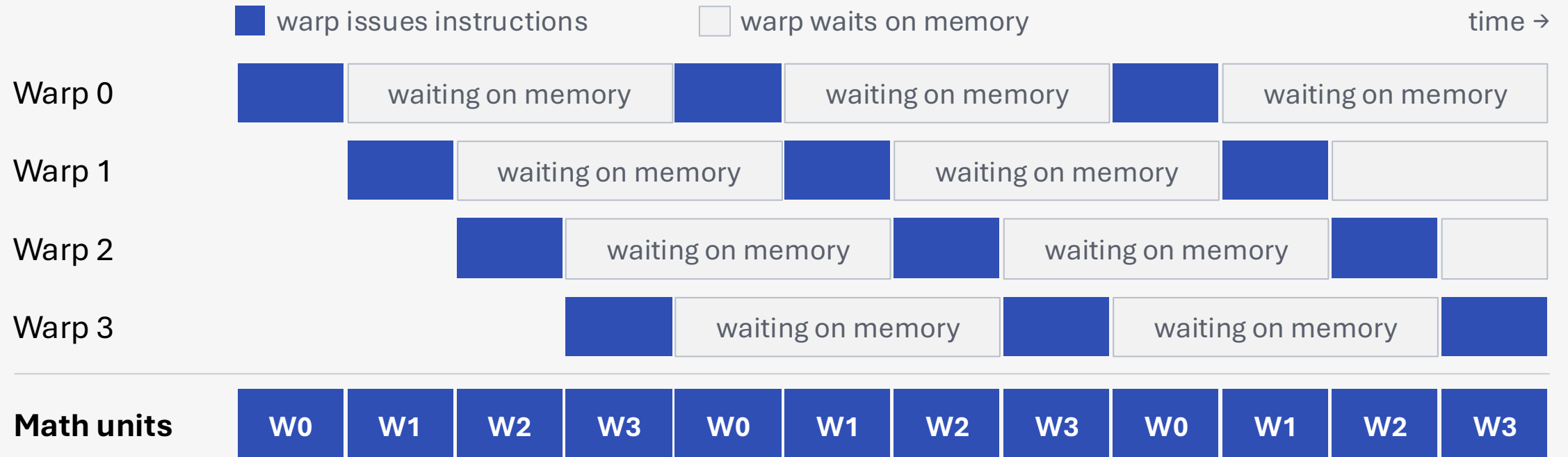
A GPU trades a few fast cores for thousands of simple ones

	64-core server CPU	NVIDIA H200 SXM
Parallel units	64 cores	132 SMs, 16,896 FP32 cores
Threads in flight	128	270,336 (2,048 per SM)
Memory bandwidth	~0.46 TB/s (DDR5)	4.8 TB/s (HBM3e)
Peak dense math	~5 TFLOPs FP32	989 TFLOPs BF16
Hides latency by	Caches, out-of-order cores	Switching to a ready warp

On a cache miss, a CPU thread runs ahead until its out-of-order window fills, then stalls. A GPU switches to another ready warp.

Why it fits LLMs: every token runs the same matmuls on the same weights, in parallel.

Switching warps hides memory latency



Why it works: while one warp waits, another issues, so the math units stay busy.

A100/H200/B200: each warp scheduler (4 in each SM) manages up to 16 warps, so 64 resident warps per SM.

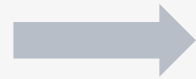
SIMT: single instruction, multiple threads

One thread's program

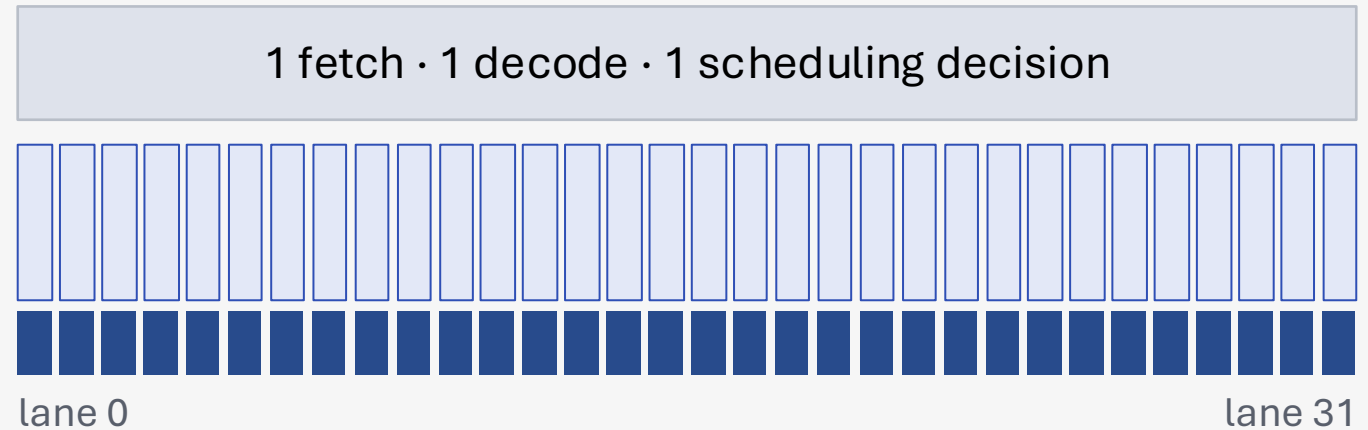
```
i = threadIdx.x  
y[i] = a*x[i] + b
```

scalar code, written once

32 threads



One warp: threads 0–31 of a block

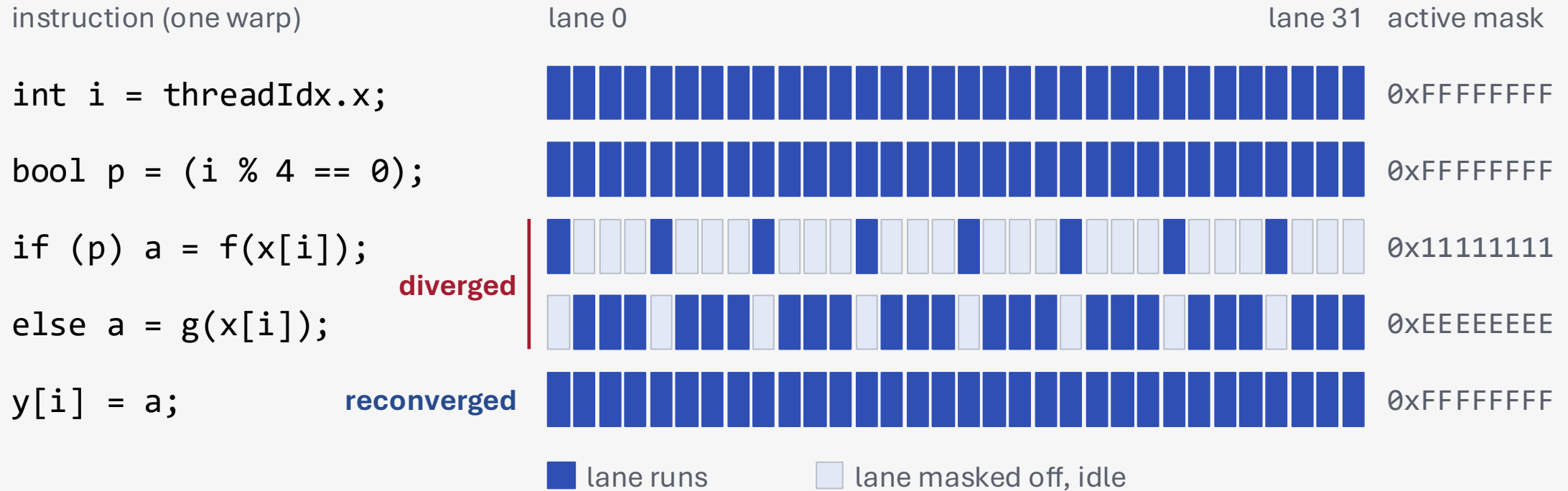


□ execution lane ■ its own registers

Why it is efficient: one warp instruction does the work of **32 scalar instructions**.

Control is paid once and shared by 32 lanes, like a SIMD vector instruction on a CPU.

Branch divergence



The cost: the warp runs both paths in turn; f uses 8 of 32 lanes, g uses 24.

The trade-offs behind the GPU

THE GPU GIVES UP



Single-thread speed. Modest clock, no speculation: serial code runs far slower.



Parallelism has to be explicit. ~270K threads fill 132 SMs; small jobs leave the chip idle.



Control flow is shared. Threads run in groups of 32; divergent branches cost throughput.



On-chip memory per thread is small. About 240 bytes per thread at full occupancy.



IN EXCHANGE IT GETS

Area and power go to **thousands of arithmetic units.**

Latency hidden by **switching warps**, not by caches and speculation.

One instruction fetch and decode drives **32 cores.**

Every thread's state stays resident, so **warp switches are free.**

Three specs set the limits of every GPU

Peak BF16 compute

A100	312 TFLOP/s
H200	989 TFLOP/s
B200	2,250 TFLOP/s
Rubin	≈ 4,000 TFLOP/s

Limits how fast the math runs

Memory bandwidth

A100	2.0 TB/s
H200	4.8 TB/s
B200	8 TB/s
Rubin	≈ 22 TB/s

Limits how fast weights and KV cache are read

Memory capacity

A100	80 GB HBM2e
H200	141 GB HBM3e
B200	180 GB HBM3e
Rubin	288 GB HBM4

Limits how many weights and KV-cache entries fit

Intuition: compute is the engine, bandwidth the fuel line, capacity the tank

Compute reaches peak only when each byte read feeds enough FLOPs

Decode, batch 1

3.3 ms per token

HBM reads

16 GB of weights

Tensor cores

idle 99.5% of the time

Decode: compute waits

Each weight is used once, so 0.016 ms of math waits on 3.3 ms of reads.

Prefill, 2,048 tokens

33 ms per prompt

HBM reads

idle once weights are loaded

Tensor cores

33 TFLOP, busy near peak

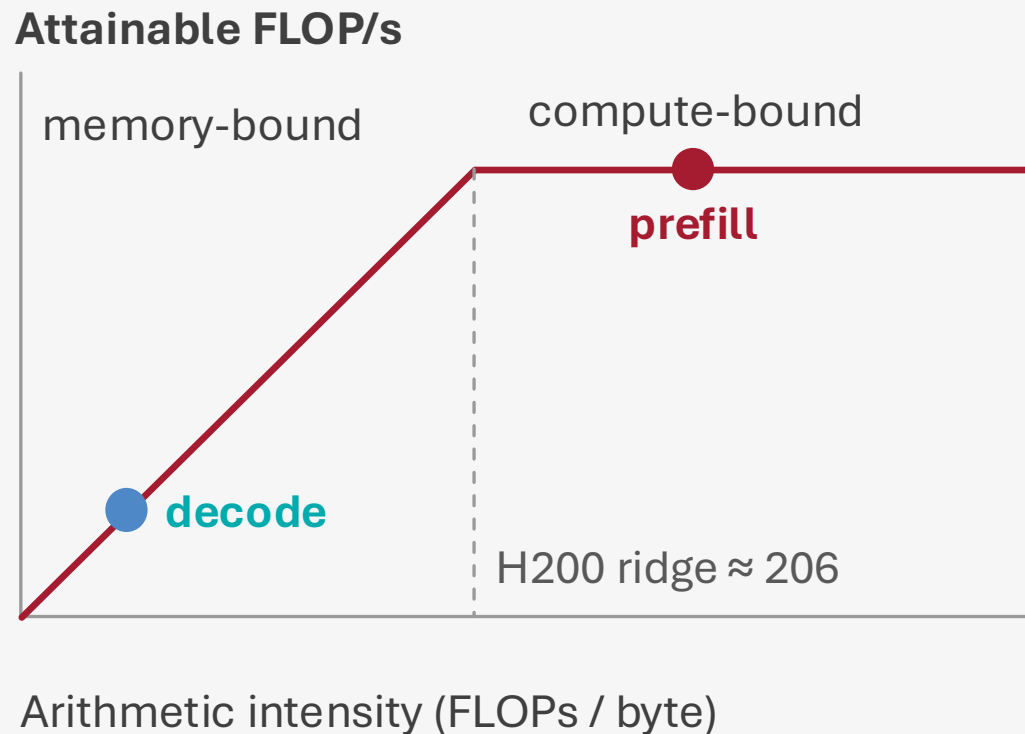
Prefill: compute is busy

Each weight feeds 2,048 tokens, so reads finish early and math sets the pace.

Kernel time $\approx \max(\text{bytes} \div \text{bandwidth}, \text{FLOPs} \div \text{peak})$.

The roofline shows whether memory or compute limits a kernel

Arithmetic intensity = FLOPs performed $\div$ bytes moved to or from HBM



Left of the ridge

memory-bound: speed = bandwidth $\times$ intensity

Right of the ridge

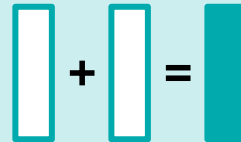
compute-bound: speed = peak FLOP/s

Ridge = peak FLOP/s $\div$ bandwidth

Common arithmetic intensity

Intensity = FLOPs ÷ bytes (b bytes per element)

Vector add, length N
1 FLOP per 3 elements moved



$$N \div 3Nb = \mathbf{1/(3b)}$$

GEMV (decode), $N \times N$
Each weight used once



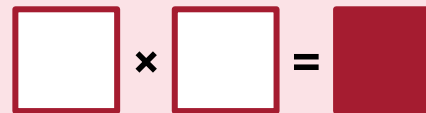
$$2N^2 \div (N^2 + 2N)b \approx \mathbf{2/b}$$

Decode attention, $n \times d$
Each KV entry used once per head



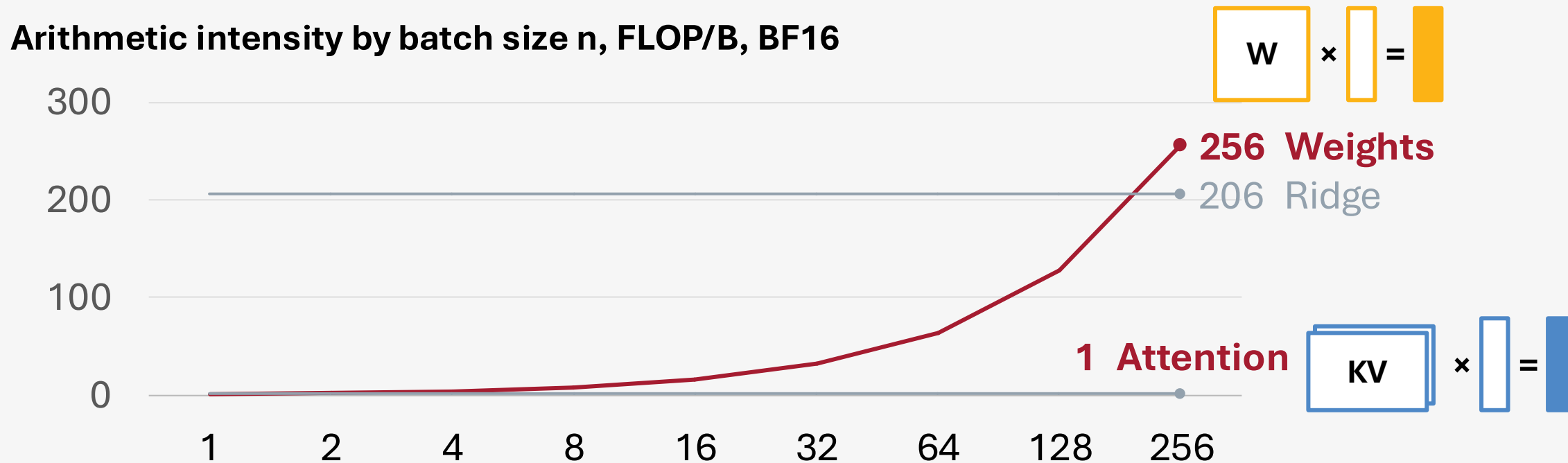
$$4nd \div 2ndb = \mathbf{2/b}$$

Square GEMM, $N \times N$
Each element reused $\sim N$ times



$$2N^3 \div 3N^2b = \mathbf{2N/(3b)}$$

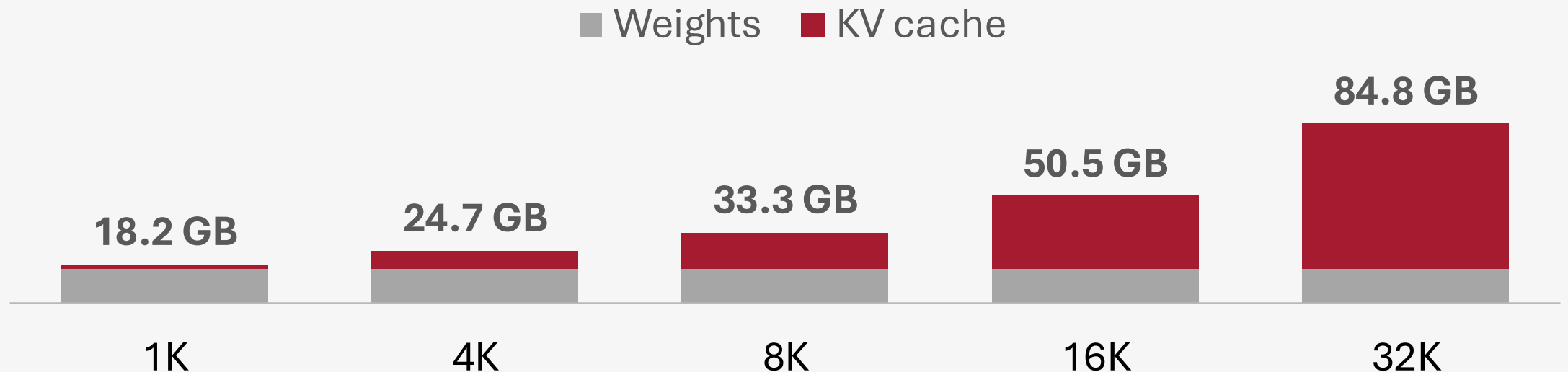
Does batching help decode?



Batching lifts weight GEMVs toward ridge, but decode attention stays at $2/b$

Does context length help decode?

HBM bytes read per decode step by context length, Llama 3 8B, batch 16, BF16



Weights: fixed 16.1 GB
read once per step at any L

KV cache: 128 KiB/token
grows with batch $\times$ L

Step $\approx$ bytes $\div$ 4.8 TB/s
3.8 ms at 1K, 17.7 ms at 32K

How busy is an H200 during batch-1 decode?

What you know

Peak compute: 989 TFLOP/s (BF16, dense)

HBM bandwidth: 4.8 TB/s

Batch-1 decode: a GEMV doing about 1 FLOP per byte of weights read

Your estimate of the share of peak compute used

A $\approx 50\%$

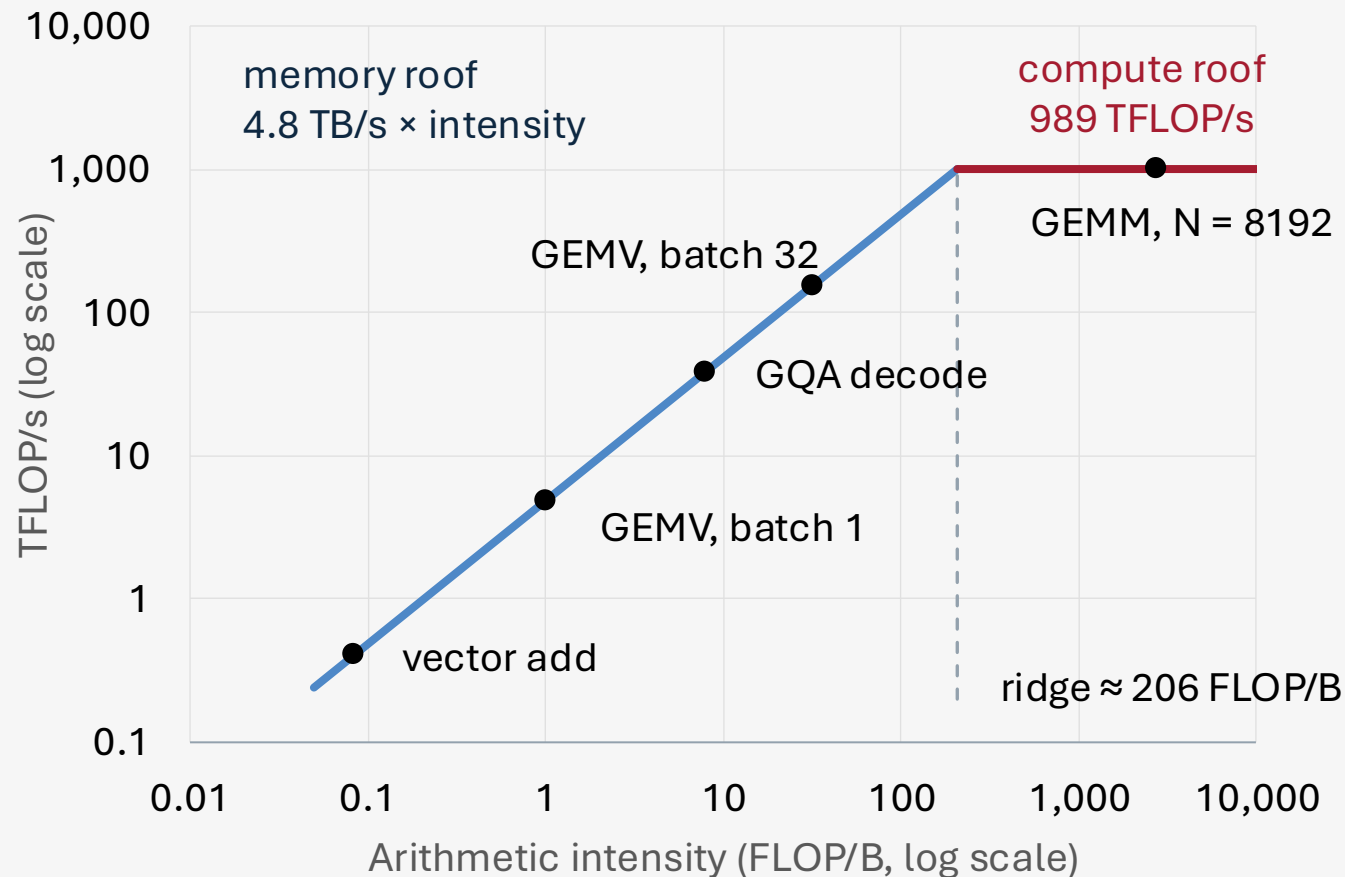
B $\approx 10\%$

C $\approx 2\%$

D $< 1\%$

Then: what would you change to raise it?

On an H200 roofline, batch-1 decode runs at 0.5% of peak compute



Decode is bandwidth-bound

Batch-1 GEMV at ≈ 1 FLOP/B reaches about 5 of 989 TFLOP/s.

Increase arithmetic intensity

Batch more, quantize, shrink KV traffic.

Prefill is compute-bound

Large GEMMs sit on the flat roof; only faster math or fewer FLOPs help.

Trend: lower precision pushes the ridge point to the right

Ridge point = dense peak $\div$ HBM bandwidth, FLOP/B

GPU	HBM	FP32	BF16	FP8	FP4
A100 SXM	2.04 TB/s	9.6	153	–	–
H100 SXM5	3.35 TB/s	20	295	591	–
H200	4.8 TB/s	14	206	412	–
B200 HGX	8 TB/s	9.4	281	563	1,125
MI355X	8 TB/s	–	315	629	1,259
Rubin (prelim.)	22 TB/s	–	182	795	1,591

FP4 pushes the ridge right

Rubin's dense NVFP4 needs $\approx$ 1,600 FLOP per byte to be compute-bound

Compute outgrows bandwidth

From A100 to Rubin, BF16 grows $\approx$ 13 $\times$ and bandwidth $\approx$ 9 $\times$, so kernels are increasingly bandwidth-bound.

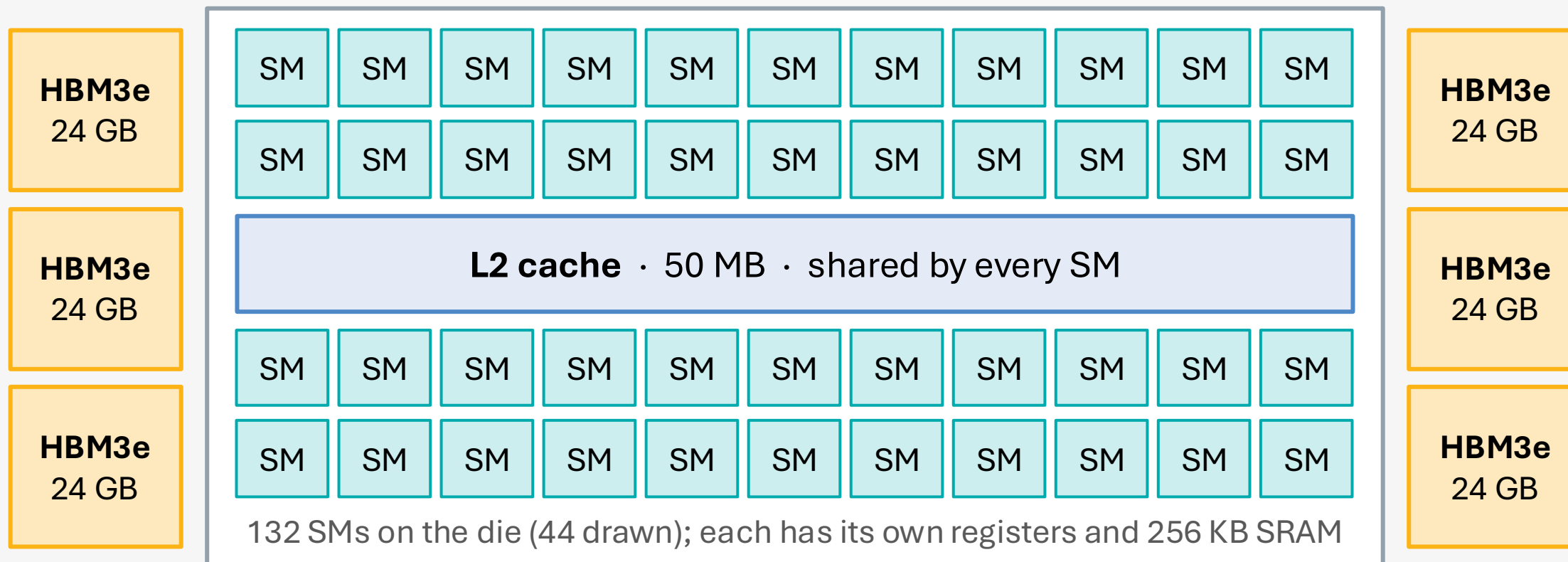
GPU basics: what to take away

- 1 Throughput machine:** a GPU trades fast single threads for thousands of parallel cores.
- 2 Three limits:** peak compute, HBM bandwidth and memory capacity bound every kernel.
- 3 Roofline:** the ridge point (206 FLOP/B on H200) separates memory-bound from compute-bound kernels.
- 4 Decode:** batching lifts weight reads toward the ridge; KV-cache reads stay memory-bound.
- 5 Trend:** newer GPUs move the ridge right; more kernels become bandwidth-bound.

GPU Architecture

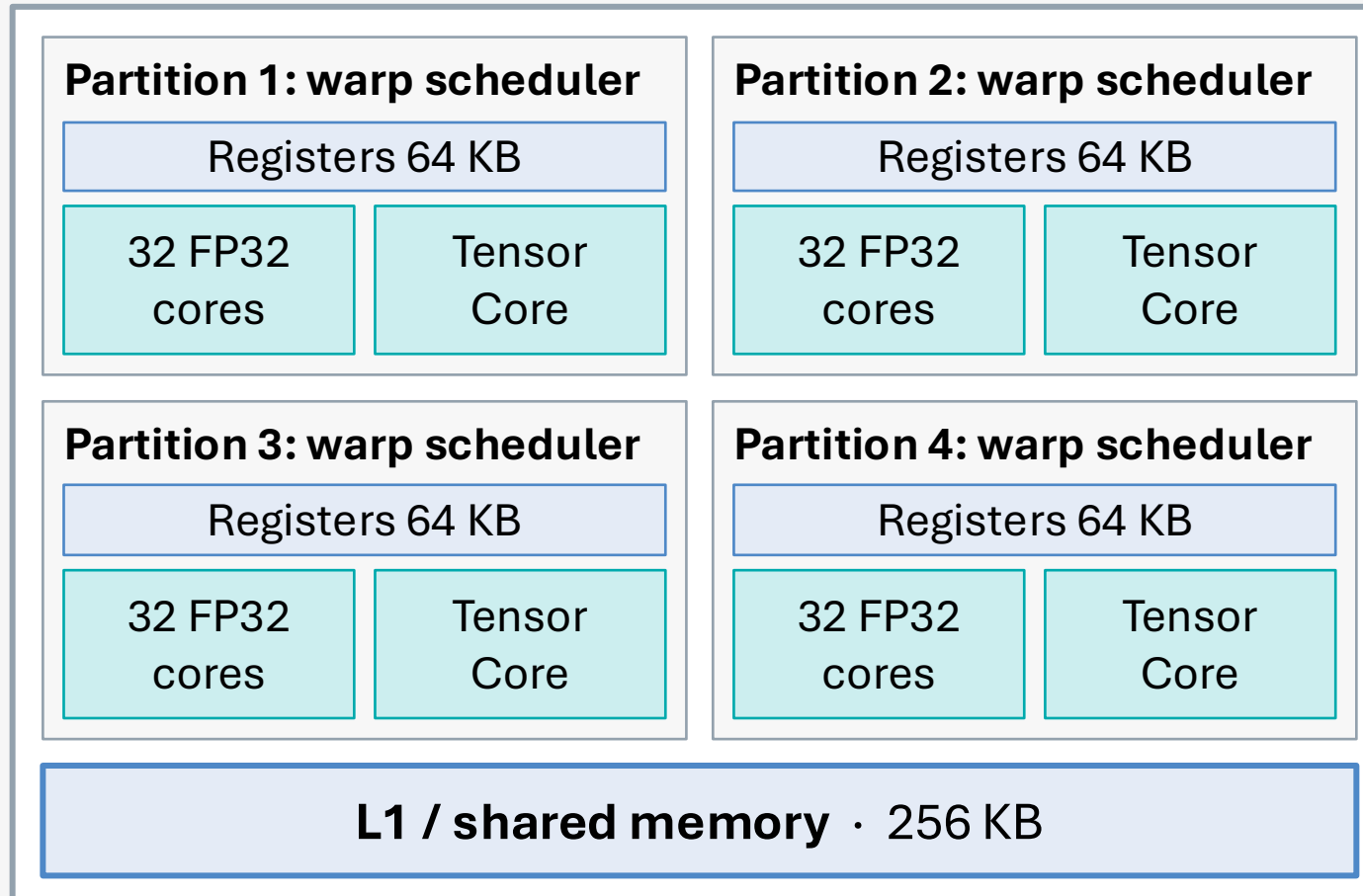
An illustration of the H200

132 SMs that share one 50 MB L2 cache and 141 GB of HBM3e



HBM3e: 6 stacks × 24 GB, 141 GB usable at 4.8 TB/s. **NVLink:** 900 GB/s to other GPUs.

Each SM splits into four partitions that each issue one warp instruction per cycle



Warp = 32 threads

One instruction runs on all 32 at once (SIMT).

Up to 64 warps

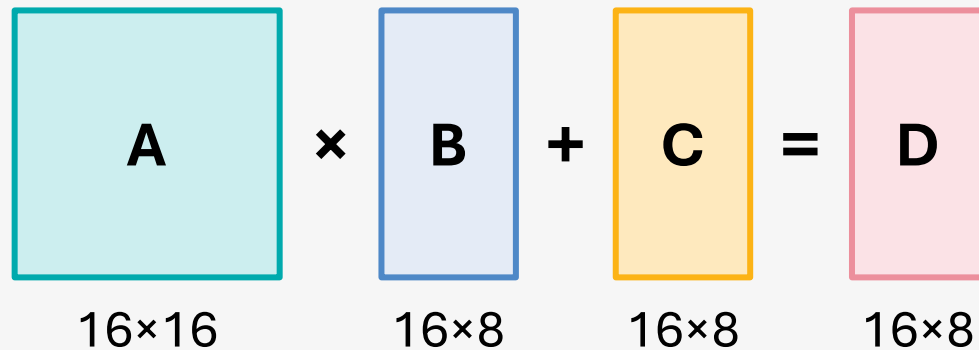
2,048 threads resident; the scheduler picks a ready warp each cycle.

Tensor Cores

Each performs a small matrix multiply-add per instruction.

LLM math is mostly matrix multiplications and Tensor Cores speed it up

One warp instruction: $D = A \times B + C$



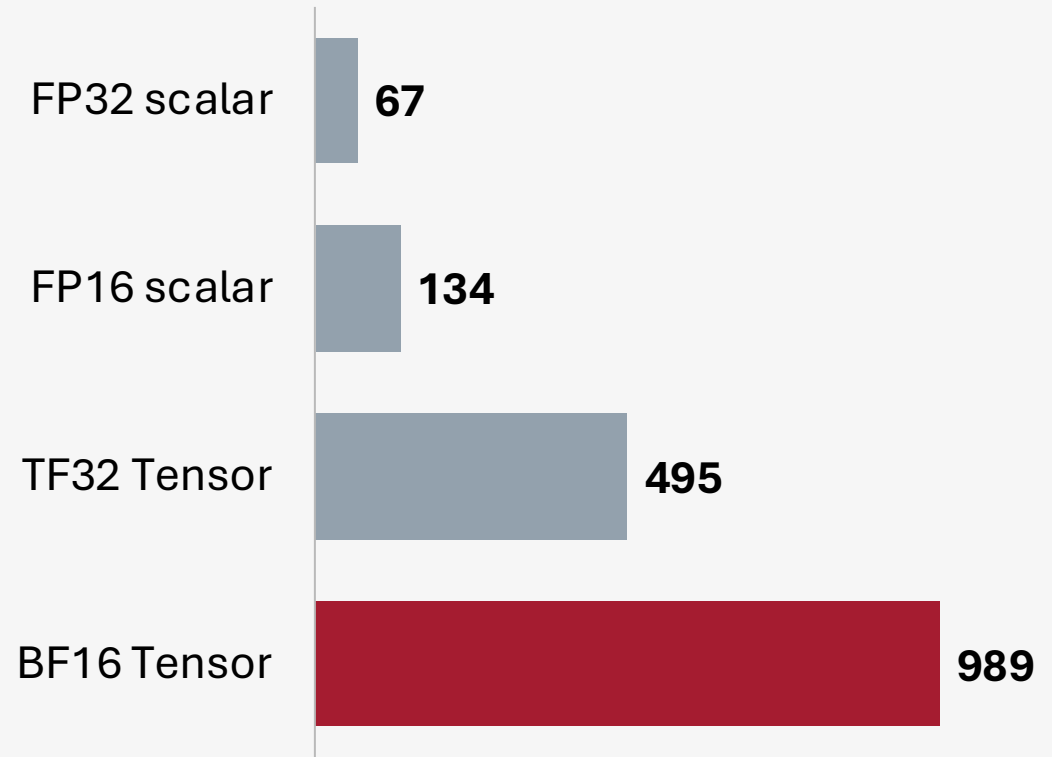
Matmuls are over 99% of LLM FLOPs

Softmax and norms use the FP32 cores.

One mma = 2,048 multiply-adds

64× an FMA's work per instruction.

Peak dense TFLOP/s, H200 SXM



Why Tensor Cores are fast



Fewer instructions per FLOP

One MMA instruction covers many multiply-adds, amortizing decode, scheduling and operand handling.



Specialized arithmetic units

Dedicated multiply-add datapaths process whole matrix tiles in parallel.



Efficient operand reuse

Each value of A feeds several output columns; each value of B feeds several output rows.



Lower-precision arithmetic

FP16, BF16 and FP8 raise throughput and shrink operands, usually with higher-precision accumulation.

The memory hierarchy

Level · H200 capacity

Registers

256 KB per SM

Shared memory / L1

256 KB pool per SM

L2 cache

50 MB

HBM3e

141 GB in 6 stacks

Host memory and peer GPUs

CPU DRAM, other GPUs over NVLink

Latency · Hopper

in the pipeline

read every instruction

≈ 30 cycles
(40 for L1 hit)

≈ 250 cycles

≈ 500 cycles

microseconds

over NVLink or PCIe

Bandwidth

3 operands

per core per clock

128 B/clock per SM

≈ 33 TB/s across 132 SMs

≈ 6 KB/clock

≈ 12 TB/s on Hopper

4.8 TB/s

H200 SXM

450 GB/s each way

NVLink 4; PCIe 5.0 ≈ 64 GB/s

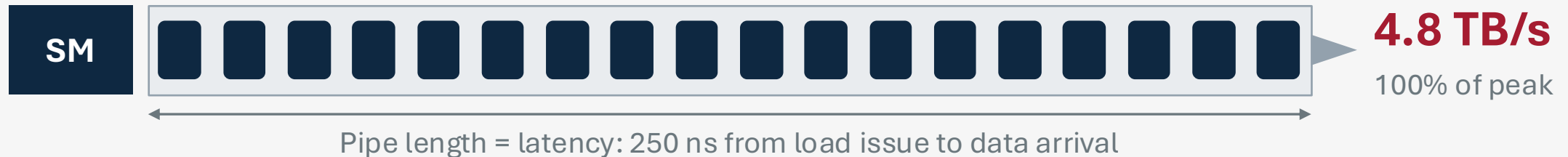
Memory is a 250 ns pipe: bandwidth depends on how many bytes are inside

Little's law: **bandwidth = bytes in flight ÷ latency**

Naive kernel: 4 KB in flight per SM = 32 warps × 32 threads × one 4 B load



Full pipe: 9 KB in flight per SM = 4.8 TB/s × 250 ns ÷ 132 SMs



The hardware fixes the pipe's length,
so the only way to raise bandwidth is to
keep more bytes inside it.

Approaches to increase bytes in flight

- Use 16-byte vector loads
- Issue four independent loads per thread
- Use bulk asynchronous copies (TMA)

Bytes in flight: each generation needs more to reach peak bandwidth

	H200 · Hopper	B200 · Blackwell	Rubin
SMs	132	148	224
HBM	141 GB HBM3e	180 GB HBM3e	288 GB HBM4
HBM bandwidth	4.8 TB/s	8 TB/s	22 TB/s
Bytes in flight, GPU	1.2 MB	2.0 MB	5.5 MB
Bytes in flight, per SM	9 KB	14 KB	25 KB
New data-path feature	TMA, clusters	TMEM, 2-SM MMA	Inline TMA descriptors



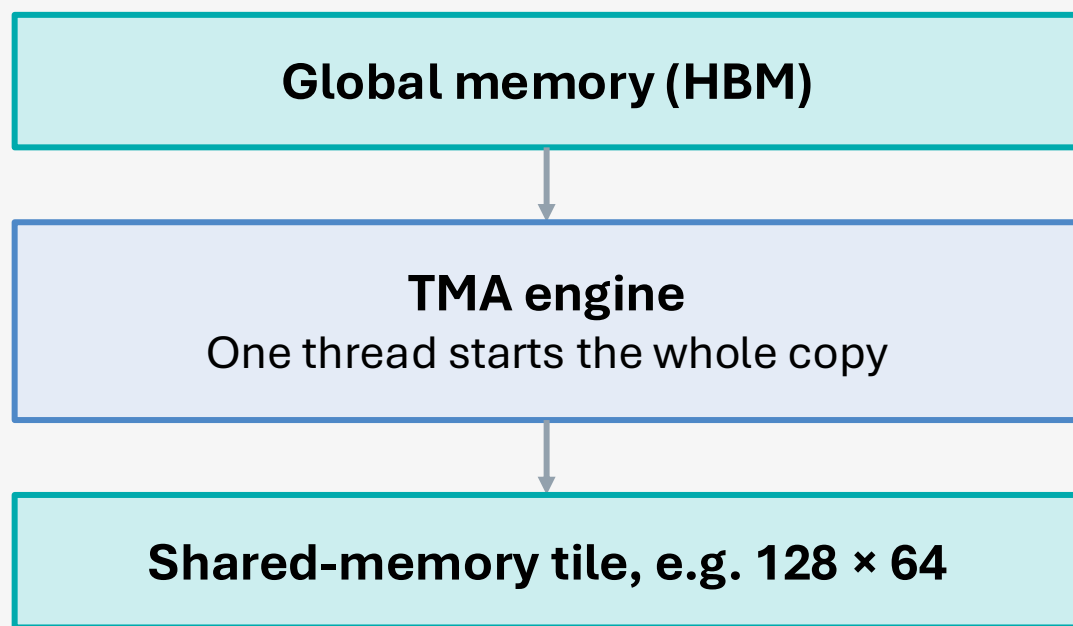
Hopper enables more bytes in flight per instruction

Feature	What it does	Effect
Coalescing	32 threads reading consecutive 4-byte words merge into one 128 B request	Fewer, fuller memory requests
16-byte vector loads	One warp instruction moves 512 B instead of 128 B	4× bytes per instruction
cp.async (Ampere+)	Copies global memory into shared memory without staging in registers	Registers stay free for more loads
TMA (Hopper)	One thread issues a bulk tile copy; hardware generates the addresses	Kilobytes in flight from one instruction
TMA multicast	One L2 read lands in the shared memory of every SM in a cluster	Less L2 traffic per SM



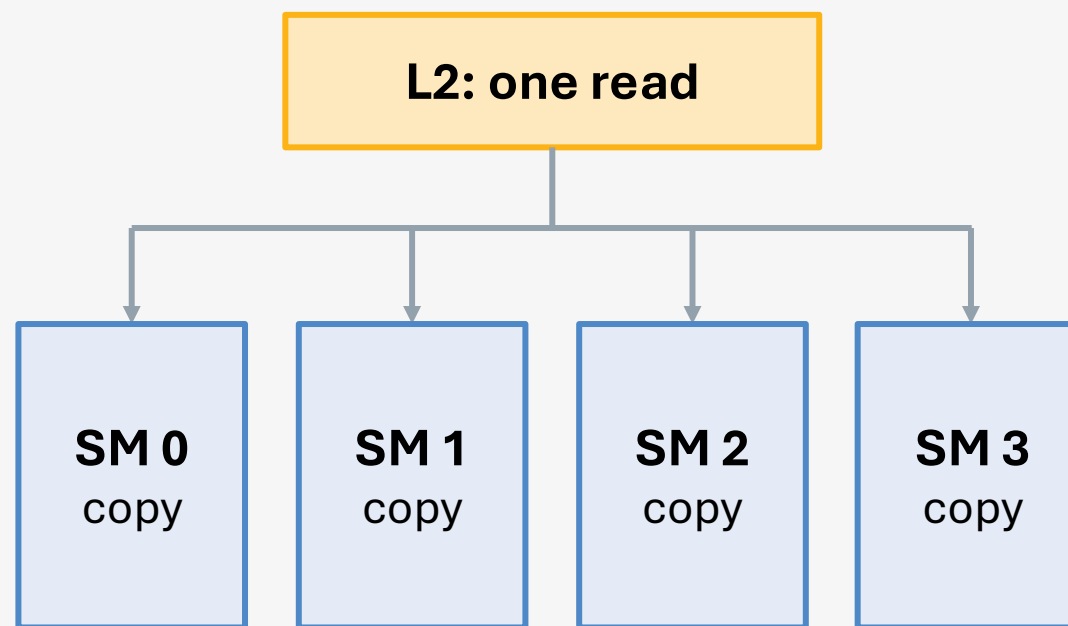
Hopper's TMA copies a whole tile with one instruction

TMA: one thread, one tile



Registers stay free, and one instruction puts **tens of KB** in flight

Cluster multicast: one read, many SMs

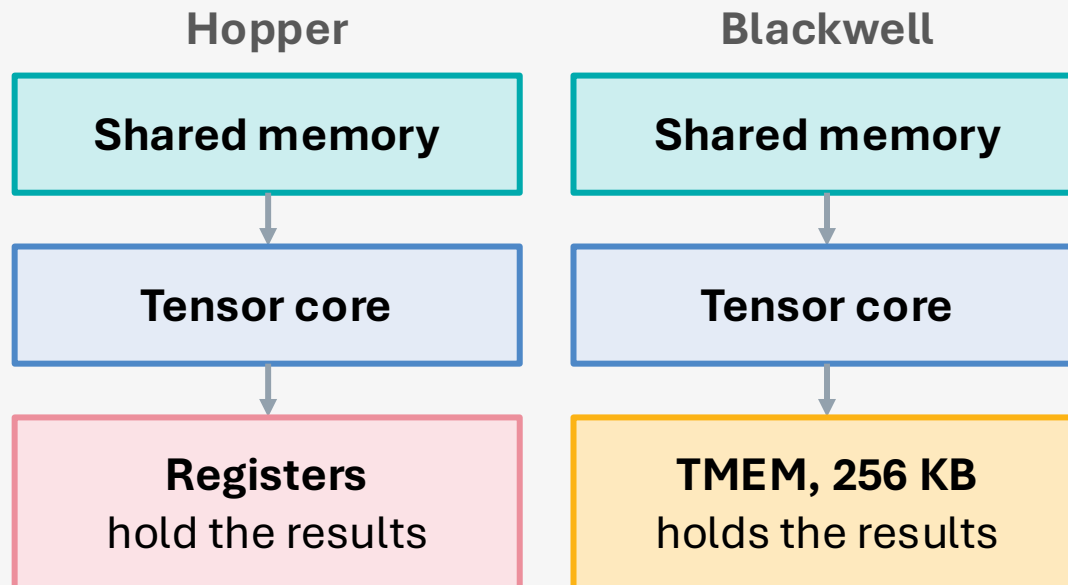


Up to 8 thread blocks in a cluster share one L2 read, so **L2 traffic drops**



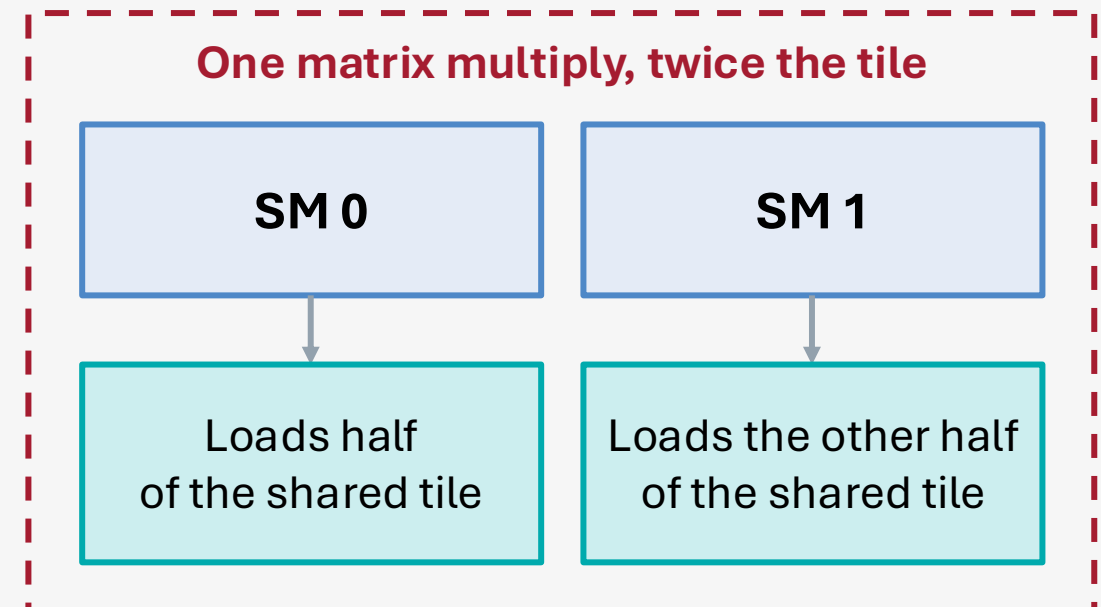
Blackwell moves results into tensor memory and pairs SMs on one multiply

TMEM: results leave the registers



Registers are freed to keep **more loads in flight**

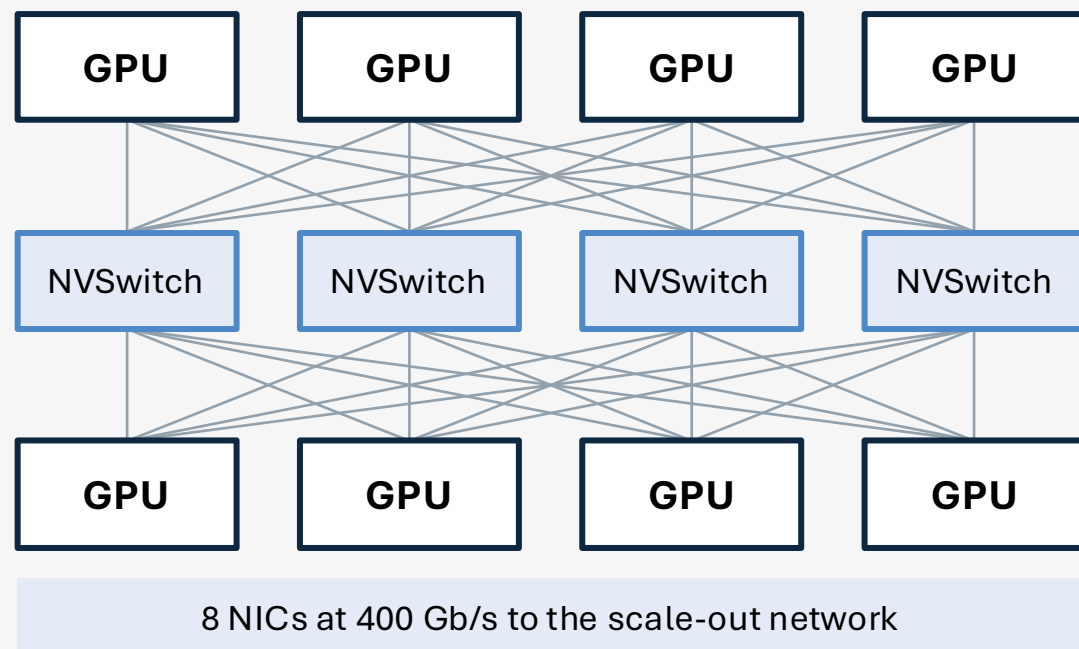
2-SM MMA: two SMs, one multiply



Each SM fetches **half the tile both SMs need**

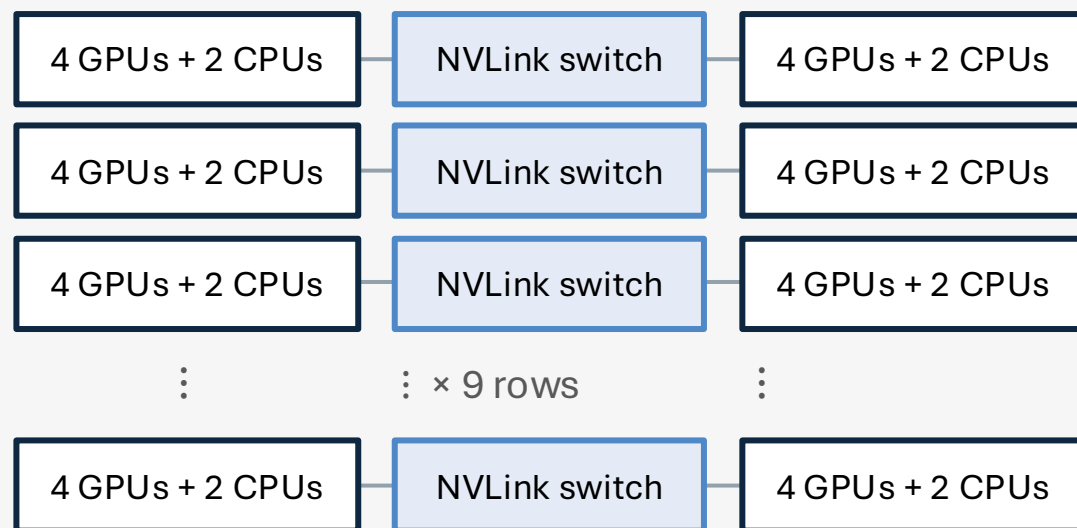
From one GPU to many: scale-up and scale-out

8-GPU node (HGX H200)



HBM 4.8 TB/s
NVLink 900 GB/s
RDMA 50 GB/s

NVL72 rack (GB200)



HBM 8 TB/s
NVLink 1.8 TB/s
RDMA 100 GB/s



Two decades of NVIDIA GPUs

2006–17: graphics → general-purpose compute

Architecture	What changed for programmers
Tesla 2006 · G80	Unified shaders; CUDA, thread blocks, shared memory
Fermi 2010 · C2050	Real L1/L2 caches, ECC, unified address space, fast FP64
Kepler 2012 · K20X	Warp shuffles, dynamic parallelism, Hyper-Q work queues
Maxwell 2014 · M40	SM split into four partitions; big per-watt gains
Pascal 2016 · P100	HBM2, NVLink, page-faulting unified memory, 2× FP16
Volta 2017 · V100	Tensor cores, independent thread scheduling, unified L1/shared

2018–26: compute → low-precision AI engines

Architecture	What changed for programmers
Turing 2018 · T4	INT8 and INT4 tensor cores, ray-tracing cores
Ampere 2020 · A100	TF32, BF16, 2:4 sparsity, cp.async, MIG, 40 MB L2
Hopper 2022 · H100	FP8, TMA, clusters and DSMEM, wgmma, confidential computing
Blackwell 2024 · B200	Two dies, FP6/FP4 and block scaling, TMEM, tcgen05, NVL72
Blackwell Ultra 2025 · B300	288 GB HBM3e, 15 PF dense NVFP4, faster exp, much less FP64
Rubin 2026 · NVL72	HBM4, NVLink 6, larger TMEM and shared memory (preliminary)

Compute outran memory, G80 → B300: peak math ~43,000×, bandwidth ~90×

Next class

- Student sharing
- GPU programming and performance optimization
- LLM serving III: batching, paging, and parallelism

Optional reading

AMD Instinct MI355X: a chiplet GPU with more HBM per package

How it differs from an NVIDIA GPU

- **8 compute dies** on one package, seen by software as one GPU
- **256 MB Infinity Cache** sits in front of HBM
- **64-thread wavefronts** instead of 32-thread warps

How you program it

- **ROCm and HIP:** CUDA-style C++; hipify converts CUDA source
- PyTorch, vLLM and SGLang run on ROCm
- Custom kernels in Triton or Composable Kernel

5.0 PF

Dense FP8

B200: 4.5 PF

288 GB

HBM3E

B200: 180 GB

8 TB/s

HBM bandwidth

B200: 8 TB/s

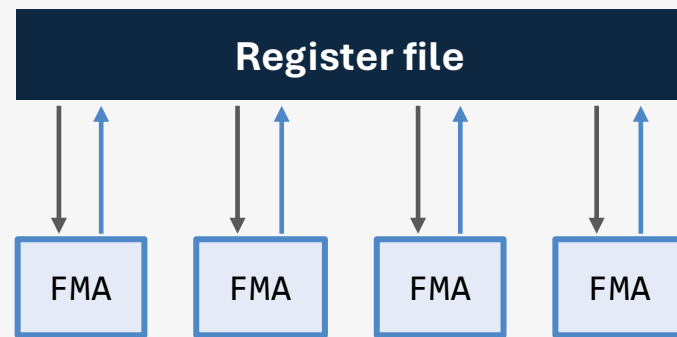
8 GPUs

Scale-up domain

GB200 NVL72: 72

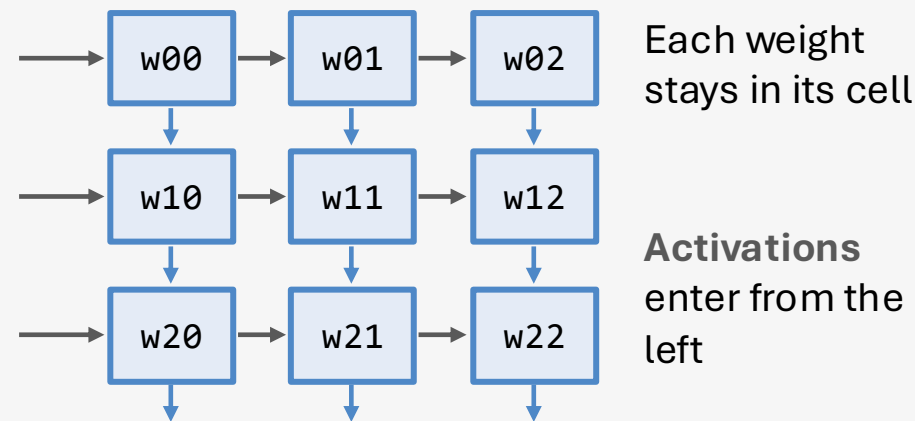
Google TPU Ironwood: systolic arrays linked in a 3D torus

SIMT lanes



Every instruction reads operands (**grey**) from the register file and writes results (**blue**) back

Systolic array (weight-stationary)



How you program it

- **JAX** or PyTorch/XLA
- **XLA** tiles matmuls onto the arrays
- **Pallas** for custom kernels

4.6 PF

Dense FP8 · B200 4.5

192 GB

HBM3E · B200 180

7.4 TB/s

HBM · B200 8

9,216 chips

3D torus · NVL72 72

Amazon Trainium3: systolic engines with software-managed SRAM

How it differs from an NVIDIA GPU

- **8 NeuronCores**, each with separate Tensor (systolic), Vector and Scalar engines
- **32 MB SRAM per core**, filled by DMA engines; no hardware cache
- **Gains come from MXFP8 and MXFP4**; BF16 per core is unchanged from Trainium2

How you program it

- **Neuron SDK** compiles PyTorch and JAX graphs
- **NKI** (Neuron Kernel Interface): Python kernels that move tiles between HBM and SRAM explicitly
- No CUDA; kernels target engines, not threads

2.5 PF

Dense MXFP8

B200: 4.5 PF FP8

144 GB

HBM3E

B200: 180 GB

4.9 TB/s

HBM bandwidth

B200: 8 TB/s

144 chips

Scale-up domain

GB200 NVL72: 72

Other accelerators: on-chip SRAM in place of HBM

Chip	On-chip SRAM	Off-chip memory	Trade-off
Cerebras WSE-3	44 GB	None	Whole wafer is one chip; larger models stream weights in
Groq LPU	230 MB	None	A 70B model at 8 bits needs about 300 chips
Microsoft Maia 200	272 MB	216 GB HBM3e	SRAM holds hot data; available only inside Azure