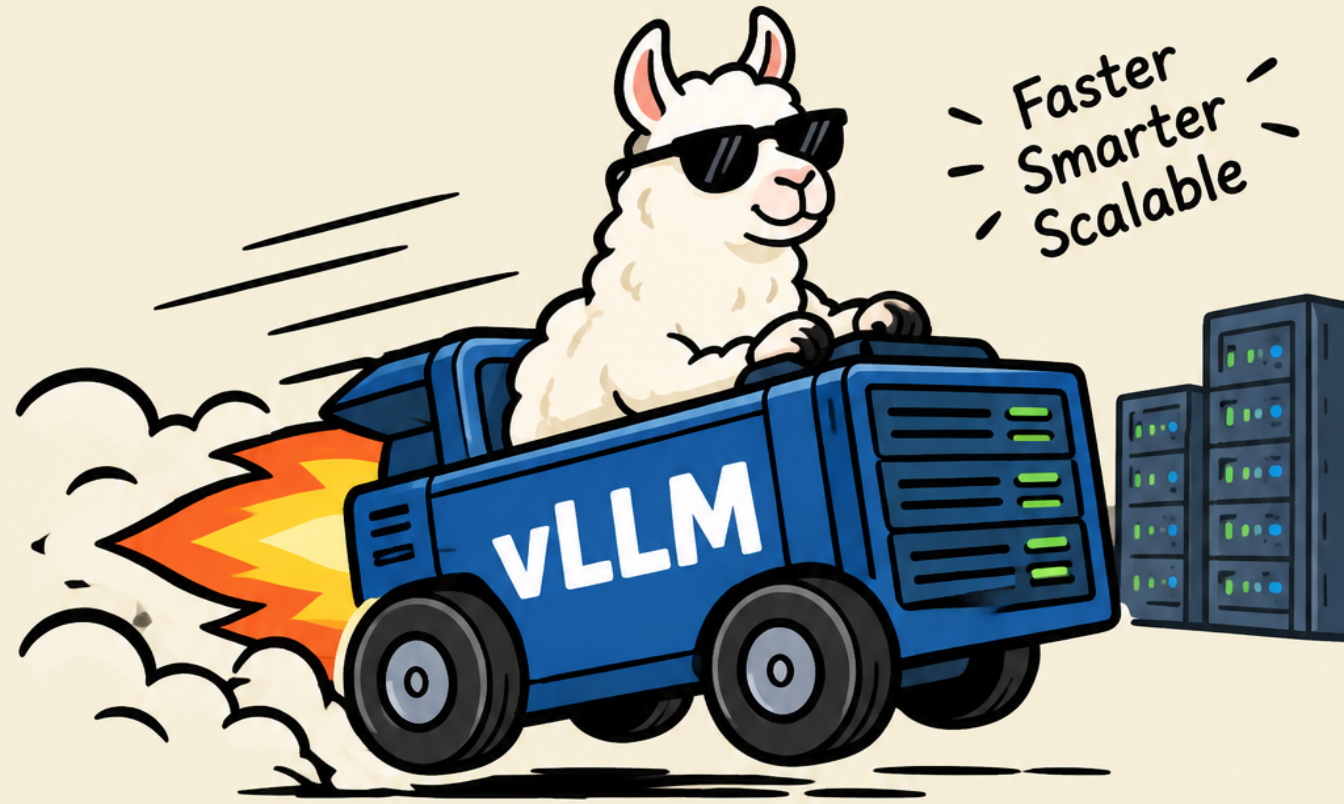
vLLM and the ENGINEERING OF FAST LLM INFERENCE

A Systems Guide to High-Performance Large Language Model Serving



ALEX HUANG

vLLM and the Engineering of Fast LLM Inference

A Systems Guide to High-Performance Large Language Model Serving

Steve Publications

This book is available at
https://leanpub.com/vllmandtheengineeringoffastllminference

This version was published on 2026-09-07

Contents

A Systems Guide to High-Performance Large Language Model Serving

This book teaches experienced engineers how to build, measure, and optimize production LLM inference systems. Using vLLM as its primary case study, it walks you from the first principles of transformer inference and GPU architecture through vLLM's internal design, advanced performance techniques, distributed deployment, and operational reliability. You will learn not just how to configure vLLM, but how to reason about memory bottlenecks, scheduling trade-offs, parallelism strategies, and cost-performance optimization so you can architect inference infrastructure that scales. The material is grounded in primary sources, source code, and measured benchmarks; version-dependent details are flagged explicitly so the underlying engineering principles endure beyond any single release.

Introduction: The LLM Inference Problem

A user opens a chat window and types a question. Their prompt arrives at a web server, which forwards it through a load balancer to an inference service. That service tokenizes the text, looks up model weights in GPU memory, runs a transformer forward pass, and begins generating tokens one by one. Each new token depends on every previous token, so the service caches intermediate attention keys and values in GPU memory and reuses them on every step. After a fraction of a second, the first word appears. Over the next few seconds, more words stream in until the model finishes.

To the user, this is a single request. To the systems engineer, it is a tightly coupled sequence of compute kernels, memory transfers, scheduling decisions, and network messages, all executing under constraints that are fundamentally different from traditional backend workloads.

This is the LLM inference problem. It is not a training problem, and it is not a standard batch-processing problem. It is its own class of systems challenge: sequential dependencies that conflict with throughput requirements, gigabytes of working memory that grow dynamically per request, and hardware that operates best under uniform, parallel loads while being asked to process wildly variable sequences.

The economic stakes are now material. Companies serving millions of requests daily spend significant amounts on GPU infrastructure for inference. In 2026, semi-public benchmarks show inference costs on current-generation hardware running in the range of a few cents to ten cents per million generated tokens for large models, depending on the architecture and precision [3]. For high-volume services, a 2x improvement in throughput per GPU is not an academic exercise. It is a direct reduction in infrastructure spend. Conversely, a failure to manage latency, memory, or reliability during a traffic spike is not just a bad metric. It is churn.

What Makes LLM Inference Hard

Naive transformer inference is straightforward. You load the model weights, tokenize the prompt, run forward passes, and stream the outputs. Hugging Face Transformers makes this a few lines of Python code. The problem is that naive inference is economically and operationally unusable at scale.

Consider what happens under even modest concurrency. Suppose you are serving a 70 billion parameter model on A100 GPUs and receive 50 concurrent requests with long prompts. Each request requires its own copy of the KV-cache, and the cache grows with every generated token. A single request with a 10,000-token prompt and 500-token output can easily consume several gigabytes of GPU memory just for its cached keys and values. Multiply that by dozens of concurrent requests, and you quickly exhaust GPU memory. Worse, because each request finishes at a different time, the memory usage pattern is jagged. Requests allocate memory at different rates, hold it for different durations, and free it unpredictably. The result is fragmentation: you have plenty of total free memory but no large enough contiguous region to serve a new request.

This is the allocation problem. Traditional systems solve it by pre-allocating the maximum possible cache per request, which wastes memory when requests use less than their allocated space. Alternatively, they use fixed batch sizes: wait until N requests arrive, process them together, then clear and repeat. Fixed batching avoids fragmentation but introduces idle time as the system waits for enough requests to fill the batch, and it cannot adapt when requests finish at different rates. Both approaches leave GPU compute capacity unused.

The compute profile is equally unforgiving. LLM inference has two distinct phases with very different characteristics. The prefill phase processes the entire input prompt in parallel across all tokens. It is compute-intensive with high arithmetic intensity, meaning the GPU does many operations per byte loaded from memory. The decode phase generates tokens one at a time. Each decode step is memory-bound because it must read all cached keys and values for every previous token, and the cache grows with each step. The decode phase consumes the vast majority of runtime for long outputs, and during decode the GPU compute units spend more time waiting for memory than doing arithmetic.

This fundamental mismatch between prefill and decode behavior means a single serving strategy cannot be optimal for both. A system tuned for fast prefill may starve decode requests, and a system tuned for high-decode throughput may delay the first token of long prompts until the entire prefill completes. Interactive users care about time to first token. Batch users care about total throughput. Both groups care about tail latency: the slowest 1 percent of requests that ruin the user experience.

Then there is the distribution problem. Real-world request patterns are not uniform. Prompts vary from a few words to tens of thousands of tokens. Outputs range from a single sentence to long essays. Requests arrive in bursts and lulls. Different endpoints serve different model sizes. A production system must handle all of this without manual reconfiguration for each workload.

These challenges are why specialized inference serving systems exist. General-purpose deep learning frameworks can run a single inference request fine, but they were not designed for continuous, concurrent serving under variable load. You need a system that manages GPU memory as carefully as an operating system manages RAM, that schedules requests with the sophistication of a production web server, and that exploits GPU hardware features at the kernel level.

A Day in the Life of an Inference Request

Tracing a single request through a production LLM inference system makes the complexity concrete.

The request starts as an HTTP POST to an OpenAI-compatible API endpoint. It carries a system prompt, conversation history, and user message, plus parameters like temperature and max tokens. The API server validates the request, applies rate limits if configured, and enqueues it for processing. At this point it is just a message in a queue.

The engine scheduler wakes up. It maintains queues of waiting requests and tracks which requests are currently running. It examines available GPU memory and the token budget for this step. The budget is controlled by a parameter called max_num_batched_tokens, which caps how many tokens the engine processes per forward pass. The scheduler decides which requests to include in the next batch. Decode requests that are already generating get priority. If budget remains, the scheduler admits new prefill requests,

or continues partial prefills from previous steps. This decision happens in milliseconds but determines whether the GPU stays busy or stalls.

For each admitted request, the scheduler ensures there is space in the KV-cache. In a system like vLLM, the cache is divided into fixed-size blocks, typically holding 16 tokens each. The scheduler allocates blocks from a free pool and updates each request's block table, which maps logical token positions to physical block locations. If no blocks are available, requests wait or existing requests are preempted, depending on the policy.

The scheduler sends its decision to the model executor. The executor prepares inputs: it assembles the batch tensor, flattens variable-length sequences into a single super-sequence, and constructs block table descriptors. On modern systems, these preparation steps are cached and reused to avoid Python overhead.

The model executor runs the forward pass. All the linear layers, activations, and attention operations execute as CUDA kernels. The attention kernel is where the KV-cache matters most: it follows the block tables to load cached keys and values from non-contiguous memory locations. Modern systems use specialized attention kernels like FlashAttention that minimize memory traffic by tiling computation and keeping intermediate results in fast on-chip SRAM.

After the forward pass produces logits for the next tokens, the sampler applies temperature scaling, top-p filtering, and sampling to select output tokens. For each running request, the new token is appended to its output and a new KV-cache block entry is created. The sampler checks whether each request has reached its maximum length or generated an EOS token. Finished requests are marked for completion.

The engine loops back to the scheduler, which updates request states, frees KV-cache blocks from completed requests, and begins the next scheduling cycle. Throughout this loop, the system may be handling dozens or hundreds of concurrent requests, each at a different stage of generation.

Once a token is generated, the detokenizer converts it to text. For streaming responses, this text is sent immediately over the HTTP connection. The time from when a request arrives until the first token is sent is called time-to-first-token, or TTFT. The time between successive tokens is inter-token latency, or ITL. TTFT is dominated by prefill work; ITL is dominated by decode work. These are the two latency metrics that users actually feel.

This entire pipeline runs under tight constraints. If the scheduler is slow,

the GPU idles. If input preparation on the CPU stalls, the GPU waits. If the attention kernel is inefficient, memory bandwidth becomes the bottleneck and throughput collapses. If KV-cache allocation fails under load, requests are dropped or delayed. Every layer must be tuned and coordinated.

Why vLLM Is Our Lens

Many inference serving frameworks exist, each with different design philosophies. Hugging Face Text Generation Inference was once widely used but is now in maintenance mode. TensorRT-LLM compiles models into highly optimized engines for NVIDIA GPUs, achieving excellent raw throughput at the cost of flexibility and build complexity. SGLang emphasizes structured generation and uses RadixAttention for aggressive prefix caching. llama.cpp focuses on CPU and edge deployment with extreme quantization support.

vLLM occupies a distinctive position: it is production-grade, broadly compatible, and its core innovations have become influential across the ecosystem. vLLM was originally developed at UC Berkeley and has grown into one of the most active open-source AI projects, with contributions from hundreds of institutions and companies and over two thousand contributors [3]. It introduced PagedAttention, a memory management technique inspired by virtual memory paging in operating systems that dramatically reduces KV-cache fragmentation and enables efficient prefix reuse. vLLM combines PagedAttention with continuous batching, sophisticated scheduling, and deep hardware optimizations including FlashAttention integration and CUDA Graph capture.

Using vLLM as the primary case study gives us three advantages. First, vLLM is widely deployed in production by major AI companies and startups, so the design decisions reflect real-world constraints, not just research curiosity. Second, vLLM is open source, so we can examine the actual implementation rather than relying solely on documentation or papers. Third, vLLM's architectural choices are principled and traceable: PagedAttention, continuous batching, prefix caching, and speculative decoding all exist to solve specific, identifiable problems. Understanding vLLM teaches you not just how to run a server, but how to think about inference system design.

This book is not a vLLM tutorial in the sense of a quick start guide. It is a deep dive that uses vLLM to explain the entire inference stack. Even if your production system uses TensorRT-LLM or SGLang or a custom solution,

the underlying bottlenecks are the same: memory bandwidth, scheduling efficiency, cache reuse, and parallelism. The principles discussed here transfer.

I should note a version caveat that affects everything that follows. vLLM has undergone a major architectural transition between V0 and V1. V0 was the original engine, with scheduler and worker colocated and prefill/decode processed in separate steps. V1, first released in alpha in January 2025 and promoted to default in version 0.8.0, separates the scheduler into a dedicated Engine Core process, unifies prefill and decode scheduling, and introduces persistent batching and improved prefix caching [4]. Many of the concepts described in this book apply to both versions, but I will call out V1-specific behavior where it differs from V0. The field also moves quickly: features like speculative decoding, expert parallelism, and quantization support evolve with each release. I will flag version-dependent details explicitly.

How to Use This Book

The book is organized as a progression from fundamentals to production mastery.

The first part establishes foundations. Chapter 1 explains transformer inference from first principles, covering the forward pass, self-attention, and why prefill and decode are fundamentally different problems. Chapter 2 covers GPU architecture: memory hierarchies, compute units, and the arithmetic intensity calculations that determine whether your bottleneck is compute or memory. Chapter 3 defines the metrics that matter for inference and how to measure them properly.

The second part dives into vLLM internals. Chapter 4 explains the KV-cache and PagedAttention, the core innovation that enables vLLM's memory efficiency. Chapter 5 covers batching and scheduling. Chapters 6 through 8 examine the vLLM architecture, its PagedAttention implementation, and its continuous batching logic. Chapter 9 is a practical guide to installing and deploying vLLM.

The third part covers optimization techniques. Chapter 10 covers quantization. Chapters 11 and 12 explain tensor, pipeline, and expert parallelism for multi-GPU and MoE models. Chapter 13 covers distributed inference across nodes. Chapter 14 examines caching strategies beyond PagedAttention.

Chapter 15 covers speculative decoding. Chapter 16 explains low-level GPU optimizations including CUDA Graphs.

The fourth part is about production engineering. Chapter 17 covers reliability, observability, and deployment infrastructure. Chapter 18 teaches systematic performance tuning. Chapter 19 covers troubleshooting and common failure modes. Chapter 20 compares vLLM with alternative inference frameworks. Chapter 21 addresses economics, capacity planning, and architectural recommendations.

You can read the book linearly, or jump to the parts most relevant to your work. The code examples are designed to be runnable: configuration files, Docker commands, and Python scripts are complete enough to execute with minimal modification. When I cite performance numbers, they come from specific experiments with stated configurations. When I speculate about future directions, I say so explicitly.

If you finish this book without learning how to tune every vLLM flag, but you understand why inference is hard, how to diagnose a bottleneck when performance degrades, and how to evaluate trade-offs between different architectures and tools, then the book has done its job.

Chapter 1: Transformer Inference from First Principles

You cannot optimize transformer inference if you do not understand the actual computation being performed. This chapter establishes the mathematical and computational foundations. We will trace a transformer forward pass, explain how self-attention works, quantify why it dominates inference cost, and distinguish the two phases of inference that require fundamentally different optimization strategies.

The Transformer Forward Pass

Modern large language models are decoder-only transformers. A decoder-only model takes a sequence of tokens as input and produces a probability distribution over the next token. It repeats this process autoregressively: each newly generated token is appended to the input sequence and used to predict the following token.

The input begins as text, which is tokenized into a sequence of token IDs. Tokenization converts strings into integers using a vocabulary learned during training, typically containing 32,000 to 150,000 tokens. Common words and subword patterns get lower IDs. A 500-character prompt might tokenize to 100-200 tokens depending on the model and text content.

These token IDs are embedded: each ID maps to a dense vector in R^d, where d is the model dimension. For a model like Llama 3.1 8B, d = 4096. The embedding layer is a lookup table of size vocab_size × d. For a vocabulary of 128,000 tokens and d = 4096 in FP16 precision, the embedding table alone occupies approximately 1.0 gigabytes.

The embedded tokens then pass through a stack of transformer layers, typically 32 to 128 layers depending on the model. Each layer contains two main components: a self-attention block and a feed-forward network, with residual connections and layer normalization around each.

The self-attention block computes interactions between all positions in the sequence. For each position, it produces three vectors: a query vector, a key vector, and a value vector, each of dimension d or a smaller head dimension. The query from position i is compared with all keys to compute attention scores, which are softmax-normalized and used as weights to combine all value vectors. The result is a new representation for position i that incorporates information from the entire sequence.

The feed-forward network is a simple two-layer MLP applied independently to each position: a linear projection up to a hidden dimension (typically 3 to 4 times d), an activation function like SwiGLU, and a projection back down. Despite its simplicity, the FFN dominates parameter count in modern transformers because of the large hidden dimension.

After the final layer, another linear projection maps the hidden state of the last token to logits over the vocabulary. These logits are converted to probabilities via softmax, and a sampling strategy selects the next token. For temperature zero, the model picks the highest-probability token greedily. For temperature greater than zero, sampling introduces randomness.

This description applies to both training and inference. The critical difference is how many tokens are processed at once and what information is available from previous steps. Training processes batches of complete sequences in parallel. Inference processes one or more sequences that grow token by token, with each new step depending on all previous results.

Self-Attention Mechanics and Complexity

Self-attention is where the computational cost of transformer inference concentrates. Understanding its mechanics is essential for understanding why inference optimization is so challenging.

Consider a sequence of n tokens, each represented as a vector in R^d. At each transformer layer, three linear projections produce query, key, and value matrices Q, K, V, each of shape n × d. The attention computation is:

Attention(Q, K, V) = softmax(QK^T / sqrt(d))V

Let's break this down operation by operation.

First, QK^T computes the dot product between every query and every key. The result is an n × n matrix of attention scores × d multiplications and additions.

Second, each row of scores is divided by sqrt(d) for numerical stability and softmax-normalized so the weights sum to one. This is O(n^2) work.

Third, the softmax weights are applied to V. Each output position i is a weighted combination of all value vectors. This is another O(n^2 × d) operation.

The attention scores matrix itself requires O(n^2) memory. For a sequence of 4,096 tokens, that is 16 million entries. In FP16, that is 32 megabytes per layer. For a 32-layer model, storing attention scores for all layers simultaneously would require over 1 gigabyte.

In training, attention scores must be stored for backpropagation, so this memory is necessary. In inference, the scores are not needed after the layer completes, so they can be recomputed. Modern kernels like FlashAttention exploit this: they compute attention in tiles, keeping only summaries in fast memory and never materializing the full n × n matrix. This reduces memory from O(n^2) to O(n) and improves performance dramatically, especially for long sequences.

The parameter count for the projections is also substantial. Each projection (Q, K, V, plus the output projection) has d × d weights. For four projections and d = 4096 in FP16, that is approximately 134 megabytes per layer for attention weights alone. Over 32 layers, that is 4.3 gigabytes. The FFN projections are larger: with a hidden dimension of roughly 3d, each layer's FFN weights dominate attention weights by a factor of about 3 to 4.

In a modern 70 billion parameter model like Llama 3.1 70B, the total weight size in FP16 is approximately 140 gigabytes. Such a model cannot fit on a single GPU and requires tensor parallelism across multiple devices.

Prefill Versus Decode: Two Different Problems

The split between prefill and decode phases is the most important distinction in LLM inference engineering. They are not just different stages of the same process. They are different computational problems with different bottlenecks.

Prefill is the processing of the input prompt. All tokens are known in advance. The model runs a standard transformer forward pass over the entire sequence, computing attention between all pairs of positions and producing hidden states for each position. During prefill, the KV-cache is built: the

key and value vectors for each token are stored so they can be reused during decode.

The prefill phase is compute-bound. With n prompt tokens, each layer performs O(n^2 × d) work in attention and O(n × d × 3d) in the FFN. The total computation scales with both the sequence length and the model size. Memory traffic is also significant, but the arithmetic intensity is high: many operations are performed per byte loaded from memory. On modern GPUs with powerful tensor cores, the prefill phase can saturate compute capacity.

Decode is the generation of new tokens one at a time. Each decode step processes a single new token position, but it must attend to all previous positions using their cached keys and values. For a sequence of length s (including prompt tokens and all generated tokens so far), the attention computation at the current step reads keys and values for all s positions. The compute is proportional to s × d, much smaller than prefill's s^2 × d.

The critical observation is that the decode step reads the same cached keys and values on every iteration, and the cache grows with each step. Over a generation of g output tokens, the total memory read for attention alone is proportional to d × s_avg × g × num_layers, where s_avg is the average sequence length during generation. This is a massive amount of memory traffic.

Because decode performs relatively little computation but reads huge amounts of data, its arithmetic intensity is very low. Estimates in the literature place decode-phase arithmetic intensity at 10 to 20 FLOPs per byte read for the attention mechanism alone, compared to 100 to 400 FLOPs per byte during prefill. The GPU's memory bandwidth becomes the bottleneck long before its compute capacity is reached.

This difference explains many practical observations. During prefill, GPU utilization can exceed 80 percent. During decode, it often drops to 20 to 40 percent because the memory subsystem is saturated while compute units sit idle. The only way to improve decode throughput is to reduce memory traffic or increase bandwidth. Quantization helps by reducing the data size. Tensor parallelism helps by distributing the load. But no software optimization can change the fundamental arithmetic intensity of the problem.

Why Attention Dominates Decode Performance

Given the decode phase's memory-bound nature, attention is the dominant cost. This may not be immediately obvious because the FFN has more parameters and performs more FLOPs per token during prefill. But during decode, attention's pattern of access makes it disproportionately expensive.

Consider what happens during a single decode step for a sequence of length s.

The attention mechanism must load keys and values for all s previous positions from the KV-cache in GPU DRAM. For a model with h attention heads and head dimension d_h, the keys alone require s × h × d_h parameters per layer. For Llama 3.1 8B with 32 heads, d_h = 128, the keys for a 2,048-token sequence occupy approximately 8.4 megabytes per layer in FP16. Values are the same size. Over 32 layers, the total KV-cache for one request at that length is roughly 540 megabytes.

The FFN also requires loading weights for each layer. The FFN weight size is approximately 6 × d × 3d per layer in typical implementations. For d = 4096, that is about 384 megabytes per layer, or over 12 gigabytes total for 32 layers. These weights are the same for every request, so they can be cached in GPU memory permanently.

Here is the key distinction: FFN weights are loaded once and reused for all decode steps. KV-cache entries are loaded on every single decode step because each step depends on all previous positions. Over a generation of 500 tokens, the KV-cache for one request is read approximately 500 times, totaling over 270 gigabytes of memory traffic for attention alone on a single request.

This repeated access pattern is why KV-cache management is so critical. Reducing cache size through quantization or more efficient architectures directly reduces the bandwidth requirement. Systems like vLLM optimize not just the attention kernel, but the memory layout and access patterns of the cache itself.

Modern models increasingly use variants of multi-head attention to reduce KV-cache size. Multi-Query Attention, or MQA, uses a single key and value head shared by all query heads, reducing cache size by a factor equal to the number of heads. Grouped-Query Attention, or GQA, is a compromise: query heads are grouped and each group shares a single key-value head. Llama 3.1

uses GQA with 8 groups in the 8B variant and 8 groups in the 70B variant. For the 70B model with 64 query heads, this reduces the KV-cache by 8x compared to standard multi-head attention.

The Role of Positional Embeddings

The self-attention mechanism as described has no notion of order: it treats tokens as a set rather than a sequence. Positional embeddings address this by adding position-dependent information to each token's representation.

Early transformers used fixed sinusoidal positional encodings, adding deterministic functions of position to the input embeddings. This approach works but has limitations: the model must learn to extract positional information from the encodings, and extrapolation beyond training sequence length is unreliable.

Modern models use learned positional embeddings or more sophisticated schemes. RoPE, or Rotary Positional Embeddings, has become the dominant approach in current models. RoPE encodes position information through a rotation applied to query and key vectors. The rotation angle depends on the position, so the dot product between vectors captures relative position implicitly.

RoPE has two advantages relevant to inference optimization. First, it generalizes better to sequence lengths beyond training, allowing models to process longer prompts than they were trained on. This is valuable for applications requiring long context. Second, and more subtly, RoPE's mathematical structure can sometimes be exploited in attention kernel optimization because it is a deterministic transformation of the key and query vectors rather than an additional tensor that must be loaded and added.

Another relevant development is the use of sliding window attention in some models. Instead of attending to all previous positions, each position attends only to a fixed window of nearby tokens. This reduces both computation and KV-cache size for long sequences. Models like Mistral 7B and Phi use sliding windows of 4,096 tokens for most attention layers while using full attention for some layers.

Sliding window attention creates interesting optimization opportunities for serving systems. Because older tokens outside the window are no longer needed, their KV-cache blocks can be freed during generation. This reduces

memory pressure for long-running requests. Serving systems like vLLM can be configured to handle sliding window models, freeing cache as positions fall out of the window.

FlashAttention and Modern Attention Kernels

Standard attention implementation is straightforward but inefficient on GPUs. It materializes the full n × n attention score matrix in global memory, then performs softmax and value projection. This approach has two problems: O(n^2) memory for scores, and excessive reads and writes to global memory.

FlashAttention, introduced by Dao et al. in 2022 [1], solves both problems through IO-aware tiling. The core idea is to process attention in small blocks, keeping intermediate results in fast on-chip SRAM and never writing the full attention matrix to global memory.

Here is the high-level algorithm. The sequence is divided into tiles. For each query tile, we iterate over key-value tiles: compute the partial attention scores for that tile, maintain running statistics (max and sum for softmax normalization), and accumulate the partial output. At the end, we use the accumulated statistics to normalize and produce the final output. The entire process reads the input sequences and KV-cache only twice but writes intermediate results to SRAM, not global memory.

The performance gains are substantial. FlashAttention achieves 2x to 4x speedup over optimized baseline attention implementations in both training and inference, with linear instead of quadratic memory scaling for the attention scores.

FlashAttention-2, released in 2023 [2], improves parallelism and work partitioning. It parallelizes over the sequence length dimension as well as the batch and head dimensions, improving GPU occupancy. It also restructures the computation to reduce register pressure and improve memory access patterns.

FlashAttention-3, released in 2024, targets the Hopper architecture specifically, using asynchronous instructions and warp-specialized pipelines. It achieves even better utilization on H100 and later GPUs, though the gains are architecture-specific.

vLLM integrates FlashAttention as its default attention backend when available. The integration is not trivial: vLLM's PagedAttention requires attention

to read from non-contiguous memory blocks, which conflicts with FlashAttention's assumptions about memory layout. vLLM handles this through its own custom attention kernels that incorporate FlashAttention's IO-aware principles while supporting paged memory access.

There is an important practical distinction to understand. FlashAttention is primarily a training optimization: during training, recomputing attention scores saves memory that would otherwise be held for backpropagation. During inference, the recomputation benefit disappears because scores are not needed after the forward pass. However, FlashAttention's reduced memory traffic and better memory hierarchy usage still improve inference performance, especially during prefill with long sequences. During decode, where memory bandwidth is already the bottleneck, the attention kernel choice matters less than the KV-cache size and access patterns.

Modern serving systems support multiple attention backends. In addition to FlashAttention, systems may support FlashInfer, a kernel optimized for inference-specific patterns like variable sequence lengths and sparse attention. vLLM's attention backend selection depends on the model architecture, hardware, and configuration options. The choice of backend affects both performance and compatibility: not all backends support all attention variants like sliding windows or grouped-query attention.

This chapter has established the computational profile of transformer inference. The prefill phase is compute-intensive and benefits from high arithmetic intensity and optimized kernels. The decode phase is memory-intensive and bottlenecked by bandwidth. Attention is the dominant cost during decode because it requires repeated access to a growing cache. These characteristics drive every optimization discussed in the rest of this book: memory management, batching, caching, quantization, and parallelism. The goal of inference engineering is to work within these constraints to maximize throughput and minimize latency.

Chapter 2: GPU Architecture for LLM Inference

LLM inference is a GPU-bound workload. Understanding how GPUs work is not optional decoration. It is the foundation for reasoning about performance, capacity, and cost. This chapter explains the GPU hardware constraints that drive inference optimization decisions.

GPU Compute Units: SMs, CUDA Cores, and Tensor Cores

A modern NVIDIA GPU is a massively parallel processor divided into streaming multiprocessors, or SMs. Each SM is a complete compute unit containing CUDA cores for floating-point and integer arithmetic, special function units for transcendental operations, and load/store units for memory access. SMs operate independently and concurrently: the GPU dispatches work across SMs and relies on their parallelism to achieve throughput.

The number of SMs varies by GPU model. The A100 SXM has 108 SMs. The H100 SXM has 132 SMs. The consumer-grade RTX 4090 has 128 SMs. More SMs mean more parallel execution units, which matters for throughput.

Within each SM, CUDA cores execute individual instructions. A100 SMs contain 64 CUDA cores each, for a total of 6,912 on the chip. H100 SMs have 128 CUDA cores each, for 16,896 total. CUDA cores handle general-purpose computation and some parts of neural network operations.

Tensor cores are the specialized units that matter most for LLM inference. Introduced in the Volta architecture and refined through Turing, Ampere, Ada Lovelace, and Hopper, tensor cores accelerate matrix multiply-accumulate operations in low precision. They are designed specifically for the kind of operations that dominate transformer workloads.

Each generation adds improvements. Ampere (A100) introduced sparsity support and improved INT8 performance. Hopper (H100) introduced native

FP8 support and fourth-generation tensor cores with higher throughput. The Blackwell architecture (B200) continues this trajectory with even higher FP8 and FP4 throughput.

For LLM inference, the relevant precision formats are:

- FP16 and BF16: 16-bit floating-point formats used for model weights and activations. BF16 is now preferred over FP16 for most models because it has better numerical stability and matches the training precision of many modern models. A100 tensor cores perform FP16 and BF16 matmuls at 312 teraFLOPs in TF32-equivalent mode, or 624 teraFLOPs with structured sparsity.
- FP8: 8-bit floating-point introduced in Hopper. FP8 provides higher throughput and smaller memory footprint while maintaining reasonable numerical precision for inference. H100 achieves 1,979 teraFLOPs in FP8, nearly 6x the FP16 rate.
- INT8 and INT4: Integer formats used primarily for quantized inference. INT8 is widely supported and commonly used with quantization methods like AWQ and GPTQ. INT4 is aggressive but becoming practical with careful calibration.

The distinction between different precisions matters for both performance and compatibility. Not all models train in all formats. A model trained in BF16 may lose accuracy if loaded in FP16 due to precision differences. A model can run in FP8 inference if the quantization calibration is correct and the hardware supports FP8 tensor cores.

The GPU Memory Hierarchy: DRAM, SRAM, Registers

GPU memory is organized in a hierarchy with very different characteristics at each level. Understanding this hierarchy is critical because inference performance is fundamentally limited by how quickly data can move through it.

The largest and slowest level is GPU DRAM, typically HBM (High Bandwidth Memory). HBM is stacked on the same package as the GPU die and connected through a wide memory bus. The A100 SXM has 80GB of HBM2e memory with 2,039 GB/s peak bandwidth. The H100 SXM has 80GB of HBM3 with 3,352 GB/s

peak bandwidth, a 64 percent improvement. The H100 NVL variant has 94GB with 3,888 GB/s bandwidth.

Memory bandwidth matters more than memory capacity for inference throughput. During decode, the GPU spends most of its time reading KV-cache entries and model weights from DRAM. Higher bandwidth means more tokens per second. A system bottlenecked by memory bandwidth on an A100 will improve proportionally if moved to an H100, assuming everything else is equal.

Within each SM, SRAM serves as L1 cache and shared memory. The size is small: A100 SMs have 192KB of L1/shared memory. SRAM is orders of magnitude faster than DRAM but limited in capacity. The key optimization strategy is to keep frequently accessed data in SRAM through tiling and caching.

Registers are the fastest storage, located within each SM and assigned to individual threads. There are thousands of registers per SM (A100: 64KB per SM). Registers hold per-thread local data like loop counters and intermediate computation results. Register pressure affects occupancy: if a kernel requires too many registers per thread, fewer threads can run concurrently on each SM, reducing parallelism.

This hierarchy creates an optimization challenge. DRAM is large but slow. SRAM is fast but small. Registers are fastest but extremely limited. Kernel optimization is largely about keeping data in the fastest possible level of the hierarchy for as long as possible. FlashAttention succeeds because it tiles computation so that intermediate results stay in SRAM instead of being written to DRAM and read back.

For inference systems, the practical implications are:

- Model weights live in DRAM because they do not fit in SRAM. They are loaded into SRAM for each layer computation and reused as much as possible.
- KV-cache lives in DRAM and is loaded repeatedly during decode. Reducing KV-cache size directly reduces memory traffic.
- Intermediate activations should be kept in SRAM or registers as much as possible through kernel fusion and tiling.

Memory Bandwidth as the Primary Bottleneck

As established in Chapter 1, LLM inference decode is memory-bound. This means the limiting factor is how fast data can be read from DRAM, not how fast the GPU can compute.

To understand why, we need to look at the actual numbers. Consider a decode step for a single token on a model like Llama 3.1 8B with a sequence length of 1,000 tokens.

The attention mechanism must load KV-cache entries for all 1,000 previous positions. For 32 attention layers, 32 heads, 128-dimensional heads, and FP16 precision: 1,000 × 32 × 32 × 128 × 2 bytes = 262 megabytes of keys and values.

The FFN weights for all layers must also be loaded. For approximately 8 billion parameters in FP16: 16 gigabytes total. These weights are loaded once and stay in DRAM.

Over a generation of 250 output tokens, the KV-cache is repeatedly loaded. Average sequence length during generation is approximately 1,125 tokens. Total KV-cache traffic is roughly 250 × 300 megabytes = 75 gigabytes. Model weight traffic is 16 gigabytes × 250 = 4 terabytes. Total memory traffic for this single request is approximately 4.08 terabytes.

On an A100 with 2,039 GB/s bandwidth, if memory were the only bottleneck, the theoretical minimum time would be 4,080 GB : 2,039 GB/s = 2 seconds for 250 tokens, or 125 tokens per second. On an H100 with 3,352 GB/s, the minimum would be 1.22 seconds, or 205 tokens per second.

Real-world performance is lower because not all memory accesses are perfectly efficient, compute overhead exists, and kernel launch latency adds up. But the orders of magnitude are correct. This arithmetic shows why:

- Memory bandwidth is the first constraint to consider when sizing infrastructure.
- Quantization helps directly because it reduces data size. FP8 KV-cache halves the traffic compared to FP16.
- Tensor parallelism helps because it distributes memory traffic across multiple GPUs, each with its own bandwidth.
- GPU compute upgrades that do not increase memory bandwidth provide diminishing returns for decode throughput.

The relationship between memory bandwidth and tokens per second is approximately linear for a given model and sequence length distribution. This makes bandwidth a key metric for hardware procurement decisions.

Arithmetic Intensity and Roofline Analysis

Arithmetic intensity is the ratio of operations performed to bytes transferred: FLOPs per byte. It is the fundamental metric that determines whether a workload is compute-bound or memory-bound.

A workload with high arithmetic intensity performs many operations per byte loaded. Matrix multiplication on large matrices is a classic example. A workload with low arithmetic intensity does few operations per byte. Scattered memory reads with little computation are another example.

Every GPU has a roofline defined by its peak compute and memory bandwidth. The roofline model plots achievable performance as a function of arithmetic intensity. Below the roofline knee, performance is limited by bandwidth. Above it, performance is limited by compute.

For LLM inference, the prefill phase sits above the knee: arithmetic intensity is high enough that compute is the bottleneck. The decode phase sits below the knee: arithmetic intensity is too low, so bandwidth is the bottleneck.

We can estimate arithmetic intensity for attention in decode. Each decode step computes attention between one query and all cached keys. For sequence length s and head dimension d_h:

- FLOPs: approximately $2 \times s \times d\_h^{2 \text{ for the QK matmul, plus } 2 \times s \times d\_h}2$ for the weighted value computation.
- Bytes read: KV-cache entries for s positions, roughly $2 \times s \times h \times d\_h \times 2$ bytes in FP16.

Dividing FLOPs by bytes gives roughly d_h/h bytes of computation per byte read. For h = 32 and d_h = 128, this is about 8 FLOPs per byte for attention. For the full layer including FFN, arithmetic intensity rises but remains in the low teens to twenties.

Compare this to a GPU's effective arithmetic intensity at the roofline knee. For an A100 with 312 TFLOPS FP16 and 2,039 GB/s bandwidth, the knee is at

approximately 153 FLOPs per byte. Decode-phase arithmetic intensity of 10-20 FLOPs per byte is well below this. The GPU cannot use its full compute capacity because memory is the limiting factor.

This analysis has practical implications:

1. During decode, buying more compute (more CUDA cores) does not help if bandwidth is unchanged.
2. Quantization that reduces precision from FP16 to FP8 doubles arithmetic intensity from roughly 15 to 30 FLOPs per byte. This is a significant improvement but may still be below the knee, so bandwidth savings still dominate.
3. Increasing batch size during decode increases arithmetic intensity because more tokens share the same memory reads for model weights. This is why batching improves decode throughput up to the point where KV-cache traffic saturates bandwidth.
4. Prefill optimization targets compute efficiency: better kernel fusion, higher tensor core utilization, better matrix multiplication libraries. Decode optimization targets memory efficiency: smaller cache, better memory access patterns, fewer kernel launches.

PCIe, NVLink, and Interconnect Topology

For single-GPU inference, the GPU's internal memory and compute architecture dominates. For multi-GPU inference, the interconnect between GPUs matters enormously.

PCIe is the standard interface for connecting GPUs to the host and to each other through PCIe switches. PCIe Gen4 x16 provides approximately 16 GB/s in each direction. PCIe Gen5 doubles that to approximately 32 GB/s. These numbers are tiny compared to GPU memory bandwidth (2-4 TB/s), so PCIe is not suitable for heavy GPU-to-GPU communication.

NVLink is NVIDIA's high-speed interconnect designed specifically for GPU-to-GPU communication. NVLink speeds have increased with each generation:

- NVLink 2.0: 300 GB/s per GPU pair
- NVLink 3.0 (A100): 600 GB/s aggregate per GPU pair

- NVLink 4.0 (H100): 900 GB/s aggregate per GPU pair
- NVLink 5.0 (Blackwell): 1.8 TB/s

The distinction between per-link speed and aggregate speed matters. A GPU may have multiple NVLink links, and the aggregate bandwidth is the sum across all links. The NVLink Switch System allows many GPUs to be interconnected, not just pairs.

For tensor parallel inference, GPUs communicate frequently during every forward pass. Each layer requires all-to-all communication to exchange partial results of matrix multiplications. If the interconnect is slow, GPUs spend significant time waiting for data instead of computing.

A simple calculation shows the impact. For a model with 70B parameters split across 8 GPUs using tensor parallelism, each GPU holds approximately 17.5B parameters, or about 35GB in FP16. During each forward pass, GPUs exchange intermediate activations that are a fraction of this size, maybe 100-500MB per layer. With 80 layers and 8 GPUs, total communication per forward pass could be tens of gigabytes. At NVLink 3.0 speeds (600 GB/s), this takes milliseconds. At PCIe Gen4 speeds (16 GB/s), it takes seconds. The difference is dramatic.

The practical consequence is that multi-GPU inference should always use NVLink-connected GPUs when possible. Consumer GPUs like RTX cards typically lack NVLink and must use PCIe, which severely limits their multi-GPU performance for tensor parallelism. For this reason, enterprise workloads prefer data center GPUs (A100, H100) over consumer equivalents even when compute capacity is similar.

NVLink topology also affects multi-node inference. Within a node, GPUs are connected via NVLink and an NVSwitch. Between nodes, communication goes through the network, typically InfiniBand or high-speed Ethernet. Network bandwidth and latency are much worse than NVLink, so multi-node inference has higher communication overhead. vLLM's multi-node support handles this through Ray and careful collective communication optimization, but the overhead is unavoidable.

CUDA Streams and Concurrency Primitives

CUDA provides streams for executing operations concurrently on the GPU. A stream is an ordered sequence of operations executed in order on the device.

Multiple streams can execute concurrently, enabling overlap of computation and memory transfers, parallel execution of independent kernels, and better GPU utilization.

For inference systems, streams enable several important optimizations:

1. Overlap of prefill and decode computation. In a mixed batch, different parts of the computation may be independent and run on different streams.
2. Overlap of memory transfers with computation. While the GPU computes on one batch, it can load the next batch's data in the background.
3. Overlap of CPU and GPU work. The CPU can prepare inputs for the next step while the GPU is executing the current step.

vLLM uses streams extensively. The execution loop is structured so that data preparation, kernel launches, and result processing can overlap. In V1, this overlap is improved because the Engine Core process can schedule the next batch while GPU workers are executing the current batch, with communication through CUDA IPC and shared memory regions.

Another critical primitive is the CUDA event, which marks a point in a stream's execution. Events enable synchronization: one stream can wait for an event in another stream before proceeding. This allows fine-grained control over execution ordering without blocking the entire GPU.

CUDA Graphs, discussed in detail in Chapter 16, build on streams by recording a sequence of operations as a graph that can be replayed efficiently, eliminating kernel launch overhead and enabling more sophisticated concurrency patterns.

Modern inference systems also use pinned (page-locked) memory for faster CPU-GPU transfers. Pinned memory cannot be swapped by the OS, so the GPU can access it directly through DMA without going through the page table. The trade-off is reduced system memory flexibility and potential impact on system performance if too much memory is pinned.

This chapter has established the GPU hardware constraints that govern inference performance. Memory bandwidth is the dominant bottleneck for decode. Memory hierarchy awareness drives kernel design. Arithmetic intensity determines which optimizations matter. Interconnect speed limits multi-GPU scaling. CUDA streams enable concurrency and overlap. Every feature in

vLLM, from PagedAttention to continuous batching to quantization support, is a response to these hardware realities.

Chapter 3: Measuring What Matters: Inference Metrics and Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Latency Metrics: TTFT, ITL, and End-to-End Delay

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Throughput Metrics: Tokens per Second and Requests per Second

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

GPU Utilization and Efficiency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Workload Profiles: Interactive, Batch, and Hybrid

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Designing Reproducible Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Cost Models: Tokens, GPUs, and Dollars

Chapter 4: The KV-Cache and Memory Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Why the KV-Cache Exists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

KV-Cache Memory: How Much and Why It Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Fragmentation and the Allocation Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

PagedAttention: The Core Innovation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Block Tables and Virtual Memory for KV-Cache

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Eviction Policies and Cache Thrashing

Chapter 5: Batching and Scheduling Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Static Batching and Its Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Continuous Batching: Keeping the GPU Busy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Request Scheduling Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Preemption and Speculative Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Impact of Sequence Length Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Admission Control and Concurrency Limits

Chapter 6: Inside vLLM: Architecture and Execution Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The vLLM Process Model and Workers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Engine, Scheduler, and Executor Roles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Request Lifecycle in vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Model Runner and GPU Worker Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Event Loop and CPU-GPU Coordination

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

How vLLM Handles the API Server

Chapter 7: vLLM's PagedAttention Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Virtual Block Allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Block Table Construction and Maintenance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Attention Kernel with Paged KV-Cache

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Physical Memory Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Handling Variable Sequence Lengths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Interaction with the Scheduler

Chapter 8: Continuous Batching in vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Scheduler Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Running, Waiting, and Swapped States

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Preemption Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Chunked Prefill

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Integration with PagedAttention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Tuning Scheduler Parameters

Chapter 9: vLLM Configuration and Deployment Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Installation: Dependencies, CUDA, and Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Launching Your First vLLM Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

OpenAI-Compatible API Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Docker and Container Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Model Selection and Loading Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Basic Configuration Parameters

Chapter 10: Quantization for Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Why Quantize for Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Weight-Only Quantization: AWQ, GPTQ, INT8, FP8

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Activation Quantization and Perplexity Impact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

vLLM's Quantization Backends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Choosing the Right Precision for Your Workload

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Quantized Model Serving Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Chapter 11: Tensor Parallelism and Multi-GPU Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Limits of Single-GPU Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

How Tensor Parallelism Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

vLLM's Tensor Parallel Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Communication Overhead and NVLink

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Performance Scaling Curves

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Configuration for Multi-GPU Deployments

Chapter 12: Pipeline Parallelism and Expert Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Pipeline Parallelism Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Pipeline Bubble and Scheduling Challenges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

vLLM's Pipeline Parallelism Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Mixture of Experts and Expert Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

DeepSeek and Qwen MoE Models in vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Hybrid Parallelism Strategies

Chapter 13: Distributed Inference Across Nodes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Multi-Node Communication with Ray

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Network Topology and Bandwidth Planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Fault Tolerance Across Nodes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Load Balancing Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Kubernetes Deployments with vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Operational Complexity of Distributed Serving

Chapter 14: Caching, Prefix Caching, and Context Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Prefix Caching in vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Hash-Based Cache Lookup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Cache Validity and Model Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

System Prompts and Repeated Contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Multi-Query and Grouped-Query Attention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Long-Context Optimization Techniques

Chapter 15: Speculative Decoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Speculative Decoding Algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Draft Model Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Token Tree Speculation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

vLLM's Speculative Decoding Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Measuring Acceptance Rates and Speedup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

When Speculative Decoding Helps and Hurts

Chapter 16: CUDA Graphs and Kernel Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

CUDA Kernel Launch Overhead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

CUDA Graphs in vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Attention Backend Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

FlashAttention Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Profiling Kernel Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

When Low-Level Optimization Matters

Chapter 17: Production Serving Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Health Checks and Readiness Probes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Autoscaling Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Rate Limiting and Admission Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Observability: Metrics, Logs, and Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Authentication and API Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Rolling Upgrades and Model Versioning

Chapter 18: Performance Engineering and Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Establishing Performance Baselines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Profiling CPU and GPU with vLLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Diagnosing Bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Tuning for Interactive Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Tuning for Batch Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Configuration Decision Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Chapter 19: Troubleshooting and Failure Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Out-of-Memory Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

CUDA and Driver Incompatibilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Scheduling and Throughput Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

GPU Utilization Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Distributed System Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Version and API Compatibility Issues

Chapter 20: The Ecosystem: vLLM Versus Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Hugging Face Text Generation Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

TensorRT-LLM: NVIDIA's Approach

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

SGLang and Structured Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

llama.cpp and CPU/Edge Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Feature Matrix and Workload Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Chapter 21: The Economics and Future of LLM Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Hardware Economics: GPUs and Inference Cost

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Capacity Planning Methodology

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Emerging Techniques and Trends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

What Will Remain Stable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Architectural Recommendations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Conclusion: Engineering Judgment in Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

The Inference Engineering Mindset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Key Principles That Endure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Building Versus Buying

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

A Final Architecture Recommendation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Back Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Glossary of Key Terms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Quick Reference: Key vLLM Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

Performance Tuning Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vllmandtheengineeringoffastllminference.