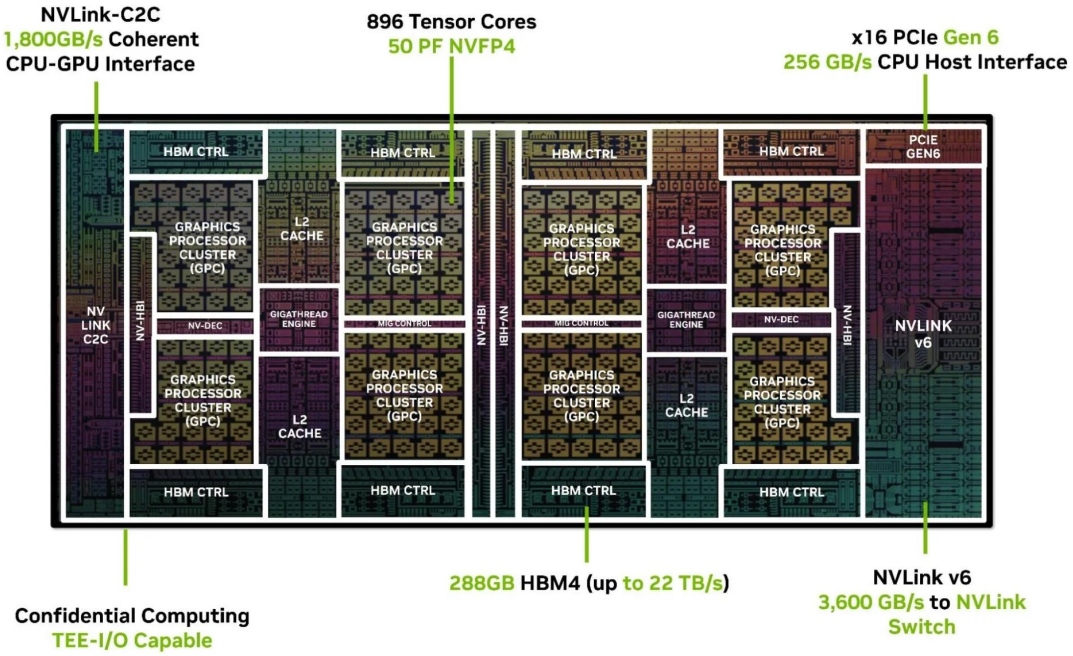
GPU architecture and workload evolution

Workload changes → bottleneck changes → architecture changes

UTILIZING THE ARITHMETIC UNITS

The useful question is whether those units can be fully utilized and which key factors determine their utilization.



Rubin GPUs contain up to 224 SMs and 288GB HBM4 Memory. Available SM count and HBM varies by SKU.

Image: NVIDIA Rubin GPU architecture, July 2026

Six design decisions that determine GPU utilization

Each decision removes a different reason for execution units to wait.

1 Provide enough parallel work

Many SMs process independent thread groups at the same time.

2 Keep warp state on chip

Keep waiting warps in registers so ready warps can run.

3 Keep reused data nearby

Registers, shared memory, and caches reduce repeated HBM accesses.

4 Match hardware to operations

CUDA cores, Tensor Cores, SFUs, and load/store units use different data paths.

5 Overlap movement and execution

Copy future inputs while current arithmetic runs. Wait before using them.

6 Scale beyond one die

Die links and NVLink carry data needed by work on other processors.

Image and SM counts: NVIDIA Blackwell Ultra architecture deep dive, August 2025

WORKLOAD MILESTONE I

Computer Graphics

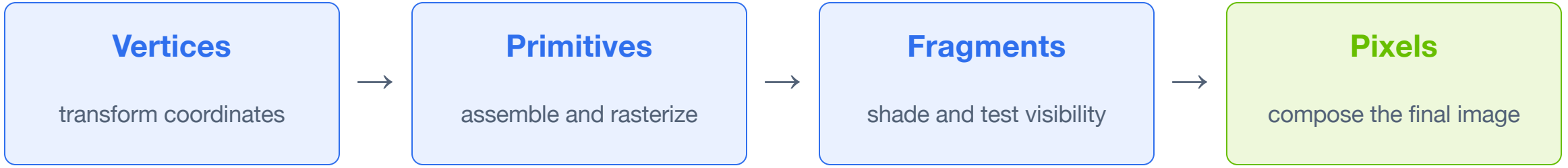
How can we render a complex scene within one frame deadline?

Workload	Render many vertices and fragments with similar operations.
Bottleneck	The machine must finish a large amount of parallel work per second.
Architectural response	Replicate arithmetic units and specialize the rendering stages.

Graphics origins: GeForce 256, 1999

Graphics rewards parallel throughput

Many vertices and fragments need similar calculations within one frame deadline.



Parallel work

Many scene elements can be processed at the same time.

Throughput pressure

At 60 frames/s, the entire frame has about 16.7 ms.

Hardware response

Parallel arithmetic and specialized rendering stages increase total work/s.

Source: NVIDIA CUDA Programming Guide, Introduction. Frame budget is a teaching calculation.

WORKLOAD MILESTONE II

Scientific and General-Purpose Computing

How can the same parallel hardware run numerical programs directly?

Workload	Simulations and numerical methods update large arrays in parallel.
Bottleneck	Graphics interfaces restrict general data access and cooperation.
Architectural response	Expose programmable threads, on-chip storage, and synchronization.

General programming: G80 and CUDA, 2006

Scientific computing needed a general programming model

The arithmetic fit the GPU, but numerical arrays had to pass through graphics interfaces.

EARLY GPGPU

Numerical arrays

for example, temperatures

→

Textures and shaders

encode computation as graphics

→

Rendered outputs

recover the numerical results

CUDA

Arrays → parallel kernel threads → numerical results

Workload

Simulations and numerical methods expose parallel updates and reusable data.

Programming bottleneck

Graphics APIs make general data access and cooperation awkward.

Hardware and software

CUDA exposes threads, local storage, and synchronization directly.

Sources: NVIDIA CUDA Programming Guide; CUDA Refresher: Reviewing the Origins of GPU Computing.

Why CUDA needed registers and shared memory

General programs need persistent values, flexible memory access, and synchronization.

1999

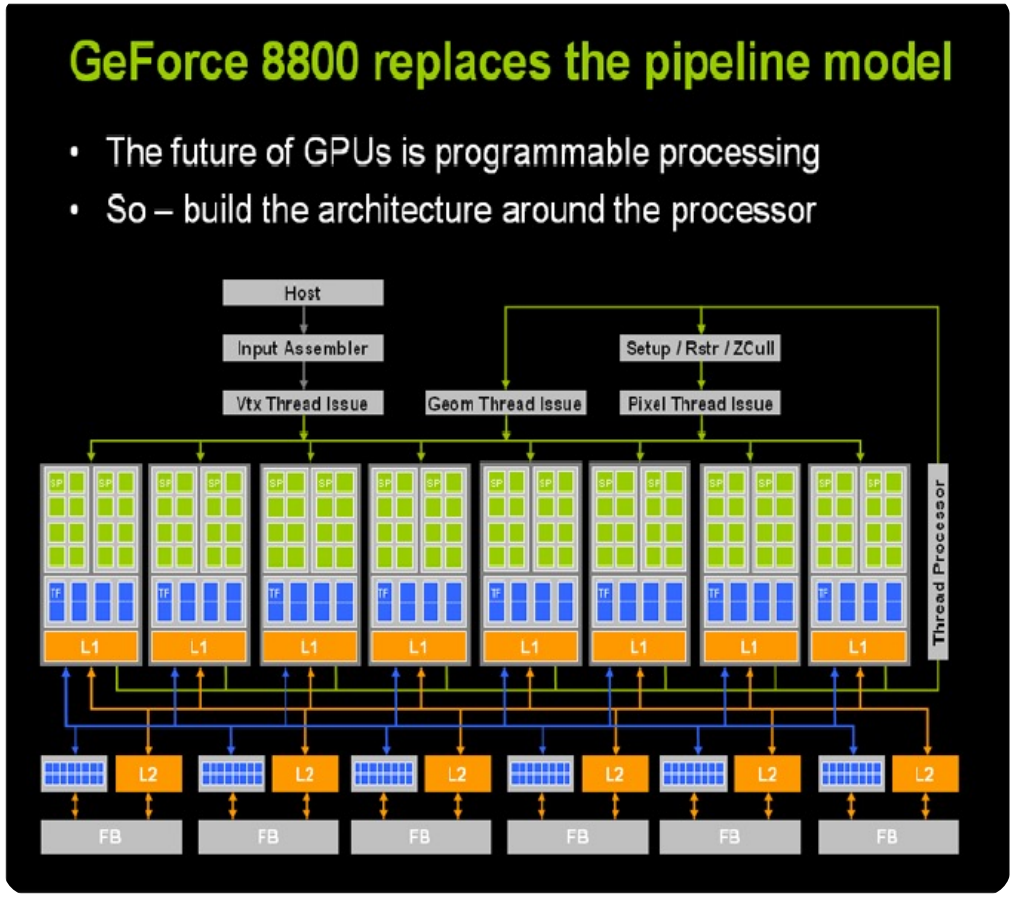
GeForce 256

Specialized graphics stages processed vertices and pixels. General algorithms had to work within graphics memory and execution interfaces.

2006

GeForce 8800 / G80

CUDA mapped general programs onto unified processors. G80 combined 128 stream processors with on-chip registers, 16 KB of shared memory per SM, and warp scheduling to reuse data and tolerate off-chip latency.



GeForce 8800 GTX anchor: 128 stream processors, about 0.52 TFLOPS, 768 MB GDDR3, and 86.4 GB/s memory bandwidth

Image and facts: NVIDIA GeForce 8800 Architecture Technical Brief; NVIDIA Fermi Architecture Whitepaper

A CUDA thread: private state, shared execution

A thread runs one instance of a GPU function (kernel). A warp groups 32 threads for instruction issue.

What belongs to one thread?

Identity — threadIdx identifies its position within a block.

Control state — Program counter: next instruction. Call stack: function-call context. Per-thread control state since Volta.

Private values — Registers hold operands and results. Local memory holds other private data and can hold spills.

One instruction, separate register values

```
Illustrative instruction: R2 = R0 + R1
```

Thread	R0	R1	R2 after
0	2	3	5
1	7	4	11

R0 in thread 0 and R0 in thread 1 name different values. Physical register capacity is shared, but each thread has its own allocation.

How resident warps make progress

Stalled
Next instruction must wait.

Eligible
Ready, awaiting selection.

Selected
Issues an instruction, then continues.

Issue is not completion. Registers stay allocated while waiting. Sources: CUDA Programming Guide and Nsight Compute.

Threads and warps

A warp groups 32 threads for instruction scheduling.

PROGRAMMING VIEW

THREAD

One execution of kernel code

A thread has its own ID, registers, and control state.

BLOCK

A cooperating thread group

A block runs on one SM. Its threads can share memory and synchronize.

WARP

The hardware scheduling group

The SM partitions consecutive block threads into groups of 32.

CONCRETE EXAMPLE

One block contains 64 threads

Hardware creates two warps from consecutive thread IDs.

WARP 0	0-7	8-15	16-23	24-31
WARP 1	32-39	40-47	48-55	56-63

INSTRUCTION ISSUE

The scheduler selects a ready warp. One instruction applies to the warp's active threads.

BRANCH DIVERGENCE

When threads choose different paths, the warp executes each path with a different active mask. Some lanes remain idle.

Source: NVIDIA CUDA Programming Guide 13.2, SIMT Execution Model and Hardware Multithreading

Following an instruction through the SM

Fetch and issue select the operation, then registers supply the data.

CONTROL PATH

L0 instruction cache
fetch next instruction

→

Decode + dispatch
`FFMA R8,R12,R16,R20`

→

Scoreboard
are R12/R16/R20 ready?

→

Issue
reserve target pipe

OPERAND AND RESULT PATH

Register-file slice

`R12: a lane vector`
`R16: b lane vector`
`R20: c lane vector`

→

Operand collection

Read the source lane vectors and route them to the reserved execution pipeline.

→

FP32 pipeline

`active lanes perform a × b + c`

→

Writeback + wakeup

Write each thread's R8. Its dependent instruction can then become ready.

LATENCY AND THROUGHPUT

One instruction takes time to complete. Independent instructions can overlap in the pipeline. This is a conceptual data path, not a physical stage or register-port specification.

Sources: NVIDIA Blackwell Ultra SM diagram; CUDA Programming Guide; Volta microbenchmarking (measured, generation-specific)

CUDA cores perform one operation per active thread

The warp shares an instruction while each thread supplies its own operands.

ISSUED WARP INSTRUCTION

```
FFMA R8, R12, R16, R20
R8 = R12 × R16 + R20
```

REGISTER OPERANDS

```
lane 0: a0, b0, c0
lane 1: a1, b1, c1
lane 2: a2, b2, c2
...
lane 31: a31, b31, c31
```

ONE BLACKWELL ULTRA SM PARTITION

32 warp lanes

All active threads receive the operation. A false per-thread condition prevents that thread from writing its result.

lane 0	lane 1	lane 2	lane 3
lane 28	lane 29	lane 30	lane 31

Green lanes write results. Gray lanes keep their previous values (lanes 3 and 30 shown).

PIPELINE RESULT

Execute and write back

Each active lane computes its own result and writes it to that thread's destination register.

WHAT LIMITS THROUGHPUT

Dependent instructions wait for results. A saturated arithmetic pipe also delays issue. The scheduler hides these delays by selecting another eligible warp.

Blackwell Ultra: four SM partitions × 32 CUDA cores = 128 CUDA cores per SM. Counts and supported instruction rates vary by architecture.

Sources: NVIDIA Blackwell Ultra architecture; CUDA Binary Utilities instruction reference

How a warp becomes ready to issue

Resident warps retain their state while dependencies delay their next instruction.

RESIDENT WARP POOL

Warp A — ELIGIBLE — FFMA operands ready

Warp B — WAITING — global load pending

Warp C — WAITING — thread-block barrier

Warp D — ELIGIBLE — MUFU operands ready

→

WARP SCHEDULER

A warp can issue when:

1. its next instruction is available
2. the required operands are ready
3. the target pipeline can accept work

Selected this cycle: Warp A

→

DISPATCH TARGET

CUDA-core pipe	FFMA, integer, compare
Tensor Core pipe	matrix multiply-accumulate
SFU pipe	MUFU special functions
Load/store path	memory requests

State stays on chip

Selecting a resident warp requires no operating-system register save or restore.

Occupancy measures residency

Active resident warps / maximum resident warps. More warps help only if work is ready.

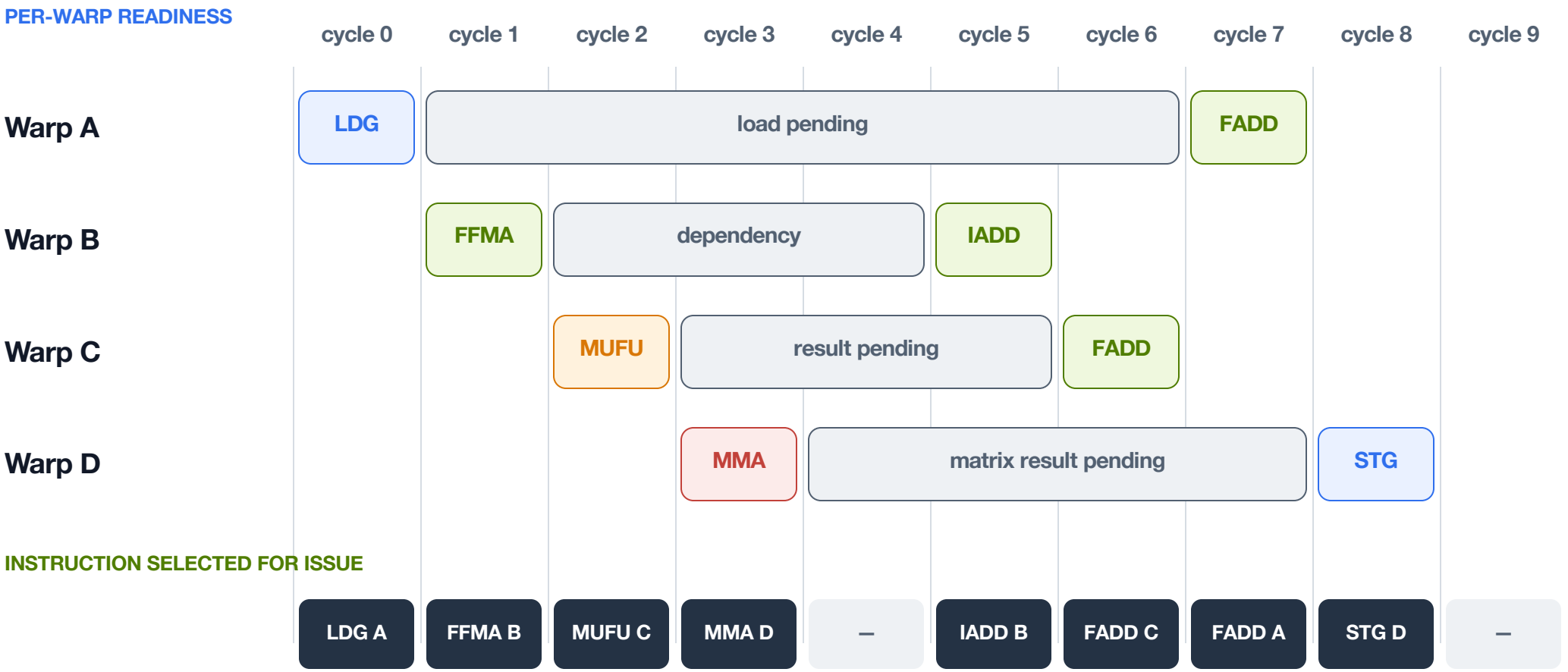
The scoreboard tracks dependencies

Hardware records unfinished results. A dependency, barrier, or unavailable pipe keeps a warp from issuing.

Sources: CUDA Programming Guide; NVIDIA Nsight Compute Profiling Guide

Instruction issue and completion happen at different times

Other warps can issue while an earlier instruction is still in flight.



Latency hiding fills issue slots with independent work while earlier instructions wait.

Illustrative one-issue-per-cycle schedule. Waiting intervals are not measured instruction latencies.

Inside a Blackwell Ultra streaming multiprocessor

Modern example: Blackwell Ultra shows where threads and warps execute.

An SM is the complete processing block

A CUDA core is one kind of execution resource inside the SM.

1 Warp schedulers choose ready work

Four schedulers manage assigned warps. Issuing starts work, which can finish later.

2 Registers supply each thread's operands

Threads share physical capacity, but each thread retains its own register values.

3 Specialized units execute different instructions

CUDA cores handle general arithmetic. Tensor Cores handle matrix operations. SFUs handle functions such as exponentials.

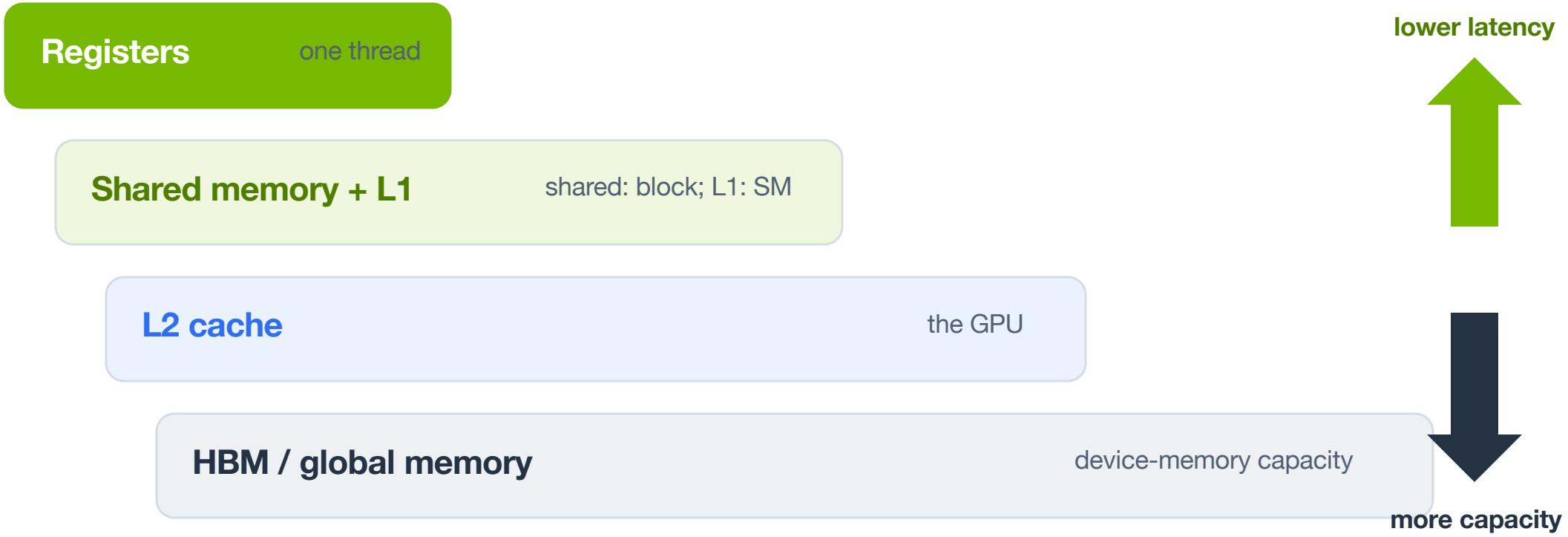
4 On-chip storage feeds the units

Shared memory, caches, registers, and Tensor Memory reduce repeated traffic to HBM.

Image and specifications: NVIDIA Blackwell Ultra architecture deep dive, August 2025

Where GPU data live

Reusing nearby data reduces traffic to larger, more distant storage.



Reuse: load a tile from HBM, retain it on chip, and use it many times.

Source: NVIDIA CUDA Programming Guide, GPU Memory and GPU Device Memory Spaces

Fermi added a general cache hierarchy

Historical response: Fermi added general caching in 2010.

2010 · FERMI

General-purpose caching

Each SM gained an L1 cache. A unified L2 cache served load, store, and texture requests across the GPU.

CONFIGURABLE ON-CHIP STORAGE

64 KB per SM

```
48 KB shared + 16 KB L1
or
16 KB shared + 48 KB L1
```

Unified L2: 768 KB

Tesla C2050: 14 SMs, 448 cores, 1.03 TFLOPS, 3 GB GDDR5, 144 GB/s

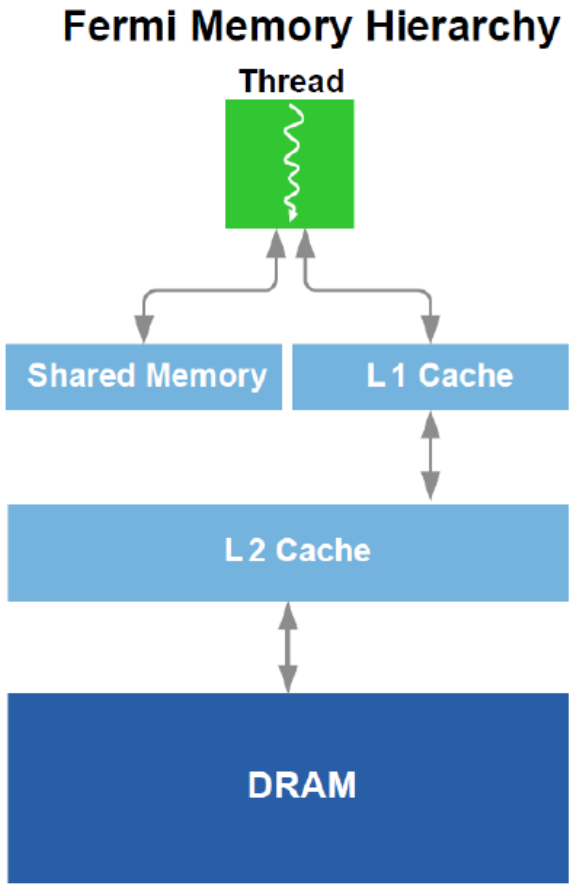


Image and specifications: NVIDIA Fermi Compute Architecture Whitepaper, pp. 15–16

Coalescing groups a warp's memory requests

Useful bytes and requested cache sectors can differ by a large factor.

WARP ADDRESS VECTOR

TRANSACTION COUNT FOR 32 THREADS × ONE 4-BYTE WORD

Contiguous

`addresses 0, 4, 8, …, 124`

`0–31 B` `32–63 B` `64–95 B` `96–127 B`

4 sectors = 128 B requested / 128 B useful

Stride ≥ 32 bytes

each lane lands in a different sector

... 32 sectors

32 sectors = 1024 B requested / 128 B useful

These counts describe requested cache sectors. Cache hits and reuse determine how much traffic reaches HBM.

Source: CUDA Programming Guide, device-memory access and coalescing rules

WORKLOAD MILESTONE III

Deep Learning Training

How can we train large models with dense matrix work across GPUs?

Workload	Repeat forward, backward, and weight-update computations.
Bottleneck	Large matrix operations, training state, and GPU exchanges limit scale.
Architectural response	Use Tensor Cores, high-bandwidth memory, and accelerator links.

Matrix specialization: Volta through Hopper

Training rewards matrix throughput and cooperation

Large batches reuse weights; training state and cross-GPU exchanges add memory and link pressure.

Forward

compute predictions

→

Backward

compute gradients

→

Update

adjust model parameters

Updated weights feed the next training iteration.

Matrix arithmetic

Large matrix operations motivate Tensor Cores and suitable low precision.

Resident state

Weights, activations, gradients, and optimizer state need HBM.

Distributed work

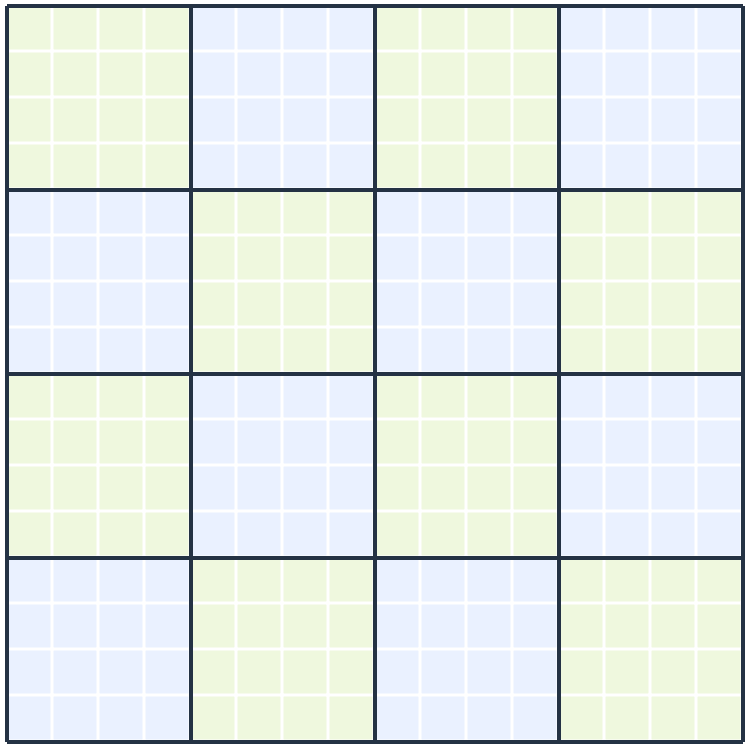
Gradient and partial-result exchanges motivate NVLink and NVSwitch.

Sources: NVIDIA Tensor Core programming guide; NVIDIA Matrix Multiplication Background User's Guide.

Matrix tile size and Tensor Core work

Count the arithmetic separately from the hardware cycles that execute it.

OUTPUT TILE D: 16 × 16, WITH K = 16



256 outputs × 16 terms = 4,096 FMAs

→

WARP INSTRUCTION

`mma / wmma`

32 threads collectively supply fragments of A and B and collectively receive fragments of C/D.

WMMA hides fragment layout. PTX documents thread mappings for specific MMA forms.

One tile contains many multiply-adds

→

CONCRETE HISTORICAL ANCHOR

Volta Tensor Core

`4 × 4 × 4 MMA`

A 4×4 tile × a 4×4 tile, accumulated into a 4×4 result.

64 FMA operations / clock

FP16 products, FP32 accumulation

16 output regions × 4 updates across K = 64 small MMA operations. This is a work count, not a cycle count.

Sources: NVIDIA Volta Architecture whitepaper; CUDA WMMA programming model; PTX ISA

Copying data before computation

A consumer must wait for its input, but independent work can continue.

Conventional staging Global load writes a register; a shared-memory store reads that value.

Direct copy Dedicated hardware moves the payload from global to shared memory.

Asynchronous execution Launch the copy, do independent work, then wait before reading the destination.

Two different properties Direct describes the data path. Asynchronous describes completion timing.

The register saving comes from the direct path, not from asynchrony alone.

Higher bandwidth and direct copies feed matrix work

HBM supplies more bytes while asynchronous copies reduce staging overhead.

2016 · PASCAL P100

HBM2 moved memory onto the package

16 GB HBM2 at 732 GB/s increased usable bandwidth. NVLink accelerated GPU-to-GPU and GPU-to-CPU transfers. Hardware page faults enabled demand-paged Unified Memory.

2020 · AMPERE A100

Software could stage data without register detours

A100 80 GB SXM: 2.039 TB/s HBM2e and 40 MB L2. Direct global-to-shared copies bypass payload registers. Threads can continue, but must wait before using the copied data.

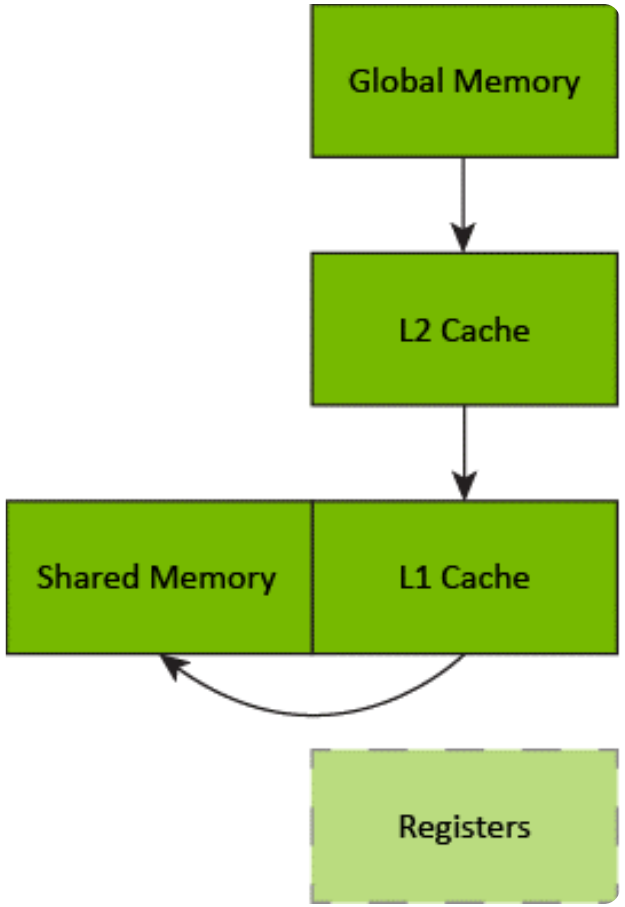
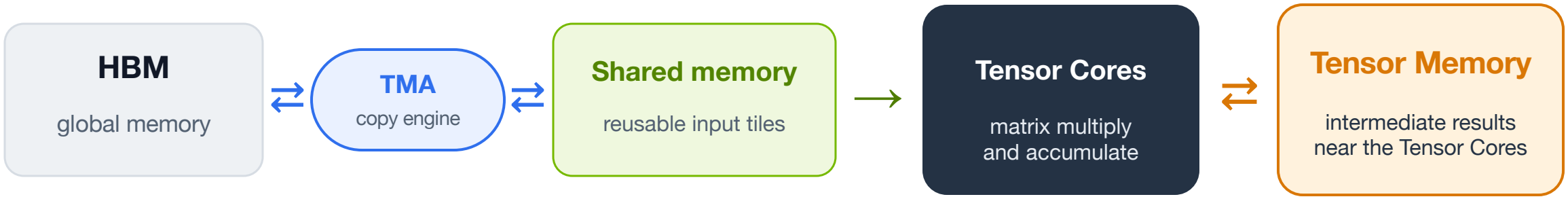


Image: NVIDIA, “A better journey through the memory hierarchy,” 2020

TMA moves tiles and Tensor Memory stores results

A transfer engine and a storage resource solve different problems.

INPUT PATH



2022 · HOPPER

TMA first appears

One thread can launch an asynchronous 1D–5D transfer. Hardware handles addresses, strides, and bounds while other threads compute.

2024–2025 · BLACKWELL

Tensor Memory joins the SM

Blackwell Ultra provides 256 KB of TMEM per SM for matrix intermediates. Specialized instructions manage and access it.

2026 · RUBIN

TMA descriptors become easier to reuse

A kernel can override pointer or stride fields, reusing a descriptor for expert tensors with a common layout.

Sources: NVIDIA Hopper and Blackwell Ultra architecture deep dives; NVIDIA Rubin GPU architecture

Tensor Core instruction families and operand storage

The matrix operation stays the same, while participation and operand locations change.

$$D = A \times B + C$$

A and B are input tiles. C is the old accumulator. D is the updated accumulator.

SOURCES VARY BY INSTRUCTION

Operand A

registers, shared memory, or TMEM

Operand B

registers or shared memory

→

Tensor Core operation

The collective instruction describes matrix tile shapes, data types, layouts, and accumulation behavior.

many multiply-adds per instruction

→

Accumulator

Registers on earlier instruction families

Tensor Memory (TMEM)

INSTRUCTION FAMILIES

Volta–Ampere `mma`

A warp cooperates. Operands and accumulators use registers.

Hopper `wgmma.mma_async`

A 128-thread warpgroup launches asynchronous matrix work. Shared memory can supply operands.

Blackwell `tcgen05.mma`

An elected thread launches matrix work. TMEM holds the accumulator. A may use TMEM, B uses shared memory.

A tile operation coordinates many multiply-adds. One issuing thread does not perform all the arithmetic.

Sources: NVIDIA PTX ISA; NVIDIA CUTLASS tcgen05 programming guide; Hopper architecture deep dive

Overlapping tensor copies and matrix computation

Two buffers let the next input tile arrive while the current tile is in use.

STEADY-STATE PIPELINE

Stage N + 1: move

TMA

HBM → shared buffer 1

Stage N: compute

WGMMA / tcgen05

shared or TMEM operands → matrix pipe

Stage N – 1: consume

epilogue + store

accumulator → registers / memory

BUFFER AND SYNCHRONIZATION STATE

Shared buffer 0 — consumed by N

Shared buffer 1 — filled for N+1

barrier / transaction count declares when the producer has finished

HOPPER

`wgmma.mma_async`

Four warps form a 128-thread warpgroup. commit_group and wait_group delimit asynchronous MMA batches.

BLACKWELL

`tcgen05.mma`

One thread launches the collective operation. The accumulator resides in TMEM; software allocates, uses, and deallocates TMEM explicitly.

Wait for input arrival before reading. Wait for matrix completion before reusing buffers or consuming results.

Sources: NVIDIA Hopper architecture; PTX ISA; CUTLASS tcgen05 programming guide

Cooperation across SMs and across GPU dies

Hopper adds shared-memory cooperation while Blackwell connects two compute dies.

2022 · HOPPER H100 SXM

A new level between shared memory and L2

Thread-block clusters can access distributed shared memory across cooperating SMs. The Tensor Memory Accelerator moves multidimensional tiles while threads compute.

80 GB HBM3

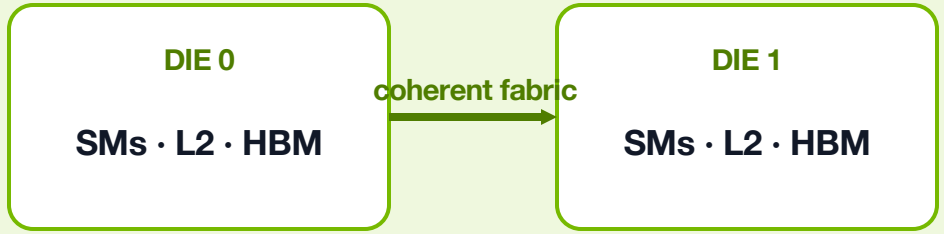
3.35 TB/s HBM bandwidth

50 MB L2 · up to 228 KB shared memory per SM

2025 · BLACKWELL ULTRA B300

Two dies act as one CUDA GPU

Each locality domain has nearby L2 slices and HBM controllers. Coherent L2 caching preserves one address space, but remote traffic crosses the die-to-die fabric.



B300: 288 GB HBM3e and 8 TB/s

160 SMs, 640 Tensor Cores, 256 KB Tensor Memory per SM

Sources: NVIDIA Hopper Architecture In-Depth; NVIDIA Blackwell Ultra architecture deep dive

WORKLOAD MILESTONE IV

Large Language Model Inference

What limits response time once matrix multiplication becomes fast?

Workload	Process a prompt, then generate dependent tokens one step at a time.
Bottleneck	Small-batch decode can spend more time moving data than multiplying.
Architectural response	Reduce traffic and improve memory, attention, and communication.

Serving models: Blackwell Ultra and Rubin

Prefill and decode

Inference uses learned weights without updating them.

Prefill	Process the known prompt positions together. Reuse weights across positions.
Decode	Generate the next token, then use it to begin the following step.
Retained state	Keep earlier attention keys and values in the KV cache.
Changing balance	Small-batch decode often offers less weight reuse than prefill.

Batch size and context length change the balance; these are common tendencies.

The attention computation

Compare a query with keys, turn scores into weights, then combine values.

1 Compare

Q contains queries. K contains keys. Row dot products form scores $QK^T / \sqrt{d}$.

2 Normalize

Softmax turns each row of scores into nonnegative weights that sum to one.

3 Combine

Multiply the weights by V, the values, to form the attention output.

d = number of components in each key. Softmax includes exponentials and reductions.

What can delay the next token

Matrix work, other arithmetic, memory access, and communication compete for time.

Which resource or dependency sets the completion time?

1

Matrix computation

Matrix multiplications

2

Memory movement

Weights and KV cache

3

Other arithmetic

Exponentials, reductions, and scaling

4

Communication

Collectives across accelerators

Required bytes ≤ available bytes. Ideal time ≥ max(work / compute rate, bytes / bandwidth).

What moves? How much? How often? Through what path? Who waits?

Why small-batch decode can be memory limited

Compare the ideal arithmetic time with the time needed to read the operands.

Prefill: known prompt

token 1 token 2 token 3 token 4

Process many positions together.
Often compute-intensive when reuse is high.

Decode: dependent new tokens

new token → next token → next token

Read weights and stored attention keys/values.
Small batches often face memory limits.

Illustrative single-sequence decode step

8B matrix weights × 2 bytes = 16 GB; add 1 GB of old KV reads. All state fits in device memory.

Ideal matrix time

16 GFLOPs / 500 TFLOP/s = 0.032 ms

Ideal read time

17 GB / 2 TB/s = 8.5 ms

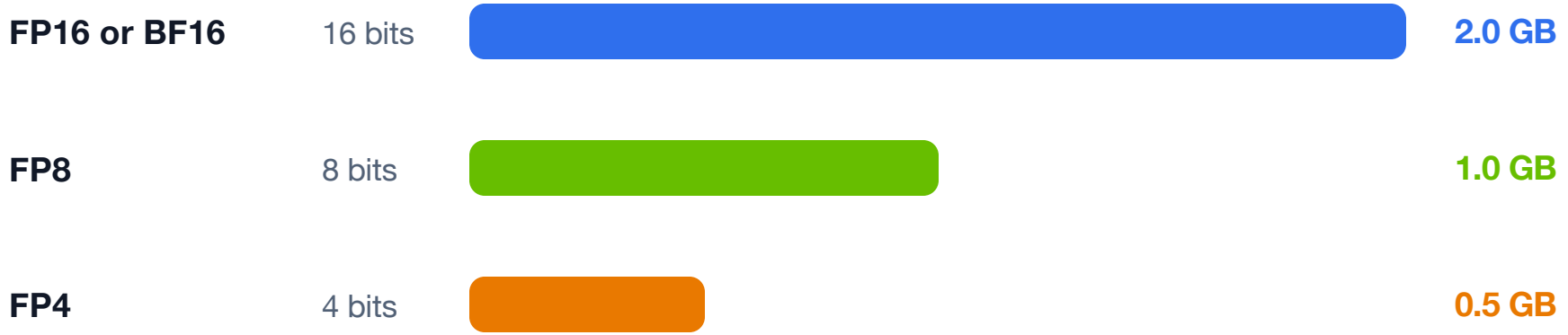
About 266× longer to read. More matrix units alone cannot remove this limit.

Source: NVIDIA, Mastering LLM Techniques: Inference Optimization. Toy assumptions, not a benchmark; other costs excluded.

Low precision reduces bytes and increases matrix throughput

The same element count can use fewer bytes without reducing the arithmetic count.

STORAGE FOR ONE BILLION VALUES



Compute effect

The same multiply-add count can execute at a higher supported rate.

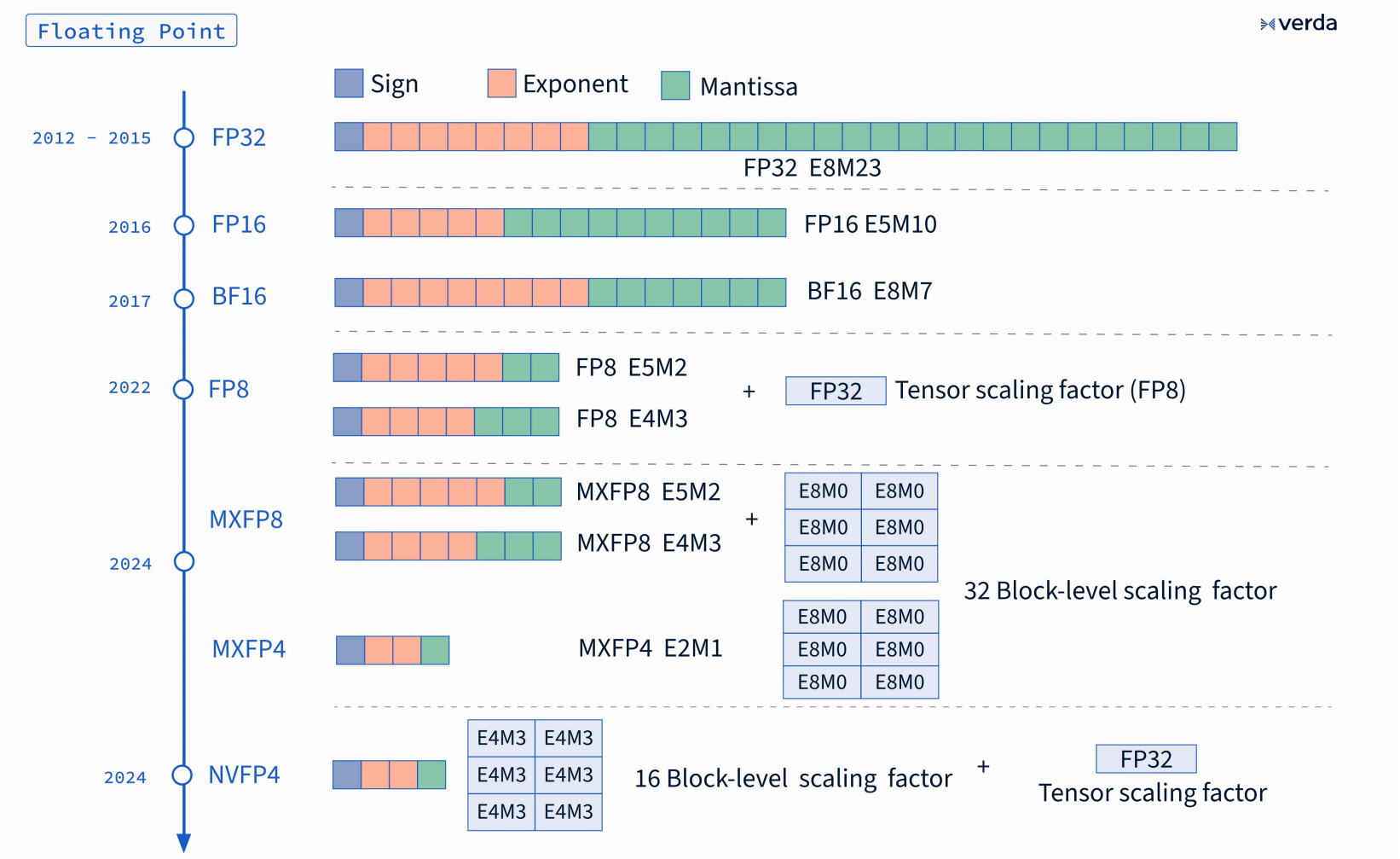
Memory effect

Value-only bytes = element count × bits per value / 8. Scales and metadata add space.

Value bits only. Decimal GB. Encoding scales, metadata, and padding are additional.

Low precision reduces bytes and increases matrix throughput

The same element count can use fewer bytes without reducing the arithmetic count.



Why softmax matters more after faster matrix multiplication

A tenfold improvement in one phase can produce a much smaller total speedup.

Total speedup = 100 / (9 + 10) = 5.3×

Softmax grows from 10% to 53% of the total. Faster exponentials can reduce part of that cost, while reductions and data movement still matter.

Special function units and exponential throughput

Exponentials use a specialized path that can limit softmax performance.

INSTRUCTION PATH

CUDA expression

```
y = exp2(x)
```

→

PTX virtual instruction

```
ex2.approx.f32
```

→

SASS machine instruction

```
MUFU.EX2
```

→

SFU pipeline

evaluate active lanes and write results to registers

COMMON SFU MACHINE OPERATIONS

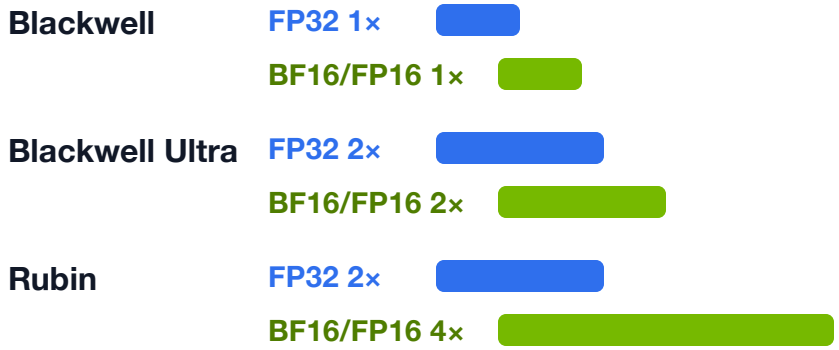
`EX2`	base-2 exponential
`LG2`	base-2 logarithm
`RCP`	reciprocal
`RSQ`	reciprocal square root
`SIN / COS`	trigonometric approximation

ACCURACY AND SCHEDULING

The hardware instruction supplies an approximate primitive. A compiler or math library may add range reduction and refinement when the requested function needs more accuracy.

EXPONENTIAL THROUGHPUT PER SM

per clock per SM, relative to Blackwell

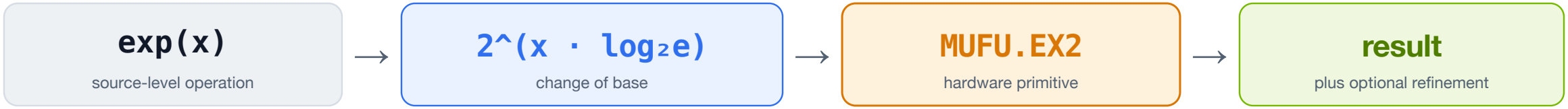


Sources: CUDA Binary Utilities; NVIDIA Blackwell Ultra softmax deep dive; NVIDIA Rubin architecture

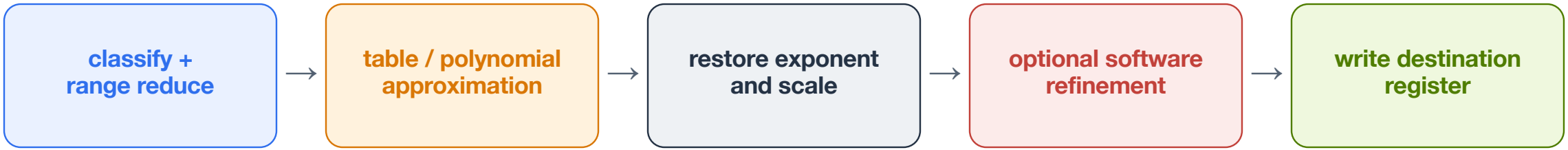
How a fast exponential uses the SFU

A change of base connects exp(x) to the base-two exponential primitive.

NATURAL EXPONENTIAL EXAMPLE



CONCEPTUAL MATH-FUNCTION PIPELINE



WORKED EXAMPLE

x = 1
t = 1 × 1.4427
2^t ≈ 2.7183 ≈ exp(1)

WHAT THE DIAGRAM MEANS

Reduce the input range, approximate, then restore scale.
The boxes explain the mathematics.
The internal NVIDIA circuit is not public.

Sources: CUDA PTX ISA and Binary Utilities; central SFU pipeline is explicitly conceptual

Matching instructions to execution units

A kernel needs the right mix of arithmetic, matrix, special-function, and memory capacity.

Resource	Work it performs	Typical instructions or operations	Common uses
CUDA cores	General arithmetic across active warp lanes	Floating-point fused multiply-add, integer arithmetic, compare	Elementwise math, indexing, control calculations
Tensor Cores	Matrix multiply-accumulate on small tiles	MMA, WGMMA, and newer Tensor Core instructions	Matrix multiplication, convolution, attention projections
SFUs	Transcendental and special math	Exponential, sine, cosine, reciprocal, reciprocal square root	Softmax exponentials, activations, normalization
Load/store units	Address generation and memory requests	Loads, stores, and atomic operations	Feed registers and shared memory, then write results
Texture units	Cached sampling and filtering	Texture fetch and interpolation	Graphics textures and read-only spatial lookup

NVIDIA uses SFU for Special Function Unit. Warp schedulers coordinate the units but do not perform their arithmetic.

Sources: NVIDIA Blackwell Ultra architecture; CUDA Binary Utilities instruction reference

Communication can delay the next layer

GPUs must combine dependent results before the next computation can use them.

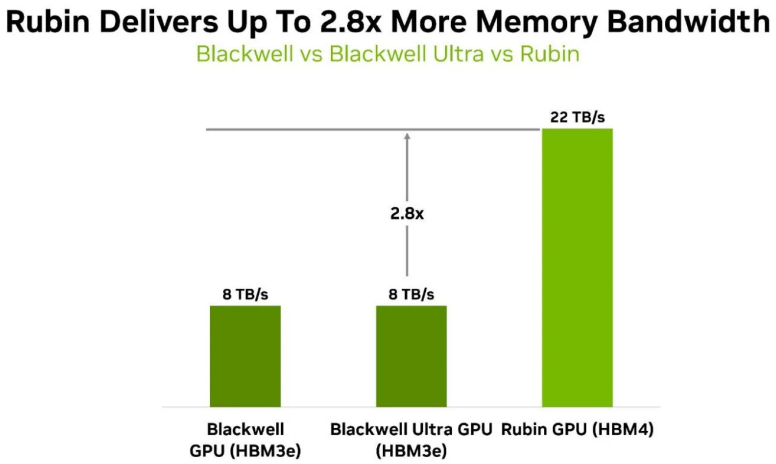
ONE DISTRIBUTED LAYER, LOGICAL COMMUNICATION VIEW

If compute time falls faster than communication time, GPUs spend a larger fraction of each layer waiting for data from their peers.

Example: 8 ms compute + 2 ms exchange becomes 4 + 2 = 6 ms after compute rate doubles.

Rubin memory and communication bandwidth

Different links serve local memory, neighboring GPUs, and the host.



Rubin GPU: 224 SMs and 896 Tensor Cores

ON-PACKAGE MEMORY

Up to 288 GB HBM4
Up to 22 TB/s peak bandwidth

MOVEMENT ENGINES AND CACHES

TMA supports inline descriptor updates. The centralized L2 retains data on chip. These mechanisms reduce movement and setup costs.

SCALE-UP FABRIC

NVLink 6: 3.6 TB/s aggregate per GPU to the switch

Design lesson **Modern GPU optimization means choosing both a storage level and a movement mechanism for every reused tensor tile.**

Image and specifications: NVIDIA Rubin GPU architecture, July 2026

Attention revisited: mapping work onto GPU hardware

Return to the attention algorithm and identify the hardware that serves each stage.

1

Load query, key, and value tiles

TMA + load/store

Move tensor tiles from HBM into shared memory and registers.

2

Form attention scores

Tensor Cores

Compute QK^T, then divide by the square root of the key dimension.

3

Normalize the scores

CUDA cores + SFUs

Subtract row maximum, exponentiate, sum, then divide. The row weights sum to one.

4

Create the context vectors

Tensor Cores + CUDA cores

Multiply the normalized weights by V to form the attention output.

STORAGE USED THROUGHOUT

Registers hold thread-local values. Shared memory stages reusable tiles. Tensor Memory can hold Tensor Core intermediate results on Blackwell-class hardware.

Exact instruction placement depends on the GPU architecture, compiler, and library implementation.

Sources: NVIDIA Blackwell Ultra architecture and softmax execution deep dives

WORKLOAD MILESTONE V

Agentic Computing

What should the machine optimize when the model must act and wait?

Workload	Alternate model execution with tools and observations.
Bottleneck	Persistent state and irregular model–tool dependencies add waiting.
Architectural response	Explore closer integration of compute, memory, and task scheduling.

Emerging workload: model and tool execution

Agentic computing adds tools and persistent state

The complete task alternates between model execution and events outside the GPU.

Heterogeneous execution

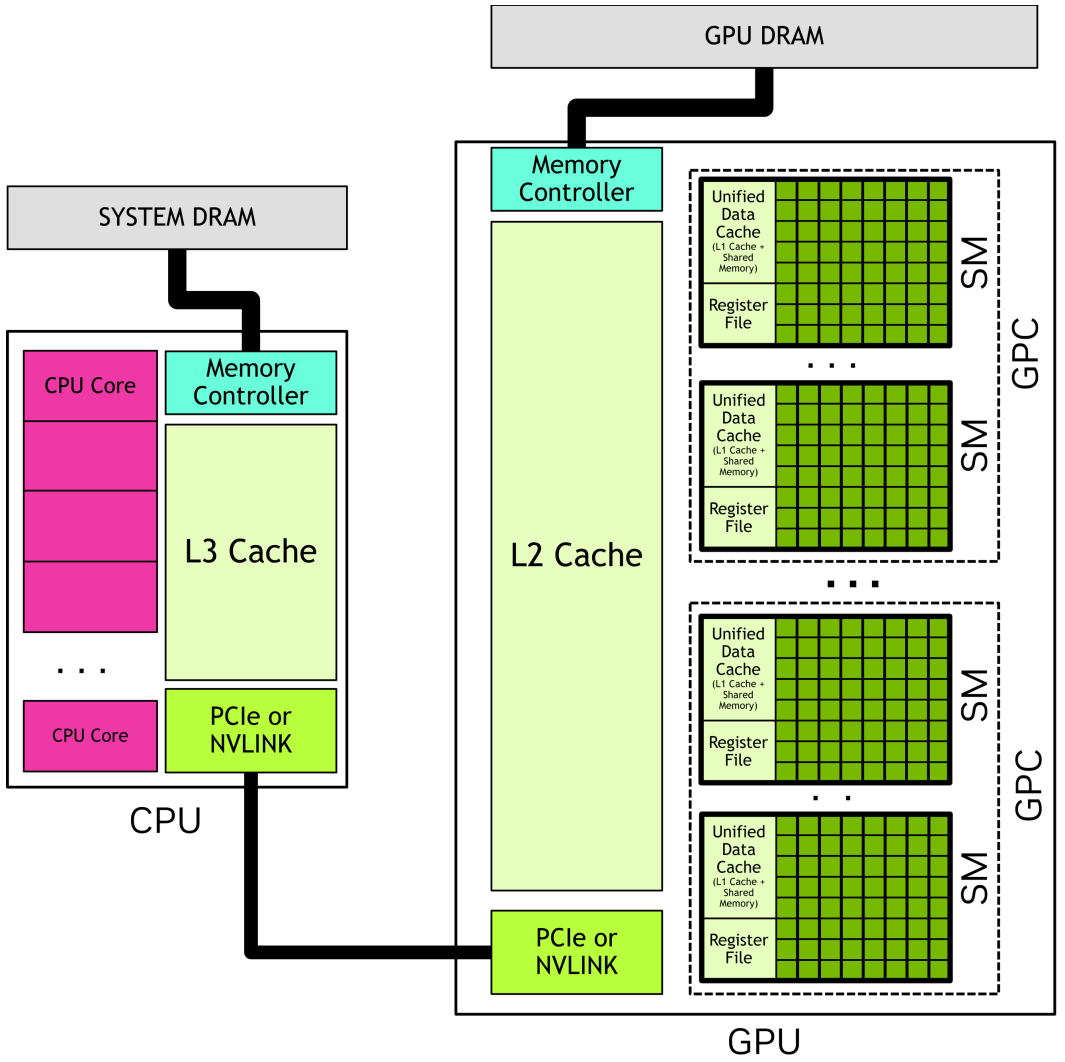
GPU model work alternates with CPU tools, storage, and networks.

Persistent, irregular state

Context and tool results survive pauses, branches, retries, and verification.

Architectural question

Can closer CPU–GPU and memory integration reduce task-level waiting?



Source: Yao et al., ReAct (2022). Integration is a research hypothesis, not a product forecast.

Workloads reshape architecture

Workload changes → bottleneck changes → architectural changes

Workload	Pressure on the machine	Architectural response
Graphics	Many similar scene operations	Parallel arithmetic and rendering stages
Scientific computing	General numerical data and cooperation	CUDA threads, local storage, synchronization
Training	Dense matrix work, state, exchanges	Tensor Cores, HBM, accelerator links
Inference	Weight/context traffic and attention	Less traffic, more bandwidth, balanced execution
Agentic computing	Tools, persistent state, irregular waits	Tighter integration? Measure complete tasks.

These workloads coexist. Architectural responses also enable new workloads.

If agentic AI becomes dominant, what should the computer look like?

Sources: NVIDIA CUDA and inference documentation; ReAct. Final row is an open research question.

Reference: GPU evolution in numbers

Capacity, bandwidth, and arithmetic throughput describe different limits.

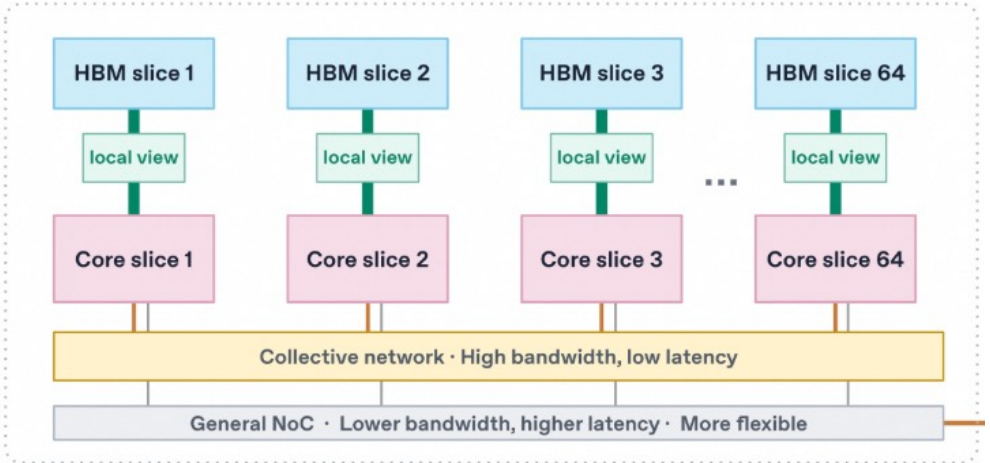
Product	Parallel structure	Peak FP32	Memory	Bandwidth	On-chip storage anchor
GeForce 8800 GTX	128 stream processors	about 0.52 TFLOPS	768 MB GDDR3	86.4 GB/s	16 KB shared memory per SM
Tesla C2050	14 SMs, 448 cores	1.03 TFLOPS	3 GB GDDR5	144 GB/s	64 KB L1/shared per SM, 768 KB L2
Tesla P100 SXM2	56 SMs, 3,584 cores	10.6 TFLOPS	16 GB HBM2	732 GB/s	256 KB registers per SM, 4 MB L2
A100 80 GB SXM	108 SMs, 6,912 cores	19.5 TFLOPS	80 GB HBM2e	2.039 TB/s	192 KB L1/shared per SM, 40 MB L2
H100 SXM	132 SMs, 16,896 cores	67 TFLOPS	80 GB HBM3	3.35 TB/s	up to 228 KB shared per SM, 50 MB L2
Blackwell Ultra B300	160 SMs, 640 Tensor Cores	75 TFLOPS	288 GB HBM3e	8 TB/s	256 KB Tensor Memory per SM
Rubin GPU	224 SMs, 896 Tensor Cores	130 TFLOPS	288 GB HBM4	22 TB/s	centralized L2, capacity not public

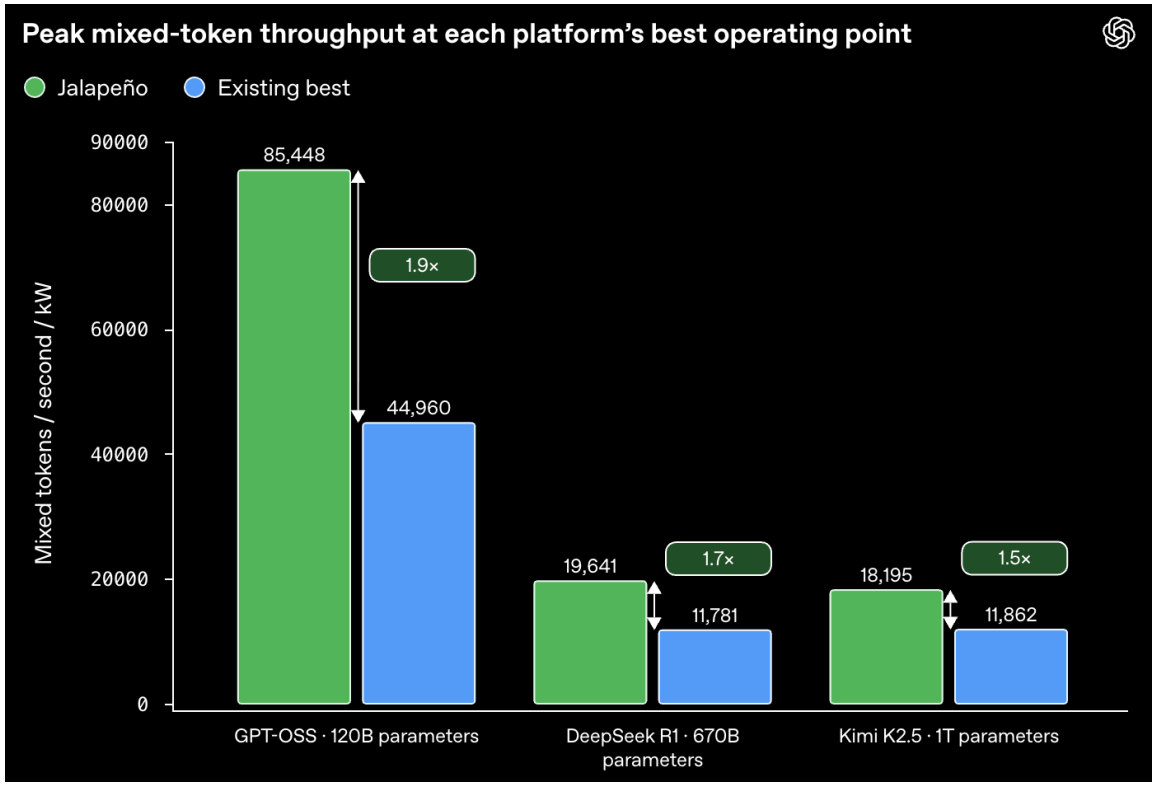
B300 and Rubin both list 288 GB. Bandwidth rises from 8 to 22 TB/s, a 2.75× peak-rate increase.

Sources: NVIDIA product datasheets and architecture whitepapers; Rubin specifications are preliminary

Jalapeño: local memory paths and explicit placement

64 compute slices pair with local HBM regions. Software places data near the computation that uses it. Local tensors, explicit communication, and predictable synchronization.





A shorter local memory path

No shared L2 appears in this diagram. Local HBM access avoids a chip-wide shared-cache path.

Two communication paths

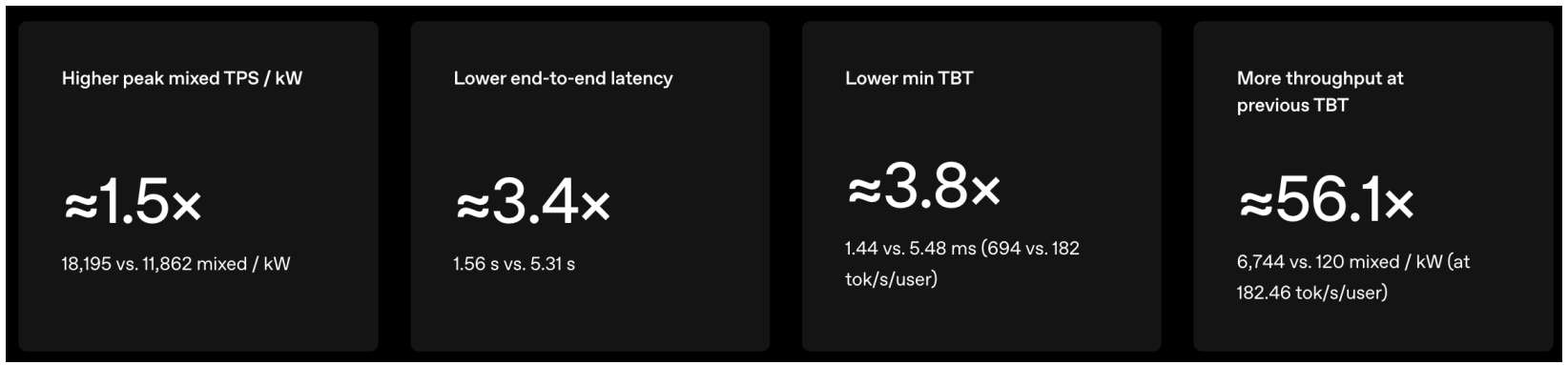
Collective network: common cross-core exchanges.
General NoC: flexible traffic and access to Ethernet.

Per chip: 216 GiB HBM4 at 15.4 TB/s

Matrix peak: 13.4 PFLOP/s for MXFP4 × MXFP4

Trade-off: software must plan data placement and cross-slice communication.

Source: OpenAI, Hot Chips 2026, slides 25 and 32. Diagram excerpt. NoC = network on chip. Detailed cache design is not specified here.


Higher peak mixed TPS / kW
≈1.5×
18,195 vs. 11,862 mixed / kW
Lower end-to-end latency
≈3.4×
1.56 s vs. 5.31 s
Lower min TBT
≈3.8×
1.44 vs. 5.48 ms (694 vs. 182 tok/s/user)
More throughput at previous TBT
≈56.1×
6,744 vs. 120 mixed / kW (at 182.46 tok/s/user)