Deep Learning and Transformers

Chapter 22: From PyTorch to Accelerator Silicon

[Author Names]

2026

Chapter 22

From PyTorch to Accelerator Silicon

Chapter Overview

You understand the mathematics of transformers. You know that attention is a softmax-weighted average over value vectors, that feed-forward networks are two linear projections with a nonlinearity, and that the whole structure repeats for dozens or hundreds of layers. But there is an enormous gap between understanding the math and understanding how it executes. When you write `output = self.attention(x)`, what happens? Which chip does the multiplication? How does data get there? Why does one serving configuration produce ten times the throughput of another on identical hardware?

This chapter is not a catalog of tricks. It is a framework for reasoning about trade-offs. Seven layers of engineering separate your Python code from the transistors. Each layer is an optimization surface with its own controls, constraints, and failure modes. Decisions at one layer propagate to others. An engineer who understands these interactions can diagnose bottlenecks, select optimizations in the right order, and avoid wasted effort.

We use NVIDIA H100 as the primary worked example and explicitly map every concept to AMD MI300X, Google TPU v5p, and AWS Trainium2. For PyTorch quantization implementation code (dynamic, static, and QAT), see Chapter **??**.

Learning Objectives

1. Understand the seven-layer inference stack from model definition to accelerator silicon
2. Reason about the prefill vs. decode asymmetry and its optimization implications
3. Apply tensor, pipeline, and expert parallelism with correct communication patterns
4. Understand frontend runtimes: eager dispatch, lazy tensors, graph capture, compilation, and replay
5. Configure inference engines: continuous batching, PagedAttention, and scheduler knobs
6. Analyze compute-bound vs. memory-bound regimes using roofline reasoning
7. Apply model compression: quantization (FP8, INT8, INT4), pruning, and distillation
8. Diagnose bottlenecks systematically and select optimizations by regime
9. Evaluate hardware across NVIDIA, AMD, TPU, and Trainium platforms

Three Optimization Objectives

Every optimization targets one or more of three objectives. They often conflict, so you must choose which to prioritize before optimizing:

Latency: time from request arrival to first/last token. Dominates interactive and real-time applications. Sensitive to serialized dependencies, interconnect latency, and runtime overhead.

Throughput: tokens per second across all concurrent requests. Dominates batch and offline workloads. Sensitive to accelerator utilization, batching efficiency, and memory capacity.

Cost-efficiency: tokens per dollar (or per watt). Dominates at scale. Often requires trading latency for throughput via larger batches, aggressive quantization, or fewer devices.

Numbers in this chapter are a mix of conceptual examples, representative measurements from published benchmarks, and platform-specific values. Each is labeled accordingly. Exact performance depends on model, hardware, software versions, and workload shape. Always measure on your own stack.

22.1 The Two Phases of Inference: Prefill and Decode

Almost every optimization decision depends on which phase dominates your workload, and the two phases have completely different performance characteristics.

Prefill (processing the prompt). The entire prompt is processed in one forward pass. Every token in the prompt is handled simultaneously. If the prompt has 500 tokens, the model runs matrix multiplications on all 500 at once—for example, a $[500 \times 8192]$ activation matrix times a $[8192 \times 8192]$ weight matrix. Large matrix multiplications keep the GPU's compute cores busy. Prefill is *compute-bound*.

Decode (generating the response). After prefill, the model generates one token at a time. Each token requires a full forward pass through all layers, but the "batch" of new tokens is tiny—just one new token (or a handful if batching multiple requests). The matrix multiplication is now $[1 \times 8192]$ times $[8192 \times 8192]$. The weight matrix is exactly the same size as during prefill, but the activation vector is tiny. The GPU loads the entire weight matrix from memory just to multiply it by a single row. Decode is *memory-bandwidth-bound*.

This distinction is critical because different bottlenecks require different optimizations:

- Making the compute cores faster (e.g., FP8) helps prefill much more than decode.
- Reducing memory traffic (e.g., quantizing weights so there is less to load) helps decode much more than prefill.
- Reducing communication latency (e.g., faster all-reduce) helps decode more, because decode's per-token compute is so short that even a few microseconds of communication overhead is proportionally large.

A chatbot generates thousands of decode tokens per request but prefills once, so decode performance usually dominates. A document summarization service might process very long prompts and generate short summaries, so prefill dominates instead.

22.2 The Seven-Layer Inference Stack

The path from PyTorch to silicon follows seven conceptual layers, numbered 7 (closest to you) down to 1 (bare metal). The top layers are portable; the bottom layers are device-specific. A single software tool (e.g., XLA) may span multiple conceptual layers. The layer model is an analytical framework, not a claim about software module boundaries.

How the layers interact: Layer 7 determines communication patterns—how often devices must talk and how much data they exchange. Layer 4 determines request shape distributions—how variable or regular the work is. Layers 5 and 2 determine how much overhead is optimized away. Layers 3 and 1 set the execution efficiency ceiling—the physical limits of how fast things can run.

Key Point 22.1 (Misconception Alert)**.** *A single tool may span multiple layers. XLA implements both graph optimization (Layer 5) and final lowering (Layer 2). vLLM orchestrates scheduling*

#	Layer	What It Controls	Portable?	Example Implementations
7	Model Definition	Architecture dims, parallelism, sharding	Yes	Same `nn.Module` on all backends
6	Frontend Runtime	Eager dispatch, lazy tracing, compiled capture	Mostly	CUDA/HIP eager; XLA lazy; `torch.compile`
5	Graph Compilation	Fusion, memory plan, comm scheduling	IR portable	TorchInductor; XLA HLO; NeuronX Compiler
4	Inference Engine	Batching, KV-cache mgmt, orchestration	Concepts yes	vLLM; TensorRT-LLM; NeuronX Serving
3	Device Kernels	GEMM, attention, all-reduce implementations	Concepts only	cuBLAS+NCCL; rocBLAS+RCCL; XLA; Neuron
2	Device Compiler	Final lowering to native ISA, scheduling	No	nvcc→SASS; LLVM→AMDGPU; XLA→TPU ISA
1	Accelerator HW	Compute units, memory hierarchy, interconnect	No	H100+NVLink; MI300X+IF; TPU+ICI; Trn2

Table 22.1: The seven-layer inference stack. Layers 7–5 are portable across vendors. Layers 3–1 are device-specific.

> *(Layer 4) and inserts collectives (affecting Layer 3). Do not assume one-to-one mapping between tools and layers.*

22.2.1 Backend Cheat Sheet

This reference maps every concept to its vendor-specific implementation. Later sections cite it rather than repeating the mapping.

Concept	NVIDIA (CUDA)	AMD (ROCm)	Google TPU	AWS Trainium
Graph capture	`torch.compile`	`torch.compile`	XLA lazy/compile	`torch_neuronx.trace`
IR	Inductor/Triton	Inductor+HIP	StableHLO/HLO	Neuron HLO
GEMM library	cuBLAS/CUTLASS	rocBLAS/CK	XLA GEMM	Neuron GEMM
Attention kernel	FlashAttention-2/3	CK FlashAttn	Pallas attention	Neuron attention
All-reduce	NCCL	RCCL	XLA collectives	Neuron collectives
LLM engine	vLLM, TRT-LLM	vLLM ROCm	vLLM TPU	NeuronX Serving
Graph replay	CUDA Graphs	hipGraphs	XLA executable	Neuron NEFF
Intra-node link	NVLink/NVSwitch	Infinity Fabric	ICI torus	NeuronLink
Inter-node fabric	InfiniBand NDR	InfiniBand NDR	ICI across pods	EFA (3.2 Tb/s)

Table 22.2: Backend cheat sheet: vendor-specific implementations of each inference stack concept.

22.2.2 Latency and Bandwidth Ladder

22.3 Layer 7: The Model Definition

When you define a transformer in PyTorch, you lock in decisions that propagate through every layer below.

Tensor shapes: hidden dimension H, head count, KV-head count, FFN intermediate size, layer count. These determine every GEMM shape, every collective message size, and the memory footprint breakdown.

Link Type	Example	BW (bidi)	Latency	Scope	Use Case
NVLink 4.0	H100 (8 GPUs)	900 GB/s	∼1–2 μs	Intra-node	TP, EP
NVLink 5.0	B200 (72 NVSwitch)	1.8 TB/s	∼1–2 μs	Multi-rack	Extended TP
ICI torus	TPU v5p pod	Multi-TB/s	∼1–3 μs	Within pod	TP/EP
Infinity Fabric	MI300X (8 GPUs)	∼896 GB/s	∼2–3 μs	Intra-node	TP, EP
NeuronLink	Trn2 (16 chips)	∼1.6 TB/s	∼few μs	Intra-inst	TP
InfiniBand NDR	Multi-node	400 Gb/s/port	∼5–10 μs	Cross-node	PP, param sync
EFA	AWS instances	3.2 Tb/s/inst	∼5–15 μs	Cross-node	PP, cross-inst EP

Table 22.3: Interconnect latency and bandwidth ladder. TP (frequent, latency-sensitive) goes on the fastest link. PP (infrequent, large) tolerates slower links. EP (all-to-all) requires high bisection bandwidth.

Divisibility constraints: head counts must divide evenly by TP degree. KV-head counts must also divide. FFN dimensions affect kernel tile efficiency.

Communication patterns: TP degree fixes the number and frequency of all-reduces. PP depth fixes stage boundary point-to-point transfers. EP determines all-to-all topology requirements.

Memory footprint classes: model weights, KV-cache per token, activation memory per layer. Each scales differently with parallelism.

Key Point 22.2 (What Layer 7 Cannot Fix Later)**.** *Poor divisibility (e.g., 8 KV-heads with TP-3 gives 8/3 which is not integer), hidden dimensions misaligned with kernel tile sizes (typically multiples of 128), or an architecture whose communication volume exceeds interconnect capacity at the target TP degree.*

22.3.1 Why Multi-Device Execution Is Required

Multi-device execution is required for three distinct reasons, and they are not equivalent:

Parameter memory fit. A 70B model needs ∼140 GB in BF16 ($70 \times 10^9 \times 2$ bytes). An H100 has 80 GB, a TPU v5p has 95 GB, and a Trainium2 has 96 GB—the weights alone do not fit. MI300X at 192 GB can hold 70B weights plus meaningful KV-cache on one device, but this is the exception.

KV-cache fit. Even if weights fit, the KV-cache can be enormous. For a 70B model with 128K context length and 80 layers, a single request at full context can require 10+ GB. A few concurrent requests exhaust remaining memory.

Throughput and latency targets. Even if everything fits, a single device may not meet requirements. A single GPU must stream all 140 GB of weights through memory for every decode token—that alone takes ∼42 ms at 3.35 TB/s on H100. Splitting weights across 4 GPUs means each streams only 35 GB, roughly 4× faster.

22.3.2 Tensor Parallelism: Splitting Weight Matrices

What All-Reduce Actually Does

Imagine four devices, each holding a number: Device 0 has 10, Device 1 has 20, Device 2 has 30, Device 3 has 40. An all-reduce(sum) computes the sum ($10 + 20 + 30 + 40 = 100$) and distributes the result to every device. After the all-reduce, all four devices hold the value 100.

In LLM inference, the "numbers" are tensors—vectors or matrices of thousands of floating-point values. When TP splits a weight matrix across devices, each device computes a partial result. The all-reduce sums these partial results so every device gets the complete answer.

Two critical characteristics of all-reduce determine TP's performance impact:

Frequency: a standard transformer block requires two all-reduces per layer (one after the attention output projection, one after the FFN down-projection). For an 80-layer model, that is 160 all-reduces per token generated.

Message size: during decode, the data being summed is a vector of size $[H]$ in the chosen precision. For $H = 8192$ in BF16: $8192 \times 2 = 16{,}384$ bytes = 16 KB. At 900 GB/s of NVLink bandwidth, 16 KB transfers in nanoseconds—but the latency of initiating the all-reduce ($\sim$1–2 μs) dominates. This is why decode TP performance is *latency-bound*, not bandwidth-bound.

Column-Parallel and Row-Parallel GEMMs

TP splits weight matrices across devices using two deliberately paired patterns.

Column-parallel GEMM (no communication needed). Split weight matrix $\mathbf{W}$ with shape $[8192 \times 8192]$ by columns: Device 0 gets $[8192 \times 4096]$, Device 1 gets $[8192 \times 4096]$. Input $\mathbf{x}\,[1 \times 8192]$ goes to both devices. Each produces half the output. No communication needed—but each device only has half the result.

Row-parallel GEMM (all-reduce required). The next weight matrix is split by rows. Each device multiplies its local input half by its local weight half, producing partial sums $[1 \times 8192]$. These must be summed via all-reduce to get the correct result.

Key Point 22.3 (The Megatron-LM Pairing)**.** *By alternating column-parallel then row-parallel within each transformer block, the column-parallel output feeds directly as the row-parallel input. Only the row-parallel step needs an all-reduce. A standard transformer block has two such pairs (attention: QKV column-parallel → output row-parallel; FFN: gate/up column-parallel → down row-parallel), so each layer needs exactly two all-reduces. This pattern works identically on any hardware that supports all-reduce.*

Per-layer communication volume. Bytes per all-reduce per device = $H \times p$, where p = bytes per element. Per layer: $2 \times H \times p$. For decode with $H = 8192$, BF16: $2 \times 8192 \times 2 = 32$ KB per layer. For 80 layers: $\sim$2.6 MB total.

Example 22.1 (Complete Per-Device Walkthrough: TP-2)**.** Trace one transformer layer with TP-2. $H = 8192$, 32 query heads (GQA: 8 KV-heads, $d_{\text{head}} = 256$), FFN= 22016. Decode: $B = 1$, $S = 1$. All values BF16.

Step 1 — QKV Projection (column-parallel, NO communication). Both devices receive the same input $\mathbf{x}\,[1 \times 8192]$. Each multiplies by its local Q, K, V column shards. Device 0 produces: $\mathbf{Q}_0\,[1 \times 4096]$ (heads 0–15), $\mathbf{K}_0\,[1 \times 512]$ (KV-heads 0–3), $\mathbf{V}_0\,[1 \times 512]$.

Step 2 — Attention (local, NO communication). Each device runs attention on its own heads against its own KV-cache shard. Completely independent per device.

Step 3 — Output Projection + ALL-REDUCE. Row-parallel step. Each device computes partial = $[1 \times 4096] \times [4096 \times 8192] = [1 \times 8192]$. All-reduce sums them. Data transferred: $8192 \times 2 = 16$ KB per device.

Steps 4–5 — FFN (same column→row pattern). Column-parallel gate+up (no communication), local SwiGLU, row-parallel down + all-reduce (another 16 KB).

Total for this layer: 2 all-reduces, 32 KB moved. For 80 layers: 160 all-reduces, $\sim$2.6 MB. At NVLink latencies of $\sim$1–2 μs per all-reduce, communication overhead is $\sim$160–320 μs per token. On slow interconnects (InfiniBand at $\sim$5–10 μs), the same 160 all-reduces cost 800–1600 μs—up to 30% overhead. This is why TP must use the fastest available link.

22.3.3 Pipeline Parallelism

While TP splits every layer across devices, PP assigns entire layers to different devices. Device 0 computes layers 0–39; Device 1 computes layers 40–79. Each device holds complete, unsharded weights for its assigned layers.

Communication is point-to-point: after Device 0 finishes, it sends the activation tensor to Device 1. During prefill with a 2048-token prompt: activation = $[1 \times 2048 \times 8192] \times 2$ bytes = 32 MB. But this

transfer happens only once per stage boundary, not once per layer. During decode ($S = 1$): only 16 KB. PP therefore tolerates higher-latency links.

Pipeline bubble: the key drawback. With simple scheduling, the fraction of time wasted to bubbles is roughly 1/PP_degree. Micro-batching reduces this by feeding multiple small batches in rapid succession. Decode is particularly vulnerable because each micro-batch has so little compute that communication latency between stages becomes proportionally larger.

22.3.4 Expert Parallelism

EP is specific to Mixture-of-Experts (MoE) models, where each transformer layer has multiple "expert" FFN blocks and a router that sends each token to only 1–2 experts. EP places different experts on different devices. The communication pattern is all-to-all: every device may need to send tokens to every other device.

All-to-all is the most interconnect-demanding collective. This is where bisection bandwidth matters: if you split 8 devices into two groups of 4, the total bandwidth between the groups determines how fast all-to-all runs. NVSwitch provides full bisection bandwidth for 8 GPUs. InfiniBand and EFA between nodes typically do not provide sufficient bisection bandwidth for efficient cross-node EP.

Routing skew risk: if the router disproportionately sends tokens to a few experts, those devices become bottlenecks while others idle. Load-balancing losses and capacity factors in the model architecture (Layer 7) directly affect hardware utilization at Layers 3/1.

Multiple specialized EP kernel libraries have emerged for inter-node communication:

Aspect	DeepEP	PPLX-Kernels	UCCL-EP	NCCL a2a
GPU vendor	NVIDIA only	NVIDIA only	NVIDIA + AMD	NVIDIA (+AMD)
NIC requirement	Mellanox CX (IB)	EFA or IB	Any RDMA	Any
GPU-to-NIC path	IBGDA (direct)	NVSHMEM+libfabric	CPU proxy	CPU-initiated
FP8 support	Native	Native + block scaling	Native	Via framework
Cloud deployment	On-prem IB	AWS (EFA) + IB	AWS, GCP, on-prem	Any

Table 22.4: EP kernel library comparison for inter-node MoE communication.

22.3.5 Context Parallelism and Sequence Parallelism

Context Parallelism (CP) addresses what happens when the sequence is so long that the KV-cache for a single request does not fit on one device. CP splits the sequence across devices. Ring attention is the most common implementation: devices are arranged in a logical ring, each computing attention for its local chunk against its local KV block, then passing its KV block to the next device.

Sequence Parallelism (SP) is a refinement of TP that reduces memory usage. In standard TP, activations between GEMMs are replicated on every device. SP uses reduce-scatter (each device gets $1/P$ of the result) followed by a deferred all-gather. Between GEMMs, each device holds only $1/P$ of the activations. Total communication volume is the same, but peak activation memory drops by the TP degree.

22.3.6 Parallelism Selection Framework

Scenario	Start with	Why	Watch out for
Dense model, doesn't fit	TP first, then PP	TP distributes every layer	TP beyond NVLink domain
Decode latency critical	TP, minimize degree	Each TP AR adds latency	Over-aggressive TP
Long context, high concurrency	TP + CP or KV quant	KV-cache dominates memory	CP ring overhead
MoE model	EP first, TP within	EP native to MoE	All-to-all needs bisection BW

Table 22.5: Parallelism selection framework: start from requirements and work down.

22.4 Layer 6: Frontend Runtime

This layer determines how Python model code translates into work that lower layers execute. It controls execution semantics (eager vs. lazy vs. compiled dispatch), Python-to-accelerator overhead, and how operations are grouped before graph compilation.

22.4.1 Eager Mode (NVIDIA, AMD)

Default PyTorch on CUDA/ROCm: each operation dispatches immediately. When your code calls `torch.matmul(x, W)`, PyTorch's C++ dispatcher selects a kernel, launches it on the GPU, and returns control to Python. Each dispatch involves Python interpreter overhead, C++ dispatch, kernel launch, and potentially synchronization.

For large operations (e.g., a big prefill GEMM that takes milliseconds), this overhead is negligible. But during decode with TP-4, individual GEMMs on tiny tensors may take only 5–15 μs of actual compute. If every dispatch adds 3–5 μs of overhead, the overhead is 20–50% of compute time.

22.4.2 Lazy Tensor Mode (TPU, Trainium)

On TPU, PyTorch/XLA records operations into a graph instead of executing each immediately. Operations accumulate until an explicit barrier (`xm.mark_step()`), at which point XLA compiles the entire graph, caches the result, and dispatches it as a single unit. The advantage: XLA can optimize many operations together. The downside: debugging is harder because errors surface only at compilation time. Trainium supports both lazy tensors (via XLA) and `torch.compile` (via torch-neuronx).

22.4.3 Graph Capture, Compilation, and Replay

These three concepts are frequently conflated but are distinct steps with different costs and benefits.

Definition 22.1 (Graph Capture, Compilation, and Replay)**. Step 1: Graph capture** records what operations to perform, in what order, with what tensor shapes—without executing them. Mechanisms: `torch.compile` (Dynamo traces Python), XLA lazy tensors, `torch_neuronx.trace`. Capture alone provides almost no performance benefit.

Step 2: Graph compilation (Layer 5) optimizes the captured graph: fusing operations, planning memory layout, scheduling communication. Compilation can take seconds to minutes, but produces much faster execution.

Step 3: Graph replay (Layer 2) re-executes a compiled graph with new input data—as long as tensor shapes are identical. On NVIDIA: CUDA Graphs. On AMD: hipGraphs. On TPU: XLA cached executables. On Trainium: Neuron NEFFs. Replay skips both capture and compilation entirely.

The three steps are cumulative: capture alone does little, capture + compilation gives moderate benefit, capture + compilation + replay gives the full benefit. For production serving, all three are essential.

Key Point 22.4 (Why Replay Requires Static Shapes)**.** *The compiled execution plan includes specific memory addresses, buffer sizes, and kernel launch configurations that depend on tensor dimensions. If you change the batch size from 16 to 17, every buffer size changes and the old plan is invalid. Shape bucketing (Section 22.5) addresses this by pre-compiling plans for a fixed set of shapes.*

22.4.4 Process Groups and Communicators

Multi-device collectives require process groups that define which devices participate together. In a TP-4, PP-2 deployment on 8 devices: two TP groups exist (devices $[0, 1, 2, 3]$ and $[4, 5, 6, 7]$). Four PP groups exist as device pairs across stages (e.g., $[0, 4]$, $[1, 5]$). Model code (Layer 7) defines group shapes; the serving framework (Layer 4) instantiates them; the communication library (Layer 3) automatically selects the best algorithm.

22.5 Layer 5: Graph Compilation

Graph compilation transforms a captured operation graph into an optimized execution plan. It controls which operations are fused, how memory is allocated and reused, how communication is scheduled relative to computation, and what IR is emitted for the device compiler.

22.5.1 Four Optimization Mechanisms

1. Host overhead elimination. Replaces per-op Python dispatch with a single graph launch. Most impactful for decode, where individual operations are tiny.

2. Memory traffic reduction via fusion. Combines operations that share intermediate tensors into fused kernels. Without fusion, each intermediate is written to HBM and read back. Fusion keeps intermediates in fast on-chip memory. Most impactful for prefill, where intermediate tensors are large.

3. Memory layout planning. Optimizes tensor arrangement in memory (row-major vs. column-major) to match downstream kernel access patterns. Poor layout can halve bandwidth utilization.

4. Communication scheduling. Overlaps collective communication with computation by scheduling them on separate hardware resources. Impactful whenever TP all-reduce is on the critical path.

22.5.2 Fusion Taxonomy

Vendor documentation uses "fusion" to mean three very different things. Confusing them leads to incorrect performance reasoning.

Operator fusion combines multiple logical operations into a single kernel launch. Example: GEMM $\rightarrow$ bias add $\rightarrow$ ReLU becomes one kernel. Intermediate results never leave on-chip memory.

Kernel fusion is a specialized, hand-written kernel implementing a complex multi-step computation internally. The canonical example is FlashAttention: instead of computing $\mathbf{QK}^\top$ (writing the enormous $N \times N$ attention matrix to HBM), applying softmax, then multiplying by $\mathbf{V}$, FlashAttention does all of this inside a single kernel by tiling the computation into blocks that fit in shared memory (228 KB on H100). For a 2048-token sequence, the naive attention matrix would be $2048 \times 2048 \times 2$ bytes = 8 MB—FlashAttention eliminates this entirely from HBM traffic.

Schedule overlap runs communication and computation simultaneously on different hardware resources. A GPU has Tensor Cores for compute and NVLink controllers for communication—physically different silicon that can operate in parallel. This is not fusion; the operations remain separate.

Example 22.2 (Layer Norm + Dropout Fusion)**.** Consider layer normalization followed by dropout on a tensor of shape $[32, 128, 768]$.

Unfused: Layer norm kernel reads/writes 3.1M elements; dropout kernel reads/writes 3.1M elements. Total memory traffic: 24.8 MB (FP16). Time: 0.18 ms.

Fused: Single kernel reads/writes 3.1M elements. Total traffic: 12.4 MB. Time: 0.10 ms. The 1.8× speedup comes from halving memory traffic. For a 12-layer transformer, this fusion appears 24 times per forward pass.

22.5.3 Static Shapes and Shape Bucketing

Graph replay requires that every tensor has exactly the same shape as when compiled. Real inference requests have variable sequence lengths. The solution is *shape bucketing*: the inference engine rounds each request's dimensions up to the nearest bucket boundary (e.g., $[64, 128, 256, 512, 1024, 2048, 4096]$). Each bucket has a pre-compiled graph. More buckets means less padding waste but a larger compile cache.

22.5.4 When Compilation Hurts

Compile time: initial compilation can take seconds to minutes. For short-running jobs, this overhead may dominate.

Graph breaks: if the compiler encounters an unsupported operation, it falls back to eager mode for that region. A single graph break can negate most compilation benefits.

Shape churn: on XLA and Neuron, every new tensor shape triggers a new compilation. Without bucketing, variable-length requests cause continuous recompilation.

For production serving, compilation costs are amortized over millions of requests and almost always worthwhile. For development and debugging, eager mode provides faster iteration.

22.6 Layer 4: Inference Engines

Inference engines are purpose-built LLM serving runtimes. They control which requests to process and when (admission and scheduling), how to group requests (batching), where to store the KV-cache and when to evict it (memory management), and how to distribute work across devices (orchestration). End-to-end gains from a well-configured inference engine are typically 3–20× over naive PyTorch inference.

22.6.1 Continuous Batching

In static batching, you collect N requests, pad them to the same length, process the entire batch, and wait for every request to finish. If one request generates 10 tokens and another generates 500, the first request's GPU resources sit idle until the 500-token request completes.

Continuous batching (also called iteration-level batching) checks the state of every request at every decode iteration. When a request finishes, the scheduler immediately inserts a new request into the freed slot. The GPU stays saturated instead of idling.

Scheduler Knobs

Max batch size limits concurrent active requests. During decode, each request contributes one token to the per-iteration GEMM. A batch of 16 means $[16 \times H]$; a batch of 256 means $[256 \times H]$. Increasing batch size improves arithmetic intensity until the GEMM becomes compute-bound. Too low: poor GPU utilization. Too high: KV-cache memory exhaustion and increased inter-token latency (ITL).

Max tokens per iteration limits total tokens (prefill + decode) per scheduler step. A large prefill can monopolize an iteration, spiking ITL for all active decode requests. Chunked prefill—splitting long prompts across iterations—is the standard solution.

Admission policy determines which waiting request to admit next. FIFO is simplest but causes head-of-line blocking. Priority queues enable SLO differentiation. Shortest-job-first minimizes average latency but requires length estimation.

Preemption policy determines what happens when KV-cache memory runs out. Options: swap KV to CPU (preserves state, costs PCIe bandwidth), recomputation (discards KV, re-prefills later), or request rejection.

22.6.2 PagedAttention: Virtual Memory for the KV-Cache

The KV-cache stores key and value vectors from every token processed so far. The naive approach allocates a contiguous block per request sized for the maximum context length, wasting 60–80% of KV memory on average.

Knob	Controls	Increase →	Decrease →	Trade-off
Max batch size	Concurrent requests	↑ Throughput, ↑ ITL	↓ Throughput, ↓ ITL	Throughput vs. latency
Max tokens/step	Prefill chunk size	↑ Prefill tput, ↑ ITL spikes	Better ITL protection	TTFT vs. ITL
Admission policy	Request ordering	(varies by policy)	(varies)	Fairness vs. efficiency
Preemption policy	Memory overflow	(swap/recompute/reject)	(varies)	Latency vs. memory

Table 22.6: Continuous batching scheduler knobs and their trade-offs.

PagedAttention, inspired by OS virtual memory, divides the KV-cache into small fixed-size pages (e.g., 16 tokens per page). A page table maps each request to its pages. As a request grows, new pages are allocated on demand. When a request finishes, pages are immediately freed.

Benefits: no internal fragmentation; memory utilization jumps dramatically; prefix caching allows shared system-prompt pages across requests.

Cost: attention kernels must follow the page table to find each KV block, adding pointer indirection. Optimized block-sparse attention kernels minimize this overhead.

22.6.3 vLLM and PagedAttention

vLLM is a production-grade LLM serving system that maximizes throughput by managing KV-cache memory with near-zero waste. Its core contribution is PagedAttention: instead of storing each sequence's KV-cache in a single contiguous buffer, vLLM partitions the cache into fixed-size "KV blocks" and maintains a block table mapping logical positions to physical blocks. This indirection enables fine-grained sharing and compaction.

During inference, new tokens are appended by allocating additional KV blocks and updating the block table, without relocating existing data. When a request finishes, its blocks are recycled immediately. Empirical evaluations show vLLM delivers 2–4× higher throughput than earlier systems at similar latency, with gains increasing for longer contexts.

22.6.4 Multi-Device Orchestration

In vLLM, linear layers are wrapped in TP-aware modules at model load time, weights are sharded across devices, and all-reduce calls are inserted after each row-parallel GEMM. For decode's tiny all-reduce messages (∼16 KB), vLLM includes custom small-message kernels that bypass general-purpose NCCL overhead. The engine architecture (scheduler, cache manager, router) is backend-agnostic: vLLM supports CUDA, ROCm, and TPU backends.

22.6.5 Request Lifecycle

1. **Admission:** scheduler checks KV-cache page availability and batch capacity. If insufficient, queues, preempts, or rejects.
2. **Prefill:** full prompt processed in one forward pass. Large, compute-heavy GEMMs. KV-cache pages allocated as KV vectors are computed.
3. **Decode loop:** one token per iteration. Scheduler selects batch → graph replay launches compiled decode graph → local GEMMs and attention → all-reduces synchronize TP → logits sampled.
4. **Completion:** KV-cache pages reclaimed, batch slot freed, next waiting request admitted.

22.6.6 Throughput vs. Latency Tuning Profiles

22.7 Layer 3: Device Kernels and Collectives

This layer provides hardware-tuned implementations of compute (GEMM, attention) and communication (all-reduce, all-to-all) operations. It controls memory access patterns, on-chip data reuse, instruction scheduling, and collective algorithms.

Dimension	Interactive (latency-optimized)	Batch (throughput-optimized)
Target metric	TTFT, ITL	Tokens/second/dollar
Batch size	Small (1–16)	Large (64–512)
TP degree	Moderate (TP-2 or TP-4)	Higher OK (TP-4 or TP-8)
Graph replay	Essential for decode	Helpful, less critical
Quantization	Conservative (FP8 or BF16)	Aggressive (FP8 + KV quant)
KV-cache	PagedAttention + prefix caching	PagedAttention + large page pool

Table 22.7: Tuning profiles for interactive vs. batch workloads on the same hardware.

22.7.1 The Memory Hierarchy Problem

The gap between an untuned implementation and a hardware-optimized kernel is often 10–20× for GEMM. The reason is the memory hierarchy:

Level	H100	MI300X	TPU v5p	Relative Cost
Registers	256KB/SM	Similar/CU	VRF/MXU	1× (fastest)
Shared/Local memory	228KB/SM	64KB LDS/CU	CMEM (on-chip)	∼5–10× slower
L2 cache	50MB	256MB	Compiler-managed	∼20–50× slower
HBM (main memory)	80GB, 3.35TB/s	192GB, 5.3TB/s	95GB, 2.76TB/s	∼100–500× slower

Table 22.8: Memory hierarchy across accelerator platforms. The actual arithmetic is fast; moving data from HBM to compute cores is what costs time.

A naive matrix multiplication loads each element from HBM multiple times. A tuned kernel *tiles* the computation: it loads a block into fast shared memory, does all multiplications that need that block, then loads the next block. FlashAttention is the poster child: standard attention writes the $N \times N$ score matrix to HBM and reads it back multiple times. FlashAttention tiles all three steps (Q×K, softmax, ×V) so the score matrix never leaves shared memory.

22.7.2 Compute-Bound vs. Memory-Bound: The Roofline

Definition 22.2 (Arithmetic Intensity and the Roofline)**.** Arithmetic intensity measures FLOPs per byte loaded from memory:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes transferred}} \tag{22.1}$$

A kernel is compute-bound if its arithmetic intensity exceeds the hardware's ops:byte ratio; memory-bound otherwise. For H100: 990 TFLOPS/3.35 TB/s ≈ 295 FLOPs/byte.

Example 22.3 (Decode vs. Prefill Roofline Analysis)**. Decode** ($B = 1$): GEMM $[1 \times 8192] \times [8192 \times 8192]$. Compute: $2 \times 8192^2 = 134$M FLOPs. Weight matrix: 128 MB. Intensity ≈ 1 FLOP/byte. Massively below the 295 balance point—*entirely memory-bandwidth-bound.* Tensor Cores finish instantly, then sit idle waiting for weights.

Prefill ($B = 256$): GEMM $[256 \times 8192] \times [8192 \times 8192]$. Compute: 34.4B FLOPs. Weights still 128 MB. Intensity ≈ 269 FLOPs/byte—near the balance point. Compute and memory are roughly balanced.

This is why FP8 helps prefill much more than decode: prefill is near-compute-bound, so doubling TFLOPS roughly doubles throughput. Decode is bandwidth-bound, so doubling TFLOPS does nothing.

22.7.3 Coalesced Memory Access and Bank Conflicts

When threads in a warp access global memory, the hardware attempts to combine accesses into a single transaction. Coalesced access (consecutive threads accessing consecutive locations) allows one 128-byte transaction instead of 32 separate ones. For transformer operations, ensuring consecutive threads load consecutive elements of the query matrix allows full bandwidth utilization.

Shared memory is divided into 32 banks. When multiple threads access the same bank at different addresses, a bank conflict serializes the accesses. Careful padding of shared memory tiles eliminates conflicts.

Example 22.4 (Shared Memory Optimization in Attention)**.** Computing $\mathbf{A} = \mathbf{Q}\mathbf{K}^\top$ with tiles loaded into shared memory:

Unoptimized: Tile 64×64 FP16 values (8192 bytes). 32-way bank conflicts when accessing columns. Effective bandwidth: 50 GB/s (3% of peak).

Optimized with padding: Tile 64×72 FP16 values (9216 bytes, 8 elements padding per row). No bank conflicts. Effective bandwidth: 1.4 TB/s (88% of peak). The 12.5% increase in shared memory usage yields 28× bandwidth improvement.

22.7.4 Flash Attention

Standard attention implementations compute $\mathbf{A} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{A}$ to HBM, read it back for softmax, write the result, read it again for multiplication with $\mathbf{V}$. This results in $O(n^2)$ memory reads and writes.

Flash Attention restructures the computation to work on tiles that fit in shared memory, computing attention for one tile at a time and only writing the final output. This reduces memory traffic from $O(n^2)$ to $O(n)$.

Example 22.5 (Flash Attention Performance)**.** BERT-base with sequence length 512, batch size 16, on A100 GPU:

Standard attention: 48 GB memory traffic, 8.2 ms, 45% bandwidth utilization.

Flash Attention: 12 GB traffic (4× reduction), 3.8 ms (2.2× speedup), 82% utilization.

At sequence length 2048: 3.5× speedup. At 8192: 5.2× speedup while also enabling sequences that would otherwise exceed memory.

22.7.5 Collective Communication

All-reduce, all-gather, reduce-scatter, all-to-all, and point-to-point are the five collective primitives. Each platform implements them through specialized libraries (see Table 22.2). The library automatically selects the best algorithm: latency-optimized for small messages (decode's 16 KB), bandwidth-optimized (ring or tree) for large messages (prefill's multi-MB).

22.7.6 Kernel Failure Modes

Shape misalignment: dimensions not aligned to tile sizes (e.g., not multiples of 128) cause padding waste or fallback to slower kernels.

Tiny messages: all-reduce for small decode tensors is latency-dominated regardless of bandwidth.

Register pressure: aggressively fusing many operations increases register usage, potentially reducing occupancy and overall throughput.

Local speedup, global bottleneck: a 2× faster kernel does not help if the bottleneck is scheduler overhead (Layer 4) or interconnect latency (Layer 1). Always verify that optimizing a kernel helps the end-to-end metric.

22.8 Layers 2 and 1: Device Compiler and Accelerator Hardware

22.8.1 Layer 2: Device Compiler and Runtime

Layer 2 takes the optimized graph from Layer 5 and produces native machine instructions. It controls register allocation, instruction scheduling, memory coalescing, occupancy tuning, and execution dispatch.

Aspect	NVIDIA CUDA	AMD ROCm	Google TPU	AWS Trainium
Compiler	nvcc → PTX → SASS	hipcc → LLVM → AMDGPU	XLA HLO → TPU ISA	Neuron Compiler
User control	High	High	Low (XLA manages)	Low (Neuron manag
Execution	CUDA streams + Graphs	HIP streams + hipGraphs	XLA scheduling	Neuron runtime + N
Profiling	Nsight, nvprof	rocprof, ROCm SMI	XLA/TPU profiler	Neuron monitor

Table 22.9: Device compiler and runtime across platforms.

Most Layer 2 effects are influenced indirectly via Layer 4/5 choices (graph structure, shape regularity), not by direct user intervention.

22.8.2 Layer 1: Accelerator Hardware

Hardware sets the ceiling on execution efficiency. Software stack maturity can dramatically affect how much of the ceiling is actually reached—a mature stack may achieve 60–70% of peak while an immature one achieves 20–30%.

Spec (per chip)	H100 SXM	B200 SXM	MI300X	TPU v5p	Trn2
Compute units	132 SMs	192 SMs	304 CUs	4 MXUs/chip	32 NeuronCores
BF16 peak TFLOPS	990	∼2,250	1,307	∼459	∼380
HBM capacity	80 GB HBM3	192 GB HBM3e	192 GB HBM3	95 GB HBM2e	96 GB HBM
HBM bandwidth	3.35 TB/s	8 TB/s	5.3 TB/s	2.76 TB/s	∼1.6 TB/s
FP8 support	Native	Native	Native	INT8/FP8-like	Native FP8

Table 22.10: Accelerator hardware comparison (February 2026 snapshot). All values are per-chip peak specifications. Sustained performance is typically 40–70% of peak.

22.8.3 How to Read Spec Sheets Without Fooling Yourself

Peak TFLOPS ≠ LLM performance. Decode is memory-bandwidth-bound. A chip with 2× the TFLOPS but similar HBM bandwidth will not decode 2× faster.

HBM bandwidth matters, but access patterns matter more. Coalesced sequential access achieves near-peak bandwidth. Random or strided access may achieve only 30–50% of peak.

Interconnect latency dominates decode TP performance. During decode, all-reduce messages are tiny (∼16 KB). A 2× higher-bandwidth interconnect with the same latency provides zero decode benefit.

Software stack maturity can outweigh raw specs. A chip with inferior specs but a mature, optimized software stack often outperforms a faster chip with immature software.

22.8.4 Workload-to-Hardware Fit

22.9 Quantization: Precision, Performance, and Trade-offs

22.9.1 Quantization Taxonomy

Quantization means using fewer bits to represent numbers. Standard transformer inference uses BF16 (2 bytes per number). Quantization reduces this to 8 bits (FP8 or INT8, 1 byte) or even 4 bits (INT4,

Workload Profile	Key Hardware Dimension	Why
Decode-heavy, low-latency	Interconnect latency + HBM BW	Weight loading + AR latency dominate
Long-context, high-concurrency	HBM capacity + KV strategy	KV-cache dominates memory
Prefill-heavy, batch throughput	Compute TFLOPS + HBM BW	Large GEMMs are compute-bound
MoE with EP	Bisection bandwidth + topology	All-to-all needs every device to talk

Table 22.11: Matching workload profiles to the hardware dimension that matters most.

0.5 bytes). Three potential benefits: less memory used, less data to move, and more operations per clock cycle.

Crucially, quantization is not one technique—it is a family targeting different tensors:

Weight quantization (GPTQ, AWQ, GGML): reduces weight memory footprint. Decode benefits most: less data to load from HBM per token. Does NOT reduce communication volume because activations being all-reduced remain at higher precision.

Activation quantization (FP8 compute): uses FP8 for the actual matrix multiplication. Benefits both compute throughput (2× TFLOPS on FP8-capable hardware) and communication volume. Most impactful for prefill because prefill is compute-bound.

KV-cache quantization (INT8/FP8 KV): stores KV vectors at lower precision. Cuts KV memory per token in half, enabling more concurrent requests or longer contexts.

Communication precision: running all-reduce at FP8 instead of BF16 halves data volume. Most relevant during prefill where messages are large.

22.9.2 Quantization Fundamentals

Definition 22.3 (Quantization)**.** Map floating-point weights to lower precision:

$$w_{\text{quant}} = \text{round}\left(\frac{w_{\text{float}}}{s}\right) + z \tag{22.2}$$

where s is the scale factor and z is the zero-point.

Example 22.6 (INT8 Quantization)**. FP32 weight:** $w = 0.137$. **Range:** $w \in [-1.0, 1.0]$. **Scale:** $s = 2.0/256 = 0.0078125$.

Quantize:

$$w_{\text{INT8}} = \text{round}\left(\frac{0.137}{0.0078125}\right) = \text{round}(17.54) = 18 \tag{22.3}$$

Dequantize: $w' = 18 \times 0.0078125 = 0.1406$. **Error:** $|0.137 - 0.1406| = 0.0036$ (2.6% relative).

22.9.3 Post-Training Quantization (PTQ)

PTQ takes a fully trained FP32 model and converts it to lower precision through a calibration process. A small calibration dataset (a few hundred to a few thousand examples) is processed in FP32 while collecting activation statistics. These statistics determine scale factors mapping the continuous FP32 range to the discrete INT8 range.

The primary advantage is simplicity: no retraining required. Modern frameworks provide built-in PTQ APIs. The limitation: quantization errors accumulate through deep networks. Layers with wide activation distributions or outlier values are especially sensitive.

22.9.4 Quantization-Aware Training (QAT)

QAT incorporates quantization effects directly into training. In the forward pass, weights and activations are quantized to the target precision using the same formula as inference. The backward pass operates in full FP32 precision. This allows the optimizer to adjust weights to positions robust to rounding and learn activation distributions that quantize well.

Example 22.7 (BERT-base Quantization Results).

Precision	GLUE Score	Speedup
FP32 (baseline)	84.5	1.0×
FP16	84.4	1.8×
INT8 (PTQ)	82.1	2.9×
INT8 (QAT)	84.2	2.9×

QAT recovers most accuracy lost in PTQ while maintaining the same speedup.

22.9.5 Accuracy and Stability Trade-offs

Calibration: FP8 and INT8 require scaling factors derived from representative data. Poor calibration clips important values.

Outlier sensitivity: some models have occasional very large activation values critical for quality. Techniques like SmoothQuant redistribute outlier magnitude from activations to weights before quantizing.

Accumulation precision: FP8 multiply with FP32 accumulate is generally safe; FP8 accumulate can cause compounding rounding errors.

Model-specific: some architectures (e.g., Llama family) tolerate FP8 well. Others show noticeable quality regression. Always evaluate on your specific model and task.

22.9.6 Quantization Decision Heuristics

Bottleneck	Strategy	Why
Memory capacity (weights)	Weight-only quant (INT4/INT8)	Reduces weight memory directly
Memory capacity (KV-cache)	KV quantization (INT8/FP8)	Frees memory for more requests
Compute throughput (prefill)	FP8 activation + compute	2× TFLOPS on FP8 hardware
TP communication (BW-bound)	FP8 communication	Halves all-reduce volume
Quality-sensitive	Start BF16, add conservatively	Measure regression first

Table 22.12: Quantization strategy selection based on the dominant bottleneck.

22.10 Model Pruning

22.10.1 Pruning Strategies

Model pruning removes parameters to reduce computational and memory requirements. Two main approaches exist:

Magnitude-based pruning removes weights with small absolute values: if $|w_{ij}| < \tau$, the weight is set to zero. The threshold τ can be set globally, per-layer, or adaptively.

Structured vs. unstructured pruning. Unstructured pruning removes individual weights, creating sparse matrices that require specialized sparse kernels to realize speedups. Structured pruning removes entire architectural units (neurons, attention heads, layers), directly reducing matrix dimensions visible to standard dense libraries. Structured pruning achieves lower compression ratios but provides immediate speedups without custom kernels.

22.10.2 Iterative Pruning

Algorithm 1: Iterative Magnitude Pruning

Input: Model, sparsity target s_{target}
for *sparsity* $s = 0$ **to** s_{target} *by steps* **do**
    Train model to convergence
    Prune Δs lowest-magnitude weights
    Fine-tune model

Example 22.8 (Attention Head Pruning)**.** BERT-base: 12 layers × 12 heads = 144 heads.
Finding: Can remove 50% of heads with minimal impact.
Procedure: (1) Compute importance score per head. (2) Rank by importance. (3) Prune lowest 50% (72 heads). (4) Fine-tune.
Result: 50% fewer attention operations. GLUE score: 84.5 → 83.8 (0.7 point drop). 1.5× faster inference.

22.11 Knowledge Distillation

Definition 22.4 (Distillation Objective)**.**

$$\mathcal{L} = \alpha \mathcal{L}_{\text{CE}}(y, y_{\text{student}}) + (1 - \alpha)\mathcal{L}_{\text{KD}}(y_{\text{teacher}}, y_{\text{student}}) \tag{22.4}$$

where:

$$\mathcal{L}_{\text{KD}} = \text{KL}\left(\frac{\exp(z_t/T)}{\sum \exp(z_t/T)} \middle\| \frac{\exp(z_s/T)}{\sum \exp(z_s/T)}\right) \tag{22.5}$$

T = temperature (typically 2–5), higher = softer probabilities.

22.11.1 DistilBERT Approach

DistilBERT reduces depth by half (6 layers vs. BERT-base's 12) while maintaining hidden size 768. The student is initialized by copying every other layer from the teacher. Training combines three losses: distillation loss (match teacher's output distribution), masked language modeling loss, and cosine distance loss between hidden states.

Results: 66M parameters (vs. 110M), ∼60% faster inference, retains 97% of BERT's GLUE performance. This combination of compression, speedup, and accuracy retention makes DistilBERT compelling for production deployments.

22.12 Bottleneck-Driven Optimization

Do not apply optimizations by popularity. Apply them by bottleneck. A 10× kernel speedup is worthless if the bottleneck is scheduling overhead. A perfect scheduler is worthless if the bottleneck is interconnect latency. The single most important skill in inference optimization is correctly identifying where time is actually being spent.

22.12.1 Bottleneck Taxonomy

Every inference workload is dominated by one (occasionally two) bottleneck regimes. Identifying the regime tells you which family of optimizations will help:

Bottleneck Regime	Symptom	Dominant In
Compute-bound	Utilization >85%; more batch doesn't help	Prefill with large batches
HBM bandwidth-bound	Low utilization despite available work	Decode (single-token GEMMs)
Interconnect BW-bound	Large collective messages, high link utilization	Prefill TP all-reduce, EP a2a
Interconnect latency-bound	Small messages, latency dominates	Decode TP all-reduce
Host/runtime overhead-bound	Low utilization, dispatch visible in profile	Decode without graph replay
Memory capacity-bound	OOM or aggressive KV eviction	Long context + high concurrency
Scheduler/policy-bound	Low utilization despite available requests	Bursty traffic, poor scheduling

Table 22.13: Bottleneck taxonomy for LLM inference workloads.

22.12.2 Diagnosis Workflow

A systematic approach to optimization:

1. **Define the target metric.** Latency (TTFT, ITL, P50, P99)? Throughput (tokens/s)? Cost (tokens/dollar)?
2. **Segment the workload.** What fraction is prefill vs. decode? What is the sequence length distribution?
3. **Profile.** Use vendor profilers (Nsight, rocprof, XLA profiler, Neuron profiler) to measure host time, kernel time, communication time, idle time, memory utilization.
4. **Classify the bottleneck.** Match profiler signals to the taxonomy above.
5. **Choose the optimization family.** Compute-bound → FP8, kernel optimization. BW-bound → quantization, memory access optimization. Latency-bound → graph replay, custom collectives. Overhead-bound → compilation, better scheduling. Capacity-bound → quantization, KV optimization, more devices.
6. **Re-measure.** Verify the optimization helped end-to-end. A local improvement that shifts the bottleneck elsewhere may show no system-level gain.

22.12.3 Optimization Impact Summary

Optimization	Layer	Gain	Bottleneck Relieved	Helps	Trade-off
`torch.compile`/XLA	5	1.5–3×	Host overhead	Decode > prefill	Compile time; gra
FlashAttention	3	2–4× attn	HBM BW (attention)	Prefill > decode	Kernel compatibili
Continuous batching	4	3–8× tput	Scheduler/utilization	Both	Tail latency risk
PagedAttention	4	2–4× concurrency	Memory capacity (KV)	Both	Indirection overhe
Fused GEMM+AllReduce	3	1.3–1.8× TP	Interconnect (TP path)	Prefill > decode	Kernel complexity
FP8 compute	3+1	1.5–2×	Compute + HBM BW	Prefill > decode	Accuracy risk
Graph replay	2	1.2–2× decode	Host overhead	Decode ≫ prefill	Static shape requi

Table 22.14: Optimization impact summary. All gain values are representative ranges from published benchmarks, not universal guarantees.

22.12.4 Case Study: Interactive Chatbot (Decode-Heavy)

Target: P50 inter-token latency < 30 ms. 70B model on 8×H100. Decode dominates because each response generates hundreds of tokens.

Bottleneck analysis: per-token GEMMs are memory-bandwidth-bound. TP all-reduce is latency-bound (16 KB messages). Host dispatch overhead is significant without graph replay.

Priority 1 (Layer 4): continuous batching with moderate batch sizes (8–32).

Priority 2 (Layers 5/2): graph compilation + CUDA Graph replay for decode.

Priority 3 (Layer 3): custom small-message all-reduce bypassing NCCL overhead.

Priority 4 (Layer 7): TP-4 rather than TP-8. Each additional TP degree adds serialized all-reduce latency per token.

22.12.5 Case Study: Batch Document Summarization (Prefill-Heavy)

Target: maximize tokens/second/dollar. 70B model on 8×H100. Processing thousands of long documents offline.

Bottleneck analysis: large GEMMs are compute-bound. TP all-reduce messages are large (bandwidth-relevant). Memory capacity constrains batch size.

Priority 1 (Layer 4): continuous batching with large batches (128–512).

Priority 2 (Layer 3): FP8 GEMMs. 2× compute throughput maps directly to ∼2× prefill throughput.

Priority 3 (Layer 3): fused GEMM+all-reduce. Large prefill GEMMs provide enough compute time to overlap communication.

Priority 4: KV-cache quantization (FP8 KV) to free memory for more concurrent documents.

22.13 Production Deployment

22.13.1 Serving Frameworks

TorchServe provides native PyTorch serving with REST/gRPC APIs, dynamic batching, model versioning, and integrated monitoring. Natural deployment path for PyTorch-trained models.

Triton Inference Server supports models from PyTorch, TensorFlow, ONNX, and TensorRT within a single infrastructure. Sophisticated scheduling can execute multiple models concurrently on the same GPU.

ONNX Runtime provides framework-agnostic inference with graph-level optimizations (operator fusion, constant folding, dead code elimination). Typically delivers 1.5–2× speedup over PyTorch inference; combined with INT8 quantization, 3–4×. See Chapter **??** for ONNX export code.

TensorRT leverages detailed knowledge of NVIDIA GPU architectures for aggressive layer fusion, kernel auto-tuning, and INT8 calibration. For BERT-base: 2–3× speedup over PyTorch in FP16; 4–5× with INT8.

22.13.2 Distributed Inference with Ray and Kubernetes

When inference workloads exceed single-machine capacity, they are typically deployed on a Kubernetes cluster with Ray Serve handling request routing and autoscaling. A Ray cluster runs inside Kubernetes: a head pod manages state, worker pods host GPU-backed model replicas. Kubernetes provisions and recovers pods; Ray's autoscaler adjusts replica count based on request rate and queue length.

Designing such a system requires coordinating multiple scaling layers. Each replica needs appropriate concurrency limits and batch sizes to avoid GPU memory oversubscription. Ray's autoscaler must react quickly to sustained load without oscillations. Kubernetes node autoscaling must align with Ray's resource requests. Observability across all layers is essential for diagnosing stalled scaling, replica imbalance, or OOM events.

22.13.3 Deployment Checklist

Performance optimization: quantize to INT8/FP16 where accuracy permits; export to ONNX/TensorRT for graph-level optimization; tune batch size for latency–throughput balance; enable KV caching for autoregressive generation.

Reliability engineering: distinguish transient vs. permanent failures with appropriate retry logic; set request timeouts; implement health checks for load balancer integration; enable model versioning for safe rollouts.

Monitoring and observability: track latency at p50/p95/p99 (not just averages); measure throughput in requests/second; monitor GPU utilization to identify headroom; break down error rates by type.

22.13.4 Hardware Selection

CPU vs. GPU Trade-offs

CPUs suit small models (<100M parameters) with batch size 1–4 and latency-critical single-request scenarios. A distilled BERT (66M parameters) achieves 15 ms latency on modern CPU with batch size 1.

GPUs become the clear choice for training (any size), large models (>100M), batch inference (batch > 8), and throughput-oriented applications. BERT-large achieves 2.5 seq/s on CPU vs. 45 seq/s on T4 GPU (18× advantage).

Cost analysis often favors GPUs despite higher instance cost: T4 GPU at $0.35/hour processes 162K sequences/hour ($0.0000022/seq) vs. CPU at $0.10/hour processing 9K sequences/hour ($0.000011/seq)—GPU is 5× more cost-effective.

Training Hardware Selection

- **<1B parameters:** single V100 or A100. BERT-base trains in ∼3 days on V100.
- **1–10B parameters:** 4–8 A100 GPUs with NVLink. GPT-2 (1.5B) trains in ∼2 weeks on 8×A100.
- **10–100B parameters:** 16–64 A100 GPUs with NVLink + InfiniBand. Requires pipeline and tensor parallelism.
- **>100B parameters:** hundreds to thousands of GPUs. Training costs reach millions of dollars.

Inference Hardware Selection

- **Batch throughput:** A100 or A10 GPUs with large batches (32–128). BERT-large: 520 seq/s on A100 with batch 64.
- **Low-latency:** T4 or A10 with TensorRT + INT8. BERT-base: 5 ms latency on T4 with batch 1.
- **Cost-optimized:** CPU or T4 with quantized models. Distilled BERT on CPU: ∼$0.000005/inference.
- **Edge:** mobile CPUs, edge TPUs, Jetson. MobileBERT (25M params): 30 ms on mobile CPU.

22.14 Key Takeaways and Engineering Decision Checklist

22.14.1 Principles

1. Parallelism at Layer 7 is typically the most consequential decision. It determines communication patterns, memory distribution, and constraints on every layer below.

2. The top of the stack is portable. Layers 7–5 work across vendors. Invest in understanding these first.

3. Layers 4 and 3 often deliver the largest end-to-end gains. Inference engine optimizations and custom kernels provide the biggest multipliers, but only after parallelism and compilation are correctly configured.

4. Communication follows the Bandwidth Ladder. TP on fastest link, PP tolerates slower links, EP needs bisection bandwidth.

5. Diagnose before optimizing. Profile to classify the bottleneck regime. Apply the optimization family that addresses that regime. Re-measure.

6. Numbers require context. Every speedup claim is scoped to a workload phase, hardware configuration, and software version.

22.14.2 Engineering Decision Checklist

1. What is the objective metric? Latency (TTFT, ITL)? Throughput (tokens/s)? Cost (tokens/-dollar)?
2. Is the workload prefill- or decode-dominated?
3. What is the bottleneck regime? Profile first.
4. Is parallelism required for fit or chosen for performance?
5. Is shape regularity sufficient for graph replay? If not, implement bucketing.
6. Is the interconnect appropriate for TP/EP?
7. What precision trade-off is acceptable? Measure accuracy impact before committing.
8. Does the optimization help end-to-end? Always measure the system-level metric.

22.15 Worked End-to-End Token Trace

One complete decode step for a 70B model on TP-4 (4×H100, NVSwitch). BF16. $B = 16$ concurrent requests.

#	Layer	What Happens	Time (illustrative)	Bottleneck
1	4: Scheduler	Select batch of 16 reqs, allocate KV pages	∼10–50 μs	Host overhead
2	5/2: Replay	Launch compiled decode graph for $[B=16, S=1]$	∼5–10 μs	Host overhead
3	3: GEMM	QKV projection (column-parallel)	∼15–30 μs	HBM BW-bound
4	3: Attention	FlashAttention: 8 local heads, KV-cache lookup	∼10–40 μs	HBM BW-bound
5	3: GEMM+AR	Output proj (row-par) + all-reduce 256 KB	∼15–30 + 2–5 μs	HBM BW + latency
6	3: FFN	Gate+Up (col-par), SwiGLU, Down (row-par) + AR	∼30–60 + 2–5 μs	HBM BW-bound
7	Repeat	×80 layers	∼5–15 ms total	Accumulated
8	4: Complete	Detokenize, check stop, reclaim KV	∼5–20 μs	Host overhead

Table 22.15: End-to-end decode step trace. Decode is dominated by memory-bandwidth-bound GEMMs with small but frequent latency-bound all-reduces, bookended by host scheduling overhead.

22.16 Exercises

Exercise 22.1. Quantize BERT-base to INT8:

1. Use PyTorch quantization APIs
2. Calibrate on 1000 examples
3. Measure: (a) Model size, (b) Inference speed, (c) GLUE accuracy
4. Compare PTQ vs QAT

Exercise 22.2. Implement attention head pruning:

1. Compute importance scores for all heads
2. Prune 25%, 50%, 75% of heads

3. Fine-tune after pruning
4. Plot accuracy vs sparsity

Exercise 22.3. Optimize inference pipeline:

1. Baseline: PyTorch FP32
2. Convert to ONNX, measure speedup
3. Apply INT8 quantization
4. Implement dynamic batching
5. Report final throughput improvement

Exercise 22.4. Analyze the prefill vs. decode asymmetry:

1. For a model with $H = 4096$ and 32 layers, compute the arithmetic intensity of a single-token decode GEMM and a 512-token prefill GEMM
2. Determine whether each is compute-bound or memory-bound on an H100 (3.35 TB/s, 990 TFLOPS)
3. Calculate the theoretical speedup from switching to FP8 (2× TFLOPS) for each phase
4. Explain why the speedup differs between phases

Exercise 22.5. Tensor parallelism communication analysis:

1. For a 70B model with $H = 8192$ and 80 layers, calculate the total all-reduce data volume per decode token with TP-4 in BF16
2. Calculate the communication time assuming NVLink latency of 2 μs per all-reduce
3. Repeat for InfiniBand at 8 μs per all-reduce
4. If decode compute takes 10 ms per token, what percentage overhead does communication add in each case?
5. At what TP degree does InfiniBand communication overhead exceed 50% of compute time?

Exercise 22.6. Flash Attention implementation study:

1. Implement standard attention with separate kernels
2. Analyze memory traffic for different sequence lengths
3. Study Flash Attention paper and implementation
4. Benchmark Flash Attention vs standard attention

5. Plot speedup as a function of sequence length

Exercise 22.7. Inference engine configuration:

1. Deploy a 7B model using vLLM with default settings
2. Measure throughput and P50/P99 latency under load
3. Experiment with max batch size: 1, 8, 32, 128
4. Enable PagedAttention and measure memory utilization improvement
5. Compare static batching vs. continuous batching throughput
6. Identify the bottleneck regime at each batch size using profiling

Exercise 22.8. Bottleneck diagnosis:

1. Choose a transformer model and deployment scenario (interactive chatbot or batch processing)
2. Profile the deployment using available tools
3. Classify the bottleneck using the taxonomy in Table 22.13
4. Select and apply the appropriate optimization from Table 22.14
5. Measure end-to-end improvement and verify the bottleneck shifted
6. Document the before/after metrics and explain why the optimization helped

Exercise 22.9. Hardware selection analysis:

1. Choose a transformer model and deployment scenario
2. Estimate throughput requirements and latency constraints
3. Compare cost per inference for CPU, T4, and A100
4. Calculate break-even point where GPU becomes cost-effective
5. Recommend hardware configuration with justification

22.17 Solutions

Full solutions for all exercises are available at `https://deeplearning.hofkensvermeulen.be`.