

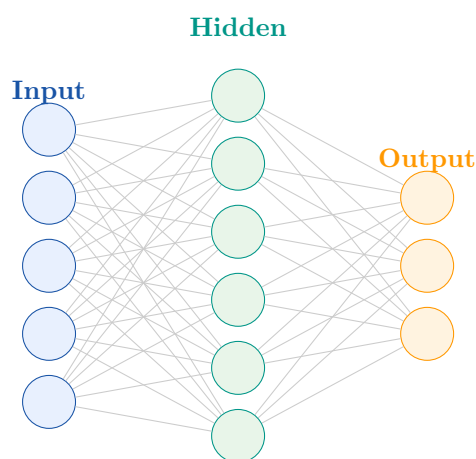
---

# Efficient LLM Inference

## Systems, Algorithms & Production Engineering

### Interview Pocket Notes

---



### Author

**AI Engineering Insider**

Lamhot Siagian

<https://www.linkedin.com/in/lamhotsiagian>

Second Edition - 2026

---

*Covering Quantization, Speculative Decoding, KV Cache, FlashAttention,  
vLLM, MoE, Edge Inference, Production Systems & Inference-Time Scaling*

---

# Contents

<b>1</b>	<b>The Inference Problem</b>	<b>1</b>
1.1	Training vs. Inference: Fundamentally Different Goals . . . . .	1
1.2	The Token Generation Loop Explained . . . . .	1
1.3	Latency, Throughput, and Cost: The Iron Triangle . . . . .	2
1.4	Memory Bandwidth: The True Bottleneck . . . . .	2
1.5	The Roofline Model . . . . .	2
<b>2</b>	<b>Hardware Foundations for Inference</b>	<b>5</b>
2.1	GPU Architecture for the Inference Engineer . . . . .	5
2.2	Tensor Cores and Mixed Precision . . . . .	5
2.3	Memory Hierarchy and Bandwidth . . . . .	5
2.4	CPU and Edge Hardware . . . . .	6
2.5	Emerging Hardware Trends . . . . .	6
<b>3</b>	<b>Transformer Inference Mechanics</b>	<b>9</b>
3.1	Anatomy of a Transformer Forward Pass . . . . .	9
3.2	The KV Cache: What It Is and Why It Matters . . . . .	9
3.3	Prefill vs. Decode: Two Very Different Workloads . . . . .	10
3.4	Attention Variants: MHA to MLA . . . . .	10
3.5	Positional Encodings at Inference Time . . . . .	10
<b>4</b>	<b>Quantization</b>	<b>13</b>
4.1	Why Quantization Matters . . . . .	13
4.2	Number Formats . . . . .	13
4.3	Post-Training Quantization Methods . . . . .	13
4.3.1	GPTQ . . . . .	13
4.3.2	AWQ (Activation-Aware Weight Quantization) . . . . .	14
4.3.3	QuIP# and AQLM . . . . .	14
4.4	Quantization-Aware Training (QAT) . . . . .	14
4.5	Activation Quantization Challenges . . . . .	14
<b>5</b>	<b>Speculative Decoding</b>	<b>17</b>
5.1	The Core Insight: Draft and Verify . . . . .	17
5.2	Token Tree Verification . . . . .	17
5.3	Medusa: Multi-Head Speculative Decoding . . . . .	17
5.4	EAGLE: Efficient Autoregressive Generation . . . . .	18
5.5	When Speculative Decoding Helps (and When It Doesn't) . . . . .	18
<b>6</b>	<b>KV Cache Optimization</b>	<b>20</b>
6.1	The KV Cache Memory Crisis . . . . .	20
6.2	PagedAttention: Virtual Memory for KV Caches . . . . .	20
6.3	Prefix Caching (Prompt Caching) . . . . .	20
6.4	StreamingLLM: The Attention Sink Phenomenon . . . . .	21
6.5	KV Cache Quantization . . . . .	21

<b>7</b>	<b>Kernel Engineering and FlashAttention</b>	<b>23</b>
7.1	Why Custom CUDA Kernels Matter . . . . .	23
7.2	FlashAttention: IO-Aware Exact Attention . . . . .	23
7.3	The Online Softmax Trick . . . . .	24
7.4	Triton: Pythonic Kernel Programming . . . . .	24
<b>8</b>	<b>Serving Systems Architecture</b>	<b>27</b>
8.1	Anatomy of a Production Inference Server . . . . .	27
8.2	Batching Strategies . . . . .	27
8.2.1	Static Batching . . . . .	27
8.2.2	Continuous (Iteration-Level) Batching . . . . .	27
8.2.3	Chunked Prefill . . . . .	27
8.3	Disaggregated Prefill and Decode . . . . .	27
8.4	SLO-Aware Scheduling . . . . .	28
<b>9</b>	<b>Parallelism for Large Model Inference</b>	<b>30</b>
9.1	Why Parallelism Is Necessary . . . . .	30
9.2	Tensor Parallelism . . . . .	30
9.3	Pipeline Parallelism . . . . .	30
9.4	Mixture of Experts Inference . . . . .	30
<b>10</b>	<b>Long-Context and Memory Management</b>	<b>33</b>
10.1	The Quadratic Attention Problem . . . . .	33
10.2	FlashAttention for Long Context . . . . .	33
10.3	Ring Attention . . . . .	33
10.4	Context Compression . . . . .	33
10.5	State Space Models as Linear Attention . . . . .	33
<b>11</b>	<b>Edge and On-Device Inference</b>	<b>36</b>
11.1	The Edge Inference Landscape . . . . .	36
11.2	Memory Constraints and Quantization for Edge . . . . .	36
11.3	Runtime Frameworks . . . . .	36
11.4	Neural Architecture Search for Efficiency . . . . .	36
<b>12</b>	<b>Inference-Time Compute Scaling</b>	<b>39</b>
12.1	The New Scaling Paradigm . . . . .	39
12.2	Chain-of-Thought and Its Inference Cost . . . . .	39
12.3	Process Reward Models and Search . . . . .	39
12.4	DeepSeek-R1 and o1-Style Architectures . . . . .	40
<b>13</b>	<b>Observability and Production Engineering</b>	<b>42</b>
13.1	The Production Inference Stack . . . . .	42
13.2	Key Metrics in Depth . . . . .	42
13.3	Failure Modes and Debugging . . . . .	42
13.4	Cost Modeling . . . . .	43
<b>14</b>	<b>The Future of Inference</b>	<b>46</b>
14.1	Diffusion-Based Language Models . . . . .	46
14.2	Multi-Token Prediction . . . . .	46
14.3	Neuromorphic and Photonic Computing . . . . .	46
14.4	The Inference-First Design Philosophy . . . . .	46
	<b>Appendix A: Key Formulas Reference</b>	<b>49</b>

<b>Appendix B: Annotated Paper Reading List</b>	<b>50</b>
<b>Appendix C: Glossary</b>	<b>51</b>
<b>Appendix D: Hardware Comparison</b>	<b>53</b>

# Chapter 1

## The Inference Problem

### 1.1 Training vs. Inference: Fundamentally Different Goals

When a large language model is *trained*, the objective is to minimize a loss function over a massive dataset. The system processes data in large batches, accumulates gradients across many GPUs, performs backward passes, and updates billions of parameters. Throughput is the supreme metric: we want to process as many tokens per second as possible, amortized across the training run. Latency of an individual forward pass is largely irrelevant — what matters is overall training time measured in days or weeks.

*Inference* is the mirror image. Here the model parameters are frozen. The task is to compute a forward pass — transforming an input prompt into a predicted output — as fast and cheaply as possible, for each individual request, often from real users who are waiting for a response.

#### Definition: Inference

Inference is the act of running a trained neural network model on new, unseen input data to produce a prediction or output, without updating the model's parameters.

The goals of inference diverge sharply from training in three key dimensions:

- **Latency over throughput:** A user waiting for a response experiences latency directly. A 200 ms response feels instant; a 3-second response feels slow.
- **Memory efficiency:** At inference time, we typically do not need to store optimizer states or gradient buffers, but we do need to manage the KV cache for autoregressive generation.
- **Cost per request:** In production, you serve millions of requests. Cost per request directly determines profitability.

### 1.2 The Token Generation Loop Explained

Modern LLMs are *autoregressive*: they generate text one token at a time, where each new token is conditioned on all previously generated tokens. This creates the fundamental loop of LLM inference.

#### The Autoregressive Generation Loop

1. Receive a prompt (sequence of input tokens).
2. Run a **prefill** forward pass over the entire prompt in parallel, computing attention keys and values for all input tokens and caching them.
3. Autoregressively **decode**: at each step, feed the last generated token as input, compute attention over the KV cache, and sample the next token.
4. Repeat step 3 until a stop token is generated or a maximum length is reached.

The prefill phase is compute-bound: we process many tokens in parallel, fully utilizing the GPU’s tensor cores. The decode phase, however, is fundamentally memory-bandwidth-bound: we process a single token at a time, but must load the entire KV cache from GPU memory for every step. This asymmetry drives much of the complexity in inference systems.

### 1.3 Latency, Throughput, and Cost: The Iron Triangle

Inference optimization operates within a trilemma. Improving one dimension often comes at the expense of another:

Metric	Definition	How to Improve
<b>Latency</b>	Time from request to completion. Key sub-metrics: TTFT (Time to First Token) and TPOT (Time Per Output Token).	Reduce model size, increase batch size carefully, use speculative decoding, optimize KV cache.
<b>Throughput</b>	Tokens generated per second across all concurrent users.	Increase batch size, use continuous batching, tensor parallelism.
<b>Cost</b>	USD per 1M tokens generated.	Quantize the model, use efficient serving frameworks, optimize hardware utilization.

The iron triangle manifests concretely: maximizing throughput requires large batch sizes, but large batches increase per-request latency. Minimizing latency often requires serving requests individually, which wastes GPU utilization and increases cost. Inference engineering is the art of navigating these tradeoffs intelligently for your specific application requirements.

### 1.4 Memory Bandwidth: The True Bottleneck

A common misconception is that inference is compute-bound — that we need more FLOPS. In reality, for autoregressive LLM decoding, the bottleneck is almost always *memory bandwidth*: the speed at which we can load model weights from GPU memory (HBM) to the GPU’s compute cores.

Consider a 7-billion parameter model stored in 16-bit precision. The model weights occupy approximately:

$$7 \times 10^9 \times 2 \text{ bytes} = 14 \text{ GB}$$

An NVIDIA A100 GPU provides  $\approx 2$  TB/s of HBM bandwidth. To load all weights once (one decode step), the theoretical lower bound on latency is:

$$\frac{14 \text{ GB}}{2000 \text{ GB/s}} = 7 \text{ ms}$$

This is before any compute. If we run with batch size 1, we spend this time loading weights to generate just one token. With batch size 64, we amortize the same weight load across 64 requests simultaneously, dramatically improving throughput without changing latency much.

### 1.5 The Roofline Model

The Roofline model provides a clean framework for understanding whether a given operation is compute-bound or memory-bound. For any operation:

Achieved Performance (FLOP/s) =  $\min(\text{Peak FLOP/s}, \text{Arithmetic Intensity} \times \text{Peak Bandwidth})$

where *Arithmetic Intensity* is measured in FLOP per byte of memory accessed.

#### Practical Roofline Insight

Matrix-vector multiplications (the core of the decode step with batch size 1) have low arithmetic intensity ( $\sim 1\text{--}2$  FLOP/byte) and are solidly in the memory-bound regime. Large matrix-matrix multiplications (prefill with long prompts) have high arithmetic intensity and approach the compute bound. This is why prefill and decode behave so differently and why they benefit from different optimizations.

#### Interview Preparation — Mock Q&A

**Q1.1. What is the difference between prefill and decode in LLM inference, and why does it matter for system design?**

**Answer:** The prefill phase processes the entire input prompt in a single parallel forward pass, making it **compute-bound** because we multiply large token matrices against the weight matrices. The decode phase generates tokens one at a time autoregressively, making it **memory-bandwidth-bound** because we load all model weights for each single-token step. This distinction matters enormously for system design: prefill and decode have very different optimal batch sizes, GPU utilization profiles, and latency sensitivities. Modern systems like Splitwise and Mooncake physically separate prefill and decode onto different GPU pools, allowing each to be independently scaled and optimized.

**Q1.2. Explain the iron triangle of inference. How would you make a decision when optimizing for a latency-sensitive chatbot application?**

**Answer:** The inference iron triangle describes the tension between latency, throughput, and cost — improving one typically degrades another. For a latency-sensitive chatbot, I would prioritize **TTFT and TPOT**. Concretely: (1) Use a smaller or quantized model to reduce per-step computation. (2) Keep batch sizes small or use continuous batching with strict SLO-aware scheduling. (3) Apply speculative decoding to generate multiple tokens per step. (4) Use prefix caching to skip re-computing the KV cache for repeated system prompts. I would accept higher cost per token as the trade-off for low latency.

**Q1.3. Why is memory bandwidth, rather than raw compute (FLOP/s), typically the bottleneck in LLM decoding?**

**Answer:** During autoregressive decoding with a small batch size, the GPU must load the entire model weights from HBM for every token generated. A 7B-parameter model in FP16 is 14 GB; even on an A100 with 2 TB/s bandwidth, this takes approximately 7 ms per step. Meanwhile, the actual computation on a single token is trivially small compared to what the GPU's tensor cores can handle. The arithmetic intensity of matrix-vector multiply (the key decode operation) is roughly 1–2 FLOP per byte, far below the  $\sim 300$  FLOP/byte threshold needed to be compute-bound. Only by increasing batch size do we amortize the weight-loading cost and approach compute-bound behavior.

**Q1.4.** How do you measure and report inference performance? What metrics would you instrument in a production serving system?

**Answer:** I would track: **TTFT** (time to first token, critical for user-perceived responsiveness), **TPOT** (time per output token, determines streaming smoothness), **throughput** (tokens/second across all users), **GPU utilization and memory utilization**, **goodput** (fraction of requests meeting SLO), **queue depth** (to detect overload), and **cost per million tokens**. I'd set up percentile dashboards (p50, p95, p99) rather than averages, since tail latency is what breaks user experience. I'd also separately track prefill latency vs. decode latency to diagnose bottlenecks independently.

**Q1.5.** A colleague argues that since FLOP/s keeps growing faster than memory bandwidth, inference will soon be compute-bound. How do you respond?

**Answer:** This is the classic *bandwidth wall* argument, and I'd partially agree with the observation but disagree with the conclusion. It is true that compute FLOP/s has grown faster than HBM bandwidth over recent GPU generations. However, the arithmetic intensity of single-batch decode is so low (1–2 FLOP/byte) that even with large FLOP/s improvements, we'd still be memory-bound unless batch sizes increase dramatically or model architectures change. Additionally, techniques like quantization deliberately trade compute for reduced memory traffic, keeping us in the memory-bound regime. The path forward is architectures and systems that increase arithmetic intensity per byte loaded — like GQA, MLA, and efficient KV caching — not simply hoping for faster compute.

## Chapter 2

# Hardware Foundations for Inference

### 2.1 GPU Architecture for the Inference Engineer

Modern GPUs are organized hierarchically. At the top level, an NVIDIA H100 SXM5 contains:

- **132 Streaming Multiprocessors (SMs)**, each containing 128 CUDA cores and 4 fourth-generation Tensor Cores.
- **80 GB HBM3** memory with  $\approx 3.35$  TB/s bandwidth.
- A **shared memory / L1 cache** of 256 KB per SM, orders of magnitude faster than HBM.
- An **L2 cache** of 50 MB shared across all SMs.
- **NVLink 4.0** providing 900 GB/s bidirectional bandwidth for multi-GPU communication.

The memory hierarchy matters deeply for inference optimization. Operations that can be performed in shared memory (like fused attention in FlashAttention) avoid the slow round-trip to HBM entirely.

### 2.2 Tensor Cores and Mixed Precision

Tensor Cores are specialized matrix-multiplication units that operate on small tiles (e.g.,  $16 \times 16 \times 16$ ) of operands in low-precision formats. The H100's fourth-generation Tensor Cores support:

Format	Peak TFLOP/s	Use Case
FP64	67	Scientific computing
TF32	989	Training default
FP16/BF16	1,979	Inference standard
FP8	3,958	Quantized inference
INT8	3,958	Quantized inference

The key insight is that inference can exploit lower precision than training because we only need to preserve the *forward pass* fidelity — no gradient accumulation in high precision is needed. This is why INT8 and FP8 inference delivers  $\sim 2\times$  the theoretical peak throughput of FP16.

### 2.3 Memory Hierarchy and Bandwidth

Understanding where your data lives at any moment is critical for inference performance:

GPU Memory Hierarchy (H100 SXM)			
Level	Capacity	Bandwidth	Latency
Registers	~256 KB/SM	>20 TB/s	~1 cycle
Shared Mem (L1)	228 KB/SM	~10 TB/s	~20 cycles
L2 Cache	50 MB	~12 TB/s	~150 cycles
HBM3 (VRAM)	80 GB	3.35 TB/s	~600 cycles
NVLink	Multi-GPU	900 GB/s	microseconds
PCIe	CPU-GPU	64 GB/s	microseconds

FlashAttention’s core insight is that standard attention computes full attention matrices in HBM, but by tiling operations so they fit in shared memory, we avoid most HBM traffic and achieve near-optimal memory efficiency.

## 2.4 CPU and Edge Hardware

Not all inference happens on data center GPUs. CPU inference is critical for latency-sensitive applications, on-premises deployments, and cost optimization:

**Modern CPUs for inference** use AVX-512 VNNI (Vector Neural Network Instructions) to perform INT8 matrix-vector multiplications efficiently. Intel’s 4th-gen Xeon Scalable processors can deliver ~10 TFLOP/s of INT8 throughput, sufficient for small models.

**Apple Silicon** (M3 Ultra) unifies CPU and GPU memory into a single pool (up to 192 GB) with ~800 GB/s bandwidth. This unified memory architecture makes it exceptionally good for inference on large models that would otherwise require multiple discrete GPUs.

**Qualcomm Hexagon DSPs** and **NPU**s in mobile SoCs are purpose-built for low-power neural network inference, delivering several TOPS (Tera-Operations Per Second) at milliwatt power budgets.

## 2.5 Emerging Hardware Trends

- **Google TPU v5p:** Custom matrix engine with inter-chip interconnects enabling pods of thousands of chips for large-model inference.
- **Groq LPU:** Deterministic inference accelerator with no DRAM — all weights stored in on-chip SRAM, enabling ~500 token/s per chip with ultra-low latency.
- **Cerebras WSE-3:** Wafer-scale engine with 44 GB of on-chip SRAM, eliminating the HBM bottleneck entirely for models that fit.
- **Photonic computing:** Early-stage technology using light instead of electrons for matrix multiplication, promising dramatically lower power consumption.

Interview Preparation — Mock Q&A

**Q2.1.** Explain the difference between HBM bandwidth and NVLink bandwidth. When does each become the bottleneck?

**Answer:** HBM (High Bandwidth Memory) is the on-device memory attached to a single GPU — on an H100, it provides 3.35 TB/s and is the bottleneck during single-GPU inference when loading model weights token by token. NVLink is the high-speed inter-

connect between multiple GPUs on the same node, providing 900 GB/s total bandwidth on H100 NVLink. NVLink becomes the bottleneck during **tensor-parallel inference**: when we shard a large model across multiple GPUs, each attention head's output must be all-reduced across GPUs after every layer. If the model is sharded across 8 GPUs, every all-reduce consumes NVLink bandwidth. For very large models (175B+), NVLink can limit the scaling efficiency of tensor parallelism.

**Q2.2. Why does Apple Silicon perform competitively for LLM inference despite lower peak FLOP/s than discrete GPUs?**

**Answer:** Apple Silicon uses a **unified memory architecture** where CPU, GPU, and Neural Engine all share the same physical memory pool — up to 192 GB on M3 Ultra. This eliminates the PCIe bottleneck that plagues discrete GPU setups (which must copy data across a 64 GB/s bus). The unified memory bandwidth of 800 GB/s is comparable to discrete GPUs, and crucially, a 70B-parameter model (140 GB in FP16) can fit entirely in memory without quantization, which is impossible on any single discrete GPU currently. For memory-bandwidth-bound decode workloads, bandwidth per dollar, not FLOP/s per dollar, is the relevant metric.

**Q2.3. What is the significance of Tensor Core formats (FP16 vs INT8 vs FP8) for inference? When would you choose each?**

**Answer:** Tensor Cores accelerate matrix multiplications in specific low-precision formats, with INT8/FP8 delivering  $\sim 2\times$  the throughput of FP16 on the H100. I'd choose **FP16/BF16** for models where accuracy is paramount and the hardware supports it natively — it's the safe baseline. **INT8** (via GPTQ, AWQ, or LLM.int8()) is excellent for weight quantization on models up to 70B parameters, providing roughly  $2\times$  memory reduction with minimal accuracy loss on most tasks. **FP8** (supported natively on H100) is emerging as the sweet spot — better dynamic range than INT8, hardware-accelerated, and increasingly used in production systems like TensorRT-LLM. I'd pick FP8 for new deployments on H100-class hardware.

**Q2.4. How would you profile an LLM inference workload to identify whether it is compute-bound or memory-bandwidth-bound?**

**Answer:** I'd use **NVIDIA Nsight Systems** for a timeline view (SM activity, HBM transfers, PCIe transfers) and **Nsight Compute** for kernel-level profiling (achieved FLOP/s vs. peak, memory throughput vs. peak). Concretely: (1) Measure achieved memory bandwidth as a fraction of peak HBM bandwidth. (2) Measure achieved FLOP/s as a fraction of peak Tensor Core FLOP/s. (3) Compute the operational intensity of key kernels. If memory bandwidth is near-saturated while FLOP/s utilization is low, we're memory-bound. I'd also run a sweep over batch sizes: if throughput scales linearly with batch size, we're in the memory-bound regime. If it plateaus, we've hit compute saturation.

**Q2.5. Describe the trade-offs between deploying inference on a Groq LPU versus an NVIDIA H100.**

**Answer:** The **Groq LPU** prioritizes deterministic ultra-low latency: with all weights stored in on-chip SRAM, there is no DRAM latency, yielding extremely consistent  $\sim 500$  token/s single-stream throughput. However, SRAM is expensive and limited, so LPUs can only serve models up to a certain size per chip, and scaling to very large models requires many chips with high cost. The **H100** offers far more flexibility: 80 GB HBM per GPU, support for virtually any model size across multiple GPUs, programmability

via CUDA, and an ecosystem of optimized libraries. I'd choose Groq for latency-critical applications with fixed model sizes (e.g., a production chatbot with a known 7B model). I'd choose H100 for flexibility, large models (70B+), or high-throughput batch inference where absolute latency is less critical.

## Chapter 3

# Transformer Inference Mechanics

### 3.1 Anatomy of a Transformer Forward Pass

A standard decoder-only transformer (such as GPT, LLaMA, or Mistral) processes tokens through a stack of  $L$  identical layers. Each layer contains two sub-components: a **Multi-Head Attention (MHA)** block and a **Feed-Forward Network (FFN)**, each followed by layer normalization and a residual connection.

For a sequence of  $n$  input tokens with model dimension  $d$ :

1. **Embedding:** Map token IDs to vectors  $\mathbf{X} \in \mathbb{R}^{n \times d}$ .
2. **For each layer**  $\ell = 1 \dots L$ :
  - (a) Compute queries, keys, values:  $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ ,  $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ ,  $\mathbf{V} = \mathbf{X}\mathbf{W}_V$
  - (b) Compute attention:  $\text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}$
  - (c) Apply FFN with gate:  $\text{FFN}(\mathbf{x}) = \sigma(\mathbf{x}\mathbf{W}_1) \odot (\mathbf{x}\mathbf{W}_3) \cdot \mathbf{W}_2$
3. **Language model head:** Project to vocabulary logits and sample.

### 3.2 The KV Cache: What It Is and Why It Matters

Without caching, autoregressive generation would require recomputing keys and values for all previous tokens at every step — an  $O(n^2)$  cost in the sequence length. The **KV cache** solves this: after the prefill phase, we store the key and value matrices for every layer for every token processed, and reuse them in subsequent decode steps.

#### Definition: KV Cache

The KV cache is a data structure that stores the key ( $\mathbf{K}$ ) and value ( $\mathbf{V}$ ) tensors from the attention computation for all previously generated tokens, allowing the decode step to attend to prior context without recomputation.

**KV cache memory footprint** for a single sequence:

$$\text{Memory} = 2 \times L \times n \times n_{\text{heads}} \times d_{\text{head}} \times \text{bytes\_per\_element}$$

For LLaMA-2 70B (80 layers, 64 heads, 128 head dim, FP16, 4096 context):

$$2 \times 80 \times 4096 \times 64 \times 128 \times 2 \approx 10.7 \text{ GB}$$

This enormous memory cost is what makes KV cache management the central challenge of LLM serving.

### 3.3 Prefill vs. Decode: Two Very Different Workloads

#### Prefill Phase

Processes all  $n$  prompt tokens in a single forward pass in parallel. The dominant operation is GEMM (General Matrix Multiplication) of shape  $[n, d] \times [d, 4d]$  for the FFN layers. This is **compute-bound** for long prompts (large  $n$ ), achieving high GPU utilization on Tensor Cores.

#### Decode Phase

Processes one token per step. The dominant operation is GEMV (General Matrix-Vector Multiply) of shape  $[1, d] \times [d, 4d]$ . This is **memory-bandwidth-bound**: we load  $\sim 140$  GB of weights (for a 70B model in FP16) per step to perform a trivially small computation. GPU utilization of Tensor Cores is typically  $<10\%$ .

### 3.4 Attention Variants: MHA to MLA

The explosive growth in context length has driven a series of innovations in attention architectures to reduce KV cache size:

**Multi-Head Attention (MHA)**: Standard attention with  $H$  heads. Each head has its own  $\mathbf{K}$  and  $\mathbf{V}$  matrices. KV cache proportional to  $H$ .

**Multi-Query Attention (MQA)**: All query heads share a single set of  $\mathbf{K}$  and  $\mathbf{V}$  matrices. Reduces KV cache by  $H \times$  but can hurt model quality.

**Grouped Query Attention (GQA)**: Groups of  $G$  query heads share one set of  $\mathbf{K}$ ,  $\mathbf{V}$ . Used in LLaMA-3, Mistral, Gemma. A principled middle ground between MHA and MQA.

**Multi-head Latent Attention (MLA)**: Introduced in DeepSeek-V2. Projects keys and values into a low-dimensional latent space  $c \ll d$ , storing only the latent vectors in the KV cache. Reduces KV cache by  $\sim 5\text{--}13 \times$  vs. MHA with minimal quality degradation.

KV cache size ratio: MHA : GQA : MLA  $\approx 8 : 2 : 1$

### 3.5 Positional Encodings at Inference Time

**Rotary Positional Embeddings (RoPE)** are now the dominant positional encoding scheme for decoder-only LLMs. RoPE encodes position by rotating query and key vectors in the complex plane:

$$\mathbf{q}_m = \mathbf{q} \cdot e^{im\theta}, \quad \mathbf{k}_n = \mathbf{k} \cdot e^{in\theta}$$

so that  $\langle \mathbf{q}_m, \mathbf{k}_n \rangle$  depends only on the relative position  $m - n$ . At inference time, RoPE allows **context length extrapolation** via NTK-aware scaling or YaRN, enabling models trained on 4K context to generalize to 128K+ contexts with appropriate position interpolation.

#### Interview Preparation — Mock Q&A

**Q3.1.** Explain Multi-Query Attention vs. Grouped Query Attention. Why are they important for inference?

**Answer:** In standard Multi-Head Attention, each of the  $H$  attention heads has its own

separate  $\mathbf{K}$  and  $\mathbf{V}$  projection matrices. This means the KV cache grows linearly with the number of heads. Multi-Query Attention (MQA), proposed by Shazeer (2019), uses a single shared  $\mathbf{K}$  and  $\mathbf{V}$  for all query heads, reducing the KV cache by  $H\times$ . However, sharing across all heads can degrade model quality. Grouped Query Attention (GQA) is a generalization: heads are divided into  $G$  groups, and each group shares one  $\mathbf{K}/\mathbf{V}$  pair. This reduces the KV cache by  $H/G\times$  while preserving more capacity than MQA. For inference, these techniques directly reduce memory pressure, enabling longer contexts and larger batch sizes. LLaMA-3 70B uses GQA with 8 KV heads for 64 query heads, an  $8\times$  KV reduction.

**Q3.2. Walk me through what happens to the KV cache during a 1000-token generation on a 7B model.**

**Answer:** During prefill, all prompt tokens are processed in parallel, and their keys and values are computed and stored in the KV cache for all layers. For a 7B model (32 layers, 32 heads, 128 head dim, FP16), the KV cache per token per layer is  $2 \times 32 \times 128 \times 2 = 16,384$  bytes. For 1000 generated tokens on top of a 512-token prompt: the KV cache grows from  $512 \times 16,384 \times 32$  bytes at start to  $1512 \times 16,384 \times 32 \approx 790$  MB at the end. During each decode step, we read the entire KV cache (growing with each step), compute attention of the new query against all cached keys, and append the new key/value pair to the cache. This growing memory read cost is why decode latency increases slightly over long generations.

**Q3.3. What is the attention complexity bottleneck and how do modern systems address it?**

**Answer:** Standard attention has  $O(n^2)$  complexity in both time and space: the attention matrix  $\mathbf{QK}^\top$  is  $n \times n$ . For  $n = 128,000$  tokens, this matrix is  $128,000^2 \times 2$  bytes  $\approx 32$  GB — too large to materialize in HBM. Modern systems address this through: (1) **FlashAttention**: tile-based computation that avoids materializing the full attention matrix in HBM; (2) **Sparse attention patterns** (Longformer, BigBird): only attend to a subset of tokens; (3) **Linear attention approximations**: approximate the softmax attention with kernel methods to achieve  $O(n)$ ; (4) **State Space Models** (Mamba): replace attention entirely with a recurrent structure that processes long sequences in  $O(n)$  time and  $O(1)$  memory.

**Q3.4. How does RoPE enable context length extrapolation? What goes wrong if you naively extend context beyond training length?**

**Answer:** RoPE encodes position using rotation matrices applied to query and key vectors, where the rotation angle for dimension  $i$  is  $\theta_i = 1/10000^{2i/d}$ . This means high-frequency dimensions rotate quickly with position while low-frequency dimensions rotate slowly. If you naively extrapolate beyond the training context length  $n$ , the high-frequency dimensions encounter rotation angles they've never seen in training, leading to attention score distributions that are out-of-distribution. The model loses its ability to distinguish relative positions accurately. **YaRN** and **NTK-aware scaling** address this by rescaling the base frequency  $\theta$  based on the extension ratio, ensuring all dimensions remain in-distribution at the new context length.

**Q3.5. Why is the decode phase memory-bandwidth-bound while prefill is compute-bound? What are the implications for serving system design?**

**Answer:** During decode, we process a single token, meaning the matrix operations are matrix-vector multiplications (GEMV) where one dimension is 1. GEMV has arithmetic

intensity of  $\sim 2$  FLOP per byte (loading the weight matrix once to do one dot product), far below the roofline threshold for compute-bound operation. The GPU tensor cores are severely underutilized. During prefill, we process  $n$  tokens simultaneously, meaning operations are matrix-matrix multiplications (GEMM) with much higher arithmetic intensity ( $\sim 2n/d$  FLOP per byte), approaching compute saturation for large  $n$ . For serving design, this implies we should: (1) use continuous batching to increase effective batch size during decode; (2) consider disaggregated prefill-decode serving (Splitwise, Mooncake) where prefill runs on compute-optimized nodes and decode runs on memory-bandwidth-optimized nodes.

## Chapter 4

# Quantization

### 4.1 Why Quantization Matters

A 70B-parameter model in FP32 requires 280 GB of GPU memory — more than three A100 80GB GPUs just to hold the weights. In FP16 or BF16, this drops to 140 GB. With INT8 quantization, it becomes 70 GB (fits on one H100 with room for KV cache). With INT4, it drops to 35 GB, fitting comfortably on a single GPU. Quantization is not an academic curiosity — it is the difference between serving a model and not serving it.

### 4.2 Number Formats

Format	Bits	Range	Characteristics
FP32	32	$\pm 3.4 \times 10^{38}$	Full precision, training standard
BF16	16	$\pm 3.4 \times 10^{38}$	Same exponent range as FP32, less mantissa precision
FP16	16	$\pm 65,504$	Common inference format; overflow risk
FP8 E4M3	8	$\pm 448$	H100 native; inference and fine-tuning
FP8 E5M2	8	$\pm 57,344$	More range, less precision
INT8	8	$-128$ to $127$	Widely supported; $2\times$ size saving vs FP16
INT4	4	$-8$ to $7$	$4\times$ size saving; requires careful calibration
INT2	2	$-2$ to $1$	Extreme compression; significant quality loss

### 4.3 Post-Training Quantization Methods

#### 4.3.1 GPTQ

GPTQ (Frantar et al., 2022) is a layer-wise second-order PTQ method. It quantizes weights row by row, updating remaining weights to compensate for the quantization error of already-quantized weights using the inverse Hessian:

$$\delta \mathbf{W} = -\frac{w_q - w}{[\mathbf{H}_F^{-1}]_{qq}} \cdot [\mathbf{H}_F^{-1}]_{:,q}$$

GPTQ achieves near-lossless 4-bit weight quantization for models  $\geq 30$ B parameters and is the foundation of many popular quantization tools (AutoGPTQ, ExLlamaV2).

### 4.3.2 AWQ (Activation-Aware Weight Quantization)

AWQ (Lin et al., 2023) observes that not all weights are equally important — weights corresponding to large-magnitude activations are more sensitive to quantization error. AWQ protects these salient weights by scaling channels before quantization:

$$\mathbf{Q}(\mathbf{W} \cdot \text{diag}(\mathbf{s})^{-1}) \cdot \text{diag}(\mathbf{s}) \approx \mathbf{W}$$

This preserves the mathematical equivalence while reducing the effective quantization range of critical channels. AWQ consistently outperforms GPTQ at 4-bit precision and is widely used in production.

### 4.3.3 QuIP# and AQLM

QuIP# (Tseng et al., 2024) uses incoherence processing — randomly rotating weights and activations before quantization — to flatten the distribution of values and enable near-lossless 2-bit quantization. AQLM (Additive Quantization of Language Models) uses a codebook-based approach where weight vectors are encoded as sums of codebook entries, enabling 2-bit equivalent compression with quality closer to 4-bit methods.

## 4.4 Quantization-Aware Training (QAT)

PTQ methods calibrate on a small dataset and apply quantization post-hoc, which limits their accuracy recovery. QAT incorporates fake quantization operators into the forward pass during fine-tuning:

$$\hat{x} = \text{round}\left(\frac{x}{s}\right) \cdot s$$

where  $s$  is the learned quantization scale. Gradients flow through the rounding operation via the Straight-Through Estimator. QAT yields higher accuracy than PTQ at extreme bit-widths (2–3 bit) but requires significant compute budget.

## 4.5 Activation Quantization Challenges

While weight quantization is relatively straightforward (weights are static and their distribution can be analyzed offline), **activation quantization** is much harder. Activations have dynamic ranges that vary by input, and LLMs exhibit a well-documented phenomenon: a small number of channels (“outliers”) have activation magnitudes  $100\times$  larger than typical channels. Naively quantizing activations to INT8 forces the quantization scale to accommodate these outliers, causing all normal-range activations to collapse to just a few distinct values.

Solutions include:

- **LLM.int8()** (Dettmers et al.): Decompose matrix multiplication, keeping outlier channels in FP16 and the rest in INT8.
- **SmoothQuant**: Migrate the quantization difficulty from activations to weights via per-channel scaling:  $\mathbf{Y} = (\mathbf{X}\text{diag}(\mathbf{s})^{-1}) \cdot (\text{diag}(\mathbf{s})\mathbf{W})$ .
- **FP8 activations**: The wider dynamic range of FP8 compared to INT8 naturally accommodates outliers.

## Interview Preparation — Mock Q&amp;A

**Q4.1. You need to deploy a 70B LLM on 2 A100 80GB GPUs. What quantization strategy would you choose and why?**

**Answer:** Two A100 80GB GPUs give us 160 GB total GPU memory. A 70B model in FP16 requires 140 GB for weights alone, leaving only 20 GB for KV cache and activations — uncomfortably tight. I would apply **AWQ 4-bit quantization**, reducing the weight footprint to ~35 GB. This leaves ~125 GB for KV cache, batch activations, and overhead. AWQ 4-bit typically preserves >99% of the original model’s benchmark scores on general reasoning tasks. I would also evaluate GPTQ as an alternative. I’d calibrate on domain-specific data if the deployment use case is specialized. For the serving framework, I’d use vLLM with AWQ support and tensor parallelism across the two GPUs.

**Q4.2. Explain the difference between weight-only quantization and weight-and-activation quantization. When does each apply?**

**Answer:** **Weight-only quantization** (W4A16, W8A16) quantizes the model parameters to INT4 or INT8 but keeps activations in FP16 at runtime. Weights are dequantized back to FP16 before the matrix multiplication. This reduces model size and memory bandwidth for weight loading (critical for decode), but the actual GEMM computation still runs in FP16. Methods: GPTQ, AWQ. **Weight-and-activation quantization** (W8A8, W4A4) quantizes both weights and activations, enabling the matrix multiplication itself to run in INT8 or INT4. This provides additional compute throughput from Tensor Core acceleration. However, it requires careful handling of activation outliers. Methods: SmoothQuant, LLM.int8(). For purely memory-bound decode workloads, W4A16 is often sufficient. For compute-bound prefill or large batch sizes, W8A8 or FP8 provides additional speedup.

**Q4.3. What is the Straight-Through Estimator and why is it needed for QAT?**

**Answer:** The rounding operation in quantization ( $\hat{x} = \text{round}(x/s) \cdot s$ ) has a zero derivative almost everywhere — it is a staircase function. This means that naive backpropagation through a quantization node produces zero gradients, making learning impossible. The **Straight-Through Estimator (STE)**, introduced by Bengio et al. (2013), approximates the gradient of the round function as 1 (identity) within the quantization range:  $\frac{\partial \hat{x}}{\partial x} \approx \mathbf{1}_{|x| \leq \text{threshold}}$ . This is mathematically unjustified but empirically effective — it allows gradients to flow through the quantization operation during the backward pass as if the rounding had not occurred, enabling the model to learn weight values that “round well.”

**Q4.4. How does SmoothQuant address the activation outlier problem? What are its limitations?**

**Answer:** SmoothQuant observes that activation outliers are difficult to quantize (large dynamic range, few extreme values) but weights are easy (smooth distributions). The key insight is that we can *migrate* the quantization difficulty from activations to weights via a mathematically equivalent transformation: if  $\mathbf{Y} = \mathbf{X}\mathbf{W}$ , then  $\mathbf{Y} = (\mathbf{X}\text{diag}(\mathbf{s})^{-1}) \cdot (\text{diag}(\mathbf{s})\mathbf{W})$  for any positive scaling vector  $\mathbf{s}$ . By choosing  $s_j = \max(|\mathbf{X}_{:,j}|)^\alpha / \max(|\mathbf{W}_{j,:}|)^{1-\alpha}$ , we equalize the ranges. **Limitations:** (1) Requires calibration data to estimate per-channel activation statistics; (2) The optimal  $\alpha$  is task-dependent; (3) Does not fully eliminate outliers in all layers; (4) Some models (particularly with very large vocabulary or embedding layers) still exhibit quantization sensitivity.

**Q4.5. Compare GPTQ and AWQ. Which would you use in production and under what conditions?**

**Answer:** **GPTQ** uses second-order (Hessian) information to minimize the layer-wise quantization error, sequentially updating weights to compensate for previously quantized weights. It is computationally expensive to run but produces high-quality quantized models, especially for larger models where the Hessian approximation is more accurate. **AWQ** is simpler and faster — it identifies the 1% of weight channels with highest activation magnitudes and protects them by scaling before quantization, requiring only forward passes for calibration. In practice, AWQ tends to outperform GPTQ at 4-bit precision on most tasks and is faster to apply. I would use **AWQ in production** due to its speed, quality, and broad community support (AutoAWQ). I'd consider GPTQ for specialized tasks where careful calibration on domain data is worthwhile, or when using ExLlamaV2's highly optimized GPTQ kernel.

## Chapter 5

# Speculative Decoding

### 5.1 The Core Insight: Draft and Verify

Speculative decoding (Leviathan et al., 2023; Chen et al., 2023) exploits a fundamental asymmetry: *verifying* whether a sequence of tokens would have been generated by a large model can be done in a single parallel forward pass, while *generating* those tokens autoregressively requires one forward pass per token.

The algorithm is elegant:

1. Use a small **draft model** (3–7B parameters) to autoregressively generate  $K$  candidate “draft” tokens cheaply.
2. Run a single forward pass of the large **target model** over all  $K$  draft tokens in parallel.
3. **Verify** each draft token against the target model’s distribution using rejection sampling. Accept tokens that match and reject (with a correction step) where they diverge.
4. The result is provably equivalent to sampling from the target model distribution.

The expected speedup is  $\frac{1}{1-\alpha^K}$  where  $\alpha$  is the per-token acceptance rate. For  $\alpha = 0.8$  and  $K = 5$ , the expected accepted tokens per step is  $\approx 3.3$ , yielding a theoretical  $3.3\times$  decode throughput increase.

### 5.2 Token Tree Verification

Basic speculative decoding generates a single draft sequence. **Tree-based speculative decoding** generates a *tree* of draft sequences: at each position, the draft model proposes multiple candidate tokens, creating a branching tree structure. The target model verifies all tree branches in a single forward pass using a tree-masked attention pattern. This dramatically increases the probability of high acceptance rates by exploring multiple alternatives simultaneously.

### 5.3 Medusa: Multi-Head Speculative Decoding

Medusa (Cai et al., 2024) eliminates the need for a separate draft model. Instead, it adds  $K$  additional “Medusa heads” — lightweight linear layers — on top of the target model. Each head  $k$  predicts token  $t + k$  (the  $k$ -th future token) in a single forward pass:

$$\text{logits}_{t+k} = \text{MedusaHead}_k(\mathbf{h}_t)$$

where  $\mathbf{h}_t$  is the last hidden state at position  $t$ . This enables the model to self-speculatively propose multiple future tokens. Medusa-2 adds self-distillation to further improve draft quality, achieving  $2\text{--}3\times$  speedup on generation benchmarks.

## 5.4 EAGLE: Efficient Autoregressive Generation

EAGLE (Li et al., 2024) addresses a key limitation of Medusa: predicting future tokens from only the last hidden state loses information about the exact token sequence, limiting acceptance rates. EAGLE instead trains a lightweight draft model that takes as input both the hidden states *and* the feature embeddings of previously generated tokens, enabling it to produce much more accurate next-token predictions. EAGLE-2 further introduces a **dynamic draft tree** that adapts the tree structure based on the confidence of draft predictions, achieving 3–4× speedup on coding and reasoning tasks.

## 5.5 When Speculative Decoding Helps (and When It Doesn't)

### Conditions Favorable to Speculative Decoding

- **Single-stream, low-latency serving:** The target model is underutilized with batch size 1. Speculative decoding amortizes idle compute.
- **High acceptance rate:** Tasks with predictable outputs (code completion, structured generation, factual QA) see high  $\alpha$  and large speedups.
- **Sufficient GPU memory:** Need room for both draft and target model.

### Note

Speculative decoding provides *no throughput benefit* in high-batch-size serving scenarios, where the GPU is already saturated. It is a latency optimization, not a throughput optimization.

### Interview Preparation — Mock Q&A

**Q5.1. Prove that speculative decoding produces outputs from the same distribution as the target model.**

**Answer:** The proof relies on **rejection sampling**. Suppose the draft model proposes token  $x$  with probability  $q(x)$  and the target model's distribution is  $p(x)$ . We accept the draft token with probability  $\min(1, p(x)/q(x))$ . If rejected, we sample from the residual distribution  $p'(x) \propto \max(0, p(x) - q(x))$ . The marginal acceptance probability at any step is  $\sum_x q(x) \cdot \min(1, p(x)/q(x)) = \sum_x \min(q(x), p(x)) = 1 - \text{TV}(p, q)$ . The final output distribution is: accepted tokens follow  $q(x) \cdot \min(1, p(x)/q(x))$ , and the correction term ensures the remaining probability mass matches  $p(x)$ . The combined distribution integrates exactly to  $p(x)$ .

**Q5.2. What is the acceptance rate  $\alpha$  in speculative decoding, and what factors influence it?**

**Answer:** The acceptance rate  $\alpha$  is the probability that a single draft token is accepted by the target model's rejection sampling criterion. It is determined by the overlap between the draft model's distribution  $q(x)$  and the target model's distribution  $p(x)$ :  $\alpha = 1 - \text{TV}(p, q) = \sum_x \min(p(x), q(x))$ . Factors that increase  $\alpha$ : (1) **Task predictability**: coding and structured text have more deterministic next tokens; (2) **Draft-target**

**alignment:** a draft model from the same family or a distilled version of the target; (3) **Temperature:** lower sampling temperature makes both distributions more peaked and likely to agree; (4) **Smaller speculative window:** fewer draft tokens ( $K$ ) reduces compounding rejection probability. Typical values are  $\alpha = 0.7$ – $0.9$  on code generation, and  $0.5$ – $0.7$  on open-ended chat.

**Q5.3. How does Medusa differ from a separate draft model approach? What are the trade-offs?**

**Answer:** A **separate draft model** is an independent small model with its own weights, KV cache, and forward pass. Its advantage is flexibility: the draft model can be any compatible smaller model, and its quality can be improved independently. The downside is memory overhead (storing two models) and complexity (managing two KV caches). **Medusa** adds lightweight linear heads on top of the target model's hidden states, requiring no separate model. It uses the target model's internal representations directly, which are richer than a smaller draft model's. The trade-off: Medusa heads are less expressive than a full draft model (they see only the last hidden state, not the full context), so acceptance rates may be lower. Medusa-2 addresses this with self-distillation. Medusa is simpler to deploy; a separate draft model can be higher quality if carefully chosen.

**Q5.4. In what scenario would you *not* recommend speculative decoding?**

**Answer:** I would not recommend speculative decoding in: (1) **High-batch-size production serving:** When GPUs are compute-saturated at batch sizes of 32+, the parallel verification step consumes compute that could otherwise serve additional requests; throughput decreases. (2) **Memory-constrained environments:** Speculative decoding requires additional memory for the draft model (a separate model approach) or Medusa heads. If GPU memory is fully committed to the KV cache and weights, it's infeasible. (3) **Tasks with inherently low acceptance rates:** Open-ended creative generation with high temperature and diverse vocabulary produces low  $\alpha$  ( $< 0.5$ ), meaning frequent rejections and little speedup. (4) **Very short outputs:** The overhead of the draft loop is fixed; for 1–5 token outputs, speculative decoding provides no benefit.

**Q5.5. Describe the EAGLE architecture and explain why it achieves higher acceptance rates than basic Medusa.**

**Answer:** EAGLE (Extrapolation Algorithm for Greater Language-model Efficiency) trains a lightweight autoregressive draft model that takes as input not just the hidden state  $\mathbf{h}_t$  from the target model (as Medusa does) but also the **token embedding** of the most recently generated token  $\mathbf{e}_t$ . This joint input  $[\mathbf{h}_t; \mathbf{e}_t]$  is fed into a single transformer decoder layer to predict the next-step hidden state  $\mathbf{h}_{t+1}$ , from which future tokens are predicted. The key advantage over Medusa is **autoregressive draft generation:** EAGLE can generate multiple draft tokens in sequence, conditioning each on the previous, rather than predicting all from the same fixed hidden state. This sequential conditioning captures inter-token dependencies that Medusa's parallel heads miss, yielding acceptance rates 5–15 percentage points higher. EAGLE-2 adds confidence-based dynamic tree expansion.

## Chapter 6

# KV Cache Optimization

### 6.1 The KV Cache Memory Crisis

As shown in Chapter 3, the KV cache for a single long-context request can consume tens of gigabytes of GPU memory. For a serving system with hundreds of concurrent requests, naively pre-allocating maximum-length KV caches wastes most of the allocated memory (since actual request lengths are unpredictable) and severely limits batch size. This was the central unsolved problem in LLM serving before 2023.

### 6.2 PagedAttention: Virtual Memory for KV Caches

PagedAttention (Kwon et al., 2023), the core innovation in vLLM, applies operating-system virtual memory concepts to KV cache management.

Instead of allocating a contiguous block of memory for each request’s KV cache, PagedAttention divides the KV cache into fixed-size **blocks** (pages) of  $B$  tokens each. A **block table** maps each request’s logical KV cache positions to physical blocks in GPU memory — blocks need not be contiguous.

#### PagedAttention Key Properties

- **No pre-allocation:** Blocks are allocated on-demand as generation proceeds.
- **No internal fragmentation:** Wasted memory is at most  $(B - 1)$  tokens per request.
- **Copy-on-write:** Multiple requests sharing the same prefix (e.g., a system prompt) can share physical blocks until their paths diverge.
- **Flexible eviction:** Blocks from one request can be swapped to CPU or disk to make room for another.

PagedAttention requires a modified attention kernel that gathers non-contiguous blocks during the attention computation — a non-trivial engineering challenge that vLLM implements via custom CUDA kernels.

### 6.3 Prefix Caching (Prompt Caching)

Many production workloads share a long common prefix — a system prompt, a document, or a few-shot example — across thousands of requests. Without caching, every request recomputes the KV cache for this shared prefix from scratch.

**Prefix caching** (also called prompt caching, implemented in vLLM, SGLang, and offered by Anthropic/OpenAI as a feature) stores the KV cache for common prefixes and reuses them across requests. This can reduce prefill cost by 90%+ when the prefix is long relative to the user prompt.

## 6.4 StreamingLLM: The Attention Sink Phenomenon

For applications requiring indefinitely long context (streaming conversations, long-running agents), we cannot store the KV cache for all tokens — it grows without bound. Naïve sliding window attention (keeping only the most recent  $W$  tokens’ KV cache) causes catastrophic performance degradation. Why? LLMs develop “**attention sinks**”: the first few tokens in the sequence (regardless of their semantic content) receive disproportionately large attention scores across all layers. Removing them destroys the model’s internal representations.

**StreamingLLM** (Xiao et al., 2023) keeps a small “sink” of the initial tokens (typically 4) along with a sliding window of recent tokens, enabling infinite-length LLM inference with bounded memory:

$$\text{KV retained} = \text{sink tokens} \cup \text{recent window}$$

## 6.5 KV Cache Quantization

The KV cache itself can be quantized independently of model weights, since it is read-once per attention step and any quantization error only affects the current decode step rather than accumulating. INT8 KV cache quantization is now standard in frameworks like TensorRT-LLM and vLLM, providing  $\sim 2\times$  KV memory reduction. More aggressive schemes apply INT4 or even FP8 to the KV cache with careful per-head or per-channel scaling.

### Interview Preparation — Mock Q&A

**Q6.1. Explain PagedAttention. How does it differ from pre-allocated contiguous KV cache management?**

**Answer:** Traditional KV cache management pre-allocates a contiguous block of GPU memory sized for the maximum possible sequence length for each request. This wastes enormous memory because most requests are much shorter than the maximum. Additionally, it fragments memory: once a block is released, the gap may be too small for a new block, leading to external fragmentation. PagedAttention divides KV cache memory into fixed-size blocks ( $\sim 16$  tokens per block) and maintains a block table per request mapping logical positions to physical blocks. Blocks are allocated on demand as new tokens are generated and freed immediately when a request completes. This eliminates pre-allocation waste, reduces fragmentation to at most  $B - 1$  tokens per request, and enables copy-on-write sharing of common prefix blocks across requests. The result is typically  $2\text{--}4\times$  higher batch size for the same GPU memory.

**Q6.2. How does prefix caching work, and what kind of application benefits most from it?**

**Answer:** Prefix caching stores the computed KV cache for a prefix of tokens (e.g., a system prompt) and reuses it for subsequent requests that share that prefix. The system maintains a hash map from prefix token sequence (or content hash) to physical KV cache blocks. When a new request arrives, the serving system checks if any of its leading token positions match a cached prefix; if so, the prefill step for those tokens is skipped entirely. Applications that benefit most: (1) **API products with fixed system prompts**: e.g., a customer service bot with a 2000-token system prompt — every user message benefits from the cached prefix; (2) **Multi-turn conversations**: subsequent turns can reuse KV for all prior turns; (3) **Batch document processing**: e.g., 1000 questions about the

same document share the document's KV cache. Real-world latency reduction can be 50–90% for the prefill phase.

**Q6.3. What is the attention sink phenomenon? Why does naïve sliding window attention fail for StreamingLLM?**

**Answer:** When LLMs are trained on fixed-length sequences, they learn to direct large amounts of attention toward the first few tokens of every sequence — these are called “attention sinks.” The first tokens act as aggregators that absorb excess attention probability mass that the softmax requires to normalize. These sink tokens are not semantically important; they simply become structural anchors. Naïve sliding window attention, which discards the oldest tokens to maintain a fixed-size KV cache, inadvertently removes these sink tokens when the window slides past the beginning. Without the sinks, the attention normalization breaks: softmax scores over the remaining tokens become very different from the training distribution, causing outputs to degrade catastrophically. StreamingLLM fixes this by always retaining the first 4 initial sink tokens in the KV cache alongside the recent window, restoring normal attention behavior.

**Q6.4. How would you design a KV cache eviction policy for a system serving requests with highly variable lengths?**

**Answer:** I would implement a priority-based eviction policy combining several signals: (1) **Recency (LRU)**: evict the least recently accessed blocks first; (2) **Sequence completion probability**: use the request's current position and typical length distribution to estimate how close it is to completion — prefer evicting requests likely to complete soon; (3) **Block shareability**: protect blocks that are shared by multiple requests (prefix cache blocks) as they have high amortized value; (4) **Eviction cost**: prefer evicting from requests that can cheaply recompute the KV cache (short prompts) over those with long, expensive prefills. I would also implement block swapping to CPU DRAM as a second tier before outright eviction, since CPU memory is abundant. vLLM's preemption mechanism is a practical example: it pauses and re-schedules requests when memory pressure is high.

**Q6.5. Explain KV cache quantization. Can you quantize the KV cache more aggressively than model weights? Why or why not?**

**Answer:** KV cache quantization stores the keys and values (the attention cache) in reduced precision (INT8, INT4, or FP8) rather than FP16. There are good arguments for **more aggressive** quantization of KV cache than weights: (1) KV cache values are read once per attention step and do not accumulate errors across steps (each decode step reads fresh from the cache); (2) The attention operation is inherently normalized (softmax), providing some robustness to key quantization errors; (3) There is more redundancy across heads and layers in KV cache than in weight matrices. However, there are also counterarguments: activations (which the KV cache is derived from) exhibit outlier phenomena similar to activation quantization, making them harder to quantize than weights. In practice, INT8 KV cache is now well-established with minimal quality loss; INT4 KV cache is feasible with per-head or per-token scaling but requires careful implementation.

## Chapter 7

# Kernel Engineering and FlashAttention

### 7.1 Why Custom CUDA Kernels Matter

High-level frameworks like PyTorch compose operations from library primitives: a linear layer is a call to `torch.nn.Linear`, which calls into cuBLAS for the GEMM, then writes results to HBM, then reads them back for the next operation. This *eager execution* model creates massive HBM traffic for intermediate tensors that could have been kept in fast on-chip memory.

**Kernel fusion** combines multiple operations into a single GPU kernel, keeping intermediate results in registers or shared memory and avoiding HBM round-trips. For a fused LayerNorm + Linear + GeLU kernel, we reduce HBM traffic by  $3\times$  compared to three separate kernels.

### 7.2 FlashAttention: IO-Aware Exact Attention

Standard attention materializes the full  $n \times n$  attention matrix in HBM:

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}), \quad \mathbf{O} = \mathbf{P}\mathbf{V}$$

For  $n = 16,384$  and FP16, this matrix is  $16,384^2 \times 2 \approx 537$  MB. Reading and writing this matrix to HBM takes  $\sim 0.5$  ms on an A100 — and this must happen twice per forward pass (once for  $\mathbf{P}$  computation, once for the backward pass during training).

FlashAttention (Dao et al., 2022) eliminates this by **tiling**: decomposing the attention computation into blocks that fit in SRAM, computing the numerically stable softmax incrementally using the online softmax trick, and never materializing the full attention matrix in HBM.

#### FlashAttention Complexity

	Standard Attention	FlashAttention
HBM reads/writes	$O(n^2)$	$O(n^2/M)$ (fewer passes)
SRAM usage	$O(n^2)$	$O(M)$ (tile size)
Speed	Baseline	2–4 $\times$ faster
Sequence length limit	HBM limited	Longer contexts feasible

**FlashAttention-2** (Dao, 2023) improved parallelism by splitting the work across thread blocks along the sequence length dimension and reducing non-matrix-multiply operations. **FlashAttention-3** (Shah et al., 2024) further exploits H100-specific features: asynchronous memory copies with WGMMMA (Warpgroup Matrix Multiply Accumulate) instructions and FP8 precision, achieving  $\sim 75\%$  of H100 peak FLOP/s utilization.

### 7.3 The Online Softmax Trick

The core mathematical challenge of tiling attention is computing the exact softmax when we process the sequence in chunks. The online softmax (Milakov & Gimelshein, 2018) maintains a running maximum  $m$  and a running sum  $\ell$  to compute the softmax without seeing all values first:

$$m_i = \max(m_{i-1}, \max_j s_{ij}), \quad \ell_i = e^{m_{i-1}-m_i} \ell_{i-1} + \sum_j e^{s_{ij}-m_i}$$

At the end,  $\mathbf{O} = \mathbf{O}_{\text{running}}/\ell$  gives the exact softmax output. This is numerically stable and requires only  $O(1)$  extra state per row of the attention matrix.

### 7.4 Triton: Pythonic Kernel Programming

CUDA kernel development requires expertise in C++ and GPU-specific optimizations. Triton (Tillet et al., 2019) is a Python-based DSL for writing GPU kernels that abstracts away low-level details while generating highly optimized code:

```

1 import triton
2 import triton.language as tl
3
4 @triton.jit
5 def fused_softmax_kernel(X, Y, stride, N,
6                          BLOCK: tl.constexpr):
7     row = tl.program_id(0)
8     x_ptrs = X + row * stride + tl.arange(0, BLOCK)
9     x = tl.load(x_ptrs, mask=tl.arange(0, BLOCK) < N)
10    x_max = tl.max(x, axis=0)
11    x = x - x_max
12    numerator = tl.exp(x)
13    denominator = tl.sum(numerator, axis=0)
14    y = numerator / denominator
15    tl.store(Y + row * stride + tl.arange(0, BLOCK),
16            y, mask=tl.arange(0, BLOCK) < N)

```

Triton handles shared memory allocation, warp-level synchronization, and vectorization automatically, allowing ML engineers to write high-performance kernels without deep CUDA expertise.

#### Interview Preparation — Mock Q&A

**Q7.1.** Explain why FlashAttention is faster than standard attention. What is the key bottleneck it addresses?

**Answer:** Standard attention materializes the full  $n \times n$  attention matrix ( $\mathbf{QK}^\top$ ) in HBM. For long sequences, this matrix can be hundreds of megabytes to gigabytes, and reading/writing it to HBM is the dominant cost — not the actual arithmetic. FlashAttention is faster because it is **IO-aware**: it tiles the attention computation into blocks that fit in the GPU’s on-chip SRAM (shared memory), computing the softmax incrementally using the online softmax trick. Intermediate results stay in fast SRAM and are never written to HBM. The final output for each block is accumulated and written to HBM once. The

HBM traffic is reduced from  $O(n^2)$  to  $O(n \cdot d)$ , matching the theoretical minimum for reading inputs and writing outputs. On an A100, FlashAttention achieves  $2\text{--}4\times$  speedup over standard attention for long sequences.

**Q7.2. What is kernel fusion and why does it improve inference performance?**

**Answer:** Kernel fusion is the combination of multiple computational operations (that would normally each launch a separate GPU kernel) into a single kernel that retains intermediate results in on-chip memory (registers or shared memory) rather than writing them to HBM between operations. Without fusion: each operation writes its output to HBM, and the next reads it back — creating redundant memory traffic. With fusion: intermediate tensors live in registers or shared memory at  $>10$  TB/s effective bandwidth rather than the 2–3 TB/s of HBM. Common fusion patterns in LLM inference: (1) QKV projection fused with RoPE application; (2) LayerNorm fused with the following linear layer; (3) SiLU/Gelu gating fused with the FFN up-projection. Fused kernels in TensorRT-LLM and vLLM can reduce kernel launch overhead and HBM traffic simultaneously, yielding 10–40% latency improvements over unfused baselines.

**Q7.3. How does the online softmax trick enable tiled attention computation?**

**Answer:** The standard softmax  $\text{softmax}(\mathbf{x})_i = e^{x_i} / \sum_j e^{x_j}$  requires seeing all values in the row before computing any output, precluding tiling. The online softmax processes the row in chunks, maintaining a running maximum  $m$  and exponential sum  $\ell$ . When a new chunk arrives with values  $\mathbf{x}'$ : (1) Update running max:  $m' = \max(m, \max(\mathbf{x}'))$ ; (2) Rescale existing sum:  $\ell' = \ell \cdot e^{m-m'} + \sum e^{x'_j - m'}$ ; (3) Rescale existing output:  $\mathbf{O}' = \mathbf{O} \cdot e^{m-m'} / \ell' + \mathbf{V}' \cdot e^{\mathbf{x}' - m'} / \ell'$ . At completion,  $\mathbf{O}'$  equals the exact softmax-weighted sum. This enables FlashAttention to process  $Q, K, V$  in tiles, producing exact results without materializing the full attention matrix.

**Q7.4. What improvements did FlashAttention-3 introduce for the H100, and how does it exploit H100-specific features?**

**Answer:** FlashAttention-3 was specifically engineered for the H100’s new hardware capabilities: (1) **WGMMMA (Warpgroup Matrix Multiply Accumulate)**: The H100 introduced new asynchronous matrix multiply instructions at the warpgroup level, allowing FA3 to overlap memory copy and compute operations within a tile, hiding memory latency; (2) **TMA (Tensor Memory Accelerator)**: Hardware-managed asynchronous transfers between HBM and shared memory, reducing software overhead for data movement; (3) **FP8 support**: FA3 supports FP8 attention computation, doubling the effective Tensor Core throughput for attention; (4) **Ping-pong pipelining**: Two warpgroups alternate between loading the next tile and computing on the current tile, keeping both the Tensor Cores and memory system fully busy. Together, FA3 achieves  $\sim 75\%$  of the theoretical peak H100 FLOP/s on attention, vs.  $\sim 35\%$  for FA2.

**Q7.5. When would you use Triton instead of writing raw CUDA? What are the trade-offs?**

**Answer:** I’d use **Triton** when: (1) I need a custom kernel (e.g., a novel fused operation) but the team lacks deep CUDA expertise; (2) The kernel is “medium complexity” — not a simple elementwise operation (covered by PyTorch) but not requiring hand-tuned warp-level primitives; (3) Rapid iteration is important — Triton’s JIT compilation means I can experiment quickly. Triton’s **advantages**: Python syntax, automatic shared memory tiling, simpler debugging, and generated code that is often within 10–20% of

hand-optimized CUDA for many workloads. I'd use **raw CUDA** when: (1) Maximum performance is critical (e.g., the attention kernel in FlashAttention which requires extremely precise control of memory access patterns and warp synchronization); (2) I need fine-grained control over warp-level primitives, bank conflict avoidance, or asynchronous copy pipelines (TMA/WGMMA on H100). The trade-off is development time (CUDA: weeks; Triton: days) vs. peak performance.

## Chapter 8

# Serving Systems Architecture

### 8.1 Anatomy of a Production Inference Server

A production LLM inference server is a complex distributed system. Its core components are:

- **Frontend / API Gateway:** Receives HTTP requests, authenticates, rate-limits, and enqueues.
- **Scheduler:** The brain of the serving system. Decides which requests to run in the current batch, managing GPU memory and SLO constraints.
- **Execution Engine:** Runs the actual GPU forward passes, manages KV cache blocks.
- **Tokenizer:** Converts text to token IDs and back (typically runs on CPU).
- **Sampler:** Applies temperature, top-k, top-p, and other sampling logic to logits.

### 8.2 Batching Strategies

#### 8.2.1 Static Batching

The simplest approach: collect a fixed number of requests, pad them all to the length of the longest sequence, and process the padded batch. Highly inefficient: padding wastes compute and memory proportional to length variance.

#### 8.2.2 Continuous (Iteration-Level) Batching

Pioneered by Orca (Yu et al., 2022) and implemented in vLLM and TGI. Instead of batching at the request level, batch at the *iteration* (decode step) level. At every decode step, the scheduler can add new requests (whose prefill has completed) to the running batch, and completed sequences are immediately removed. This keeps GPU utilization high by eliminating the “long tail” problem of static batching.

#### 8.2.3 Chunked Prefill

Sarathi-Serve (Agrawal et al., 2024) observed that a single long prefill request can monopolize the GPU for hundreds of milliseconds, starving decode-phase requests (“prefill piracy”). Chunked prefill divides the prefill computation into smaller chunks that are interleaved with decode steps, providing more predictable latency across all request types.

### 8.3 Disaggregated Prefill and Decode

The fundamental tension between prefill (compute-bound) and decode (memory-bound) suggests they should be served on different hardware optimized for each regime. Disaggregated serving (Splitwise, Mooncake) physically separates:

- **Prefill nodes:** Maximize compute utilization (large batches, long prompts). Can use cheaper, older compute-optimized GPUs.
- **Decode nodes:** Maximize memory bandwidth, support high concurrency. Benefit from newer GPUs with high HBM bandwidth per parameter.

The challenge is **KV cache migration**: after prefill completes, the KV cache must be transferred from the prefill node to the decode node (over NVLink, InfiniBand, or RDMA), adding latency. Systems like Mooncake (from Kimi/Moonshot AI) use a distributed KV cache pool on secondary storage to decouple this entirely.

## 8.4 SLO-Aware Scheduling

Production systems must respect Service Level Objectives (SLOs) — typically expressed as p95 or p99 TTFT and TPOT targets. A naive FIFO scheduler may starve short requests behind long ones. SLO-aware scheduling techniques include:

- **Priority queuing:** Shorter/higher-priority requests jump ahead.
- **Preemption:** Pause a long-running request when its SLO trajectory is safe, free its KV cache blocks, and serve a higher-priority request.
- **Request routing:** Route requests to different server pools based on estimated length and server load.
- **Admission control:** Reject or queue requests when the system is overloaded to prevent SLO violations.

### Interview Preparation — Mock Q&A

#### Q8.1. Explain continuous batching. Why is it superior to static batching for LLM serving?

**Answer:** Static batching groups requests of similar length, pads shorter ones, and processes the batch until all requests complete. The problem: a batch of 32 requests where one is 10× longer than the others means 31 GPUs wait idle for that one request. Compute wasted on padding tokens is permanent. Continuous batching (Orca, vLLM) operates at the iteration level: at every decode step, the scheduler checks whether any running requests have completed (emit <EOS>) and replaces them immediately with newly arrived requests from the queue. No padding across requests of different lengths — each request occupies only the tokens it actually needs. GPU utilization stays consistently high because there is no “batch wait” for stragglers. In practice, continuous batching improves throughput by 5–23× over static batching at similar latency, according to the Orca paper’s benchmarks.

#### Q8.2. What is “prefill piracy” and how does chunked prefill address it?

**Answer:** Prefill piracy describes the situation where a request with a very long prompt monopolizes the GPU during its prefill phase — potentially for hundreds of milliseconds — while all decode-phase requests in the batch stall. During prefill, the GPU is computing GEMM over a large token matrix; decode requests that are in the middle

of generating tokens must wait, causing their TPOT (time per output token) to spike. Chunked prefill (Sarathi-Serve) divides the prefill of a long prompt into small chunks (e.g., 512 tokens per chunk) and interleaves each chunk with decode steps. This means a 4096-token prefill contributes only 512 tokens of compute before yielding to the decode batch, reducing decode TPOT degradation by 5–10× while adding only minimal overhead to total prefill time.

**Q8.3. Describe the architecture of vLLM. What are its key innovations and where are its limitations?**

**Answer:** vLLM (Virtual LLM) is a high-throughput serving system built around three core innovations: (1) **PagedAttention**: non-contiguous block-based KV cache memory management eliminating fragmentation; (2) **Continuous batching**: iteration-level batching for high GPU utilization; (3) **Custom attention kernels**: CUDA kernels for paged attention computation. The architecture: requests arrive at an API server, are tokenized, then scheduled by the **LLMEngine** which manages KV blocks via a **BlockManager**, and executed on GPU workers via a custom execution engine. Limitations: (1) vLLM’s current scheduler is not fully SLO-aware — it can starve short requests; (2) Tensor parallelism communication overhead can be significant for small models; (3) The Python-heavy scheduling layer adds latency for very short requests; (4) Disaggregated prefill/decode is not natively supported in the base vLLM; (5) Non-standard architectures (MoE, non-transformer models) require custom plugin development.

**Q8.4. How does disaggregated prefill/decode serving work? What are the engineering challenges?**

**Answer:** Disaggregated serving routes prefill computation to dedicated prefill nodes and decode computation to dedicated decode nodes. A request arrives at a router, is sent to a prefill node which runs the prompt forward pass and produces a KV cache, then the KV cache is transferred (via RDMA/InfiniBand) to a decode node which generates tokens until completion. The **engineering challenges** are substantial: (1) **KV cache transfer latency**: Transferring a 10 GB KV cache over a 400 Gb/s InfiniBand link takes ~200 ms — often longer than the prefill itself. Optimization: transfer KV blocks layer by layer, pipelining with decode; (2) **Resource balancing**: Prefill and decode nodes have different compute/memory ratios; over/under-provisioning either pool degrades overall throughput; (3) **Fault tolerance**: If a decode node fails mid-generation, the KV cache must be recovered; (4) **Routing complexity**: The router must estimate request length to balance load optimally.

**Q8.5. What metrics would you use to evaluate and compare two LLM serving systems?**

**Answer:** I would evaluate on: (1) **TTFT p50/p95/p99**: Critical for interactive applications — how fast does the first token appear; (2) **TPOT p50/p95**: Streaming smoothness — how fast do subsequent tokens arrive; (3) **Throughput (tokens/s)**: Total system output at a given hardware configuration; (4) **Goodput**: Fraction of requests meeting all SLO targets simultaneously (TTFT and TPOT); (5) **GPU utilization (MFU)**: Model FLOP Utilization — what fraction of peak GPU FLOP/s is being used productively; (6) **Cost per million tokens**: Total TCO normalized by output; (7) **Scalability**: How does performance change as concurrent users increase; (8) **Tail latency behavior under overload**: Does the system degrade gracefully or catastrophically. I’d run benchmarks across a realistic request distribution (mix of short and long prompts and outputs) rather than synthetic fixed-length benchmarks.

## Chapter 9

# Parallelism for Large Model Inference

### 9.1 Why Parallelism Is Necessary

A 405B-parameter model in BF16 requires 810 GB of GPU memory for weights alone. Even the largest individual GPU (H100 SXM with 80 GB HBM) cannot hold this model. Parallelism is not optional for frontier-scale models — it is the only way to serve them.

### 9.2 Tensor Parallelism

Tensor parallelism (Shoeybi et al., 2019 — Megatron-LM) shards individual weight matrices across GPUs. For a linear layer  $\mathbf{Y} = \mathbf{XW}$ :

**Column parallelism:** Split  $\mathbf{W}$  along columns:  $[\mathbf{W}_1|\mathbf{W}_2]$ . Each GPU computes  $\mathbf{Y}_i = \mathbf{XW}_i$  independently, then results are concatenated with a single **AllGather** collective.

**Row parallelism:** Split  $\mathbf{W}$  along rows. Each GPU computes a partial sum, then results are combined with an **AllReduce**.

In an MLP block, column parallelism on the first layer and row parallelism on the second requires only one **AllReduce** per transformer layer — efficient on NVLink but expensive over PCIe/InfiniBand.

**Scaling efficiency:** Tensor parallelism scales well up to the number of attention heads (8 GPUs for a model with 8 KV heads; beyond that, some heads have no work to do on some GPUs). Communication overhead grows quadratically with tensor parallelism degree, limiting practical scaling to 8 GPUs on a single NVLink-connected node.

### 9.3 Pipeline Parallelism

Pipeline parallelism assigns different layers of the transformer to different GPUs. GPU 0 holds layers 1–10, GPU 1 holds layers 11–20, etc. Data flows as a pipeline: GPU 0 computes and passes activations to GPU 1, which computes and passes to GPU 2, and so on.

The key challenge is **pipeline bubbles**: GPUs in later stages wait for earlier stages to produce activations. Micro-batching reduces bubble fraction (bubble fraction  $\approx (p-1)/p$  for pipeline depth  $p$  with  $m$  micro-batches,  $\approx \frac{p-1}{m+p-1}$ ).

Pipeline parallelism is communication-efficient (only activations at stage boundaries, not full AllReduce) and can scale across nodes connected by slow InfiniBand, making it suitable for very large models (>100B parameters).

### 9.4 Mixture of Experts Inference

Mixture of Experts (MoE) models (Mixtral, DeepSeek-V2, Grok-1) replace dense FFN layers with a collection of  $E$  expert networks. A router selects the top- $K$  experts for each token:

$$\mathbf{y} = \sum_{k \in \text{Top-K}} g_k \cdot \text{Expert}_k(\mathbf{x})$$

MoE models have high parameter counts but low **active parameter count per token**: Mixtral 8×7B has 47B total parameters but only 13B active per token. This makes inference faster (less compute per token) but harder to serve:

- **Expert parallelism**: Each GPU hosts a subset of experts. Tokens are routed across GPUs via `AllToAll` collectives.
- **Expert load imbalance**: Some experts may receive many tokens while others receive few, causing GPU imbalance.
- **Expert offloading**: For very large MoE models, experts are offloaded to CPU/NVMe and loaded on demand.

### Interview Preparation — Mock Q&A

**Q9.1. Explain tensor parallelism for a transformer’s MLP block. What communication collectives are required and when?**

**Answer:** In a standard MLP block  $\mathbf{Y} = \text{GeLU}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2$ , tensor parallelism across  $N$  GPUs: (1) **Column-parallel** first layer:  $\mathbf{W}_1$  is split along columns, so each GPU holds  $[\mathbf{W}_1]_{:,i:j}$  and independently computes a shard of  $\mathbf{H} = \text{GeLU}(\mathbf{X}\mathbf{W}_1)$ . No communication needed here. (2) **Row-parallel** second layer:  $\mathbf{W}_2$  is split along rows, matching the column sharding of  $\mathbf{H}$ . Each GPU computes  $\mathbf{Y}_i = \mathbf{H}_i\mathbf{W}_{2,i}$ , a partial sum. An `AllReduce` across all  $N$  GPUs combines these partial sums into the full  $\mathbf{Y}$ . Total: one `AllReduce` per MLP block, and one `AllReduce` per attention block (same structure). `AllReduce` volume:  $2 \times \text{batch size} \times \text{sequence length} \times d_{\text{model}}$  per layer. On NVLink this is fast; over PCIe it can dominate latency.

**Q9.2. When would you choose tensor parallelism over pipeline parallelism for inference?**

**Answer:** I’d choose **tensor parallelism** when: (1) The model fits on a single node’s GPUs with NVLink interconnect — TP requires frequent `AllReduce` which needs high bandwidth; (2) Minimizing latency is critical — TP reduces per-layer computation time proportionally to the number of GPUs (each GPU does  $1/N$  of the work per layer); (3) The model has many attention heads that can be evenly divided. I’d choose **pipeline parallelism** when: (1) The model is too large for a single node — PP only needs to communicate activations at layer boundaries, feasible over slower inter-node InfiniBand; (2) Throughput is more important than latency — PP introduces pipeline bubble latency but enables high batch throughput; (3) Expert parallelism for MoE models, which maps naturally to PP. In practice, production systems use **both** in combination (3D parallelism): TP within a node, PP across nodes.

**Q9.3. Explain the expert routing mechanism in an MoE model. What challenges does it create for inference serving?**

**Answer:** MoE models replace each FFN layer with  $E$  expert networks and a router. The router (usually a learned linear projection + softmax) computes routing weights for each token:  $g = \text{softmax}(\mathbf{x}\mathbf{W}_r)$ . The top- $K$  experts (usually  $K = 2$ ) are selected, and their outputs are weighted-summed. For inference serving, the challenges are: (1) **Dynamic routing**: Which expert processes which token changes with every input, making load balancing hard. Some experts may receive  $10\times$  more tokens than others in

a batch (expert collapse/load imbalance); (2) **Expert parallelism communication**: In a distributed setup, an **AllToAll** collective must route tokens to the correct expert GPU — this communication can bottleneck throughput; (3) **Memory**: Even if only 2 experts are active per token, all  $E$  experts' weights must be loaded. Expert caching strategies (preloading likely-accessed experts) can mitigate this.

**Q9.4. How does pipeline bubble fraction affect pipeline parallelism efficiency? How is it mitigated?**

**Answer:** When pipeline parallelism uses  $p$  pipeline stages and processes requests one by one, the first  $p - 1$  stages of the pipeline are idle (“bubble”) while the last stage finishes, and the first stage is idle while the last stage processes. The bubble fraction is  $(p - 1)/p$ : for  $p = 8$  stages,  $7/8$  of GPU-time is wasted. Mitigation via **micro-batching**: split the batch into  $m$  micro-batches and pipeline them. The bubble fraction becomes  $(p - 1)/(m + p - 1)$ . For  $p = 8, m = 32$ : bubble =  $7/39 \approx 18\%$ . Additional techniques: (1) **1F1B scheduling**: alternate forward and backward passes; (2) **Interleaved pipelines** (Narayanan et al.): assign multiple non-contiguous layer chunks to each GPU, reducing bubble but increasing communication; (3) For inference-only (no backward), bubble is less severe since we can simply fill with more requests.

**Q9.5. What is expert offloading for large MoE models, and what are its performance implications?**

**Answer:** Expert offloading moves expert weight matrices from GPU memory (HBM) to CPU DRAM or NVMe storage. For a model like DeepSeek-V2 with 236B total parameters ( $\sim 60$  experts in some MoE layers), only  $\sim 20$ B parameters are active per token — but all parameters must be somewhere accessible. Expert offloading works as follows: an expert-aware scheduler predicts which experts will be needed for the next batch based on the routing decisions, and prefetches their weights from CPU/NVMe to GPU ahead of time. Weights for currently unused experts are evicted. **Performance implications**: PCIe bandwidth ( $\sim 32$  GB/s) is far slower than HBM bandwidth (2–3 TB/s), so any uncached expert load adds significant latency ( $\sim$ ms per expert). This only works well when batch size is large enough (so expert access patterns are predictable) and when the expert cache hit rate is high. With 70%+ cache hit rate, expert offloading can enable serving 200B+ MoE models on a single 8-GPU node.

## Chapter 10

# Long-Context and Memory Management

### 10.1 The Quadratic Attention Problem

Attention’s  $O(n^2)$  complexity in sequence length  $n$  is the central obstacle to long-context LLM inference. For  $n = 1,000,000$  tokens (1M context, as in Gemini 1.5 Pro), the naive attention matrix would require  $1,000,000^2 \times 2$  bytes = 2 petabytes — clearly infeasible. Even for  $n = 128,000$  (128K context), the attention matrix is 32 GB.

Modern long-context approaches attack this problem from several angles.

### 10.2 FlashAttention for Long Context

As discussed in Chapter 7, FlashAttention avoids materializing the full attention matrix by tiling. This does not reduce the  $O(n^2)$  compute complexity, but it reduces the  $O(n^2)$  HBM memory requirement to  $O(n)$ , making attention computationally feasible at 128K tokens on current hardware.

### 10.3 Ring Attention

Ring Attention (Liu et al., 2023) distributes long-context attention computation across multiple devices arranged in a ring. Each device holds a chunk of the sequence’s QKV tensors, and devices pass KV chunks around the ring while computing local attention. The key insight is that the online softmax trick allows correct normalization even when a device only sees one KV chunk at a time. Ring Attention reduces per-device memory from  $O(n^2)$  to  $O(n/p)$  where  $p$  is the number of devices, enabling million-token contexts across device clusters.

### 10.4 Context Compression

Rather than computing full attention over all context, context compression techniques selectively summarize or remove low-information tokens:

**LLMLingua** (Jiang et al., 2023) uses a small language model to score each token’s perplexity given its context; low-perplexity (easily predictable) tokens are dropped. Compression ratios of 4–20× are achievable with <5% performance degradation on many tasks.

**AutoCompressor** fine-tunes the model itself to summarize long contexts into a fixed number of “summary vectors,” which are then used as a compressed KV cache.

### 10.5 State Space Models as Linear Attention

Mamba (Gu & Dao, 2023) and its successors are state space models (SSMs) that process sequences through a learned recurrent state rather than attention:

$$h_t = Ah_{t-1} + Bx_t, \quad y_t = Ch_t$$

where  $A$ ,  $B$ ,  $C$  are input-dependent in selective SSMs (S6). This achieves  $O(n)$  time and  $O(1)$  memory for inference (the hidden state  $h$  is fixed-size regardless of sequence length). At inference time, SSMs are equivalent to RNNs: each new token requires only updating the state, with no KV cache. The trade-off is that SSMs have weaker in-context retrieval capability compared to attention for tasks requiring precise recall of specific prior context.

### Interview Preparation — Mock Q&A

#### Q10.1. How does Ring Attention achieve distributed long-context inference? What is its communication cost?

**Answer:** Ring Attention distributes the sequence across  $p$  devices. Device  $i$  holds query chunk  $Q_i$  and key-value chunks  $(K_i, V_i)$ . In each of  $p$  communication steps, devices pass their KV chunk to the next device in the ring and receive the previous device's KV chunk. While waiting for the next KV chunk (overlapped with computation), each device computes partial attention scores between its local  $Q_i$  and the current KV chunk, accumulating the online softmax statistics. After  $p$  steps, each device has computed the correct attention output for its query chunk. **Communication cost:** each step transfers  $n/p$  tokens of KV cache (two tensors: K and V). Total communication per layer:  $n \times d_{\text{head}} \times n_{\text{heads}} \times 2 \times \text{precision}$ , identical to the total KV data — no extra communication overhead. Communication is pipelined with computation.

#### Q10.2. Compare Mamba (SSM) and Transformer attention for long-context inference. When would you use each?

**Answer:** **Transformer attention:**  $O(n^2)$  compute and  $O(n)$  KV cache memory. Excels at tasks requiring precise recall of specific details anywhere in the context (e.g., “What did the document say about X on page 3?”). Performance degrades gracefully as context grows if FlashAttention is used. **Mamba:**  $O(n)$  compute and  $O(1)$  state (fixed-size recurrent state). Highly efficient for very long sequences — each token requires constant computation regardless of total context length. Weakness: selective SSMs have difficulty with precise retrieval from distant context, performing worse than attention on retrieval-heavy benchmarks (RULER, passkey retrieval). **When to use Mamba:** streaming inference over very long documents, time-series prediction, applications where the model needs to track evolving state rather than retrieve specific facts. **Hybrid models** (Jamba, Zamba, Mamba-2 + attention) combine both — using SSM layers for most processing and sparse attention layers for retrieval.

#### Q10.3. How does LLMingua compress prompts? What are the risks of prompt compression?

**Answer:** LLMingua uses a small surrogate LLM to estimate the perplexity of each token given its (possibly compressed) context. Tokens with low perplexity are deemed predictable and thus low-information, making them candidates for removal. The algorithm: (1) Coarse granularity: determine which sentences or phrases to keep; (2) Fine granularity: within kept segments, remove low-perplexity tokens; (3) Target compression ratio determines aggressiveness. **Risks:** (1) The surrogate model's notion of “predictability” may differ from the target model's, causing important tokens to be dropped; (2) Removing tokens changes positional context for all subsequent tokens — models with absolute positional encodings may be particularly sensitive; (3) For retrieval-heavy tasks (precise name/number lookup), even low-perplexity tokens may be critically important; (4)

Compression is irreversible — errors cannot be recovered. Practical guideline: validate LLMingua on your specific task distribution before production deployment.

**Q10.4. Explain the difference between YaRN and NTK-aware scaling for RoPE context extension.**

**Answer:** Both YaRN and NTK-aware scaling attempt to extend a model trained with RoPE on context length  $L$  to longer contexts  $L' > L$  without fine-tuning (or with minimal fine-tuning). **NTK-aware scaling** (bloc97, 2023) rescales the RoPE base frequency: instead of base  $b = 10,000$ , use  $b' = b \cdot (L'/L)^{d/(d-2)}$ . This spreads rotations more evenly across the extended range, keeping high-frequency dimensions from wrapping. However, it applies the same scaling to all frequency dimensions. **YaRN** (Peng et al., 2023) uses a more sophisticated approach: it does not scale low-frequency dimensions at all (they already have good far-range coverage), applies NTK-style scaling to medium frequencies, and linearly interpolates high frequencies. This preserves in-distribution behavior for short sequences while enabling longer sequences. YaRN also applies an attention temperature correction (scaling attention scores by  $\sqrt{1/t}$ ) to compensate for the changed attention distribution. YaRN consistently outperforms NTK-aware scaling on perplexity and retrieval benchmarks at extended context lengths.

**Q10.5. What is RAG, and how does it relate to long-context inference? When would you choose one over the other?**

**Answer:** RAG (Retrieval-Augmented Generation) selectively fetches relevant documents from a vector database at query time, inserting only the most relevant passages into the prompt. It effectively replaces the need for a very long context window by using a retrieval system. **Long-context inference** instead puts all potentially relevant information directly in the model's context window. **Choose RAG when:** (1) The corpus is very large (millions of documents) — far exceeding any feasible context window; (2) Retrieval is precise — the query clearly maps to specific documents; (3) Latency from retrieval indexing is acceptable; (4) Cost of long-context compute is prohibitive. **Choose long-context when:** (1) The task requires synthesizing or comparing across many parts of a document (e.g., complex reasoning over a full book); (2) Retrieval is noisy — the model benefits from access to full context to distinguish relevant from irrelevant; (3) The document corpus is small enough to fit ( $<1\text{M}$  tokens). The ideal system often combines both: RAG to narrow the candidate set, then long-context to reason over the retrieved passages.

## Chapter 11

# Edge and On-Device Inference

### 11.1 The Edge Inference Landscape

Running LLMs on edge devices — smartphones, laptops, embedded systems — is increasingly important for privacy, latency, and offline capability. The constraints are severe: a flagship smartphone may have 8–16 GB of unified memory (shared between CPU and GPU) and a power budget of  $\sim 10$  watts. By contrast, an H100 GPU consumes 700 watts with 80 GB dedicated HBM.

Edge inference engineering is about getting the most intelligence within these constraints.

### 11.2 Memory Constraints and Quantization for Edge

A 7B-parameter model in FP16 requires 14 GB — exceeding the memory of most smartphones. 4-bit quantization (AWQ/GPTQ) reduces this to 3.5 GB, making 7B models feasible on high-end phones. The Apple iPhone 16 Pro with 8 GB RAM can run 4-bit Llama 3 8B with shared system memory management.

More aggressive quantization (2-bit, QuIP#) reduces to  $\sim 1.8$  GB, enabling 7B models on mid-range devices, though with some quality degradation.

### 11.3 Runtime Frameworks

**llama.cpp** (Gerganov, 2023) is a pure C/C++ implementation of LLaMA and compatible models with:

- CPU inference via AVX-512, NEON (ARM), and WebAssembly
- GPU offloading (partial or full) to Metal (Apple), CUDA, Vulkan, OpenCL
- Custom k-quant formats (Q4\_K\_M, Q5\_K\_M etc.) with group quantization for quality-size Pareto efficiency
- Extremely memory-efficient attention via mmap-based weight loading

**Apple MLX** is Apple’s native ML framework for Apple Silicon, exploiting the unified memory architecture. MLX enables running 70B models (in 4-bit) on Mac Studio with M3 Ultra (192 GB), achieving  $\sim 30$  tokens/s.

**ONNX Runtime** provides cross-platform inference with hardware acceleration across CPUs, GPUs, and NPUs from different vendors.

### 11.4 Neural Architecture Search for Efficiency

Neural Architecture Search (NAS) automates the design of model architectures optimized for specific hardware constraints. For edge deployment, NAS can discover:

- Optimal layer widths and depths for a given latency budget

- Mixed-precision assignment (different layers at different bit-widths)
- Attention head pruning patterns
- Block skipping or early exit strategies

Hardware-aware NAS (HAT, OFA) trains a “supernet” from which sub-networks of different sizes can be extracted at inference time without re-training.

### Interview Preparation — Mock Q&A

**Q11.1. You are asked to deploy a competitive language model on a smartphone with 6 GB RAM. What is your approach?**

**Answer:** Available RAM of 6 GB must be shared with the OS and other applications, leaving roughly 4–4.5 GB for the model. My approach: (1) Start with a 3B or 4B parameter model (Llama 3.2 3B, Phi-3 Mini 3.8B, Gemma 2 2B) which require only 1.5–2 GB at 4-bit; (2) Apply Q4\_K\_M quantization via llama.cpp or AWQ — this strikes the best quality-size balance for small models; (3) Use Metal GPU offloading on iOS (llama.cpp + Metal backend) for accelerated inference; (4) Implement prompt caching to avoid recomputing system prompt KV cache; (5) Apply speculative decoding with a smaller draft (e.g., a 1B model) if latency is critical; (6) Target 10–20 tokens/s which feels responsive for a chat interface. If 3B is insufficient quality, consider 7B at Q2\_K (~3.5 GB) accepting some quality degradation.

**Q11.2. Explain Apple MLX and why it is well-suited for LLM inference on Apple Silicon.**

**Answer:** MLX is Apple’s array computation library for Apple Silicon, designed with the unified memory architecture in mind. Unlike PyTorch which treats CPU and GPU as separate memory spaces requiring explicit data copies, MLX operates on a single shared memory space accessible from both CPU and GPU without copying. For LLM inference, this is transformative: model weights loaded once into system memory are immediately accessible to the GPU (Apple Silicon’s integrated GPU), eliminating the PCIe bottleneck. On an M3 Ultra with 192 GB of unified memory and 800 GB/s bandwidth, a 70B model at 4-bit (~40 GB) fits comfortably with room for KV cache, achieving ~30 tokens/s. MLX also provides a Python interface, JIT compilation, lazy evaluation, and growing model support through the `mlx-lm` package. It represents the most practical path to running frontier-sized models on consumer hardware.

**Q11.3. What is the k-quant system in llama.cpp? How does it differ from standard uniform quantization?**

**Answer:** Llama.cpp implements “k-quants” — a family of mixed-precision block quantization formats. Rather than quantizing each weight independently to  $k$  bits with a per-tensor or per-channel scale, k-quants use a hierarchical scheme: weights are grouped into blocks (typically 32 weights), and within each block, some weights are stored at higher precision than others. For example, in Q4\_K\_M: the “K” suffix indicates the super-block quantization scheme, and “M” indicates a quality tier (small, medium, large). Scale and minimum values are stored at higher precision per super-block, while individual weights use 4-bit encoding. This is superior to uniform quantization because: (1) Block-level scales adapt to local weight distributions, reducing quantization error; (2) Important weights (near the tails of the distribution) receive better representation; (3) The result-

ing quality/size ratio outperforms naive INT4 for the same bit budget. Q4\_K\_M is widely considered the best practical 4-bit format for llama.cpp.

**Q11.4. Compare llama.cpp CPU inference vs GPU inference. When would you prefer CPU inference?**

**Answer:** **GPU inference** (Metal, CUDA, Vulkan) uses the device’s parallel compute units and high bandwidth memory for maximum throughput — typically 5–20× faster than CPU for the same model. **CPU inference** uses AVX-512/NEON SIMD instructions, system RAM (much slower than GPU HBM), and sequential processing. CPU inference is slower but has advantages: (1) **No VRAM limit**: System RAM (64–512 GB on desktop) can hold much larger models than any GPU’s VRAM; (2) **Thermal**: CPUs can sustain inference longer in passive cooling environments; (3) **Availability**: CPU inference works on any machine with no GPU. I’d prefer CPU when: (1) The model is too large for the GPU’s VRAM; (2) The device has no GPU or a very weak integrated GPU; (3) Throughput is not critical and batch processing at low speed is acceptable. Partial GPU offloading (llama.cpp’s `-ngl` flag) lets you offload as many layers as fit in GPU VRAM and run the rest on CPU, balancing both.

**Q11.5. What is Hardware-Aware NAS? How does OFA (Once-For-All) enable efficient edge deployment?**

**Answer:** Hardware-Aware NAS searches for model architectures that are Pareto-optimal in accuracy vs. target hardware metric (latency, energy, memory). Instead of a single fixed model, it produces families of models with different accuracy-efficiency trade-offs, each optimized for specific hardware. **Once-For-All (OFA)** (Han et al., 2020) trains a single large “supernet” from which sub-networks of different depths, widths, and kernel sizes can be extracted without re-training. OFA uses progressive shrinking: first train the full network, then progressively train smaller sub-networks by sampling them from the full network and applying knowledge distillation from the full network. At deployment time, given a target device and latency constraint, a lookup table or predictor estimates sub-network latency, and the best-accuracy sub-network within the budget is selected. For LLM inference, OFA-style approaches can produce models with 2–5× different sizes, all derived from one training run.

## Chapter 12

# Inference-Time Compute Scaling

### 12.1 The New Scaling Paradigm

For years, the dominant scaling law was: more training compute  $\rightarrow$  better models. Chinchilla optimal training suggests spending the compute budget equally on model size and training tokens. But 2024–2025 saw the emergence of a complementary paradigm: **inference-time compute scaling** — spending more compute *at inference time* to improve output quality.

The key insight: for many tasks (especially reasoning, math, and coding), a smaller model that “thinks longer” can outperform a larger model that “answers immediately.” The question shifts from “how big a model can we train?” to “how much reasoning can we afford at inference time?”

### 12.2 Chain-of-Thought and Its Inference Cost

Chain-of-Thought (CoT) prompting (Wei et al., 2022) instructs the model to produce intermediate reasoning steps before the final answer. This dramatically improves performance on multi-step reasoning tasks but comes at a cost: a problem that requires a 10-token answer may require 500 tokens of reasoning, multiplying inference cost by  $50\times$ .

**Inference-time compute scaling question:** Given a fixed compute budget, is it better to run a larger model once, or a smaller model with more reasoning tokens?

Recent results (Snell et al., 2024, “Scaling LLM Test-Time Compute Optimally”) suggest the answer depends on problem difficulty: for hard problems, more reasoning on a smaller model is often more efficient; for easy problems, the overhead of extended reasoning is wasteful.

### 12.3 Process Reward Models and Search

A **Process Reward Model (PRM)** is trained to score the quality of intermediate reasoning steps, not just final answers. PRMs enable search: instead of following a single reasoning chain, generate multiple candidate reasoning paths and use the PRM to select or rerank the most promising one.

Common search strategies:

- **Best-of-N:** Generate  $N$  independent solutions and pick the one with the highest PRM score (or majority vote).
- **Beam search:** Maintain  $K$  partial reasoning paths, expanding the most promising at each step.
- **Monte Carlo Tree Search (MCTS):** Use the PRM as a value function in a tree search.

Best-of-N with  $N = 64$  can match or exceed the performance of a model  $10\times$  larger on math benchmarks (MATH, AIME).

## 12.4 DeepSeek-R1 and o1-Style Architectures

OpenAI o1 and DeepSeek-R1 represent a qualitative shift: rather than prompting for CoT, these models are *trained* to reason via reinforcement learning. The key mechanism: the model generates extended internal monologues (“thoughts”) before producing a final answer. These thoughts are trained via RLHF/GRPO against verifiable rewards (e.g., math answer correctness):

$$\mathcal{L}_{\text{GRPO}} = -\mathbb{E}_o \left[ \min \left( \frac{\pi_\theta(o|q)}{\pi_{\text{ref}}(o|q)} A_o, \text{clip}(\cdot, 1 \pm \epsilon) A_o \right) \right] + \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$$

The result is models that can “think” for thousands of tokens on hard problems, trading inference cost for accuracy. DeepSeek-R1-Zero demonstrates that this reasoning behavior can emerge from pure RL without supervised reasoning traces.

### Interview Preparation — Mock Q&A

**Q12.1. What is inference-time compute scaling? How does it differ from training-time compute scaling?**

**Answer:** Training-time compute scaling (Kaplan et al., 2020; Hoffmann et al., 2022 - Chinchilla) studies how model capability grows with training FLOP budget — more parameters and more training tokens produce better models. Inference-time compute scaling is a newer paradigm: given a fixed trained model, how does performance scale when we spend more compute *at inference time*? More inference compute can be spent via: (1) generating more reasoning tokens (Chain-of-Thought); (2) generating multiple candidate solutions and selecting the best (Best-of-N sampling); (3) performing tree search with a verifier or reward model; (4) iterative refinement (the model critiques and improves its own output). The key insight is that inference compute can substitute for training compute: a 7B model spending  $64\times$  more inference compute can match a 70B model on some reasoning tasks. This changes the economics of AI deployment: the compute budget shifts from a one-time training cost to a per-query inference cost.

**Q12.2. Explain Process Reward Models (PRMs). How do they differ from Outcome Reward Models (ORMs)?**

**Answer:** An **Outcome Reward Model (ORM)** assigns a single reward score to a complete solution, based only on whether the final answer is correct. It cannot distinguish between a correct answer reached through flawed reasoning from one reached through valid reasoning. An ORM cannot guide search through intermediate steps. A **Process Reward Model (PRM)** is trained to score each individual reasoning step for correctness, providing dense feedback throughout the reasoning chain. PRMs require step-level annotations in training data (labor-intensive to collect, though automated methods exist). At inference time, PRMs enable search at the step level: MCTS, beam search, and step-level reranking all become possible. PRMs are superior to ORMs for complex multi-step reasoning where the intermediate path matters (preventing the model from getting the right answer for the wrong reason), but are significantly more expensive to train and apply.

**Q12.3. What is Best-of-N sampling? When is it efficient and when is it not?**

**Answer:** Best-of-N (BoN) sampling generates  $N$  independent candidate outputs from

the model (with temperature  $> 0$ ), scores each with a reward model or verifier, and returns the highest-scoring output. It is one of the simplest inference-time compute scaling methods. **When efficient:** (1) Tasks with verifiable correct answers (math, coding, formal logic) where a verifier is cheap and reliable; (2) When diversity of generations is high — if all  $N$  samples are identical, BoN provides no benefit; (3) When  $N$  is small ( $\leq 32$ ) and the per-sample inference cost is low. **When inefficient:** (1) Open-ended tasks without a reliable evaluator (creative writing) — subjective quality is hard to score automatically; (2) Very large  $N$  — BoN quality grows as  $O(\log N)$  in many settings (the expected maximum of  $N$  samples from a distribution), delivering diminishing returns; (3) Expensive models where  $N$  samples costs  $N \times$  more than a single sample — better to use beam search or step-level refinement for the same compute budget.

**Q12.4. Describe the GRPO training objective used in DeepSeek-R1. How does it differ from PPO?**

**Answer:** GRPO (Group Relative Policy Optimization) is an RL algorithm for LLM fine-tuning that avoids the need for a separate value network (critic). Instead of estimating the advantage function  $A$  using a value network (as PPO does), GRPO computes the advantage for each output relative to the *group* of outputs generated for the same question:  $A_i = (r_i - \text{mean}(\mathbf{r})) / \text{std}(\mathbf{r})$ , where  $\mathbf{r}$  is the vector of rewards for the group. This eliminates the need to train a value function (saving memory and compute), reduces variance by normalizing within-group, and avoids the instability issues of value function estimation. Compared to PPO: GRPO is simpler (fewer hyperparameters), more memory-efficient (no value network), but may have higher variance per update since the baseline is estimated from the current batch. DeepSeek-R1-Zero demonstrates that GRPO + simple verifiable rewards alone can induce complex reasoning behaviors like self-verification and extended thinking.

**Q12.5. For a math tutoring application, would you use a larger base model or a smaller model with extended inference compute? Justify your answer.**

**Answer:** I would choose a **smaller model with extended inference compute**, for several reasons: (1) **Math is verifiable:** Correct answers can be programmatically checked, enabling reliable Best-of-N or PRM-guided search without human evaluation; (2) **Reasoning traces have pedagogical value:** A math tutor that shows step-by-step work is more useful than one that gives only the answer; CoT reasoning at inference time is a feature, not overhead; (3) **Cost efficiency:** Serving a 7B R1-style model with extended inference is cheaper per query than serving a 70B dense model, even accounting for the extra tokens; (4) **Adaptable compute budget:** For easy problems, we can use fewer reasoning tokens; for hard problems, more — providing compute-adaptive serving that a fixed large model cannot. The practical choice would be something like a 7B or 14B fine-tuned reasoning model (DeepSeek-R1-Distill or similar) with configurable inference budget, rather than a 70B baseline model.

## Chapter 13

# Observability and Production Engineering

### 13.1 The Production Inference Stack

Deploying an LLM to production is not complete when the model passes offline benchmarks. Production demands observability, reliability, cost management, and continuous improvement. The production inference stack adds several layers on top of the serving engine:

- **Logging and Tracing:** Every request logged with TTFT, TPOT, token counts, model version, and error state.
- **Metrics and Alerting:** Dashboards tracking p50/p95/p99 latency, throughput, GPU utilization, error rate.
- **A/B Testing Infrastructure:** Traffic splitting between model versions for safe rollout.
- **Capacity Planning:** Forecasting query volume and pre-provisioning hardware accordingly.
- **Cost Attribution:** Per-team or per-feature breakdown of token costs.

### 13.2 Key Metrics in Depth

**TTFT (Time to First Token):** The duration from request receipt to the first generated token being returned to the user. TTFT is dominated by the prefill computation. Users perceive TTFT as “thinking time” — a TTFT >1s makes the application feel slow even if subsequent tokens stream quickly.

**TPOT (Time Per Output Token):** The average time between consecutive output tokens during the streaming phase. Humans can read ~250 words/minute (~5 tokens/s), so TPOT < 200 ms typically feels smooth. TPOT is dominated by decode step latency and GPU memory bandwidth.

**Goodput:** The fraction of requests that meet all SLO targets simultaneously (both TTFT SLO and TPOT SLO). A system with 95% TTFT SLO compliance and 95% TPOT SLO compliance has goodput of  $0.95^2 \approx 90\%$  if these are independent. Goodput is more informative than individual metric compliance.

**Model FLOP Utilization (MFU):** The fraction of theoretical peak FLOP/s that is being used productively for model computation. A well-optimized system achieves 30–60% MFU during prefill and much lower during decode (due to memory-bandwidth bottleneck). MFU is the key metric for understanding hardware efficiency.

### 13.3 Failure Modes and Debugging

Symptom	Likely Cause	Diagnostic
High TTFT spikes	Long queuing time; prefill piracy	Queue depth monitoring, prefill latency breakdown
TPOT regression	KV cache evictions; increased batch size	KV cache hit rate, decode step latency
OOM (Out of Memory)	KV cache overflow; request surge	Memory monitoring, PagedAttention block stats
Throughput cliff	Tensor parallel communication bottleneck	GPU utilization vs. NVLink bandwidth
Quality regression	Model version issue; sampling parameter change	Model version tagging in logs; online eval

### 13.4 Cost Modeling

Understanding and minimizing cost per million output tokens is critical for business viability. The major cost components are:

$$\text{Cost per 1M tokens} = \frac{\text{GPU cost per hour} \times \text{hours}}{\text{total tokens generated}}$$

Levers to reduce cost:

1. **Maximize GPU utilization:** Batch more requests, use continuous batching.
2. **Reduce model compute:** Quantization, distillation, smaller model.
3. **Spot/preemptible instances:** 60–80% cheaper than on-demand, acceptable for batch workloads.
4. **Optimize hardware selection:** Older GPU generations (A100 vs. H100) may be more cost-efficient for memory-bound workloads.
5. **Reduce output token count:** Instruction-tune the model to be more concise.

#### Interview Preparation — Mock Q&A

**Q13.1.** How would you debug a sudden TTFT p99 spike in a production LLM serving system?

**Answer:** My investigation would follow a structured funnel: (1) **Network layer:** Check if p99 TTFT increase is correlated with increased request arrival rate — if yes, the system may be queuing. Check queue depth metrics. (2) **Prefill computation:** Isolate prefill latency from queue wait time. If prefill is spiking, check whether input prompt length distribution changed (longer prompts take longer to prefill). (3) **Resource contention:** Check GPU utilization and NVLink bandwidth — if another workload was co-deployed, it may be competing for GPU resources. (4) **KV cache pressure:** If PagedAttention block allocation rate increased, request throughput may have increased, causing compute contention. (5) **Model version or configuration change:** Compare logs to identify

any deployment in the same time window. (6) **Hardware fault**: A degraded GPU or NVLink link can cause throughput reduction. Use `nvidia-smi` to check error counts. I'd correlate all these with the spike timestamp and work backward.

**Q13.2. What is goodput and why is it more meaningful than raw throughput or individual metric SLO compliance?**

**Answer:** **Throughput** measures total tokens generated per second — but tells you nothing about whether users are getting acceptable experiences. A system can have high throughput by batching many long requests while individual TTFT violates SLOs. **Individual SLO compliance** (e.g., “95% of requests meet TTFT SLO”) is better, but multiple SLOs may be measured independently, hiding compound violations. **Goodput** is the fraction of requests satisfying *all* SLO requirements simultaneously — TTFT SLO AND TPOT SLO AND any other defined targets. It captures the rate at which the system is actually serving users satisfactorily. If TTFT compliance is 90% and TPOT compliance is 90%, goodput could be as low as 81% if violations are independent. Goodput is the metric I'd optimize for in scheduling decisions: an SLO-aware scheduler should maximize goodput, not just total throughput.

**Q13.3. How would you design an A/B testing framework for comparing two LLM versions in production?**

**Answer:** My design: (1) **Traffic splitting**: Use a consistent hash on user ID (not request ID) to split traffic deterministically — same users always hit the same model variant, reducing confounding from user preference variations. Typical split: 90/10 initially, widening as confidence grows. (2) **Metrics**: Track latency (TTFT/TPOT), quality signals (thumbs up/down, session length, follow-up questions), and cost per request for both variants. (3) **Statistical significance**: Use sequential A/B testing (e.g., SPRT) to determine when enough data has been collected, with pre-specified Type I/II error bounds. (4) **Guardrails**: Automatic rollback triggers — if safety violations, error rate, or latency degrade beyond a threshold on the test variant, automatically redirect all traffic to control. (5) **Logging**: Tag every request and response with variant ID, model version, and full timing breakdown for post-hoc analysis. (6) **Rollout stages**: Shadow mode (no user impact) → 1% → 10% → 50% → 100%.

**Q13.4. Explain how you would capacity plan for an LLM serving system expecting 10× user growth in 6 months.**

**Answer:** Capacity planning steps: (1) **Baseline measurement**: Current queries per second (QPS), average input/output token lengths, and p95 latency. (2) **Forecast QPS**: Project 10× QPS growth with seasonality (peak/off-peak). (3) **Throughput modeling**: Given current GPU count and throughput per GPU, compute GPU-hours needed at peak. Add 30–50% headroom for burstiness. (4) **Cost optimization**: Before scaling horizontally, exhaust vertical optimizations: can we apply 4-bit quantization to double throughput per GPU? Upgrade to H100s which have ~2× the memory bandwidth of A100s? Enable prefix caching for system prompts? (5) **Auto-scaling**: Implement Kubernetes-based or cloud-native autoscaling with GPU provisioning lead time baked in (GPU instances take 2–10 min to initialize). (6) **Multi-region**: Plan for geographic distribution to reduce latency and avoid single-region capacity constraints. Budget assumption: each H100 at \$2/hour can serve ~1M output tokens/hour at full utilization.

**Q13.5. How would you measure and improve Model FLOP Utilization (MFU)?**

**Answer: Measuring MFU:** Compute the number of FLOPs per forward pass (for a transformer:  $\approx 6N$  for model with  $N$  parameters per token, times tokens per second), then divide by peak GPU FLOP/s. Using NVIDIA Nsight Compute: measure “SM Throughput” as a fraction of peak. Typical values: 30–45% MFU for prefill (GEMM-bound), <10% for decode (memory-bound). **Improving MFU:** For prefill: (1) Use FlashAttention-3 to improve attention kernel utilization; (2) Profile and fuse layernorm/residual operations; (3) Use tensor parallel across multiple GPUs for large batches; (4) Ensure batch sizes are aligned to Tensor Core tile sizes (multiples of 16). For decode: MFU is inherently low and the right metric is memory bandwidth utilization instead — aim for >80% of peak HBM bandwidth. Improve by: (1) Increasing batch size to amortize weight-loading cost; (2) Quantizing weights to reduce bytes loaded per token; (3) Using speculative decoding to process multiple tokens per step.

## Chapter 14

# The Future of Inference

### 14.1 Diffusion-Based Language Models

Autoregressive generation — one token at a time — is not the only paradigm. **Diffusion language models** (MDLM, Plaid, LLaDA) apply diffusion processes to discrete token sequences: a forward process gradually masks tokens, and the model learns to denoise (unmask) them. Unlike autoregressive models, diffusion LLMs can generate all tokens in parallel (or semi-parallel), potentially achieving much higher hardware utilization during generation.

The challenge: diffusion models for text have not yet matched autoregressive models in quality, and the iterative denoising process (typically 10–100 denoising steps, each touching all tokens) can be more expensive than expected. However, recent models like LLaDA have closed the gap significantly.

### 14.2 Multi-Token Prediction

Meta’s multi-token prediction (Gloeckle et al., 2024) trains models with multiple output heads that simultaneously predict the next  $k$  tokens rather than just one. During inference, this can be used similarly to Medusa (self-speculative decoding) to generate multiple tokens per step, or the multiple heads can be used to generate better internal representations for downstream tasks. Models trained with 4-token prediction show better sample efficiency and representational quality than standard next-token prediction.

### 14.3 Neuromorphic and Photonic Computing

**Neuromorphic chips** (Intel Loihi 2, IBM NorthPole) implement neuron-like spiking computation that is extremely energy-efficient for sparse activations — orders of magnitude more efficient per operation than GPU deep learning. IBM’s NorthPole chip achieves 22 TOPS/watt for dense neural inference vs.  $\sim 2$  TOPS/watt for GPU. As LLM architectures evolve toward sparsity, neuromorphic hardware may become relevant.

**Photonic matrix multipliers** (Lightmatter, Luminous Computing) perform matrix-vector multiplication using light, achieving theoretically unlimited bandwidth and near-zero energy per multiplication. Photonic computing is early-stage but could fundamentally change the memory-bandwidth bottleneck by making weight loading essentially free.

### 14.4 The Inference-First Design Philosophy

The field is increasingly moving toward designing models with inference in mind from the start:

- **Architecture choices for KV efficiency:** MQA, GQA, MLA, and SSM layers reduce KV cache footprint.
- **Activation sparsity:** Models trained with ReLU or top-K activation functions have sparse activations that can be exploited for faster FFN computation.

- **Mixture of Depths** (Raposo et al., 2024): Dynamic computation where tokens attend to fewer layers based on difficulty.
- **Early exit**: Confidence-based early stopping at intermediate layers for easy tokens.
- **Byte-level models**: Operating on bytes rather than subword tokens removes the tokenization bottleneck and enables better multilingual performance.

### Interview Preparation — Mock Q&A

#### Q14.1. How do diffusion language models generate text differently from autoregressive models? What are the inference implications?

**Answer:** Autoregressive models generate text by predicting one token at a time, conditioning on all previous tokens — inherently sequential and memory-bandwidth-bound. Diffusion language models operate on the entire sequence simultaneously: start with a fully masked sequence of target length, then iteratively denoise (unmask) tokens over  $T$  steps using a model that predicts which tokens to unmask. This has key inference implications: (1) **Parallelism**: Each denoising step touches all positions simultaneously — fully utilizing tensor cores unlike autoregressive decode; (2) **Controllability**: Diffusion models can naturally fill in masked portions (infilling/editing) without re-generating from scratch; (3) **Cost structure**:  $T \times N$  token predictions per generation vs.  $N$  for autoregressive — potentially more expensive for the same output length; (4) **Length flexibility**: Requires knowing the output length upfront (or treating it as a hyperparameter). Current models like LLaDA use  $T = 64$  to 256 steps, though distillation can reduce this to 4–8 steps.

#### Q14.2. What is Mixture of Depths? How does it enable adaptive inference compute?

**Answer:** Mixture of Depths (Raposo et al., 2024) is a framework where not all tokens pass through all transformer layers. A router at each layer decides whether a token should be processed by that layer’s attention and FFN, or simply passed through via a residual connection (skipping the layer). Easy/predictable tokens skip layers, while complex/uncertain tokens get full processing. At inference time, this provides adaptive compute: a simple factual query may use only 30% of the model’s layers effectively, while a complex reasoning query uses all layers. This is distinct from early exit (which stops generation at a fixed layer): MoD continues to process skipped tokens through later layers. Training requires careful routing — typically using a top-K selection per layer on a per-token basis. MoD achieves similar perplexity to dense models at 50% of the FLOP cost for many token distributions.

#### Q14.3. Why might photonic computing change LLM inference fundamentally? What are the current limitations?

**Answer:** Photonic matrix multiplication uses interference of light waves to compute dot products at the speed of light, with near-zero energy cost per multiplication (photons don’t heat resistors). The theoretical benefits for LLM inference: (1) **Bandwidth wall disappears**: Optical interconnects can carry terabits per second over a single fiber, massively exceeding HBM limitations; (2) **Energy efficiency**: 100–1000× lower energy per FLOP compared to GPU CMOS; (3) **Latency**: Light-speed propagation eliminates electronic switching delays. **Current limitations**: (1) **Digital-to-optical**

**conversion:** Loading weights and activations into optical form still requires electronic ADC/DAC, which are slow and power-hungry; (2) **Precision:** Current photonic systems achieve 4–6 bit equivalent precision, sufficient for inference but requiring quantization; (3) **Programmability:** Reprogramming optical matrix elements is slower than loading new weights on a GPU; (4) **Scalability:** Current photonic chips implement small ( $64 \times 64$  to  $512 \times 512$ ) matrix multiplications, far smaller than LLM weight matrices. Lightmatter’s Passage chip is the most advanced commercial photonic inference accelerator as of 2025.

**Q14.4. How does multi-token prediction improve both training and inference efficiency?**

**Answer:** Multi-token prediction trains the model with  $k$  prediction heads, each predicting one step further into the future: head 1 predicts  $t + 1$ , head 2 predicts  $t + 2$ , etc. All heads share the same body representation. **Training benefits:** (1) More gradient signal per forward pass — the model optimizes  $k$  prediction objectives simultaneously; (2) Better long-range representations — to predict  $t + 4$ , the model must capture higher-level patterns beyond local co-occurrence; (3) Meta’s experiments show 3–4 token prediction improves benchmark performance and coding ability, especially with smaller models. **Inference benefits:** The additional prediction heads enable self-speculative decoding at no extra model size cost: use heads 2– $k$  to draft tokens  $t + 2$  to  $t + k$ , then verify with the next main forward pass. This provides Medusa-style speedup without a separate draft model. Acceptance rates are moderate ( $\sim 0.7$ – $0.8$  per head), yielding  $1.8$ – $2.5 \times$  decode speedup on coding tasks.

**Q14.5. Describe an “inference-first” model design philosophy. What architectural choices reflect this approach?**

**Answer:** The inference-first philosophy starts with the question: “What constraints does production deployment impose?” and works backward to architecture decisions, rather than maximizing benchmark scores at training time and then trying to optimize post-hoc. Key architectural choices that reflect inference-first thinking: (1) **GQA/MQA/MLA:** Fewer KV heads dramatically reduce KV cache memory, directly enabling larger batch sizes and longer contexts; (2) **SwiGLU FFN:** The gated architecture allows pruning entire experts in MoE variants based on routing scores, enabling conditional computation; (3) **Activation sparsity (ReLU variants):** Sparse activation patterns (Mistral’s claim of 89% FFN sparsity with ReLU) enable skipping large parts of the FFN computation; (4) **Reduced vocabulary:** Smaller vocabularies mean smaller embedding tables and faster sampling; (5) **Byte-level or character-level tokenization:** Eliminates tokenization bottleneck, improves multilingual handling; (6) **Depth over width:** Narrower, deeper models may be harder to train but parallelize more efficiently over GPUs. Phi-3 and Gemma represent inference-first philosophies: deliberately small models optimized for quality per parameter rather than absolute benchmark maximization.

# Appendix A: Key Formulas Reference

Concept	Formula
Attention	$\text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}$
KV Cache Size	$2 \times L \times n \times H \times d_h \times \text{bytes}$
Roofline	$\text{Perf} = \min(\text{Peak FLOP/s}, I \times B_{\text{mem}})$
Speculative Speedup	$\mathbb{E}[\text{accepted}] = \frac{1-\alpha^{K+1}}{1-\alpha}$
Pipeline Bubble	$\frac{p-1}{m+p-1}$
Quantization Scale	$s = \frac{x_{\max} - x_{\min}}{2^b - 1}$
MFU	$\frac{\text{Achieved FLOP/s}}{\text{Peak FLOP/s}}$
Cost per Token	$\frac{\text{GPU cost/hr}}{\text{tokens/hr}}$

# Appendix B: Annotated Paper Reading List

1. **Attention Is All You Need** (Vaswani et al., 2017) — The foundational transformer paper. Read this first.
2. **FlashAttention** (Dao et al., 2022) and **FlashAttention-2** (Dao, 2023) — Essential for understanding IO-aware attention computation.
3. **Efficient Memory Management for LLM Serving (vLLM/PagedAttention)** (Kwon et al., 2023) — The paper that redefined LLM serving.
4. **GPTQ** (Frantar et al., 2022) and **AWQ** (Lin et al., 2023) — Foundational quantization methods.
5. **Fast Inference from Transformers via Speculative Decoding** (Leviathan et al., 2023) — Mathematically rigorous treatment of speculative decoding.
6. **Medusa** (Cai et al., 2024) — Multi-head speculative decoding without a draft model.
7. **EAGLE-2** (Li et al., 2024) — State-of-the-art speculative decoding with dynamic trees.
8. **DeepSeek-V2** (DeepSeek, 2024) — MLA and MoE innovations for efficient inference.
9. **Mamba** (Gu & Dao, 2023) — State space models as efficient alternatives to attention.
10. **Sarathi-Serve** (Agrawal et al., 2024) — Chunked prefill for latency fairness.
11. **Scaling LLM Test-Time Compute Optimally** (Snell et al., 2024) — Formal treatment of inference-time compute scaling.
12. **DeepSeek-R1** (DeepSeek, 2025) — RL-trained reasoning with GRPO.
13. **Multi-Token Prediction** (Gloeckle et al., 2024) — Training and inference benefits of predicting multiple tokens.
14. **Mixture of Depths** (Raposo et al., 2024) — Adaptive per-layer computation.
15. **StreamingLLM** (Xiao et al., 2023) — Attention sinks and infinite-length inference.

## Appendix C: Glossary

Term	Definition
AWQ	Activation-aware Weight Quantization. PTQ method that scales channels by activation magnitude before quantization.
BF16	Brain Float 16. 16-bit format with same exponent range as FP32, used widely in LLM training and inference.
FLOP	Floating Point Operation.
GQA	Grouped Query Attention. Attention variant where groups of query heads share one KV pair.
GPTQ	Layer-wise second-order PTQ method using Hessian information for weight quantization.
HBM	High Bandwidth Memory. The type of DRAM used in modern GPUs (HBM2e, HBM3).
KV Cache	Key-Value cache. Stores attention keys and values for previously generated tokens.
MFU	Model FLOP Utilization. Fraction of peak FLOP/s used productively.
MLA	Multi-head Latent Attention. Attention variant with compressed latent KV representations.
MoE	Mixture of Experts. Model architecture with many expert FFNs and token routing.
MQA	Multi-Query Attention. All query heads share a single KV pair.
NVLink	NVIDIA's high-bandwidth GPU-to-GPU interconnect (900 GB/s on H100).
PagedAttention	KV cache memory management using virtual memory paging.
PRM	Process Reward Model. Reward model that scores intermediate reasoning steps.
PTQ	Post-Training Quantization. Quantization applied after training.
QAT	Quantization-Aware Training. Quantization incorporated during training.
RoPE	Rotary Positional Embeddings. Position encoding via complex rotation.
SSM	State Space Model. Sequential model using a learned recurrent state (Mamba, S4).
TPOT	Time Per Output Token. Duration between consecutive output tokens.
TTFT	Time To First Token. Duration from request to first output token.
TensorCore	Specialized GPU compute units for accelerated matrix multiplication.

vLLM	Virtual LLM. High-throughput serving system using PagedAttention and continuous batching.
------	---

---

# Appendix D: Hardware Comparison

GPU	VRAM	BW (TB/s)	FP16 TFLOP/s	INT8 TOPS	TDP (W)
H100 SXM5	80 GB	3.35	1,979	3,958	700
H100 PCIe	80 GB	2.0	1,513	3,026	350
A100 SXM	80 GB	2.0	312	624	400
RTX 4090	24 GB	1.0	165	330	450
RTX 3090	24 GB	0.94	71	142	350
Apple M3 Ultra	192 GB	0.8	28	56	60
Groq LPU: ~900 GB/s on-chip SRAM BW, ~1 PB/s aggregate (64 chips)					

End of Book

“The best inference is one that the user never has to think about.”