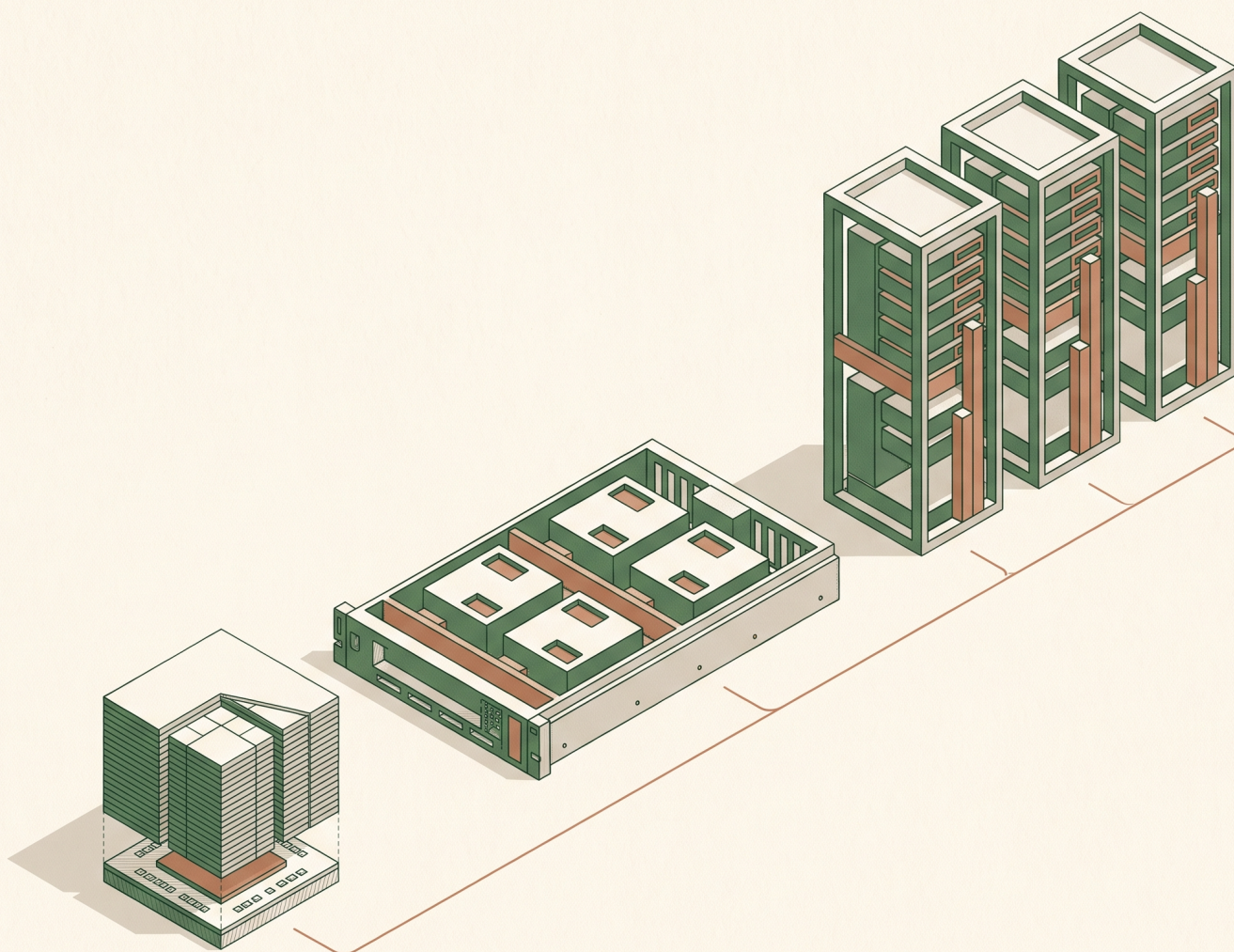


AI Infrastructure

A Beginner-to-Advanced Field Guide



Understand the machine. Follow the work. Design the system.

SEPTEMBER 2026 · ILLUSTRATED LEARNING EDITION

START HERE

You do not have to learn everything at once.

Follow the path your next decision requires. Return to the glossary when a term gets in the way.

AI infrastructure is the system around a model: processors, memory, networks, software and operations. This guide starts with the bytes on one device, follows an inference request, and builds up to distributed training and production service design.

YOUR STARTING POINT	READING ROUTE
New to AI systems	Chapters 1–9: learn the machine and follow one request. Then work Chapter 10’s memory example.
Building your first deployment	Chapters 10–13 and 17: budget memory, choose precision, define latency and verify runtime compatibility.
Preparing for distributed systems work	Chapters 14–16 and 18: count communication, separate training state, then design operations and recovery.
Practicing for an interview	Chapters 19–22: diagnose, calculate, explain a trade-off and describe the measurement that could change your decision.

The running example

A hypothetical dense 8B model with 32 attention layers, 8 KV heads and head dimension 128. It is a teaching configuration, not a named checkpoint or a promise about hardware. Each worked example states what it includes.

Read this alongside [AI Infra Foundations](#). Use [the hardware reference](#) and [the model cards](#) for dated product details. Site links may lead to public references or membership content; access is labeled on each page.

Editorial note: supplied research was reviewed and reorganized for this edition. Numerical examples and vector figures explain mechanisms; no GPU benchmark was run. The cover is conceptual artwork generated with Nano Banana. Sources are original papers, vendor documentation and maintainer references.

CONTENTS

The learning route and the reference

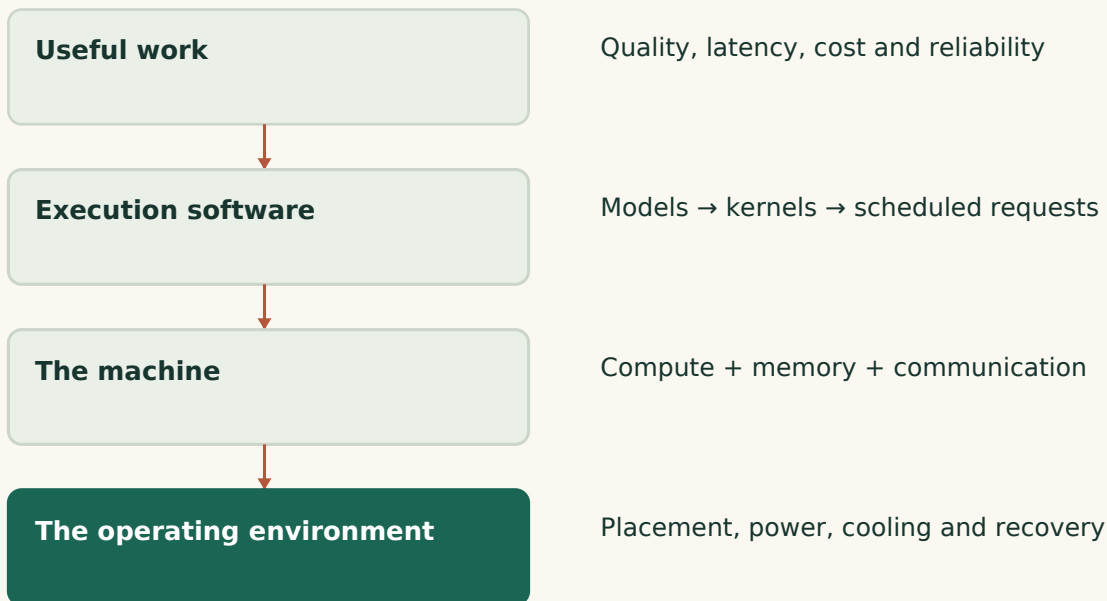
01	The whole system, in one view	4
02	Tensors: the numbers a model moves	6
03	Inside a GPU	8
04	Where the bytes live	10
05	From accelerator to server	12
06	TPUs and other accelerators	14
07	The model changes the machine you need	16
08	Follow one inference request	18
09	Attention remembers keys and values	20
10	Will the model and its users fit?	22
11	Fewer bits, different trade-offs	25
12	Find the limiting resource	28
13	Serve more work without losing responsiveness	31
14	Across devices: links and collectives	33
15	Replicate, shard, or do both	36
16	Training changes the memory budget	38
17	From checkpoint to running service	41
18	Operate the system you designed	43
19	Diagnose the symptom before tuning	46
20	Five deployment calculations	48
21	A learning path you can follow	50
22	Check your understanding	52
23	A glossary you can return to	54
24	Sources and further reading	68

01 / BEGINNER

The whole system, in one view

You call an API and receive a sentence. Between those two events, a computer has to move weights, schedule work and retain state. Start by locating those jobs.

Continue online: [Foundations of AI Infrastructure](#) · [GPU and TPU reference](#) · [Open models: start here](#)



Four layers, one service. A fault at any layer can break the user's request. The arrow shows dependency, not a mandatory software boundary.

A useful first view has four layers. The workload says what must happen: generate text, classify an image, create an embedding, or update model parameters. The execution software translates model operations into kernels and schedules requests. The machine supplies compute, memory, and communication. The operating environment supplies placement, power, cooling, access control, monitoring, and recovery.

Training changes a model's learned parameters using data and an optimization procedure. Inference applies an already trained model to an input. Serving wraps inference in a usable service: it receives requests, checks limits, routes work, batches it, streams results, and handles overload. An inference engine can also run offline without a web service. The training loop and the inference loop therefore share hardware while placing different demands on it.

A CPU handles flexible control work well: operating-system tasks, request handling, parsing, tokenization, and coordination. A GPU supplies many execution resources for parallel numerical work. A TPU is an application-specific accelerator designed by Google for machine-learning operations. These roles overlap; CPUs can run neural networks, and GPUs perform much more than matrix multiplication. An accelerator normally works with a host rather than replacing the whole computer.

A running example. A question enters an API. CPU software converts text into token IDs. A serving engine schedules those IDs on an accelerator that already holds model weights. The accelerator computes an output distribution, a token is selected, and the process repeats. During this loop, accelerator memory holds both long-lived weights and request-specific state. If several devices cooperate, intermediate tensors also cross interconnects.

The five questions to ask about any system are: What is being computed? Where are its bytes? How do they reach the compute units? What work can overlap? What happens when capacity or a deadline is exceeded? The rest of the guide makes those questions concrete.

Language models are the running example

Other workloads deserve different mental models. Embedding and reranking services often use a bounded forward pass rather than long autoregressive decoding. Diffusion generation iterates a denoising network over a sample; its steps are not LLM decode tokens. Multimodal models add image, audio, or video preprocessing and encoders. RAG adds retrieval of external information before generation; a vector database is outside the language model's KV cache.

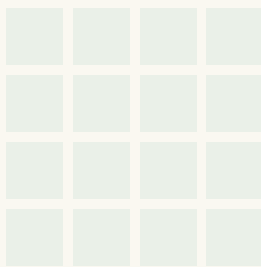
Primary references: [\[1\]](#) · [\[8\]](#) · [\[41\]](#) · [\[64\]](#) · [\[65\]](#)

02 / BEGINNER

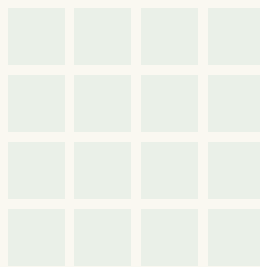
Tensors: the numbers a model moves

“8B” counts learned values. It does not tell you how many bytes they occupy, how many users fit, or how long an answer takes.

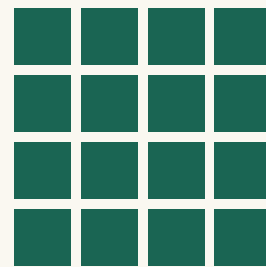
Continue online: [Model Memory Footprint](#) · [Open models: start here](#)



Shape: $4 \times 8 \times 16$



Dtype: 16 bits



Location: device

512 values $\times$ 2 bytes = 1,024 bytes

Changing the dtype changes bytes; it does not change the shape.

A tensor is an array with a shape, a dtype and a location. The small tiles are schematic; the element count follows the stated dimensions.

A scalar is one number, a vector is an ordered list of numbers, and a matrix is a two-dimensional array. A tensor generalizes arrays to any number of dimensions. A tensor's shape specifies the length of each dimension, its dtype specifies how each value is represented, and its device specifies where its storage lives. A tensor shaped 4 by 8 by 16 contains 512 values; at two bytes per value, its raw payload is 1,024 bytes.

Parameters, commonly called weights, are learned numbers in the model. Activations are intermediate results produced for a particular input. The same weights can serve thousands of requests, while each request produces different activations. A layer transforms one activation into another; a forward pass evaluates the required layers. Training additionally computes gradients, which describe how changing parameters affects the loss.

A matrix multiplication combines two arrays through multiply-and-accumulate operations. For a matrix of shape M by K multiplied by K by N, the output has shape M by N. The usual leading-order work count is $2MKN$ floating-point operations, counting a multiply and an add separately. Matrix multiplication is central to neural networks because linear layers apply learned transformations to

many input vectors. Batching creates opportunities to reuse the same weights across more vectors.

The accelerator does not understand paragraphs directly. A tokenizer converts input into discrete token IDs. A token can represent a word, part of a word, punctuation, or other symbols. Tokens do not have a universal character count. An embedding lookup maps an ID to a learned vector; image and audio inputs require their own encoders and representations. An output projection converts a final hidden vector into logits, one score per vocabulary entry, before selection of an output token.

Two shape changes have major infrastructure consequences. Increasing the batch usually increases useful parallelism and memory demand. Increasing sequence length increases the number of token representations, attention work, and often cached state. The exact growth depends on the architecture; counting parameters alone cannot capture either effect.

Useful units. A bit is a binary digit; a byte is eight bits. GB means 10^9 bytes and GiB means 2^{30} bytes. GB/s measures bytes transferred per second; Gb/s measures bits per second. FLOP is an amount of work; FLOP/s is a rate. TOPS counts trillions of operations per second, whose operation type and counting convention must be stated. A high headline number at one precision does not specify performance at another precision or tensor shape.

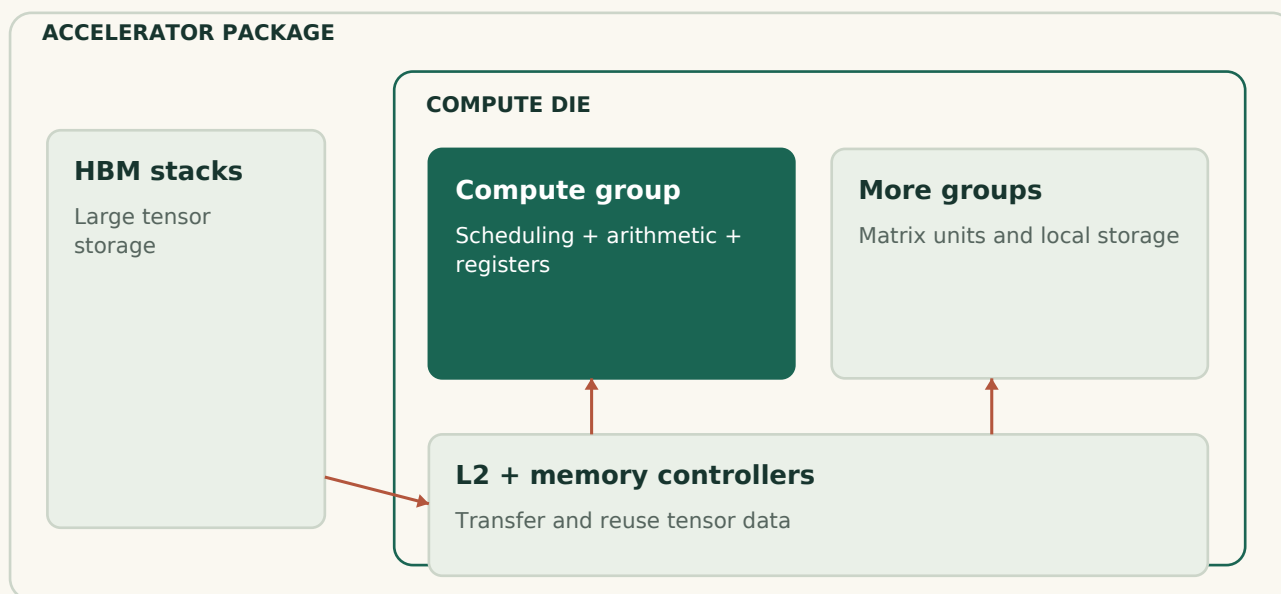
Primary references: [\[1\]](#) · [\[13\]](#) · [\[41\]](#)

03 / BEGINNER

Inside a GPU

A GPU wins when it keeps many arithmetic units supplied with useful work. Its memory system and execution rules explain when that happens.

Continue online: [GPU Execution Model](#) · [Tensor Cores and Matrix Units](#) · [Kernels and the Machine](#)



Conceptual HBM-based GPU package. Compute groups contain arithmetic, scheduling and local storage; HBM is a separate memory structure. Counts and placement are illustrative.

A GPU contains repeated compute groups, an on-chip memory system, controllers for external device memory, and interfaces to the host and other devices. In NVIDIA terminology a compute group is a streaming multiprocessor, or SM. Other vendors use different groupings and names. An SM is much more than one arithmetic core: it has scheduling machinery, arithmetic units, registers, and local storage resources.

Component	What it does	Why a serving engineer cares
Arithmetic units	Execute floating-point, integer, and other instructions	Different operations use different pipelines
Matrix units	Perform matrix multiply-accumulate operations on tiles	Fast paths require supported dtypes and suitable shapes
Warp schedulers	Issue instructions from ready groups of threads	Other ready work can run while one group waits
Registers	Hold immediate per-thread operands and results	Heavy register demand can limit concurrent work

Component	What it does	Why a serving engineer cares
Shared memory	Provides explicitly managed on-chip scratch space	Kernels reuse tiles without repeatedly loading device memory
L1 and L2 caches	Reuse recently accessed data in hardware-managed storage	Locality and access patterns affect memory traffic
Load and store units	Move operands between execution resources and memory	Poor access patterns can waste transactions
Memory controllers	Connect the compute die to device DRAM	Capacity and sustainable bandwidth limit different workloads
Copy engines	Execute supported asynchronous data transfers	Transfers may overlap compute when dependencies allow
Host and peer interfaces	Connect the GPU to CPUs, NICs, or other accelerators	Topology determines which communication paths are available

A kernel is a function executed on the accelerator. In CUDA, a launch creates a grid of thread blocks. Threads in a block can cooperate through shared memory and synchronization. NVIDIA threads execute in groups of 32 called warps, using a single-instruction, multiple-thread model called SIMT. A CUDA thread is a programming unit, not a dedicated physical core. Different branches within a warp can reduce effective parallel execution.

A Tensor Core is NVIDIA's name for a matrix-compute hardware unit. It is not an entire GPU, and a quoted count is not directly comparable with CPU core counts. Kernels normally split large matrices into tiles, load tiles, reuse operands, and accumulate partial results. The work is fast when arithmetic resources receive data efficiently; arithmetic capacity alone cannot prevent a stall.

Advanced depth. Occupancy is the fraction of an SM's supported resident warps that are active. High occupancy can help hide latency but is not a universal objective. Register usage, shared-memory allocation, instruction dependencies, memory access, and the number of ready blocks all matter. Likewise, GPU utilization reporting nonzero kernel activity does not reveal whether matrix units are near peak throughput.

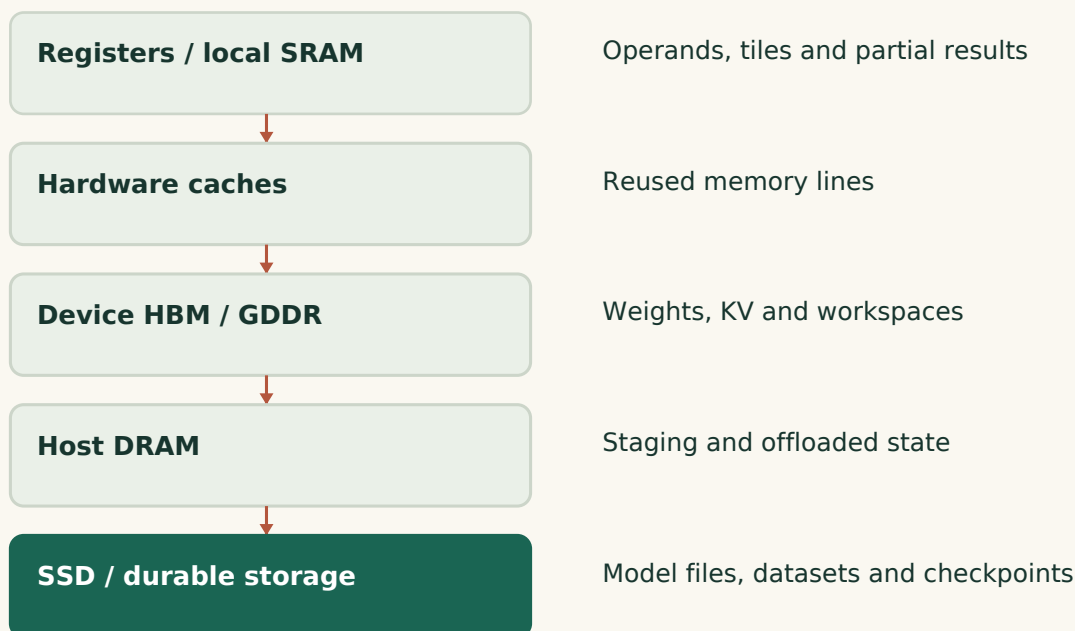
Primary references: [\[1\]](#) · [\[2\]](#) · [\[5\]](#)

04 / BEGINNER

Where the bytes live

A model can fit comfortably in memory and still generate slowly. Capacity and the speed of moving bytes answer different questions.

Continue online: [GPU Memory Hierarchy](#) · [Memory-Bound vs Compute-Bound Kernels](#)



Storage levels have different capacities, access costs and lifetimes. The connecting line indicates movement; direct paths can bypass intermediate copies.

The word memory can refer to several physically different resources. On-chip SRAM supports small, fast working sets. Device DRAM holds much larger tensors. Host DRAM holds operating-system and application data. SSDs and object stores retain files after a process or machine stops. Moving between these levels consumes time and energy.

HBM, high-bandwidth memory, is stacked DRAM with a very wide interface. Multiple DRAM dies are connected vertically, commonly through through-silicon vias, and packaged close to the accelerator. HBM is not the same physical structure as an SM's registers or SRAM. The package can place memory very close to compute without making all those bytes equally fast to access.

Storage level	Typical contents	Lifetime and access
Registers	Immediate operands and partial sums	Scoped to executing work
Shared memory or TPU local SRAM	Tiles and temporary working data	Managed for a kernel or compiled program

Storage level	Typical contents	Lifetime and access
Hardware caches	Recently used memory lines	Evicted automatically; not a reliable application allocation
HBM or GDDR	Weights, KV cache, activations, workspaces	Allocated and managed by runtime software
Host DRAM	Token buffers, CPU tensors, staging and offloaded state	Usually across host-device links for accelerator use
Local NVMe SSD	Model files, dataset cache, temporary checkpoints	Persistent medium; local availability depends on service
Shared storage or object store	Durable model versions and training data	Accessed over storage and network systems

Capacity answers whether all required bytes fit. Bandwidth answers how quickly a stream of bytes can move. Latency answers how long an individual operation waits. A workload can have plenty of free memory and still be bandwidth-bound. It can also have excellent bandwidth but fail because its allocations exceed capacity.

GDDR is graphics-oriented DRAM used by many GPUs. DDR is a family of DRAM interfaces commonly used for CPU system memory. VRAM is a broad name for GPU-accessible device memory, not a synonym for HBM. On some systems CPUs and accelerators share physical memory; on others they have separate memories linked by PCIe or coherent interfaces. Always inspect the actual system.

Unified virtual addressing gives allocations a consistent address-space view. Unified Memory can manage placement and access across processors. Neither name promises that every byte has local-device bandwidth. Page migration, remote access, coherence, and oversubscription can make a convenient allocation unexpectedly expensive.

A critical distinction. The KV cache is application state stored in allocated tensor memory. L1 and L2 are hardware caches that may temporarily cache portions of those tensors. Saying that the KV cache lives in HBM and that a kernel reuses KV data in on-chip storage is consistent: the statements refer to different levels of storage and different lifetimes.

Make each memory transaction useful

When neighboring CUDA lanes read adjacent values, the hardware can serve them with fewer memory transactions. This is coalescing. A tiled matrix multiply then goes further: it loads a small operand region into fast local storage and reuses it for several multiply-accumulates. Coalescing reduces wasted transfers; tiling reduces how often useful values must be transferred. Check the profiler before trading more registers or shared memory for another tile.

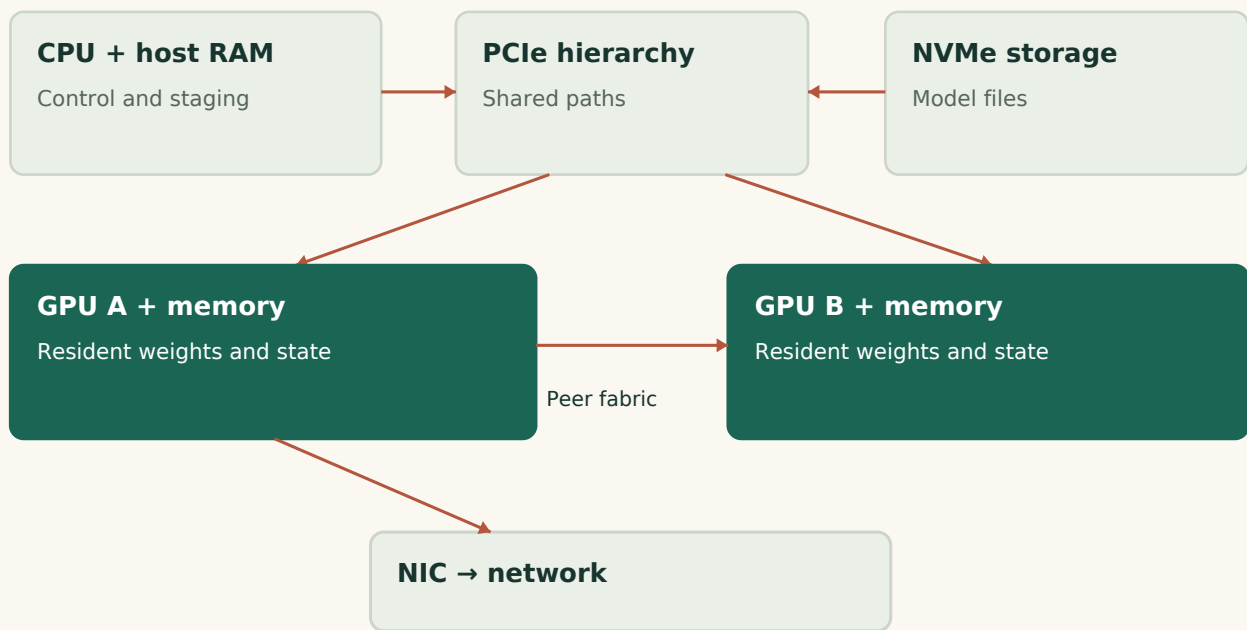
Primary references: [\[4\]](#) · [\[5\]](#) · [\[6\]](#)

05 / BEGINNER

From accelerator to server

Two GPUs with the same chip can behave differently in different servers. Follow their paths to the CPU, their peers and the network.

Continue online: [NVLink, NVSwitch and PCIe](#) · [GPU and TPU reference](#)



Each GPU retains its own device memory. PCIe, an optional peer fabric and the network are separate paths with their own sharing limits.

A server is a complete computer around the accelerators. It needs CPUs, system memory, an operating system, firmware, persistent storage, network adapters, power supplies, cooling, and management hardware. Real accelerator servers often provide separate ports or paths for compute communication, storage access, and management. The NVIDIA DGX system guide is one concrete example of this decomposition, not a universal server specification.

PCIe is a point-to-point I/O interconnect. Its generation and negotiated lane count, such as x16, affect link capacity. A CPU's PCIe root complex connects the host to an I/O hierarchy. A PCIe switch fans out connectivity, but upstream links can become shared bottlenecks. Retimers improve signal integrity on demanding links; they do not create additional end-to-end bandwidth.

NUMA, non-uniform memory access, matters in servers with multiple CPU memory domains. A GPU and NIC near one CPU socket may communicate less efficiently with CPU threads or buffers placed near another. CPU affinity, memory placement, and GPU-NIC affinity therefore belong in performance diagnosis. Two visually identical PCIe slots need not have equivalent paths to memory or the network.

A NIC connects the server to a network. An InfiniBand adapter is commonly called an HCA, or host channel adapter. A DPU or SmartNIC can offload selected infrastructure functions; it is not required to evaluate every model. A BMC, baseboard management controller, provides independent management such as power control, health information, and remote console access.

An accelerator may be a PCIe add-in card or a module on a specialized baseboard. SXM is one NVIDIA module form factor; it is not a synonym for NVLink, and its capabilities depend on the product and baseboard. Chiplet describes a package built from multiple dies. Packaging affects inter-die communication, thermal design, yield, and available connections. Software's apparent device count does not always reveal the number of physical dies.

At startup, model tensors commonly move from durable storage through local storage or host RAM into device memory. GPUDirect Storage can support a more direct storage-to-GPU data path on compatible systems. It reduces particular copies or CPU work; it does not remove the need for storage bandwidth, synchronization, or filesystem support.

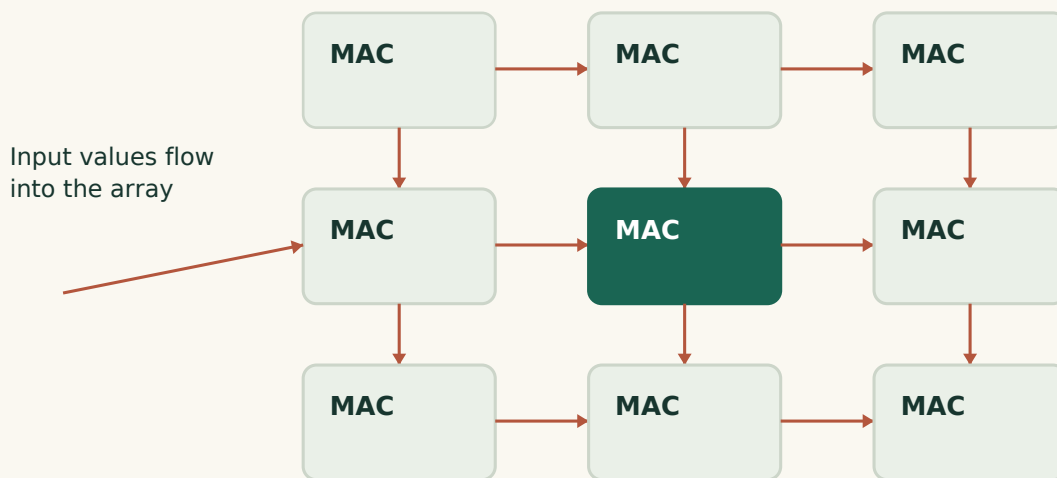
Primary references: [\[7\]](#) · [\[39\]](#) · [\[58\]](#)

06 / BEGINNER

TPUs and other accelerators

The transferable idea is data reuse. Learn how a TPU moves values through matrix units, then ask which software and communication paths support your workload.

Continue online: [TPU Architecture and Systolic Arrays](#) · [AMD Instinct and ROCm](#) · [GPU and TPU reference](#)



MAC = multiply and accumulate. Reuse happens between neighbors.

A tiny conceptual systolic array. Neighbor transfers reuse values for multiply-accumulate work; a real TPU's array size and dataflow vary by generation.

A TPU's main computational engine is a matrix multiply unit, or MXU. A systolic array arranges multiply-accumulate cells so operands and partial results can pass between neighboring cells. Reusing values within the array reduces repeated movement to larger memory. Vector and scalar units handle other operations. TPU terminology such as TensorCore describes its own organization and should not be mapped directly to NVIDIA Tensor Core counts.

TPUs still have memory and communication limits. HBM stores larger tensors, and local SRAM serves working data. Google compiler documentation describes how tensor dimensions, padding, and layouts affect efficient mapping to hardware. If a logical tensor does not align with an efficient hardware shape, extra padded work can reduce useful utilization. Supported dtypes and operator paths also vary.

XLA, Accelerated Linear Algebra, is a compiler that optimizes computation and lowers it to supported hardware backends. JAX and other frameworks can feed it programs. Compilation can fuse operations, choose layouts, and partition work. Recompilation or multiple compiled variants may be needed when important shapes change. Compiled execution is not exclusive to TPUs; GPU software also uses compilers and captured execution graphs.

ICI is Google's inter-chip interconnect within a TPU slice. A slice is a connected allocation from a TPU pod. A pod in TPU architecture describes accelerator infrastructure; it is different from a Kubernetes Pod. Multislice connects work across slices using the data-center network, or DCN, while each slice retains its ICI fabric. This introduces a hierarchy of communication costs.

AMD GPUs expose their own architecture and a software ecosystem called ROCm, with HIP as a programming interface and associated libraries and tools. The transferable concepts are kernels, memory placement, communication, and supported numerical operations. CUDA-specific source, assumptions about warp size, or one vendor's optimized kernel cannot be assumed to run unchanged elsewhere.

Other accelerators include NPUs, FPGAs, and purpose-built ASICs. An NPU is a broad category, not one shared architecture. An FPGA is reconfigurable logic; an ASIC has a more specialized fixed design. For any option, ask whether the model's operators, precision, shapes, serving features, and observability are supported. A fast chip with a missing execution path may be unusable for the workload.

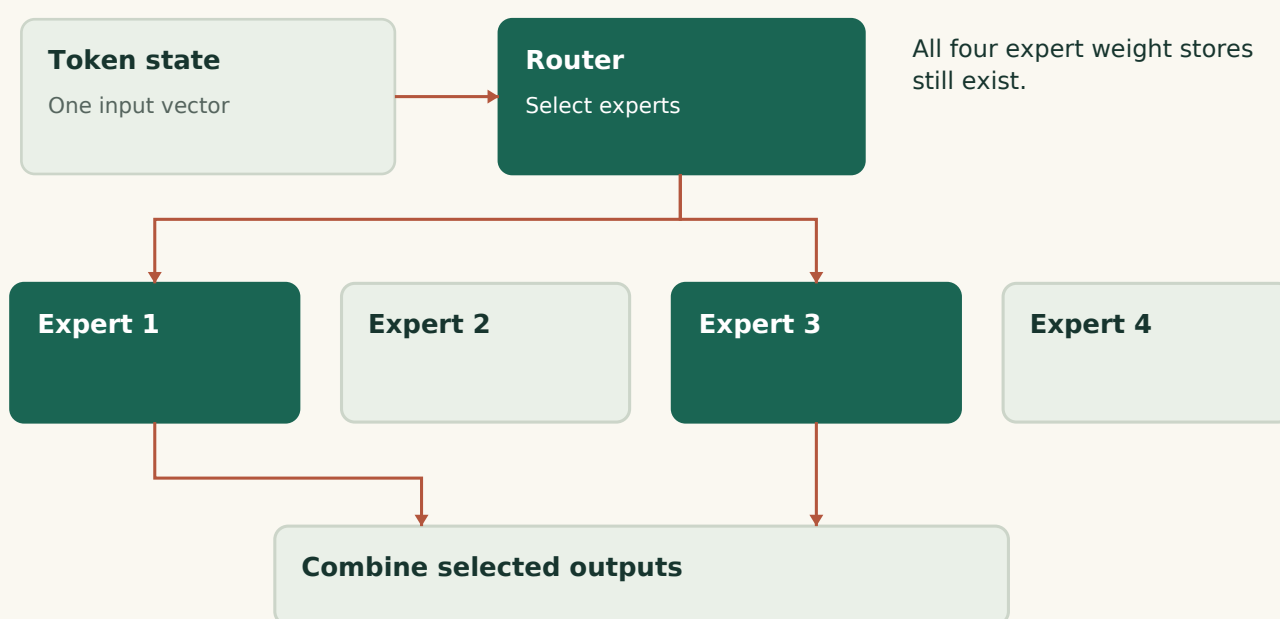
Primary references: [\[8\]](#) · [\[9\]](#) · [\[10\]](#) · [\[11\]](#) · [\[12\]](#)

07 / BEGINNER

The model changes the machine you need

Dense layers, routed experts and recurrent state create different memory and communication demands. Parameter count is only the first clue.

Continue online: [Open models: start here](#) · [Attention Variants: MHA, GQA, MQA and MLA](#) · [Open model reference](#)



A toy routed-expert layer selects two of four experts for one token. Unselected weights still need storage; another token may select different experts.

Hardware architecture describes a machine. Model architecture describes the mathematical structure executed on it. Serving architecture describes how software places and schedules that execution. These three meanings of architecture interact, but changing one does not necessarily change the others.

The original Transformer contains attention and feed-forward layers, with residual connections and normalization around sublayers. A decoder-only language model uses causal attention to generate sequences. An encoder often processes an entire input with bidirectional attention and returns representations. An encoder-decoder model encodes an input and then generates an output, adding cross-attention and potentially cached encoder-derived state.

Architecture feature	High-level meaning	Infrastructure consequence
Dense model	The usual dense layers participate for every token	Most dense weight matrices are repeatedly used

Architecture feature	High-level meaning	Infrastructure consequence
Mixture of experts	A router chooses a subset of expert networks	Active compute and total weight memory differ
Multi-head attention	Each query head has its own key and value heads	More KV state than shared-head alternatives, all else equal
Multi-query attention	Many query heads share one KV head	Smaller KV payload; specialized attention behavior
Grouped-query attention	Groups of query heads share KV heads	KV memory follows the KV-head count
Sliding or local attention	Some layers attend only to a bounded region	Their cache can be bounded by a window
Latent-compressed attention	A compact latent representation replaces some expanded cached state	Requires an architecture-specific memory formula
State-space or recurrent layer	Maintains and updates recurrent state	State does not follow the ordinary full-attention KV formula
Hybrid model	Combines attention and other layer types	Sum each layer family's actual state requirements

Mixture of experts, or MoE, typically routes tokens through selected feed-forward experts. A model can have many total parameters while activating a smaller portion per token. Unless the design offloads or streams experts, the full set of deployed expert weights still occupies memory somewhere. Distributed experts introduce token dispatch, return traffic, and possible load imbalance. MoE sparsity is different from sparse attention and from hardware-supported sparsity inside a matrix.

MQA and GQA share keys and values across query heads to reduce their storage and movement. With the same layers, context length, head dimension, and dtype, eight KV heads use one quarter of the raw KV storage of 32 KV heads. This is an architectural comparison, not a setting that can generally be changed arbitrarily in a trained checkpoint.

MLA, multi-head latent attention, is an example of compressed attention state. Its cached latent vectors and positional components do not fit a naive count of expanded K and V heads. State-space models, such as the selective-state-space approach in Mamba, use another state representation. Hybrid models can retain full-attention caches in some layers and recurrent state elsewhere. Always inspect the model configuration and runtime's actual allocation layout.

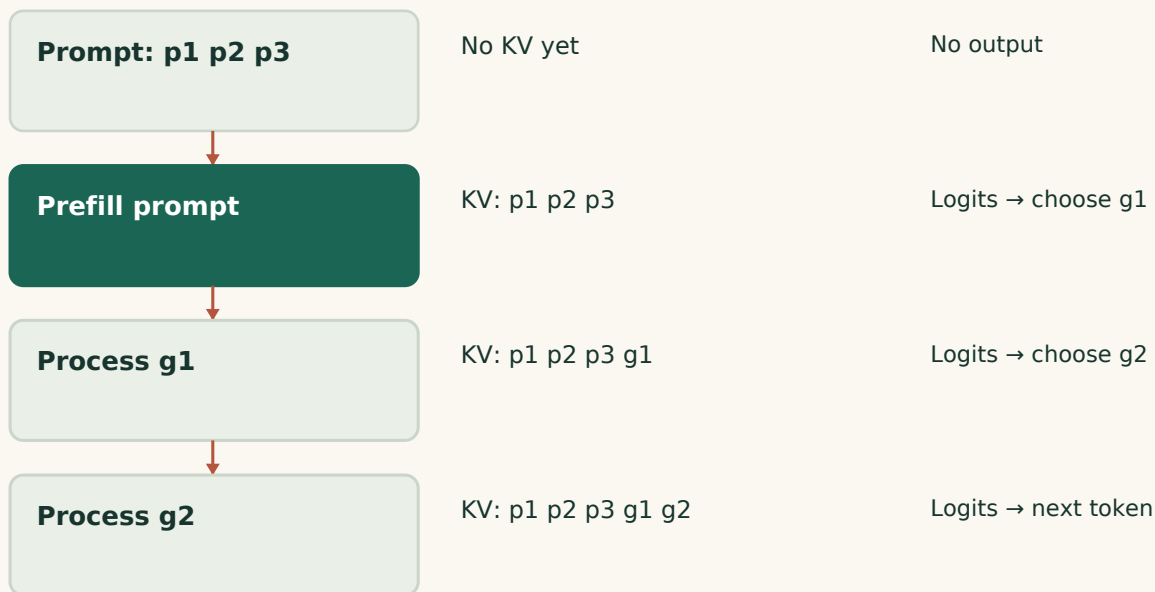
Primary references: [\[13\]](#) · [\[18\]](#) · [\[19\]](#) · [\[20\]](#) · [\[47\]](#) · [\[48\]](#) · [\[64\]](#) · [\[65\]](#)

08 / BEGINNER

Follow one inference request

Use a warm, ordinary causal language model as the example. Weights are already loaded; the request must still earn its place in the schedule.

Continue online: [Prefill vs Decode](#) · [Inference Engineering](#)



A warm causal model with a three-token prompt. Selecting a token and computing that token's K/V happen at different instants. Counts are positions per attention layer.

For this chapter assume a resident decoder-only Transformer, causal attention, one generated sequence, ordinary autoregressive decoding, and no prefix-cache hit. Different serving engines combine or move individual operations, but the data dependencies remain the useful guide.

1. Load and prepare the model. Before ordinary requests arrive, a worker loads the correct weights, tokenizer, and configuration; allocates runtime memory; and may compile, select kernels, or warm up execution. Model loading is generally not repeated for every request to an already warm worker.
2. Receive and validate the request. The service checks identity, requested model, input size, and generation limits. Optional retrieval or tool results are incorporated into the input. Chat templates and special tokens are part of the actual model input.
3. Tokenize and admit work. Text becomes token IDs, commonly on the CPU. The scheduler accounts for input length, possible output growth, available KV blocks, and other active work. A request may wait before accelerator execution begins.

4. Move input and metadata. Token IDs, positions, masks or mask metadata, and block mappings become available to the accelerator. Model weights remain resident. The literal prompt is small compared with the tensors subsequently read and produced.
5. Prefill the prompt. The model processes known prompt positions through its layers. Causal masking prevents a position from attending to later positions, even when operations across positions execute in parallel. Each attention layer creates K and V for the prompt, which are retained for reuse.
6. Select the first output token. Logits associated with the final prompt position predict the next token. Selection may use greedy choice or sampling, subject to generation constraints. Sampling can execute on the GPU or CPU depending on the engine.
7. Decode repeatedly. The selected token enters another forward pass. At each attention layer, its new K and V join existing cache state, and its query attends to the permitted past and current keys and values. Final logits predict the following token.
8. Stream and finish. Selected IDs are converted into text and returned, sometimes in chunks. An end-of-sequence token, length limit, cancellation, or another stopping rule ends generation. Live state is freed or retained under an explicit cache policy.

The first output token normally comes from prefill's final-position logits. A later decode pass is needed to compute that selected token's own layer states and predict the next token. Consequently, just after selection, the most recently emitted token may not yet have a KV entry. Memory planning usually budgets maximum retained context conservatively instead of relying on this one-token boundary.

Prefill offers parallel work across the prompt's known positions; decode has a dependency on the preceding generated token within one sequence. Independent sequences can still run together. Ordinary decoding therefore has sequential rounds even when each round performs massively parallel arithmetic. Speculative methods alter how candidate rounds are proposed and verified, discussed in [Chapter 13](#).

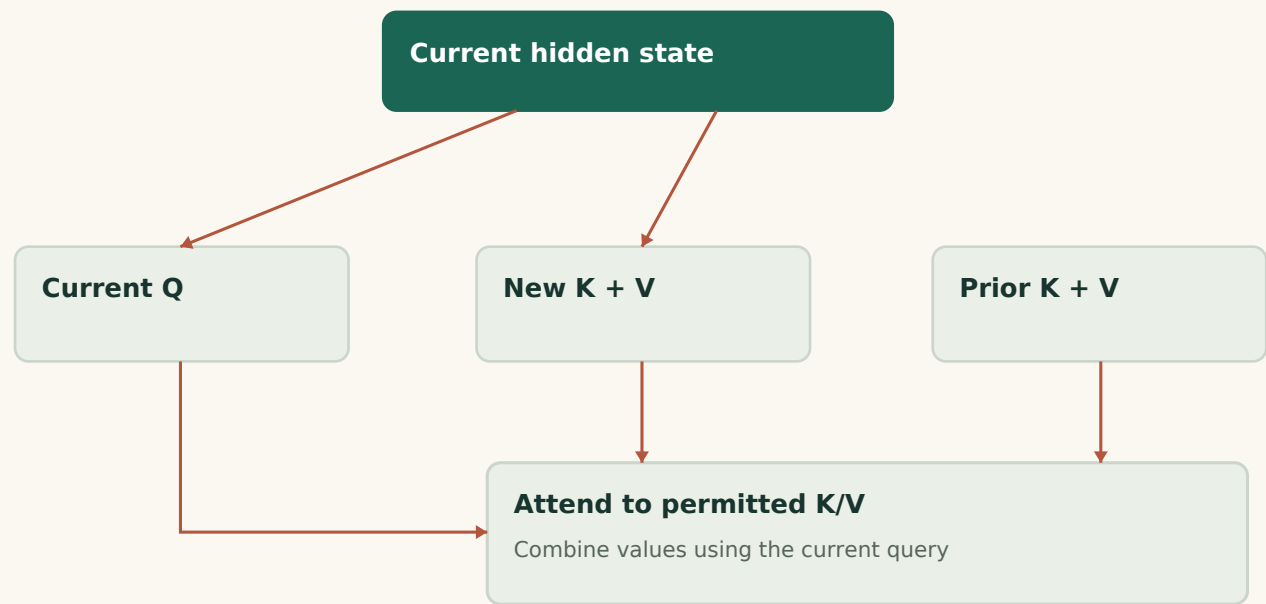
Primary references: [\[14\]](#) · [\[24\]](#) · [\[25\]](#)

09 / BEGINNER

Attention remembers keys and values

The model reuses information computed for earlier tokens. It still has to read that information and compute each new token.

Continue online: [The KV Cache](#) · [Prefix Caching and KV Reuse](#)



Then: projection + rest of block

One ordinary attention layer. Newly computed K/V join the permitted retained state. Cached representations are distinct from the learned projection weights.

Attention lets one position combine information from permitted positions. A learned projection produces a query, another produces a key, and another produces a value. The query is compared with keys to generate scores; normalized scores weight the values. Q, K, and V are tensor representations produced by model layers, not database records or copies of the original words.

For a causal decoder, earlier token states do not change when a new token is appended under the same model, positions, and preceding context. Reusing their K and V avoids recomputing the entire prefix. Earlier queries are usually unnecessary for the next output; the current query is what must look back. Each attention layer has its own cache, so caching at just the final layer would not replace the other layers' work.

Where the KV cache sits. In a common fully resident GPU server, it is stored as allocated tensors or paged blocks in GPU HBM or GDDR. The runtime allocates the blocks and tracks which sequence owns each logical position. CPU-side control metadata may map to GPU-resident payloads. A multi-device runtime may shard or replicate parts of the cache. Offloaded designs keep some state in host RAM or another tier and transfer it as needed.

At execution time, an attention kernel reads relevant cache data through the device memory hierarchy and may stage tiles in local SRAM or registers. The exact traffic depends on attention type, tiling, reuse, quantization, and cache residency. Storing more context consumes capacity; repeatedly reading it can consume bandwidth. These are distinct costs.

One complete Transformer block

In a common pre-normalization decoder block, the incoming hidden state is normalized, then projected into Q, K, and V. Positional information is applied according to the model. Attention combines permitted values, an output projection transforms the result, and a residual addition combines it with the block input. Another normalization and a feed-forward network transform the intermediate representation; another residual addition produces the block output. Other models arrange normalization differently or replace the feed-forward network with routed experts.

The feed-forward network commonly expands the hidden representation, applies an activation or gate, and projects it back down. Its intermediate activations are not the attention KV cache. These layers contribute substantial weight reads and arithmetic even when KV reuse eliminates repeated prefix computation. After the final block, a final normalization and vocabulary projection commonly produce output logits.

RoPE, rotary positional embeddings, incorporates position by rotating components of query and key representations. It helps make attention sensitive to relative position. Cache reuse must preserve the model's positional convention; increasing a configured context limit alone does not establish that a model will work accurately at that length.

Five things called a cache. A KV cache saves attention state. A prefix cache reuses compatible state for identical token prefixes. An answer cache reuses a previous response. A hardware cache reuses memory lines. A filesystem or model cache avoids downloading model files again. These mechanisms store different objects and save different kinds of work.

A prefix cache requires compatible model weights, adapters, tokenization, positions, and relevant input state; merely having similar sentences is insufficient. It mainly avoids repeated prefill of a shared prefix. It does not remove subsequent decode attention over that prefix. Retaining cache blocks also consumes memory, so cache-hit rate alone does not determine overall benefit.

Advanced depth. Local-attention layers may discard older positions, while global-attention layers keep longer histories. Cross-attention can cache projections of an encoded input. Beam search maintains multiple candidate sequences and may share prefix blocks before branches diverge. Recurrent and latent-compressed models need separate accounting. A conversation history stored in a database is not automatically a live accelerator KV cache.

Primary references: [\[13\]](#) · [\[14\]](#) · [\[15\]](#) · [\[18\]](#) · [\[19\]](#) · [\[20\]](#) · [\[66\]](#)

10 / INTERMEDIATE

Will the model and its users fit?

Budget weights, live request state and working space separately. This chapter uses one hypothetical 8B model so every number can be checked.

Continue online: [KV Cache Sizing](#) · [KV-cache memory calculator](#) · [Model Memory Footprint](#)

One device. Three separate budgets.



- **Weights: 14.90 GiB**
- **Other state + reserve: 3.00 GiB**
- **Raw KV allowance: 6.10 GiB**

$32 \text{ layers} \times 8 \text{ KV heads} \times 128 \text{ dimensions} \times 2 \text{ bytes} \times 2 = 128 \text{ KiB/token}$

Hypothetical 24 GiB device, 8B uniform BF16 weights and 3 GiB non-KV reserve. The raw KV allowance is an arithmetic ceiling, not a safe concurrency guarantee.

Start by listing every allocation class. A practical inference budget is: resident weights plus KV state plus temporary activations and workspaces plus communication buffers plus runtime allocations and fragmentation allowance. Some engines reserve most available device memory for a cache pool, so reported allocation can stay high even when relatively few cache blocks contain live request state.

Raw weight storage

For P stored parameters and b stored bits per parameter, the raw payload is P multiplied by b divided by 8 bytes. This excludes scales, zero points, padding, format metadata, higher-precision layers, and temporary conversion buffers. A hypothetical 8-billion-parameter model needs 16 billion bytes at 16 bits, which is about 14.90 GiB. At a uniform packed 4 bits, the raw weight payload is 4 billion bytes, or about 3.73 GiB. Actual load-time and resident footprints can be larger.

Ordinary full attention KV storage

For identical attention layers, equal K and V head dimensions, and no sharing or compression, the raw cache payload is:

$$\text{KV bytes} = 2 \times L \times H_{kv} \times D_{head} \times \text{bytes_per_value} \times \text{sum}(\text{retained_tokens_per_sequence})$$

The factor two counts keys and values. L is the number of attention layers, H_{kv} is the number of key/value heads, and D_{head} is each head's dimension. The sum counts actual retained tokens across active sequences. For equal-length sequences, replace the sum with batch size multiplied by retained length. This equation follows directly by counting the elements in the layer cache tensors; head sharing and cache implementations determine which elements are actually retained.

Worked example. Use 32 attention layers, 8 KV heads, head dimension 128, and two-byte KV values. One retained token across all layers costs $2 \times 32 \times 8 \times 128 \times 2 = 131,072$ bytes = 128 KiB. At 8,192 retained tokens, one sequence uses 1 GiB of raw KV payload. Sixteen equal-length sequences use 16 GiB.

Change to the example	Raw KV per sequence	Reason
8 KV heads and 8,192 tokens	1 GiB	Reference configuration
32 KV heads	4 GiB	Four times as many KV vectors
1 KV head	0.125 GiB	One eighth as many KV vectors
16,384 retained tokens	2 GiB	Twice as many stored positions
One-byte KV values	0.5 GiB	Half the payload bytes; metadata excluded

These changes isolate one variable; they do not imply that one trained model can accept every configuration. Quantizing weights alone does not halve KV storage. Context length and concurrent sequence count multiply each other in the cache budget.

Put the budget together

Suppose a hypothetical device has 24 GiB of usable memory, the model's raw weights occupy 14.90 GiB, and a measured or deliberately reserved non-KV allowance is 3 GiB. The remaining theoretical KV payload budget is 6.10 GiB. Under the 1 GiB-per-sequence assumptions above, six sequences are the arithmetic ceiling, before additional block rounding or reserve. This is a sizing exercise, not a safe production concurrency setting.

On multiple GPUs, do not divide the total by device count without checking placement. Layers, heads, experts, embeddings, and workspaces can be partitioned differently. Some KV heads can be replicated when parallelism exceeds a useful head partition. Each device must satisfy its own

peak allocation budget. Temporary loading, graph capture, and overlapping requests can exceed steady-state usage.

For hybrid attention, sum each layer's actual retained state separately. For quantized caches add scales and any higher-precision residual region. For MLA or recurrent layers use their runtime-specific state tensors. For the linked KV calculator, choose a matching architecture and confirm its displayed units and assumptions before comparing results with this GiB example.

Change one assumption

Keep the weights at BF16 and halve only KV precision. Under the raw-payload formula, each 8,192-token sequence falls from 1 GiB to 0.5 GiB. The arithmetic ceiling becomes 12 sequences. Real cache scales, quality and runtime support must be checked separately.

Primary references: [\[14\]](#) · [\[15\]](#) · [\[18\]](#)

11 / INTERMEDIATE

Fewer bits, different trade-offs

A smaller representation saves bytes by approximating values. Whether it helps depends on which tensors change and which kernels execute them.

Continue online: [Quantization for Inference](#) · [Open models: start here](#)

0.37 → code 4 → reconstructed 0.40

**Absolute reconstruction error: 0.03**

A value above 0.7 is clipped in this particular signed 4-bit example.

A toy signed 4-bit affine quantizer with scale 0.1 and zero point 0. Codes run from -8 to 7 . Real quantizers can use other formats, grouping and rounding rules.

Precision concerns how numbers are represented. Floating-point formats encode a sign, an exponent, and a significand. The exponent mainly determines range, while the significand affects resolution. Quantization maps values into a smaller representable set, usually to reduce storage, traffic, or arithmetic cost. It changes representation and can introduce error; it does not automatically reduce parameter count.

Format	Nominal storage	Interpretation
FP32	4 bytes	Common high-precision working or accumulation format
FP16	2 bytes	Half precision; limited exponent range compared with FP32
BF16	2 bytes	FP32-like exponent range with fewer significand bits
FP8	1 byte	A family of small floating-point formats with different ranges

Format	Nominal storage	Interpretation
INT8	1 byte	Integer representation, normally with scale information
INT4	0.5 byte when packed	Sixteen integer code values before scaling conventions
FP4 and block-scaled formats	Nominally 4 bits plus metadata	Small floating-point values, often with shared scaling

BF16 and FP16 consume the same raw bytes while making different range-versus-resolution trade-offs. Accumulation may use a wider format than inputs, so an operation's input dtype does not specify the precision of every intermediate calculation. Hardware support and rounding behavior must be checked.

A small quantization example

An illustrative affine integer quantizer computes $q = \text{clamp}(\text{round}(x / \text{scale}) + \text{zero_point})$ and reconstructs $\hat{x} = \text{scale} \times (q - \text{zero_point})$. With a scale of 0.1, zero point zero, and a sufficiently wide integer range, 0.37 becomes code 4 and reconstructs to 0.4. The absolute error is 0.03. Values outside the representable range are clipped, which can create much larger errors.

A scale sets the spacing of reconstructed values; a zero point maps numerical zero to a code. Symmetric schemes generally use a zero-centered mapping, while asymmetric schemes can shift it. Per-tensor scaling shares one scale widely; per-channel or per-group scaling adapts to smaller subsets at the cost of more metadata and processing. Floating-point and nonlinear quantizers need not use this exact affine rule.

What gets quantized

Weight-only quantization compresses weights while activations may remain in BF16 or FP16. Weight-and-activation quantization also compresses selected runtime tensors. KV-cache quantization targets retained attention state. These are independent decisions. W4A16 is shorthand for four-bit weights and sixteen-bit activations, but it does not specify the KV format, accumulators, grouping, or every layer.

Post-training quantization, or PTQ, converts an existing trained model, often using representative calibration inputs. Quantization-aware training, or QAT, exposes training to quantization effects so the model can adapt. Calibration coverage matters: a method tested on ordinary prose may need separate evaluation for code, long context, or other distributions.

GPTQ is a weight-quantization method using approximate second-order information. AWQ uses activation statistics to choose weight-channel scaling. SmoothQuant redistributes quantization difficulty between weights and activations through an equivalent transformation before low-precision execution. These are methods, not hardware dtypes or file formats.

QLoRA trains low-rank adapters through a frozen quantized base model. It reduces fine-tuning memory requirements, but its storage representation and compute dtype need not match. It should not be described as training every original weight with ordinary four-bit arithmetic.

Why smaller can be slower. Unpacking, dequantization, incompatible shapes, metadata traffic, or a missing optimized kernel can offset savings. A storage format supported by a loader may not have a fast kernel on the target device. Validate model quality, memory, prefill time, decode behavior, and concurrent throughput for the actual engine and hardware.

Primary references: [\[28\]](#) · [\[29\]](#) · [\[30\]](#) · [\[31\]](#) · [\[32\]](#) · [\[33\]](#) · [\[34\]](#)

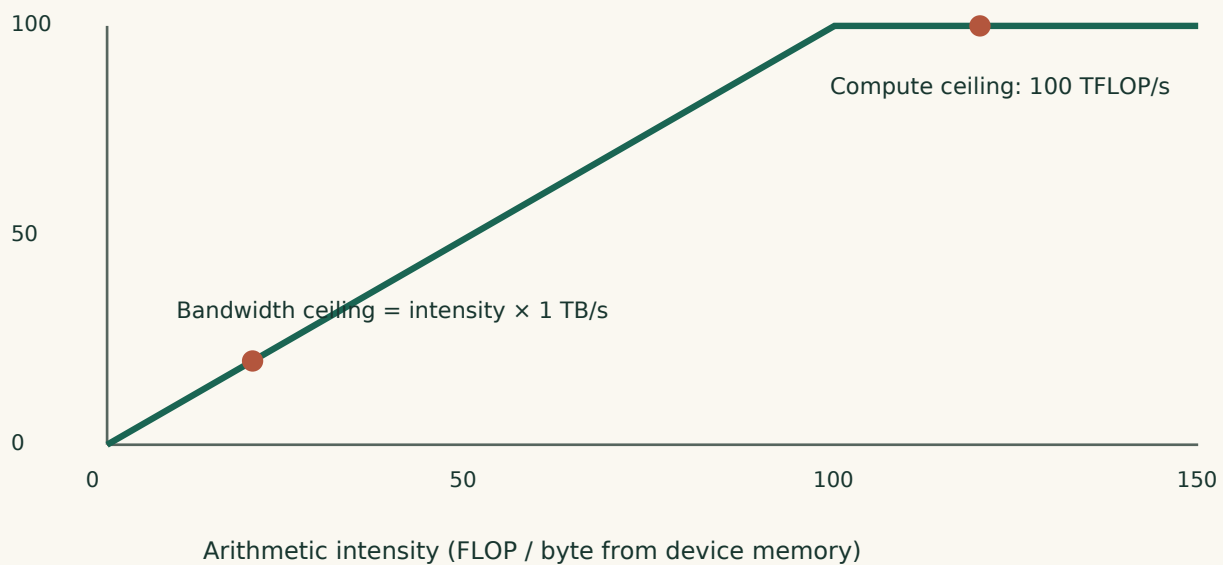
12 / INTERMEDIATE

Find the limiting resource

Before asking how to make it faster, name the resource it is waiting for. Then choose a measurement that could prove that guess wrong.

Continue online: [Roofline Model](#) · [Latency Metrics: TTFT, TPOT and Goodput](#)

Attainable performance (TFLOP/s)



Illustrative roofline with assumed 100 TFLOP/s compute and 1 TB/s bandwidth. Both axes are linear; this is a bound, not a device benchmark.

Three constraints recur: insufficient compute rate, insufficient data-transfer rate, and dependencies that keep resources waiting. Compute-bound means arithmetic throughput is the dominant constraint for the operation being examined. Memory-bandwidth-bound means moving required bytes through a particular memory interface is dominant. Communication-bound often refers to inter-device transfers. Kernel-launch overhead, CPU scheduling, synchronization, and queueing can dominate even when none of those resources reaches its peak.

The roofline model relates attainable performance to arithmetic intensity: work performed per byte moved through the memory level under study. A first-order bound is the smaller of compute capacity and bandwidth multiplied by arithmetic intensity. Tiling, fusion, or batching can raise intensity by reusing data. A kernel can be limited by HBM while another is limited by local SRAM or a vector pipeline; there is no single roofline for every operation.

A bandwidth lower bound

Suppose one hypothetical decode round must stream 16 GB of weights from device memory at a sustained 1 TB/s. The weight-transfer time alone is 16 milliseconds. Ignoring everything else, that would cap a single sequence at 62.5 tokens per second if every round produced one token and incurred that traffic. This is an optimistic bound, not a prediction: attention reads, arithmetic, communication, sampling, and scheduling can add time.

If several sequences share a round, weight data can be reused across them. Aggregate output throughput may rise without improving an individual sequence's token interval. At large batches, arithmetic or cache traffic can become dominant. Prefill often presents larger matrix operations and decode at small batch often stresses memory bandwidth, but long contexts, short prompts, MoE, and implementation details can change that balance.

Metric	Meaning	Common interpretation mistake
TTFT	Time to first token, at a declared measurement boundary	Treating it as prefill kernel time only
ITL	Interval between successive output tokens or emissions	Ignoring streaming chunks and stalls
TPOT	Often average post-first-token time per output token	Assuming every benchmark uses the same formula
End-to-end latency	Request start to completion	Comparing requests with different output lengths
Output tokens per second	Generated tokens divided by elapsed time	Mixing per-user and aggregate rates
Input tokens per second	Prompt-processing throughput	Comparing it directly with decode throughput
Requests per second	Completed requests per time interval	Ignoring varying input and output sizes
Goodput	Useful completions or tokens meeting stated requirements	Counting late, failed, or invalid work as success
p50 p95 p99	Percentiles of an explicitly chosen measurement population	Looking only at averages during overload

Serving metrics commonly include TTFT, inter-token intervals, end-to-end time, queue time, cache occupancy, and token counts. Metric definitions and names vary across versions and instruments. Always state whether timing is measured at the client, gateway, or engine.

For N output tokens and an average interval measured over the N minus one post-first-token gaps, a useful identity is total latency $\approx$ TTFT + $(N - 1) \times$ average gap. Transport buffering can change

observed intervals. When only one token is generated, that post-first-token average is undefined rather than a useful zero-latency statistic.

Queueing matters. If arrival work consistently exceeds service capacity, the queue grows. Even below average capacity, bursts and variable request sizes can create long tails. Under stable conditions, Little's law relates average in-system concurrency to arrival rate multiplied by average time in the system. At five completed requests per second and four seconds average residence time, average concurrency is twenty. This does not guarantee enough capacity for bursts or long-context requests.

Peak FLOP/s, occupancy, accelerator utilization, model FLOPs utilization, and service goodput answer different questions. Benchmark under the required quality and latency targets, and include queueing, representative lengths, concurrency, cache-hit assumptions, and failure behavior. MLPerf's defined workloads and scenarios illustrate why benchmark conditions are part of the result.

For a serving cost comparison, divide total system cost over an interval by useful completed work over the same interval. Cost per million output tokens is that ratio multiplied by one million, with input processing, idle capacity, host resources, and networking included in the stated cost boundary. If input and output work are combined, disclose their mix. A lower device-hour price can still produce more expensive useful output when the deployment has poor utilization or misses deadlines.

Work the roofline before buying more FLOP/s

For a hypothetical kernel, suppose the relevant memory path sustains 1 TB/s and the arithmetic engine can sustain 100 TFLOP/s. The intersection is 100 FLOP/byte. At 20 FLOP/byte, the bandwidth ceiling is 20 TFLOP/s; increasing compute capacity alone leaves that ceiling unchanged. If reuse raises intensity to 120 FLOP/byte, the compute ceiling becomes limiting at 100 TFLOP/s. These are assumed rates, not a GPU specification. Measure traffic at the same memory level used in the calculation.

Define useful output before calculating cost

Suppose a complete serving system costs \$6 per hour and finishes 300,000 output tokens within the required quality and latency criteria in that hour. The scoped cost is \$20 per million useful output tokens. If only 200,000 qualify, it becomes \$30. These are illustrative inputs. State whether input processing, idle time, retries, host costs and network costs are included; a cheaper GPU hour is not automatically a cheaper service.

Primary references: [\[51\]](#) · [\[53\]](#) · [\[56\]](#) · [\[57\]](#)

13 / INTERMEDIATE

Serve more work without losing responsiveness

The aim is useful throughput within the latency target. A larger batch or a clever cache is valuable only if it improves that service.

Continue online: [Continuous Batching](#) · [PagedAttention](#) · [Speculative Decoding](#)

	Round 1	Round 2	Round 3	Round 4
Sequence A	A	A	done	—
Sequence B	B	B	B	B
Sequence C	queued	prefill C	C	C

C is admitted after preparation; B continues across the change.

Conceptual scheduler rounds, not equal durations. C's prefill must finish before C can decode. New arrivals do not acquire KV merely because A leaves.

Static batching executes a prepared group together. Dynamic batching waits briefly to combine independent incoming requests. Continuous batching can change the active set between generation iterations, admitting new sequences as others finish. This avoids requiring every request in an initial group to remain until the longest sequence completes. Iteration-level scheduling is a foundational idea in Orca.

Dynamic batching trades a bounded wait for a potentially more efficient batch. It is especially straightforward for compatible stateless forward passes. Stateful generation requires the engine to manage each sequence's cache, position, stopping criteria, and attention metadata. A generic model server's dynamic batcher is therefore not automatically an LLM continuous-batching scheduler.

PagedAttention organizes logical KV blocks through mappings to physical memory blocks. Requests can grow without requiring one maximum-sized contiguous allocation per sequence, and compatible blocks can be shared. Paging reduces allocation waste; it does not remove the bytes that must store live K and V. Block rounding, metadata, and runtime reserve remain.

FlashAttention changes how attention is computed, tiling work to reduce transfers between HBM and on-chip storage and avoiding materialization of the full attention-score matrix in HBM. It computes exact attention in the algorithmic sense, subject to floating-point numerical differences. It does not make full attention linear-time in sequence length. FlashAttention and paged KV allocation solve different problems and can coexist.

Chunked prefill splits a long prompt's preparation into scheduling chunks. This can prevent a large prefill from monopolizing execution while existing users wait for decode. Chunk size trades prompt-completion speed against token responsiveness and batching opportunities. It does not change the prompt's semantics when implemented correctly.

Speculative decoding uses a draft process to propose several tokens and a target model to verify candidates together. Correct acceptance and correction rules can preserve the target sampling distribution. Speed depends on acceptance rate, draft cost, verification efficiency, and available resources. A poorly matched draft can add work. It is neither ordinary batching nor a guarantee of faster generation.

Kernel fusion combines operations to reduce launches and intermediate-memory traffic. CUDA Graphs capture compatible execution work for lower-overhead replay. Asynchronous execution uses streams and events to express work and dependencies. Overlap occurs only when dependencies and hardware resources permit it; placing two operations on different streams does not guarantee they execute concurrently.

Disaggregated prefill and decode assigns phases to different worker pools. A prefill worker prepares cache state and transfers it to a decode worker. Independent pools can use different scaling or parallelism choices, but cache transfer, routing, load balancing, and capacity fragmentation become new costs. The architecture is attractive only when those costs are justified by the workload.

Treat each optimization as a hypothesis: identify the constrained resource, state what work or bytes will change, verify output quality, then measure the desired service metric. Enabling every optimization at once makes regressions harder to explain.

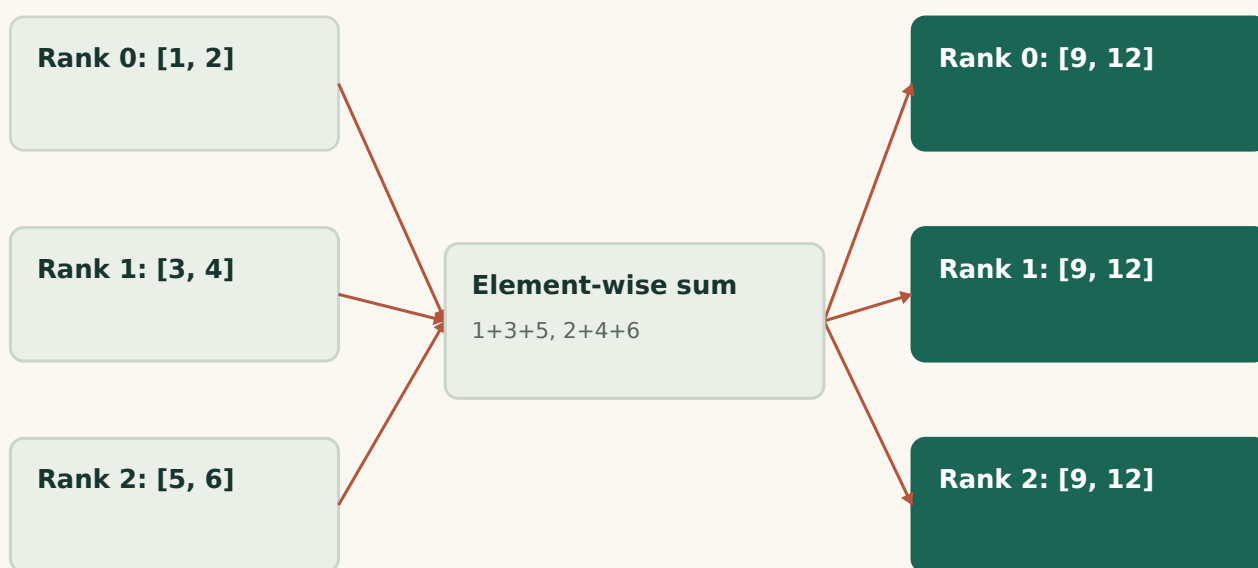
Primary references: [\[3\]](#) · [\[21\]](#) · [\[22\]](#) · [\[23\]](#) · [\[24\]](#) · [\[25\]](#) · [\[26\]](#) · [\[27\]](#)

14 / ADVANCED

Across devices: links and collectives

The same collective can cross a fast local fabric or a congested network.
Separate what the operation means from how its bytes travel.

Continue online: [NVLink, NVSwitch and PCIe](#) · [Training at Scale](#)



Sum all-reduce on three two-element vectors. Each rank receives the same element-wise sum. These arrows describe the semantic operation, not a physical network algorithm.

Scale up usually means tightly coupling accelerators in a fast local fabric. Scale out usually means connecting servers or larger groups through a network. These are architectural roles rather than fixed chassis boundaries: some accelerator fabrics span racks. The actual topology and communication path matter more than the label.

Term	What it is	How it relates to other components
PCIe	Host and device I/O interconnect	Connects CPUs, GPUs, NICs, switches, and SSDs
NVLink	NVIDIA accelerator interconnect technology	Supports high-speed peer communication on compatible systems
NVSwitch	Switching for an NVLink fabric	Enables communication among multiple connected accelerators

Term	What it is	How it relates to other components
Ethernet	A networking technology family	Carries ordinary IP traffic and can support RDMA through RoCE
InfiniBand	A switched interconnect architecture	Provides messaging and RDMA through adapters and switches
RDMA	Remote direct memory access mechanism	Accesses registered remote buffers with low data-path CPU overhead
RoCE	RDMA over Converged Ethernet	Brings RDMA transport to supported Ethernet infrastructure
GPUDirect RDMA	Direct peer access to supported GPU memory	Allows a NIC data path without mandatory host bounce copies
NCCL	A GPU communication library	Implements collectives using suitable available transports
ICI	TPU inter-chip interconnect	Connects chips within a TPU slice

NVLink describes links and fabric capabilities; NVSwitch supplies switching within compatible fabrics. Their presence does not combine all GPU memory into one uniformly fast allocation for arbitrary software. Applications still need placement and communication logic, and remote access may have different costs.

InfiniBand defines adapters, switches, addressing, transport, and management for a fabric. A subnet manager discovers and configures an InfiniBand subnet and its routes. RoCE uses Ethernet with RDMA semantics; routable RoCEv2 uses UDP/IP encapsulation. RDMA is thus not a competing cable type beside InfiniBand and Ethernet.

In RDMA, memory regions and access permissions are established before transfers. The CPU and operating system still help configure resources, post work, and manage completions. The benefit is reducing involvement and copies in the data path, not removing the CPU from the system. GPUDirect RDMA extends suitable peer paths to GPU memory and requires compatible topology, drivers, and devices.

Collective operations

Distributed models often need coordinated communication among ranks, the participating processes or workers. A rank is an identity in a communication group, not necessarily a physical machine. A process group can span devices inside one server or across many servers.

Operation	Result	Typical use
All-reduce	Combine corresponding values and give the result to every rank	Synchronize gradients or sum partial results
All-gather	Collect shards and give the combined data to every rank	Materialize sharded parameters or activations
Reduce-scatter	Combine values and retain one result shard per rank	Sharded gradient updates
All-to-all	Each rank sends different portions to other ranks	Dispatch and return tokens across MoE experts
Broadcast	Copy one rank's data to a group	Distribute shared state
Send and receive	Transfer data between selected peers	Pipeline-stage boundaries

The first five definitions follow NCCL's collective semantics. Libraries choose algorithms such as rings or trees according to message size, topology, and implementation. Participating ranks must issue compatible collective work; mismatches can hang a job.

AMD's RCCL provides corresponding collective primitives on supported AMD systems, using the available local interconnects and network transports. The semantic operation, such as all-reduce, is transferable across ecosystems; the implementation and tuning are platform-specific.

Advanced depth. A leaf-spine network connects endpoint-facing switches to a spine tier. Oversubscription means shared uplink capacity is smaller than potential downstream demand. Bisection bandwidth concerns traffic across a division of the topology. ECMP distributes flows across equal-cost paths; congestion can still create hotspots. ECN signals congestion, while PFC pauses selected traffic priorities. RoCE deployments need deliberate congestion and loss behavior; PFC is common but is not a universal requirement for every implementation. A rail-oriented layout can keep some same-position GPU traffic on one network plane, but does not make every exchange one hop. Validate the actual paths and job traffic.

Ports, cables, optics, negotiated link rates, error counters, retransmissions, and topology must all be healthy. A nominal 400 Gb/s port carries at most 50 GB/s before protocol and other overhead, not 400 GB/s. Aggregate bidirectional or multiport figures must not be compared with one-direction payload rates.

Count a ring all-reduce

In a simple bandwidth-oriented ring, reduce-scatter and all-gather each take $N - 1$ rounds. For a payload of S bytes per rank, each rank sends approximately $2 \times (N - 1) / N \times S$ bytes in total. With 8 ranks and a 1 GB tensor, this is 1.75 GB sent per rank. At an assumed 50 GB/s effective rate, payload time alone is 35 ms. This omits startup, synchronization, contention and computation; the library may choose another algorithm. Do not confuse per-rank traffic with useful application bytes.

Primary references: [\[35\]](#) · [\[36\]](#) · [\[37\]](#) · [\[38\]](#) · [\[39\]](#) · [\[40\]](#) · [\[67\]](#)

15 / ADVANCED

Replicate, shard, or do both

A replica serves independent work. A shard cooperates on the same work. Every deployment layout spends memory or communication to buy something else.

Continue online: [Training at Scale](#) · [GPU and TPU reference](#)

FOUR REPLICAS

Request 1 → full model

Request 2 → full model

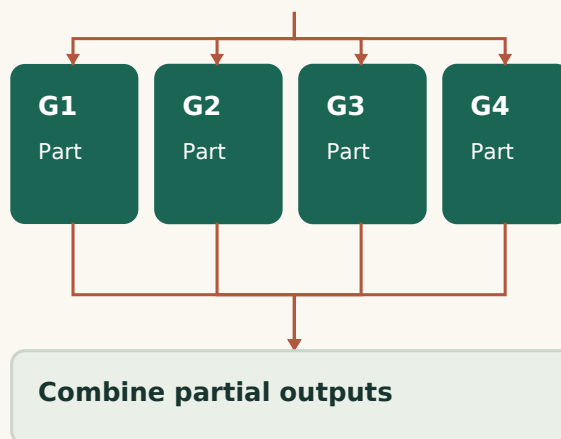
Request 3 → full model

Request 4 → full model

Independent sequences

ONE TP GROUP

One layer split across four GPUs



Two different uses of four devices. Replicas take independent requests; tensor-parallel shards cooperate on a request and communicate inside layers.

Replicate when independent workers can each hold and serve the model efficiently. Shard when memory, latency, or throughput objectives require cooperation across devices. A sharded replica is a group of devices that jointly implements one logical worker; adding another such group creates replica-level capacity.

Strategy	What is divided	Main communication or cost
Replica or data parallel inference	Independent requests across model copies	Duplicated weights; routing and capacity balance
Tensor parallelism	A layer's tensors and arithmetic across devices	Frequent exchanges of partial layer results
Pipeline parallelism	Layer groups across devices	Boundary activations and pipeline bubbles
Expert parallelism	Expert modules across devices	Token dispatch and return, often all-to-all

Strategy	What is divided	Main communication or cost
Context parallelism	Sequence positions or attention work across devices	Exchanges of attention-related state or partial results
Hybrid parallelism	Several dimensions together	More placement and communication coordination

Tensor parallelism, or TP, splits work inside layers. A simple linear layer may partition output features or input features; the choice determines whether outputs are concatenated or partial sums combined. This is why tensor parallelism usually benefits from low-latency, high-bandwidth peer links. Megatron-LM illustrates this style of intra-layer partitioning.

Pipeline parallelism, or PP, assigns successive layer groups to stages. A request's activations move from one stage to the next. Multiple microbatches can keep stages busy, but pipeline fill and drain create idle periods called bubbles. A single autoregressive sequence still depends on its previous output; putting successive layers on more devices does not remove that dependency.

Expert parallelism, or EP, places different experts on different devices. The router determines destinations, devices process their assigned token batches, and outputs return for combination. Uneven routing can produce a straggler even when aggregate arithmetic capacity looks sufficient. Total expert residency and token-exchange traffic must be budgeted separately from active parameter count.

Context parallelism divides sequence-related work. The exact collectives and cache ownership depend on the attention algorithm. Sequence parallelism is an overloaded term: some training systems use it for sharding activations of selected operations alongside tensor parallelism, rather than for distributing an entire attention context. State the implementation and dimension being split.

For ordinary attention, partitioning KV heads can reduce per-device cache, but head counts and kernel support constrain useful partitioning. Pipeline stages retain state for their own layers; replicas maintain independent request state unless a separate cache-sharing mechanism exists. A plan that divides weights evenly can still have imbalanced cache or workspace demand.

A placement rule to test. Keep frequent, latency-sensitive communication on the fastest available fabric, then use wider replication or coarser partitions where appropriate. This is a starting hypothesis, not a universal topology prescription. Actual choices depend on layer size, batch, context, network, and runtime support.

Scaling efficiency is observed speedup divided by the increase in resources. If four devices make a job only twice as fast, efficiency relative to the single-device baseline is 50 percent. That can still be worthwhile when one device cannot fit the required workload. Compare both per-request latency and goodput per unit of cost, including the complete replica group.

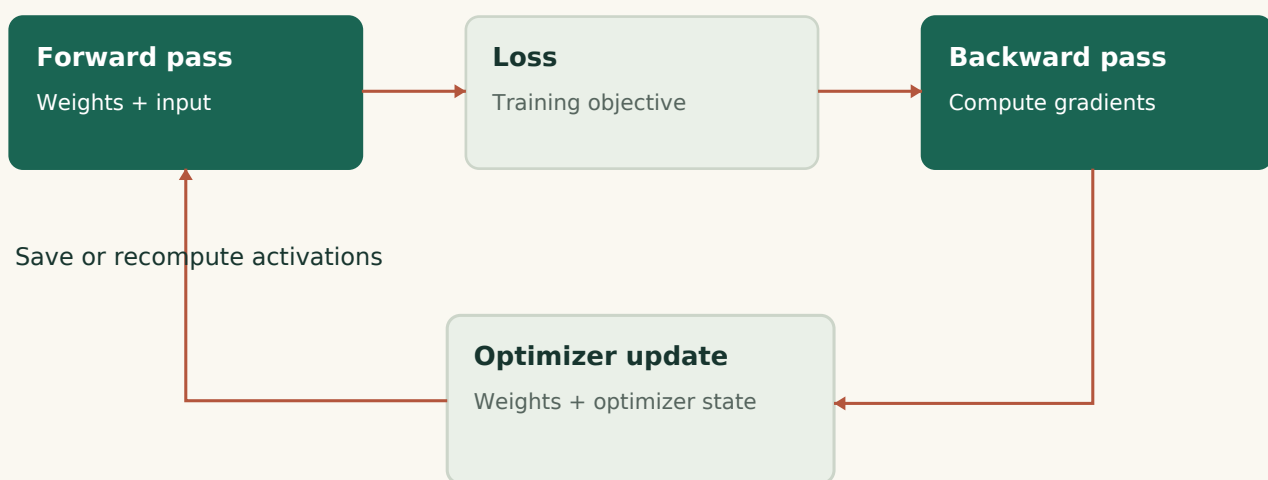
Primary references: [\[40\]](#) · [\[43\]](#) · [\[44\]](#) · [\[46\]](#) · [\[47\]](#)

16 / ADVANCED

Training changes the memory budget

An inference worker applies weights. A training worker must also remember how to change them. That extra state often decides the layout.

Continue online: [Training at Scale](#) · [Training FLOPs: 6ND](#)



Durable checkpoint: recover weights, optimizer and the required run state.

Training adds gradients and optimizer state to the forward computation. Activation recomputation and a durable recovery checkpoint solve different memory problems.

Training adds a feedback loop to inference's forward computation. A batch enters the model, a loss function measures an objective, backpropagation computes gradients, and an optimizer updates trainable parameters. A learning rate controls update magnitude. A training step usually means an optimizer update, while an epoch is one traversal of a defined dataset; streaming or token-budget training may not use epochs as its main unit.

Pretraining learns broad patterns from a large corpus. Supervised fine-tuning adapts a model with selected input-output examples. Preference optimization uses information about preferred outputs; reinforcement-learning pipelines may additionally generate rollouts and compute rewards. Their infrastructure can alternate between training and substantial inference work. Generation used during training is still inference computation, even though it supports a learning pipeline.

Why training uses more memory

Inference generally needs weights, live activations, and model-specific retained state such as KV. Full training additionally needs gradients, optimizer state, and saved forward intermediates. For an illustrative mixed-precision Adam implementation, count two bytes for model weights, two for gradients, four for an FP32 master copy, and eight for two FP32 optimizer moments: sixteen bytes per trainable parameter before activations, workspaces, and communication buffers.

For one billion trainable parameters, that illustrative persistent state is 16 GB, or about 14.90 GiB. This is not a universal multiplier. Some implementations store gradients differently, use no separate master copy, quantize optimizer state, shard state, or use a different optimizer. Measure the actual configuration.

Activation checkpointing, also called recomputation or rematerialization, saves fewer forward intermediates and recomputes selected operations during backward. This trades extra computation for lower activation memory. It differs from saving a durable checkpoint file for failure recovery.

Distributed training

In distributed data parallel training, workers generally keep model replicas, process different data, and synchronize gradients so their updates remain consistent. Gradient buckets can overlap communication with backward computation. Distributed data parallelism increases training capacity or throughput; it does not inherently shard each worker's model weights.

FSDP, Fully Sharded Data Parallel, shards model state, gathers required parameters for computation, and distributes reduced gradient shards. ZeRO partitions redundant training state in stages: stage 1 partitions optimizer state, stage 2 also partitions gradients, and stage 3 also partitions parameters. These reduce redundant storage while adding gather, scatter, and coordination costs.

Gradient accumulation processes several microbatches before an optimizer step. In a conventional replicated data-parallel setup, effective global batch equals microbatch per worker multiplied by accumulation steps multiplied by data-parallel workers. Tensor- or pipeline-parallel ranks are not additional independent data-parallel workers. Accumulation can lower activation memory per microbatch, but it changes update scheduling and must preserve intended loss normalization.

Parameter efficient adaptation

LoRA, low-rank adaptation, freezes the base weights and trains small additional low-rank matrices. It greatly reduces the number of trainable parameters and associated optimizer state; it does not eliminate the base model's forward work. Adapters can sometimes be merged into compatible weights or served separately. Multi-adapter serving introduces routing, batching, and cache-compatibility questions.

QLoRA further reduces base-weight storage during adapter training. Neither technique makes the training and inference memory budgets identical. Activations, adapter gradients, and implementation overhead still matter.

The input pipeline must deliver decoded, tokenized, shuffled or packed batches fast enough. Checkpoint files should include the state needed for the intended resume guarantee: weights, optimizer, scheduler, step, and relevant random-number and data-loader state. Distributed saves require a consistent snapshot and a completion protocol. Test restore behavior rather than assuming that a successfully created file is a usable checkpoint.

Use 6PD with its assumptions

For a conventional dense Transformer, a common first estimate of training work is $6 \times P \times D$ FLOPs, where P is the parameter count used by the approximation and D is the number of training tokens. It approximates forward plus backward matrix work and omits costs that can matter, including attention at long context, recomputation and non-model overhead. MoE needs an appropriate active-work count; its resident weights still determine storage. A nominal 1B-parameter, 10B-token exercise gives 6×10^{19} FLOPs before those corrections. Convert work to time using a measured useful rate for a comparable workload, not an unrelated peak number.

Checkpoint recovery is a design, not a file

Separate dataset reads, checkpoint writes, model distribution and collective traffic when budgeting shared links. For an assumed 1 TB checkpoint and 20 GB/s sustained payload rate, transfer alone takes at least 50 seconds. Commit metadata only after all required shards are durable, retain a known-good version, and test restoration on replacement workers. Define how much training progress can be lost and how long recovery can take.

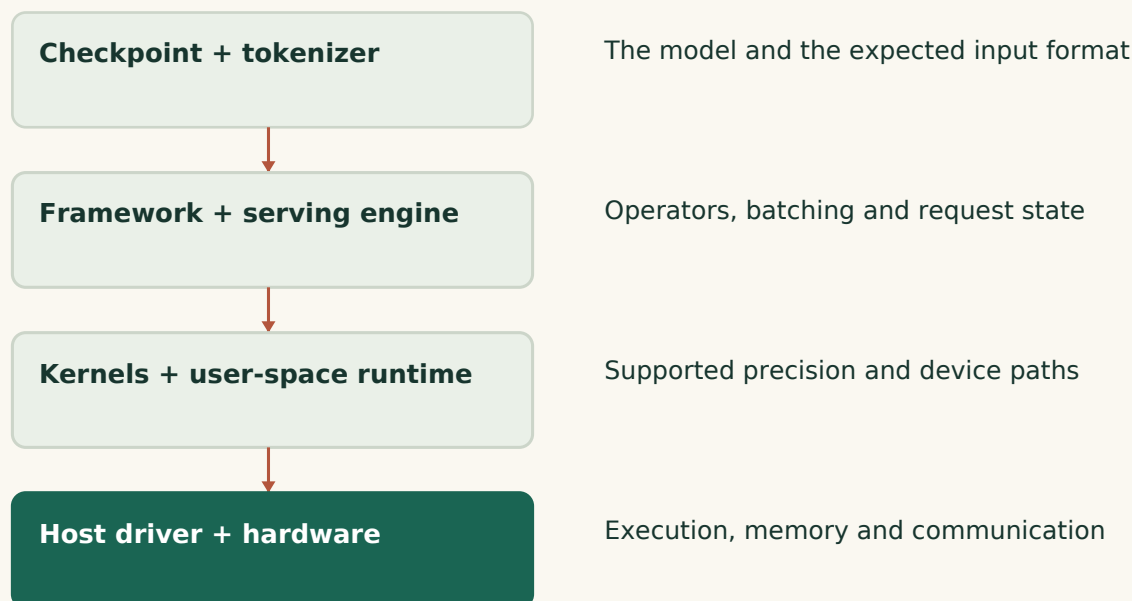
Primary references: [\[41\]](#) · [\[42\]](#) · [\[43\]](#) · [\[44\]](#) · [\[45\]](#) · [\[33\]](#) · [\[61\]](#) · [\[68\]](#) · [\[58\]](#)

17 / ADVANCED

From checkpoint to running service

Downloading weights is one step. The tokenizer, runtime, kernels, drivers and service all have to agree about what those weights mean.

Continue online: [Serving Engines: vLLM, SGLang and TensorRT-LLM](#) · [Open model reference](#)



Compatibility runs through the stack. A container can package user-space components; it still depends on supported host drivers and hardware.

The model description does not directly drive transistors. A framework constructs tensor operations; an execution backend or compiler maps them to kernels; libraries implement common primitives; a runtime and driver coordinate device execution. A serving engine adds request scheduling, generation state, and memory management. Deployment software places and operates the worker processes.

Layer	Examples or concepts	Responsibility
Application	Chat, search, coding, image generation	Defines useful output and user interaction
Gateway and router	Authentication, admission, routing, streaming	Controls access and dispatches requests
Serving engine	vLLM and other model-serving runtimes	Schedules inference and manages generation state
Framework	PyTorch, JAX, TensorFlow	Represents model operations and training logic

Layer	Examples or concepts	Responsibility
Compiler and kernel system	XLA, Triton language, graph compilers	Lowers and optimizes computation
Libraries and collectives	Matrix libraries, attention kernels, NCCL	Implements optimized numerical and communication work
Runtime and driver	Device allocation, launch, synchronization	Connects user software to hardware
Orchestration	Kubernetes, job schedulers, process supervisors	Places, restarts, and manages workers

The layers can be fused or bypassed. An engine may call specialized kernels directly rather than execute every operation through a general framework. XLA illustrates compiler optimization across devices; Triton is a language and compiler for writing efficient parallel kernels.

Two different Tritons. The Triton programming language is a kernel-development system. NVIDIA Triton Inference Server is serving software with backends, model management, and scheduling. They can participate in the same deployment, but installing one does not provide the other.

Model artefacts. A deployable model usually requires weights, architecture configuration, tokenizer and vocabulary, special-token definitions, and generation or preprocessing settings. Safetensors is a tensor-storage format designed to avoid arbitrary code execution during tensor deserialization. It does not make arbitrary companion Python code safe. GGUF is a container format for tensors and metadata used in the ggml ecosystem; it can contain different quantization types and is not itself a quantization algorithm.

Compatibility is a chain. The checkpoint architecture must match the implementation; quantization must match kernels; kernels must support the device; and user-space libraries must work with the driver. Containers package much of the software but still depend on the host driver and hardware. Record exact model revisions, image digests, runtime versions, and relevant configuration for reproducibility.

Scheduling exists at two scales. A cluster scheduler places a worker or distributed job on machines. An inference scheduler decides which requests or tokens execute in the next iteration. Kubernetes exposes accelerators through device-management mechanisms, including vendor device plugins and resource requests; that alone does not provide model-aware batching or KV management.

Kubernetes is useful for an operated service or shared platform but is not a prerequisite for learning inference. A single supervised process is enough to explore a small model.

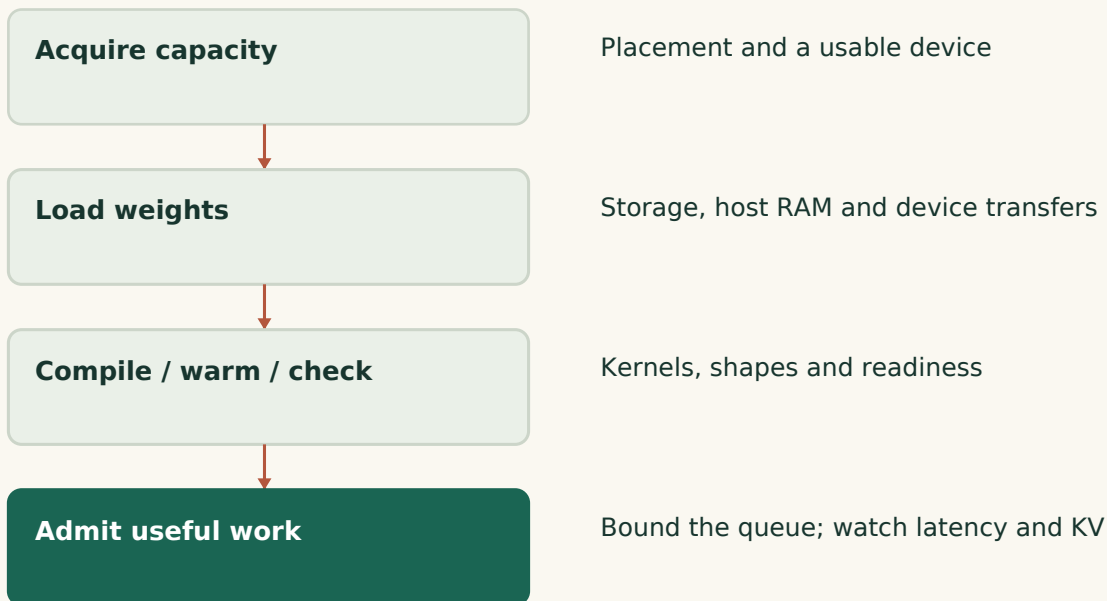
Primary references: [\[11\]](#) · [\[12\]](#) · [\[49\]](#) · [\[54\]](#) · [\[55\]](#) · [\[59\]](#) · [\[60\]](#)

18 / ADVANCED

Operate the system you designed

A deployment becomes a service when it handles bursts, failures and updates predictably. The physical rack is part of that promise.

Continue online: [Fleet and Reliability](#) · [Power and Datacenter Constraints](#)



Cold start precedes useful serving. Scale from queue and capacity evidence early enough to cover startup; readiness must reflect the intended workload.

A production service must have explicit objectives for output quality, latency, availability, and capacity. An SLI is a measured indicator; an SLO is the target for that indicator. For example, an objective could require a defined fraction of eligible requests to produce their first token within a chosen deadline. The request population and measurement boundary belong in the definition.

Admission and scaling

Admission control bounds work before expensive execution. Limit input tokens, output tokens, concurrent sequences, and queued work. Cancellation should release computation and cache state promptly. Separate interactive requests from large offline jobs when their deadlines conflict. These are design recommendations: tune the boundaries to the measured workload and available capacity.

Autoscaling must account for cold starts: obtaining hardware, downloading weights, loading memory, compiling, warming kernels, and becoming ready. A busy GPU percentage alone may not predict queue growth or KV exhaustion. Queue age, token backlog, cache pressure, latency, and per-replica capacity often give a better combined signal. A warm spare can reduce recovery time but carries an idle-capacity cost.

A load balancer must route a sequence's work to the state that serves it, or arrange explicit state transfer. Losing a worker can destroy live KV. Recreating state from retained tokens may be possible, but costs prefill and may not reproduce an interrupted streamed response exactly. Multi-device replicas create a shared failure domain: one failed rank can disrupt the whole group.

Sharing accelerators

MIG, Multi-Instance GPU, partitions supported NVIDIA GPUs into hardware-defined instances with allocated compute and memory resources. Time slicing shares access over time and does not provide the same memory or fault isolation. MPS, Multi-Process Service, coordinates CUDA work from multiple processes; it has a different isolation and resource model from MIG. None of these mechanisms multiplies physical memory capacity.

Choose sharing based on the workload's actual memory peak, interference tolerance, and isolation requirements. A logical resource fraction or virtual replica count is not a promise of a proportional latency improvement or memory entitlement.

Observe the whole path

Observe	What it can explain
Request rate, token lengths, queue age, cancellations	Offered load and time spent waiting
TTFT, token intervals, completion time, errors	User-visible performance and failures
Live KV blocks, reserved memory, allocation failures	Capacity pressure and memory-management behavior
Device activity, memory throughput, clocks, power	Whether compute, memory, or throttling constrains work
NIC traffic, link errors, collective timing	Network bottlenecks or unhealthy communication
CPU profiles, run queues, storage throughput	Preprocessing, launch, or input-pipeline starvation

Hardware tools such as DCGM expose health, telemetry, and diagnostic information on supported NVIDIA systems. A systems trace shows CPU work, GPU kernels, copies, synchronization, and overlap over time. A kernel profiler examines a specific kernel's execution and memory behavior. Use the least invasive measurement that resolves the current question; profiling itself can perturb timing.

Reliability and access

Treat model files and adapters as versioned software artefacts. Restrict access to weights and request data, validate allowed model code, protect management interfaces, and define retention for logs and caches. Avoid logging prompt contents by default when aggregate metrics suffice. Prefix-cache sharing across tenants requires deliberate isolation and authorization because cached state is derived from inputs.

Use readiness checks that represent the worker's ability to accept the intended load, graceful draining for updates, and canaries for changes to kernels or precision. Test quality as well as availability: a service can return successful HTTP responses while a tokenizer mismatch or numerical regression damages every answer. Keep rollback-compatible model and runtime artefacts.

Power cooling and storage

The GPU is only one part of server power. CPUs, memory, switches, fans, pumps, and conversion losses also consume energy. TDP or board-power ratings are not identical to measured wall power. Rack capacity must respect both electrical and cooling constraints, redundancy policy, and the consequences of losing a feed.

Air cooling removes heat through heatsinks and airflow. Direct-to-chip liquid cooling uses cold plates; a CDU, coolant distribution unit, manages an interface between cooling loops. Immersion cooling places compatible components in a dielectric fluid. These are system-level designs with different installation and maintenance requirements; no one method follows automatically from the word GPU.

PUE, power usage effectiveness, is facility energy divided by IT equipment energy over a defined boundary and interval. It measures facility overhead, not model-serving efficiency. A low-PUE facility can still run an inefficient inference service. At the workload level, define joules per useful output and account for measurement scope; energy, latency, and quality must be compared together.

Storage needs change across phases. Model startup reads large files; training pipelines stream or randomly access datasets; checkpointing writes consistent state. Metadata operations, small files, network contention, and local-cache misses can matter as much as sequential bandwidth. A recoverable system needs durable copies outside a worker's failure domain.

Work backward from the rack

Assume a hypothetical rack has a 60 kW usable electrical allocation after its redundancy policy, but its cooling design supports 48 kW. Reserve 8 kW of that limiting 48 kW for networking and other equipment. At a measured 10 kW per compute node for the target workload, four nodes fit: $\text{floor}((48 - 8) / 10) = 4$. This is an example of constraint arithmetic, not a rack recommendation. Physical space, transients, feed failure and coolant conditions may lower the limit further.

Primary references: [\[49\]](#) · [\[50\]](#) · [\[51\]](#) · [\[52\]](#) · [\[57\]](#) · [\[58\]](#) · [\[62\]](#) · [\[63\]](#)

19 / PRACTICE

Diagnose the symptom before tuning

“The GPU is slow” is an observation, not a diagnosis. Use the tables to turn it into a testable question.

Continue online: [Fleet and Reliability](#) · [Inference Autoscaling and Cold Starts](#)

Start with the observation, identify a candidate cause, and choose a measurement that can disprove it. The following table is a diagnostic guide, not a claim that each symptom has only one cause.

Symptom	Candidate causes	First useful check
Model fails while loading	Weight footprint, conversion peak, duplicate copies	Compare on-disk, temporary, and resident tensor sizes
OOM appears only under traffic	KV growth, workspaces, fragmentation, multiple versions	Plot live sequences and retained tokens against memory
Long wait before the first token	Queueing, tokenization, long prefill, cold start	Split client latency into queue and execution intervals
First token is fast but generation is slow	Weight or KV traffic, communication, decode scheduling	Profile representative decode rounds
Only long prompts disrupt other users	Prefill monopolizes a schedule or memory budget	Compare decode intervals while prefills are admitted
Adding GPUs worsens latency	Collective overhead or poor topology	Measure peer transfer and collectives for relevant sizes
High utilization with low goodput	Memory-bound kernels, low clocks, inefficient shapes	Examine useful throughput and the constrained resource
GPU intermittently idle	CPU launch gaps, input starvation, synchronization	Inspect a time-aligned CPU and GPU trace
MoE replicas show uneven load	Routing skew, small expert batches, network contention	Inspect tokens per expert and all-to-all timing
Quantized model fits but is slower	Unpacking, unsupported fast path, wrong batch regime	Compare kernel selection and per-phase time
Distributed job hangs	Rank mismatch, failed worker, incompatible collectives	Find the first failing rank and last completed collective

Symptom	Candidate causes	First useful check
Memory stays allocated after requests finish	Cache pool or allocator reserve	Compare reserved bytes with actively used blocks
Low throughput after a long run	Thermal or power limits, hardware errors, contention	Correlate clocks, power, temperatures, and error counters
Good engine benchmark but poor application latency	Retrieval, network, gateway buffering, client delays	Trace the complete request boundary

Common misconceptions to retire. More HBM capacity does not itself guarantee more bandwidth. NVLink is not InfiniBand. RDMA does not mean zero CPU involvement. Four-bit weights do not imply four-bit KV. FlashAttention is not a replacement for the KV cache. A model's active MoE parameter count is not its total weight-memory requirement. A loaded model is not evidence that the target concurrency fits.

Other recurring mistakes are treating every GPU memory as HBM, multiplying clocks and core counts into a serving estimate, assuming high occupancy is always better, comparing unqualified tokens per second, and expecting two accelerators to behave as one double-sized GPU without software support. Correct each mistake by naming the exact tensor, resource, or measurement boundary involved.

Primary references: [\[25\]](#) · [\[52\]](#) · [\[56\]](#)

20 / PRACTICE

Five deployment calculations

These five hypothetical designs connect the chapters. Recalculate one assumption at a time; none of the results is a measured benchmark.

Continue online: [Interactive calculators](#) · [GPU and TPU reference](#) · [Open model reference](#)

20 sequences: 6,144 shared + 2,048 unique tokens each

Separate KV copies



20 GiB

Perfect compatible sharing



5.75 GiB

128 KiB per unique retained token. Excludes block rounding and metadata.

The same hypothetical 20 requests can retain fewer unique KV positions when their exact compatible prefix is shared. Decode must still attend to that prefix.

Example A A small interactive assistant

Assume the 8-billion-parameter model and cache geometry from [Chapter 10](#), a 24 GiB device, and a 3 GiB non-KV reserve. At sixteen-bit weights, approximately 6.10 GiB remains for raw KV. If one sequence can retain 8,192 tokens, six concurrent sequences are the arithmetic ceiling under the simplified budget.

Now suppose prompts are short but outputs can be long. Reducing TTFT is helpful but does not establish good decode responsiveness. Measure token intervals while increasing concurrency, and admit fewer than the arithmetic ceiling if the latency objective fails. If memory is the constraint, compare an evaluated weight-quantized variant, a shorter context allowance, cache quantization, or another replica. Each option changes a different part of the budget.

The deployment plan should state the actual context limit, output cap, quality checks, queue bound, and overload behavior. A successful single-user demonstration provides none of those capacity guarantees by itself.

Example B A shared long document prefix

Suppose twenty requests share 6,144 prefix tokens and each later retains 2,048 unique tokens. With the [Chapter 10](#) geometry, twenty independent 8,192-token caches consume 20 GiB of raw payload. If compatible prefix blocks can be shared perfectly, count $6,144 + 20 \times 2,048 = 47,104$ unique retained tokens. At 128 KiB each, this is 5.75 GiB.

The arithmetic assumes exact reusable prefix state, one retained copy, no block rounding, and no extra metadata. It does not mean decode stops reading the shared context. Routing all requests toward cache locality can also create a hotspot, so evaluate cache savings alongside queue balance and latency.

Example C A model that must span devices

Suppose a model's deployed weights require 70 GiB and each of two devices has 48 GiB. Dividing weights evenly gives a hypothetical 35 GiB per device, leaving 13 GiB each for cache and all other state. Feasibility still requires checking uneven layers, quantization buffers, attention-head partitioning, and the chosen runtime's placement.

Tensor parallelism may fit the model but introduce collective work during every generation round. Two devices with weak peer connectivity can be slower than a different system with adequate single-device memory. Compare the complete design, not just summed memory. If an adequate smaller model meets quality requirements, it may simplify deployment; quality testing must establish that choice.

Example D Disaggregating a prompt

Suppose a prefill worker must transfer 1 GiB of KV state to a decode worker over a path with an effective sustained payload rate of 40 GB/s. The payload-only lower bound is 1,073,741,824 divided by 40,000,000,000, or about 26.84 milliseconds. Transfer setup, contention, partitioning, and synchronization can add time; overlap may hide some of it from the critical path.

Disaggregation must save enough interference or enable enough better placement to justify that transfer and its operational complexity. A short-prompt workload with little prefill contention may see little benefit. Measure both pools together, including network capacity and queueing.

Example E Training with a larger effective batch

Assume eight data-parallel workers, a microbatch of two sequences per worker, and four accumulation steps. The effective batch is $8 \times 2 \times 4 = 64$ sequences per optimizer update, subject to correct accumulation and normalization. Adding two-way tensor parallelism within each worker group increases the device count, not the data-parallel batch multiplier.

Use accumulation for smaller microbatches, sharding for model state, and activation checkpointing for saved intermediates. Each addresses a different part of the budget.

Primary references: [\[14\]](#) · [\[15\]](#) · [\[26\]](#) · [\[42\]](#)

21 / PRACTICE

A learning path you can follow

You can start without renting a GPU. Advance when you can explain a result and predict what will change.

Continue online: [Foundations of AI Infrastructure](#) · [Kernels and the Machine](#) · [Training at Scale](#)

The sequence below is a suggested curriculum. Advance when you can explain the result, not after an arbitrary number of days. Expensive hardware is unnecessary for the first stage; pencil-and-paper arithmetic and a CPU tensor library are enough.

Stage	What to learn	Exercise and evidence of understanding
1 Foundations	Tensors, shapes, dtypes, weights, activations	Calculate bytes for several tensors and check with a library
2 One accelerator	GPU groups, memory levels, host-device copies	Draw where one matrix multiplication's inputs and results live
3 One request	Tokens, causal attention, prefill, decode, KV	Step through a three-token prompt and two generated tokens
4 Memory planning	Raw weights, KV geometry, overhead	Reproduce Chapter 10 and explain each excluded allocation
5 Serving	Metrics, queues, batching, cache policies	Sweep prompt length and concurrency; record TTFT and intervals
6 Optimization	Quantization, paging, attention kernels	Predict which resource should improve before measuring
7 Distributed work	Collectives, TP, PP, EP, topology	Explain what crosses a link for one model partition
8 Training	Loss, backward, optimizer, sharding	Train a tiny network and compare forward-only and training state
9 Operations	SLOs, cold starts, recovery, power	Define overload behavior and demonstrate a restart or restore

Exercise 1. Use the linked memory calculators to double retained context and then double concurrent sequences. Predict the KV result first. Change only weight precision next and explain why the KV result stays unchanged.

Exercise 2. For a toy causal model, label each prompt and output token with the instant its K and V are created. Distinguish predicting a token from processing that token on a later iteration. This catches a common off-by-one misunderstanding in inference diagrams.

Exercise 3. If an accelerator is available, compare a warm and a cold request using identical inputs. Then compare one sequence with a batch. Report input tokens, output tokens, measurement boundaries, hardware, and software. Do not mix model-loading time with steady-state decode without identifying it.

Exercise 4. Draw a two-server topology with CPU sockets, GPUs, NICs, local peer links, and network switches. Choose a tensor-parallel split and mark which tensors must cross which edges. Then redraw it as two independent replicas and explain the communication that disappears.

Exercise 5. Train a tiny network, save state, restart the process, and resume. Separately enable activation recomputation if the framework supports it. Explain why the saved checkpoint file and activation checkpointing solve different problems.

Primary reading in learning order

Start with PyTorch's basic optimization tutorial for the training loop, NVIDIA's Programming Model for GPU execution, and Google's TPU architecture animations for a conceptual comparison. Continue with Hugging Face's caching explanation and cache strategies, then the request and memory chapters in this guide. These build from concrete tensors to serving behavior.

Read the PagedAttention, FlashAttention, GQA, and speculative-decoding papers after the request and memory chapters. Use the original Orca paper to understand iteration-level scheduling. For distributed work, read NCCL's collective definitions before Megatron-LM, FSDP, and ZeRO. For operations, begin with service-level objectives before studying detailed telemetry.

The Sources section provides direct links. Google TPU architecture includes conceptual animations, NCCL includes collective illustrations, and the PyTorch activation-checkpointing article includes diagrams of saved and recomputed activations. Vendor illustrations are teaching aids; they are not accurate physical floorplans or cycle-by-cycle simulations.

Primary references: [\[1\]](#) · [\[8\]](#) · [\[14\]](#) · [\[40\]](#) · [\[41\]](#)

22 / PRACTICE

Check your understanding

Use these prompts after a chapter or before an interview. A good answer names the tensor, the limiting resource and the evidence you would collect.

Continue online: [Concept library](#) · [Guided courses](#)

These are explanation prompts with short answer anchors. Expand each answer using the diagrams and calculations rather than memorizing its wording.

Question	A sound answer should include
Why use an accelerator for a neural network?	Parallel numerical work, data reuse, supported kernels, and host coordination
Why can two equally large GPUs perform differently?	Bandwidth, compute paths, precision, topology, clocks, and software
What is an SM?	A compute group containing execution, scheduling, registers, and memory resources
What is HBM?	Stacked device DRAM; distinct from on-die caches and registers
Where are weights and KV stored?	Runtime allocations in device memory in a resident design, with explicit exceptions
Why cache K and V but usually not old Q?	The new query needs past keys and values; old queries are not needed for its attention
What produces the first output token?	Final prompt-position logits from prefill in the ordinary decoder loop
Why does long context consume more memory?	More retained state in full-attention layers, plus workload-specific temporaries
Does four-bit mean every tensor uses four bits?	No; identify weights, activations, KV, scales, and accumulators separately
Why can a quantized model be slower?	Conversion overhead, kernel support, shape, and changed bottlenecks
What does FlashAttention save?	Attention intermediates and HBM traffic, not all attention arithmetic or KV
What does paging save?	Wasted allocation and duplicated compatible blocks, not required live payload

Question	A sound answer should include
What does continuous batching change?	The active sequence set between generation iterations
Does prefix caching eliminate decode over the prefix?	No; it primarily saves recomputation of compatible prefix state
Why distinguish TTFT and token intervals?	Prompt readiness and continuing output responsiveness are separate user experiences
Why are FLOP/s not tokens per second?	Workload work, bytes, dependencies, shapes, and scheduling intervene
How do NVLink and InfiniBand relate?	Different fabric roles; inspect the topology and software communication path
What does RDMA bypass?	Much data-path CPU copying and stack work, not all setup or CPU participation
What is the difference between all-reduce and all-gather?	Arithmetic combination versus collection of pieces
Why can MoE need large memory with modest active compute?	Total expert weights must reside or be supplied somewhere
Why can more devices slow one request?	Additional communication, synchronization, partitioning, and small local work
What extra memory does training need?	Gradients, optimizer state, saved activations, and implementation-specific copies
How does LoRA help?	Fewer trainable parameters and optimizer state while retaining base-model execution
What is the difference between FSDP and DDP?	Sharded state with gathers versus replicated model state and synchronized gradients
What makes a benchmark useful?	Quality, workload, boundaries, percentiles, cache assumptions, and reproducibility
What should happen when the service is full?	Bounded admission, backpressure or rejection, fair scheduling, and clear deadlines

When describing a system in an interview or design review, specify the workload first. Draw the data path, calculate capacity, identify candidate bottlenecks, propose one trade-off, and explain how you would measure whether it worked. This order turns vocabulary into an engineering argument.

Primary references: [\[1\]](#) · [\[14\]](#) · [\[40\]](#) · [\[41\]](#)

23 / REFERENCE

A glossary you can return to

Look up a term when it blocks your reading. Each definition points to the next idea rather than asking you to memorize the whole vocabulary.

Continue online: [Concept library](#) · [Open models: start here](#) · [GPU and TPU reference](#)

This glossary condenses the preceding explanations. The final column connects each term to the next concept to study. Entries describe common meanings; architecture- or runtime-specific terms can have narrower definitions in their own documentation.

Model and mathematical foundations

Term	Definition and purpose	Connection
AI infrastructure	Systems that execute and operate AI workloads	Hardware, software, and service objectives
Accelerator	A processor specialized for a class of work	Host CPU and supported kernels
Tensor	An array with shape, dtype, and storage	Memory footprint and operations
Shape	Sizes of a tensor's dimensions	Batch, sequence, and hidden dimensions
Dtype	Representation of each stored value	Precision and byte count
Layout and stride	Arrangement and address spacing of tensor elements	Kernel access patterns and copies
Parameter or weight	A learned model value	Persistent model storage
Activation	An intermediate produced for an input	Layer execution and training memory
Gradient	Derivative describing sensitivity of a loss	Backward pass and optimizer
Token	A discrete model input or output unit	Tokenizer and sequence length
Tokenizer	Conversion between input and token IDs	Model vocabulary and chat template
Vocabulary	The set of token IDs a model uses	Embeddings and output logits
Embedding	A vector representation of an item	Lookup tables, encoders, and retrieval
Hidden dimension	Width of a model's internal representation	Weight shapes and arithmetic cost
Layer	A computational transformation in a model	Activations and parameter matrices
Forward pass	Evaluation of model operations for an input	Inference and training

Term	Definition and purpose	Connection
Backward pass	Gradient computation through the forward graph	Saved or recomputed activations
Loss	Numerical objective used in training	Optimizer updates
Matmul	Matrix multiplication	Matrix units and data reuse
GEMM	General matrix multiplication primitive	Numerical libraries and tiled kernels
GEMV	Matrix-vector multiplication primitive	Small-batch linear-layer behavior
MAC	Multiply-accumulate operation	Often counted as two arithmetic operations
FLOP	One floating-point operation under a convention	Work count, not a speed
FLOP per second	Floating-point work rate	Precision-specific compute capability
TOPS	Trillions of operations per second	Requires operation type and counting convention
Logits	Unnormalized output scores	Sampling or greedy selection
Softmax	Maps scores into normalized exponential weights	Attention and output probabilities
Residual connection	Adds a bypassed representation to a sublayer result	Model block dataflow
Normalization	Rescales features using a defined statistical rule	LayerNorm, RMSNorm, and kernel fusion
MLP or FFN	Feed-forward network inside a model block	Large weight matrices or MoE experts
RoPE	Rotary positional embeddings	Position-dependent query and key transformations

GPU and server components

Term	Definition and purpose	Connection
CPU	General-purpose processor for control and computation	OS, host RAM, and accelerator coordination
GPU	Throughput-oriented parallel processor	Compute groups and device memory
SM	NVIDIA streaming multiprocessor compute group	Warps, arithmetic units, and local resources
Compute unit	Vendor-dependent grouping of accelerator resources	Inspect vendor architecture terminology

Term	Definition and purpose	Connection
ALU	Arithmetic and logic execution unit	Scalar or vector instruction pipelines
Tensor Core	NVIDIA matrix-compute hardware unit	Supported tile operations and dtypes
Thread	A logical execution instance	Warps and thread blocks
Warp	NVIDIA group of 32 threads for SIMT execution	Scheduling and divergence
Wavefront	AMD execution grouping	Size and behavior depend on architecture
SIMT	Single instruction multiple thread execution model	Per-thread state with grouped execution
Thread block	Cooperating CUDA thread group	Shared memory and synchronization
Grid	Collection of blocks in a kernel launch	Distribution of work across SMs
Kernel	Device-executed function	Launch overhead and compiled machine code
Tile	A manageable portion of a larger tensor operation	Local reuse in matrix kernels
Register	Small fast storage for executing work	Occupancy and spills
Register spill	Placement of excess register values in memory	Additional loads and stores
Shared memory	Programmer-managed CUDA on-chip scratch space	Tile reuse within cooperating work
SRAM	Static RAM commonly used on chip	Registers, caches, and local buffers
L1 cache	A nearby hardware-managed cache level	Locality within execution groups
L2 cache	A larger hardware-managed device cache level	Reuse across memory accesses
HBM	Stacked DRAM with a wide interface	Accelerator weight and state storage
GDDR	DRAM designed for graphics and accelerator use	Device memory in many GPUs
VRAM	Broad term for graphics or device memory	May be HBM or GDDR
DRAM	Dense dynamic random-access memory	Host RAM and device-memory families
DDR	Double-data-rate memory-interface family	Common CPU system memory
Memory controller	Hardware that manages DRAM transactions	Compute-to-memory bandwidth
TSV	Through-silicon via connecting stacked dies	HBM construction

Term	Definition and purpose	Connection
Interposer	Packaging interconnect structure between dies	Accelerator and memory integration
Chiplet	A die in a multi-die package	Inter-die communication and packaging
Copy engine	Hardware for supported data transfers	Overlap with computation
DMA	Direct memory access by a device	Copy engines and NIC data movement
Pinned memory	Host pages held in place for device access	Efficient asynchronous transfer paths
Unified Memory	Runtime-managed cross-processor memory access	Placement, migration, and coherence
Unified addressing	Consistent virtual address-space view	Addressing does not imply equal access speed
Page fault	Access requiring address or residency resolution	Managed-memory migration and stalls
Occupancy	Resident warps relative to supported maximum	Resource limits and latency hiding
Coalescing	Combining adjacent thread accesses efficiently	Device-memory transactions
Bank conflict	Competing accesses to a local-memory bank	Shared-memory throughput
Divergence	Different control paths inside an execution group	Reduced effective parallel execution
PCIe	Point-to-point host and device I/O interconnect	GPUs, NICs, SSDs, and switches
Lane	A serial link path within a PCIe connection	Negotiated link width and throughput
Root complex	Host-side origin of a PCIe hierarchy	CPU-device topology
PCIe switch	Fan-out component for PCIe connectivity	Shared upstream capacity
NUMA	Non-uniform access across memory domains	CPU affinity and GPU-NIC locality
SXM	A family of NVIDIA accelerator module form factors	Compatible baseboards and interconnects
NIC	Network interface controller	Server-to-network traffic
HCA	Host channel adapter, commonly for InfiniBand	RDMA transport and fabric attachment
DPU	Processor for selected infrastructure offloads	Networking, storage, and security functions
BMC	Independent server management controller	Out-of-band health and power control

Term	Definition and purpose	Connection
NVMe	Storage command interface and protocol family	SSD and storage I/O
SSD	Solid-state storage device	Model cache, data, and checkpoints

TPU and compiler vocabulary

Term	Definition and purpose	Connection
TPU	Google machine-learning ASIC family	Matrix units, memory, and ICI
ASIC	Application-specific integrated circuit	Specialized hardware design
NPU	Broad label for neural-processing accelerators	Capabilities differ by product
FPGA	Reconfigurable logic device	Custom hardware data paths
MXU	TPU matrix multiply unit	Systolic array arithmetic
Systolic array	Connected compute cells that pass data between neighbors	Operand reuse and matrix operations
Vector unit	Executes vector or elementwise operations	Normalization, activations, and softmax
Scalar unit	Handles scalar computation and control	Addresses and program coordination
VMEM	TPU vector-memory terminology for local storage	Compiler-managed working tensors
SparseCore	TPU hardware for supported sparse operations	Embedding-oriented workloads on supported generations
TPU slice	Connected allocation within a TPU pod	ICI topology and distributed execution
TPU pod	A grouped TPU infrastructure domain	Different meaning from Kubernetes Pod
ICI	TPU inter-chip interconnect	Communication inside a slice
DCN	Data-center network	Communication across TPU slices or other systems
Multislice	Execution across multiple TPU slices	Hierarchical ICI and DCN communication
XLA	Accelerated Linear Algebra compiler	Lowering, fusion, layouts, and partitioning
JAX	Array-computing and transformation framework	Compiled execution and differentiation
CUDA	NVIDIA parallel-computing platform	Programming model, runtime, and libraries

Term	Definition and purpose	Connection
ROCm	AMD accelerator software ecosystem	HIP, libraries, drivers, and tools
HIP	GPU programming interface in the ROCm ecosystem	Portability still depends on kernels and support
JIT compilation	Compiling code during execution or first use	Warm-up and shape-dependent compilation
AOT compilation	Compiling before ordinary execution	Deployment artefacts and supported shapes
Kernel fusion	Combining operations into fewer kernels	Reduced launches and intermediate traffic
CUDA Graph	Captured work graph for compatible replay	Lower launch and scheduling overhead
Stream	Ordered execution queue in a device runtime	Dependencies and possible overlap
Event	Runtime marker for progress or synchronization	Coordination across streams and host

Inference and model state

Term	Definition and purpose	Connection
Inference	Applying model parameters to inputs	Forward execution without ordinary weight updates
Serving	Operating inference for requests	Routing, scheduling, state, and reliability
Autoregressive generation	Predicting a sequence conditioned on prior tokens	Sequential decode dependencies
Prefill	Processing known input positions and preparing state	TTFT and prompt KV
Decode	Iterative generation after prompt preparation	Token intervals and growing state
Context window	Supported or configured span of model context	Token limits and retained state
Query	Attention representation that requests information	Compared with keys
Key	Attention representation used for matching	KV cache and attention scores
Value	Representation combined using attention weights	Attention output
KV cache	Retained key and value state for reuse	Application allocation, usually device memory
MHA	Multi-head attention with distinct KV heads	Higher cache-head count

Term	Definition and purpose	Connection
MQA	Multi-query attention sharing one KV head	Lower KV payload
GQA	Grouped-query attention sharing KV within groups	KV-head count controls cache size
MLA	Multi-head latent attention	Compressed, architecture-specific cached state
Causal mask	Restricts attention to allowed prior positions	Parallel prefill without future-token access
Sliding-window attention	Attention restricted to a moving region	Bounded state for affected layers
Cross-attention	Queries attend to a separate representation	Encoder-decoder or multimodal state
Dense model	Uses its dense layer paths for each token	Repeated weight access
MoE	Mixture of experts with token-dependent routing	Expert memory and communication
Router	Chooses experts or execution destinations	Load balance and dispatch
Active parameters	Parameters involved for a particular token	Compute differs from total residency
SSM	State-space model	Recurrent state rather than ordinary full KV
Prefix caching	Reusing compatible state for identical token prefixes	Less repeated prefill
Answer caching	Reusing complete previous outputs	Different validity rules from KV reuse
Paged KV	Logical cache positions mapped to physical blocks	Growth, sharing, and fragmentation
FlashAttention	IO-aware tiled exact-attention algorithm family	Avoids large HBM attention intermediates
Continuous batching	Updating the active request set between iterations	Capacity reuse while sequences finish
Dynamic batching	Combining arrivals before execution	Queue-delay and throughput trade-off
Chunked prefill	Scheduling prompt preparation in chunks	Prefill and decode interference
Speculative decoding	Drafting candidates and verifying with a target	Acceptance and verification overhead
Disaggregated serving	Separate pools for different serving phases	KV transfer and independent capacity
Greedy decoding	Selecting the highest-scoring token	Deterministic selection given identical logits

Term	Definition and purpose	Connection
Temperature	Sampling adjustment to logit distribution	Output variability and quality checks
Top-k sampling	Sampling within a chosen number of candidates	Output-selection policy
Top-p sampling	Sampling from a probability-mass candidate set	Output-selection policy
Beam search	Maintaining several candidate sequences	Branching state and cache sharing
EOS	End-of-sequence token	Stopping and cache release
RAG	Retrieval-augmented generation	External retrieval before or during generation
Embedding service	Produces vector representations	Often bounded forward-pass serving
Reranker	Scores or reorders candidate items	Retrieval pipeline and batching
Diffusion	Generation through iterative denoising	Different loop and state from LLM decoding

Numerical representation and efficiency

Term	Definition and purpose	Connection
FP32	32-bit floating-point representation	Reference or accumulator precision
FP16	16-bit floating-point representation	Reduced bytes and limited exponent range
BF16	16-bit floating point with broad exponent range	Training and inference mixed precision
FP8	Family of 8-bit floating-point representations	Scaling and hardware support
INT8 and INT4	Integer representations with 8 or 4 bits	Quantization codes and scale metadata
FP4	Family of small floating-point representations	Block scaling and supported kernels
TF32	NVIDIA math mode for selected matrix operations	Not an ordinary packed tensor-storage dtype
Mixed precision	Different formats for different tensors or operations	Input storage versus accumulator format
Accumulator	Storage for a running sum	Wider arithmetic can preserve precision
Quantization	Mapping values to a restricted representable set	Error versus storage and execution savings

Term	Definition and purpose	Connection
Dequantization	Reconstructing computational values from codes	Kernel overhead and compute dtype
Scale	Maps quantized codes to numerical magnitudes	Quantization resolution
Zero point	Code corresponding to numerical zero	Affine quantization
Calibration	Measuring representative distributions for quantization	Outliers and deployment data coverage
PTQ	Post-training quantization	Conversion of a trained model
QAT	Quantization-aware training	Adapting to quantization effects
Weight-only	Quantizing weights separately from activations	W4A16 and other mixed schemes
KV quantization	Compressing retained attention state	Context capacity and attention kernels
Group size	Number of values sharing quantization metadata	Error, metadata, and kernel trade-offs
GPTQ	Approximate second-order weight-quantization method	PTQ
AWQ	Activation-informed weight-channel scaling method	Weight quantization
SmoothQuant	Transformation for easier weight and activation quantization	Low-precision execution
NF4	A four-bit quantization codebook used in QLoRA	Distinct from ordinary integer INT4
Sparsity	Deliberate absence or non-use of values or work	Weight sparsity, attention sparsity, and MoE differ
Distillation	Training a student using a teacher's behavior	Smaller model with separately evaluated quality
Pruning	Removing selected weights or structures	Useful speedup requires executable sparsity or smaller shapes
Arithmetic intensity	Operations per byte moved at a specified level	Roofline analysis
Bandwidth-bound	Limited by transfer rate through an interface	HBM, local memory, or network
Compute-bound	Limited primarily by arithmetic throughput	Supported operations and tensor shapes
Latency hiding	Executing ready work while other work waits	Scheduling, occupancy, and overlap

Networks and parallel execution

Term	Definition and purpose	Connection
NVLink	NVIDIA high-speed peer interconnect	Local or larger accelerator fabrics
NVSwitch	Switching for compatible NVLink fabrics	Multi-accelerator peer connectivity
InfiniBand	Switched fabric with messaging and RDMA	HCAs, switches, and subnet management
Ethernet	Network technology family	IP traffic and supported RoCE
RDMA	Direct remote registered-memory operations	Reduced data-path CPU and copy work
RoCE	RDMA over Converged Ethernet	RDMA-capable NICs and network configuration
GPUDirect RDMA	Supported peer access to GPU memory	GPU-NIC data paths
GPUDirect Storage	Supported storage-to-GPU data path	Less intermediate host copying
NCCL	NVIDIA GPU collective-communication library	Uses underlying transports and topology
RCCL	ROCm collective-communication library	Collective operations on supported AMD systems
MPI	Message Passing Interface standard	Distributed processes and communication
Rank	Participant identity in a communication group	Mapping processes to devices
World size	Number of participants in a group	Distinguish parallelism dimensions
All-reduce	Combine values and distribute the result to all	Gradients or partial-result sums
All-gather	Collect all shards at each participant	Parameter or activation materialization
Reduce-scatter	Combine values and divide the result	Sharded gradients
All-to-all	Exchange participant-specific portions	MoE token dispatch
Broadcast	Send one participant's data to a group	Initialization and shared state
Tensor parallelism	Partition tensors within layers	Frequent collective communication
Pipeline parallelism	Partition successive layer groups	Stage transfers and bubbles
Expert parallelism	Partition experts	Routing and all-to-all
Context parallelism	Partition sequence-related computation	Attention exchanges and cache placement

Term	Definition and purpose	Connection
Data parallelism	Process different data with model replicas	Training synchronization or inference routing
Microbatch	Subdivision of a larger workload batch	Accumulation and pipeline scheduling
Pipeline bubble	Idle stage time during pipeline scheduling	Fill, drain, and imbalance
Scale up	Tightly coupled expansion of compute	Peer fabrics and topology
Scale out	Expansion across servers or groups	Network communication and replication
Leaf-spine	Network with endpoint-facing and spine tiers	Multiple paths and cross-rack capacity
Oversubscription	Shared capacity below possible aggregate demand	Network congestion or shared devices
Bisection bandwidth	Capacity across a division of a topology	Distributed traffic limits
ECMP	Equal-cost multipath forwarding	Distribution across network paths
ECN	Explicit congestion notification	Feedback for sender rate control
PFC	Priority-based flow control	Pausing selected Ethernet traffic classes
Subnet manager	Entity configuring an InfiniBand subnet	Discovery, addressing, and routes
QP	RDMA queue pair	Send and receive work queues
CQ	Completion queue	Reporting RDMA work completion
Memory region	Registered buffer with access metadata	RDMA permissions and transfer setup
MTU	Maximum transmission unit	Packet sizing and path compatibility
Optics or transceiver	Converts electrical and optical network signals	Link reach, speed, and health

Training and operations

Term	Definition and purpose	Connection
Optimizer	Procedure updating trainable parameters	Gradients, learning rate, and state
Adam moments	Moving gradient statistics maintained by Adam	Optimizer memory
Learning rate	Scale controlling parameter updates	Convergence and training schedule

Term	Definition and purpose	Connection
Epoch	One traversal of a defined dataset	Not always the main large-scale training unit
Gradient accumulation	Multiple microbatches before an update	Effective batch and activation memory
DDP	Distributed Data Parallel	Replicas with synchronized gradients
FSDP	Fully Sharded Data Parallel	Parameter gathers and sharded state
ZeRO	Zero Redundancy Optimizer techniques	Staged sharding of optimizer, gradient, and parameter state
LoRA	Low-rank adaptation with frozen base weights	Smaller trainable state
QLoRA	Adapter training through a quantized base	Base-weight memory savings
Activation checkpointing	Recomputing intermediates instead of retaining all	Training memory versus compute
Durable checkpoint	Saved state for later restoration	Failure recovery and resume semantics
Offload	Placing state in another memory tier	Capacity versus transfer cost
Data loader	Supplies training batches	CPU, storage, and accelerator utilization
Sequence packing	Combining examples into efficient training layouts	Padding savings and correct boundaries
TTFT	Time to first output token at a stated boundary	Queueing, prefill, and cold start
ITL	Interval between output token events	Streaming responsiveness
TPOT	A defined average time per output token	Check treatment of the first token
Throughput	Work completed per unit time	State input, output, or request units
Goodput	Useful work meeting declared requirements	Latency, quality, and error targets
Tail latency	High-percentile delay	Bursts and stragglers
SLI	Service-level indicator	Measured service behavior
SLO	Service-level objective	Target for an SLI
Admission control	Decide whether to accept additional work	Memory budgets and deadlines
Backpressure	Signal or enforce reduced offered work	Queue bounds and overload
Cold start	Preparing a worker before useful service	Provisioning, load, compile, and warm-up

Term	Definition and purpose	Connection
Autoscaling	Adjusting service capacity to demand	Lead time and workload-aware signals
Replica	An independently serving model copy or group	Load balancing and failure domains
Kubernetes Pod	Scheduling and lifecycle unit for containers	Distinct from a TPU pod
Device plugin	Exposes specialized devices to Kubernetes	Allocation, not token scheduling
Gang scheduling	Coordinated placement of a job's worker group	Avoiding unusable partial allocations
MIG	Hardware partitioning on supported NVIDIA GPUs	Compute and memory isolation
Time slicing	Sharing execution opportunities over time	Interference without MIG-equivalent isolation
MPS	CUDA Multi-Process Service	Sharing execution among processes
DCGM	NVIDIA Data Center GPU Manager	Monitoring and diagnostics
ECC	Error-correcting code mechanisms	Memory integrity and health monitoring
OOM	Out-of-memory allocation failure	Capacity, fragmentation, and peak demand
Fragmentation	Allocated or free space difficult to use efficiently	Paging and allocator behavior
Workspace	Temporary memory required by an algorithm	Peak memory beyond weights and KV

Power, cooling and model artifacts

Term	Definition and purpose	Connection
Thermal throttling	Reduced performance to manage temperature	Cooling and sustained clocks
Power cap	Limit imposed on device or system power	Throughput and efficiency trade-offs
TDP	Thermal-design or vendor power-design rating	Not identical to measured wall power
PSU	Power supply unit	Conversion and server power redundancy
PDU	Power distribution unit	Rack feeds and power monitoring
UPS	Uninterruptible power supply	Short-term power continuity
CDU	Coolant distribution unit	Cooling-loop interface and management
PUE	Facility energy divided by IT equipment energy	Facility overhead, not model efficiency
TCO	Total cost of ownership	Hardware, energy, operations, and useful output
Model registry	Catalogue of versioned model artefacts and metadata	Promotion and reproducible deployment
Safetensors	Tensor-storage file format	Safe tensor deserialization and loading
GGUF	Tensor and metadata container format	Quantized or other stored representations

Primary references: [\[1\]](#) · [\[8\]](#) · [\[14\]](#) · [\[40\]](#) · [\[41\]](#)

24 / SOURCE NOTES

Evidence you can follow

Original papers establish the mechanisms; vendor and maintainer documentation describes implementations. Examples in this guide are calculations with explicit assumptions, not vendor performance claims. Living documentation can change after this edition.

Reference numbers stay stable between revisions. Gaps in numbering reflect removed secondary reading. The update ledger records which claims were checked and which source-file errors were corrected. Use the live reference pages for changing hardware specifications, checkpoint releases and supported execution recipes.

- [1] [NVIDIA. CUDA Programming Guide 1.2 Programming Model. Living documentation.](#)
- [2] [NVIDIA. CUDA Programming Guide 2.3 Writing SIMT Kernels. Living documentation.](#)
- [3] [NVIDIA. CUDA Programming Guide 2.5 Asynchronous Execution. Living documentation.](#)
- [4] [NVIDIA. CUDA Programming Guide 2.6 Unified and System Memory. Living documentation.](#)
- [5] [NVIDIA. CUDA C++ Best Practices Guide. Living documentation.](#)
- [6] [Micron. High bandwidth memory. Living product and technical documentation.](#)
- [7] [NVIDIA. Introduction to NVIDIA DGX H100 H200 Systems. Living documentation; illustrative server architecture.](#)
- [8] [Google Cloud. TPU architecture. Living documentation.](#)
- [9] [Google Cloud. Cloud TPU performance guide. Living documentation.](#)
- [10] [Google Cloud. Cloud TPU Multislice Overview. Living documentation.](#)
- [11] [OpenXLA. XLA. Living documentation.](#)
- [12] [AMD. What is ROCm. Living documentation.](#)
- [13] [Vaswani et al. Attention Is All You Need. 2017; revised 2023.](#)
- [14] [Hugging Face. How caching works. Living Transformers documentation.](#)
- [15] [Hugging Face. Cache strategies. Living Transformers documentation.](#)
- [18] [Ainslie et al. GQA Training Generalized Multi Query Transformer Models from Multi Head Checkpoints. 2023.](#)
- [19] [Noam Shazeer. Fast Transformer Decoding One Write Head is All You Need. 2019.](#)
- [20] [DeepSeek AI. DeepSeek V2 A Strong Economical and Efficient Mixture of Experts Language Model. 2024.](#)
- [21] [Dao et al. FlashAttention Fast and Memory Efficient Exact Attention with IO Awareness. 2022.](#)
- [22] [Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. 2023.](#)
- [23] [Yu et al. Orca A Distributed Serving System for Transformer Based Generative Models. OSDI 2022.](#)

- [24] [vLLM project. Automatic Prefix Caching. Living documentation.](#)
- [25] [vLLM project. Optimization and Tuning. Living documentation.](#)
- [26] [vLLM project. Disaggregated Prefilling. Living documentation; implementation described as experimental.](#)
- [27] [Leviathan et al. Fast Inference from Transformers via Speculative Decoding. 2022; ICML 2023.](#)
- [28] [Hugging Face. Quantization. Living Optimum documentation.](#)
- [29] [Google Cloud. Improve your model performance with bfloat16. Living documentation.](#)
- [30] [Frantar et al. GPTQ Accurate Post Training Quantization for Generative Pre trained Transformers. 2022; ICLR 2023.](#)
- [31] [Lin et al. AWQ Activation aware Weight Quantization for LLM Compression and Acceleration. 2023; MLSys 2024.](#)
- [32] [Xiao et al. SmoothQuant Accurate and Efficient Post Training Quantization for Large Language Models. 2022; ICML 2023.](#)
- [33] [Dettmers et al. QLoRA Efficient Finetuning of Quantized LLMs. 2023.](#)
- [34] [vLLM project. Quantization. Living documentation.](#)
- [35] [NVIDIA. NVLink and NVLink Switch. Living product documentation; no performance ranking adopted.](#)
- [36] [NVIDIA. RDMA aware Networks Programming Guide. DOCA documentation.](#)
- [37] [InfiniBand Trade Association. About InfiniBand. 2023 page; used for architecture definitions only.](#)
- [38] [InfiniBand Trade Association. RoCE Initiative. Living documentation.](#)
- [39] [NVIDIA. GPUDirect RDMA. Living documentation.](#)
- [40] [NVIDIA. NCCL Collective Operations. Living documentation.](#)
- [41] [PyTorch. Optimizing Model Parameters. Living tutorial.](#)
- [42] [PyTorch. Getting Started with Distributed Data Parallel. Living tutorial.](#)
- [43] [PyTorch. Getting Started with Fully Sharded Data Parallel FSDP2. Living tutorial.](#)
- [44] [DeepSpeed. Zero Redundancy Optimizer. Living tutorial.](#)
- [45] [Hu et al. LoRA Low Rank Adaptation of Large Language Models. 2021.](#)
- [46] [Shoeybi et al. Megatron LM Training Multi Billion Parameter Language Models Using Model Parallelism. 2019.](#)
- [47] [Fedus et al. Switch Transformers Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. 2021; JMLR 2022.](#)
- [48] [Gu and Dao. Mamba Linear Time Sequence Modeling with Selective State Spaces. 2023.](#)
- [49] [Kubernetes. Schedule GPUs. Living documentation.](#)
- [50] [NVIDIA. Multi Instance GPU User Guide Introduction. Living documentation.](#)
- [51] [NVIDIA. Time Slicing GPUs in Kubernetes. Living documentation.](#)
- [52] [NVIDIA. DCGM Feature Overview. Living documentation.](#)
- [53] [MLCommons. MLPerf Inference Datacenter. Living benchmark documentation.](#)

- [54] [NVIDIA. Triton Inference Server Batchers. Living documentation.](#)
- [55] [Triton project. Triton documentation. Living documentation.](#)
- [56] [vLLM project. Metrics. Living documentation.](#)
- [57] [Google SRE. Service Level Objectives. Site Reliability Engineering, 2016.](#)
- [58] [NVIDIA. GPUDirect Storage Overview Guide. Living documentation.](#)
- [59] [Hugging Face. Safetensors. Living documentation.](#)
- [60] [ggml project. GGUF file format. Living specification.](#)
- [61] [PyTorch Foundation. Current and New Activation Checkpointing Techniques in PyTorch. 2025; updated 2026.](#)
- [62] [Open Compute Project. Coolant Distribution Unit. Living project documentation.](#)
- [63] [Google Data Centers. Power usage effectiveness. Living documentation; definition only.](#)
- [64] [Lewis et al. Retrieval Augmented Generation for Knowledge Intensive NLP Tasks. 2020.](#)
- [65] [Ho et al. Denoising Diffusion Probabilistic Models. 2020.](#)
- [66] [Su et al. RoFormer Enhanced Transformer with Rotary Position Embedding. 2021; revised 2023.](#)
- [67] [AMD. What is RCCL. Living documentation.](#)
- [68] [Kaplan et al. Scaling Laws for Neural Language Models. 2020; Section 2.1 gives the Transformer compute approximation.](#)

KEEP GOING

Turn the vocabulary into a decision.

You can now ask where the bytes live, what computation uses them and which path limits useful work. Apply that sequence to one small deployment before adding another optimization.

YOUR NEXT QUESTION	USE THIS RESOURCE
What should I learn next?	AI Infra Foundations, then Inference Engineering or Training at Scale.
Which accelerator has the right memory and topology?	The public GPU and TPU reference, with links to original specifications.
Which model can I run, and what might break?	Model cards, lab timelines and the Open Model Field Guide.
Can I reproduce the arithmetic?	KV-cache, training, roofline and cost calculators.

[Choose a course](#) · [Compare hardware](#) · [Read model cards](#) · [Try a calculator](#)

An interview answer worth practicing

State the workload and quality target. Draw the path. Calculate a capacity bound. Name the likely bottleneck. Propose one trade-off and say what measurement would make you reverse it.

Download the current edition and report a correction at [the guide page](#). The PDF and future editions are included while Premium membership is active. Keep the copies you download.

AI Infra Interviews · September 2026