PRACTICAL LLM INFERENCE

QUANTIZATION, GGUF AND LOCAL MODELS

UNDERSTAND
TRANSFORMERS, INFERENCE AND QUANTIZATION MATH

MASTER
GGUF INTERNALS AND MODEL FORMATS

DEPLOY
WITH LLAMA.CPP AND GPU OFFLOADING

OPTIMIZE
PERFORMANCE, MEMORY AND COST

BUILD
RELIABLE, MULTI-USER INFERENCE SERVICES

MARK LEVINSKY

Practical LLM Inference

Quantization, GGUF and Local Models

Steve Publications

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

This version was published on 2026-08-20

Contents

Quantization, GGUF and Local Models

This book teaches systems engineers how large language model inference actually works on real hardware, from transformer mechanics and quantization mathematics through GGUF internals and llama.cpp deployment to production-grade local inference services. You will learn to calculate exact memory requirements, diagnose performance bottlenecks systematically, configure GPU offloading correctly, benchmark rigorously, and build reliable multi-user inference systems that run efficiently on commodity CPUs and GPUs. No hand-waving about AI magic, just the engineering details that matter when you are responsible for making it work.

Introduction: The Local Inference Stack

Why Engineers Care About Local Inference

In 2024, OpenAI spent approximately $2.3 billion in inference compute alone, roughly fifteen times the estimated training cost of GPT-4.5 [1]. The economics of generative AI have flipped: building the model is now a relatively minor expense compared to running it. This shift has fundamentally changed what engineers need to understand about artificial intelligence systems.

For many organizations, sending every prompt through an external API is not viable. Some reasons are straightforward: data privacy requirements for medical or legal text, compliance constraints that forbid leaving certain jurisdictions, or simply the cost of high-volume inference on proprietary APIs. Other reasons are more technical: you need deterministic latency guarantees that shared cloud infrastructure cannot provide, or your application architecture requires tight integration with local services and databases.

Local LLM inference is not about running AI on a Raspberry Pi as a novelty. It is about understanding enough of the underlying mechanics to make informed decisions about hardware procurement, capacity planning, performance tuning, and reliability engineering for systems that serve real users in production environments. This book addresses that need directly.

What This Book Will Not Teach You

This book will not teach you how to prompt an AI chatbot effectively. It will not walk you through installing Ollama or LM Studio on your laptop so you can have conversations with a language model. It will not explain the difference between GPT-4 and Claude from a user perspective, nor will it recommend which foundation model is best for your use case based on qualitative assessments of output quality.

If you are looking for a guide to using AI as an end user, this is not that book. If you want to know whether local models can replace cloud APIs for

your specific application, this book will give you the technical framework to evaluate that question yourself, but it will not answer it for you. That decision depends on requirements I do not know: your latency tolerances, accuracy thresholds, budget constraints, and risk appetite.

This book assumes you are a systems engineer, ML engineer, DevOps professional, or technically-minded developer who needs to deploy local LLMs as part of a larger infrastructure. You should be comfortable with Linux command-line tools, have basic familiarity with containers, and understand concepts like memory, bandwidth, and latency from prior experience with backend systems. You do not need deep knowledge of neural network internals or transformer architecture, but you should be willing to learn how these architectures actually execute on hardware.

The Real Bottleneck: Memory Bandwidth, Not Compute

Here is the central thesis of this book: running large language models locally is constrained primarily by memory bandwidth and arithmetic intensity, not raw compute power. Understanding this fact changes everything about how you approach local inference.

Modern GPUs advertise peak performance in teraflops or petaflops. An NVIDIA A100 can deliver 19.5 TFLOPS of FP16 throughput [2]. An H100 reaches 989 TFLOPS in sparse FP16 mode [3]. These numbers are impressive and they matter for training, where the same weights are reused across massive batches of data with high arithmetic intensity. Inference is different.

When a transformer generates one token autoregressively, it must load every weight in the model from memory, perform matrix multiplications against a single-token input, then write back activations and attention states. The computation per token is fixed regardless of GPU FLOPS rating: roughly 2N floating-point operations where N is the number of non-embedding parameters [4]. For a 70 billion parameter model, each generated token requires about 140 billion FLOPS. On an A100 with 19.5 TFLOPS FP16 throughput, that would take approximately 7 milliseconds if the GPU were compute-bound and fully utilized.

But GPUs are not fully utilized during single-token generation because they spend most of their time waiting for data to arrive from memory. The arithmetic intensity of decode-phase inference is extremely low: you perform a

relatively small number of operations on each byte loaded from memory. When arithmetic intensity falls below the hardware ratio of compute-to-bandwidth, the system becomes memory-bound rather than compute-bound [5].

Consider the numbers for an A100 with 2 TB/s HBM bandwidth and 19.5 TFLOPS FP16 throughput. The ops-per-byte ratio is approximately 9.8. If the transformer inference workload has arithmetic intensity below 9.8 operations per byte loaded, memory bandwidth is the bottleneck regardless of how many FLOPS the GPU can theoretically deliver [6]. During decode-phase inference with batch size 1, that is exactly what happens: the GPU cores sit idle waiting for weights and KV-cache entries to stream through HBM.

This insight explains several counterintuitive observations about local inference:

- A slower GPU with higher memory bandwidth can outperform a faster GPU with lower bandwidth on single-user workloads
- Adding more GPUs does not speed up single-token generation unless you use tensor parallelism across layers
- CPU inference on systems with fast DDR5 or LPDDR5 memory can be surprisingly competitive for smaller models
- Quantization improves performance not primarily by reducing computation but by reducing memory traffic

The prefill phase is different from decode. When processing a long prompt, all input tokens pass through the transformer in parallel, creating high arithmetic intensity that can saturate GPU compute units. Prefill is typically compute-bound for prompts longer than approximately 480 tokens [7]. This asymmetry between phases means that optimizing inference requires different strategies for each: prefill benefits from compute acceleration and large batch sizes, while decode benefits from memory bandwidth optimization and reduced data movement.

How to Read This Book

The book is organized progressively from fundamentals through optimization to production deployment. Each chapter builds on previous material, so reading in order is recommended if you are new to transformer inference

internals. Experienced readers can skip ahead to chapters relevant to their immediate needs.

Chapters 1 and 2 establish the foundational understanding of how transformers actually execute during inference: the forward pass mechanics, why autoregressive generation creates specific performance characteristics, and how tokenization shapes everything downstream. You cannot optimize what you do not understand, so these chapters matter even if your primary interest is practical deployment.

Chapter 3 explains model weight formats and the GGUF architecture in detail. If you have ever wondered what is actually inside a .gguf file or why certain quantization types exist, this chapter provides the answer. Understanding the format helps with debugging, conversion, and making informed choices about precision levels.

Chapter 4 covers quantization mathematics and tradeoffs comprehensively. This is one of the most important chapters in the book because quantization decisions directly determine whether your model fits in available memory and how fast it runs. I explain the actual math behind different quantization schemes and provide concrete guidance on selecting appropriate precision levels for different use cases.

Chapter 5 gives you the tools to calculate exact RAM and VRAM requirements for any model configuration. You will learn formulas for weight storage, KV-cache sizing across different attention architectures, and how memory bandwidth compares across CPU and GPU platforms. This chapter is designed to be referenced when making hardware purchasing decisions or capacity planning for deployments.

Chapter 6 is a practical hands-on guide to llama.cpp: building from source on all three major platforms, acquiring models, converting weights, quantizing, and running basic inference with complete reproducible commands. If you want to start running models immediately after reading the fundamentals, this chapter gets you there.

Chapters 7 through 10 cover optimization and measurement in depth. Chapter 7 explains GPU offloading strategies including partial offloading behavior and when each approach wins. Chapter 8 dives into KV-cache design, context window implications, and memory management techniques for long contexts. Chapter 9 addresses throughput, batching, and concurrency patterns for serving multiple users. Chapter 10 teaches rigorous benchmarking

methodology: what metrics actually matter, how to measure them correctly, and how to systematically identify and fix bottlenecks.

Chapters 11 and 12 focus on production concerns. Chapter 11 walks through building real systems including a private assistant, coding assistant integration, RAG service architecture, and multi-user inference API design with containers, observability, and security considerations. Chapter 12 provides systematic troubleshooting procedures for common failures and compares llama.cpp with alternative runtimes to help you select the right tool for your workload.

The conclusion synthesizes the book's core lessons into durable engineering principles that will remain useful as specific tools and techniques evolve. The field moves quickly: new quantization formats appear, new backends are added, and model architectures change. Understanding fundamentals enables better decisions than memorizing configurations that may be obsolete in six months.

Throughout the book, inline citations [n] point to sources for factual claims. All references are collected at the end with full URLs. Code examples and commands are complete and tested where possible, though specific behavior may vary slightly across llama.cpp versions due to ongoing development. When I present my own assessment or inference rather than cited fact, I will state that explicitly.

The goal is straightforward: after reading this book, you should be able to look at any LLM model file, examine your available hardware, and determine whether the model will run, how fast it will run, what configuration changes will improve performance, and how to deploy it reliably as a production service. That is practical LLM inference.

Chapter 1: Transformer Inference Fundamentals

From Training to Inference: What Changes

A transformer model is trained on massive datasets by processing sequences of tokens in parallel and adjusting weights through backpropagation. During training, the entire sequence passes through each layer simultaneously. A batch might contain dozens or hundreds of sequences, each with thousands of tokens, all processed in a single forward pass followed by a backward pass that computes gradients for weight updates.

Inference operates under completely different constraints. The model is frozen: weights are fixed and never updated. Input arrives as individual requests rather than structured training batches. Most critically, generation is autoregressive: each new token depends on all previous tokens in the sequence, so tokens must be generated one at a time rather than in parallel [8].

This fundamental difference between training and inference shapes every optimization decision in local deployment. Techniques that accelerate training often do nothing for inference, and optimizations that dramatically improve inference performance may not help training at all. Understanding what changes between these modes is essential because it explains why inference behaves the way it does.

During training, computational cost scales with batch size times sequence length squared due to attention computation across all token pairs. The arithmetic intensity is high: each weight is reused many times across the batch and sequence, so the ratio of operations to memory accesses favors compute utilization [9]. Modern GPU clusters can keep thousands of cores busy because there is enough parallel work to saturate them.

During inference with autoregressive generation, the story changes dramatically. After processing the initial prompt, each new token requires a complete forward pass through all layers using only that single token as input.

The weights are loaded from memory, multiplied against a tiny activation tensor, and the result produces logits for one token position. This pattern repeats until generation completes or reaches a maximum length.

The computational asymmetry is stark. Training a 70 billion parameter model on a dataset of trillions of tokens might take weeks across hundreds of GPUs but amortizes that cost over massive throughput. Running inference on the same model to generate 1,000 tokens requires approximately 140 trillion FLOPS total (2N operations per token times 1,000 tokens) regardless of how many GPUs are available, because each token depends sequentially on its predecessors [10].

The Forward Pass Anatomy

Understanding the transformer forward pass is necessary for understanding inference performance. Each layer in a decoder-only transformer like LLaMA performs three main operations in sequence: normalization, self-attention, and feed-forward computation. Let us examine what each operation does and why it matters for inference.

The input to the first layer is token embeddings: vectors of dimension d_model (for example, 4,096 for LLaMA 3.1 8B) looked up from an embedding table indexed by token IDs. Each subsequent layer receives the output of the previous layer as its input and produces a modified representation that flows forward.

Normalization layers apply RMSNorm or LayerNorm to stabilize activations. These operations are element-wise and relatively cheap computationally, but they still require loading the activation tensor and applying per-channel statistics. For inference optimization purposes, normalization is rarely a bottleneck compared to attention and feed-forward operations.

Self-attention is where the transformer derives its name and most of its computational cost. Given an input sequence of token representations, self-attention computes for each token a weighted combination of all token representations in the sequence, where weights are determined by similarity between query and key vectors [11]. The operation proceeds as follows:

First, three linear projections transform each token representation into query (Q), key (K), and value (V) vectors. These projections use weight matrices W_Q, W_K, and W_V of shape d_model times d_model. For a single

token input during decode-phase inference, each projection is a matrix-vector multiplication requiring approximately 2 * d_model^2 FLOPS.

Second, attention scores are computed as the dot product between queries and keys, scaled by the square root of key dimension. For sequence length n, this produces an n times n attention matrix. During prefill with all tokens processed in parallel, computing QK^T requires $O(n^2 * d_k)$ operations where d_k is the key dimension [12]. During decode with a single new token attending to n existing tokens, only one row of the attention matrix needs computation: approximately 2 * n * d_k FLOPS.

Third, softmax normalizes attention scores into probabilities that sum to one across attended positions. This operation is element-wise over the attention scores and relatively cheap compared to matrix multiplications.

Fourth, weighted values are computed by multiplying attention probabilities with value vectors. During decode, this requires loading all n stored value vectors from memory and computing a weighted sum: approximately 2 * n * d_v FLOPS plus significant memory traffic to load those vectors.

The feed-forward network (FFN) follows self-attention in each layer. Modern transformers typically use SwiGLU activation with two projection matrices W_UP and W_GATE projecting to an intermediate dimension d_ffn (often 4 times d_model), followed by a final projection W_DOWN back to d_model [13]. For a single token, the FFN requires approximately 6 * d_model * d_ffn FLOPS across its three matrix multiplications.

The residual connections and skip layers add outputs of each sub-block to their inputs before passing forward. These are element-wise additions that cost negligible computation but require loading both tensors from memory.

For a model with L layers, generating one token during decode requires approximately 2NL FLOPS where N is total parameters, distributed across attention projections, attention scoring against cached keys, value aggregation, and FFN computations in each layer [14]. The exact distribution varies by architecture but the linear scaling with depth and parameter count holds universally.

Self-Attention and Its Costs

Self-attention is simultaneously the transformer's greatest strength and its primary performance challenge during inference. The mechanism enables

every token to attend to information from any other position in the sequence, capturing long-range dependencies that convolutional or recurrent architectures struggle with [15]. This capability is what allows transformers to maintain coherent context over thousands of tokens.

The cost comes from two sources: quadratic scaling and memory requirements. Standard self-attention computes pairwise interactions between all tokens in a sequence, resulting in O(n^2) time and memory complexity where n is sequence length [16]. During training with large batches and long sequences, this quadratic term dominates computational cost.

During inference, the situation is nuanced because of caching. When processing the initial prompt (prefill phase), attention must compute interactions between all pairs of input tokens just as in training. For a prompt of 2,048 tokens, the attention matrix has over four million entries. This computation is expensive but parallelizable: all tokens in the prompt can be processed simultaneously across GPU cores.

The key insight for inference optimization is that once tokens have been processed through attention, their key and value projections do not change. When generating the next token, there is no need to recompute keys and values for previous tokens. Instead, those projections are cached and reused [17]. The new token only needs to:

- Compute its own query, key, and value vectors (cheap: single-token linear projections)
- Attend to all cached keys from previous tokens (moderate: one row of attention scores)
- Aggregate cached values using computed attention weights (memory-intensive: load all cached values)

This caching strategy reduces the per-token cost during decode from O(n^2) to O(n), but it introduces a new constraint: the cache itself must be stored in memory. For each token in context, every layer stores key and value vectors of dimension d_k and d_v respectively. The total cache size grows linearly with sequence length and number of layers, and for large models at long contexts, this cache can exceed the model weights themselves in memory consumption [18].

Consider a concrete example. LLaMA 3.1 8B has 32 layers, uses grouped-query attention with 8 key-value heads per layer, and each head dimension is

128. For a single request with 32,768 tokens of context at FP16 precision (2 bytes per element), the KV-cache requires:

2 * 32 layers * 8 kv_heads * 128 head_dim * 32,768 tokens * 2 bytes = 4,294,967,296 bytes = approximately 4 GB [19]

For a 70B model with more layers and heads at the same context length, the cache exceeds 30 GB. This is why context window configuration directly determines whether a model fits in available memory, independent of quantization level applied to weights.

Why Autoregression Matters

Autoregressive generation means each token depends on all preceding tokens. The model predicts a probability distribution over the vocabulary for position t+1 conditioned on tokens at positions 0 through t, samples from that distribution (or selects the highest-probability token), then feeds the result back as input for predicting position t+2 [20].

This sequential dependency has profound implications for system design. Unlike training where parallelism across tokens and batch items enables massive throughput, autoregressive decode is inherently serial along the generation dimension. You cannot predict token 10 without first generating tokens 0 through 9, so no amount of GPU compute power can generate all output tokens simultaneously.

The practical consequence is that inference latency scales linearly with output length. Generating 100 tokens takes approximately ten times longer than generating 10 tokens, assuming constant per-token computation and no batching effects. For interactive applications where users expect responses within seconds, this limits practical output lengths or requires optimization to reduce per-token latency.

Autoregression also creates a specific memory access pattern during decode that is unfriendly to hardware optimization. Each generation step must load the entire model from memory (or keep it resident in fast memory), plus all cached keys and values for every token in context. The computation performed on each loaded byte is relatively small: matrix multiplications against single-token activations followed by attention over cached states [21]. This low arithmetic intensity means memory bandwidth, not compute throughput, determines generation speed.

Some research explores non-autoregressive or semi-autoregressive approaches to generation that would enable parallel token prediction and reduce latency [22]. These methods typically trade quality for speed: models that generate multiple tokens in parallel tend to produce less coherent output than autoregressive baselines. For most production use cases where output quality matters, autoregressive generation remains the standard despite its sequential constraints.

The sequential nature of autoregression does not mean inference is entirely serial. Multiple independent requests can be processed concurrently through batching techniques, and within each forward pass, operations across layers and heads can utilize parallel compute units. But along the dimension that matters most for single-request latency (tokens generated), there is no escaping the sequential dependency imposed by autoregressive prediction.

Arithmetic Intensity and Compute Density

Arithmetic intensity measures how many floating-point operations a computation performs per byte loaded from memory. It is defined as:

arithmetic_intensity = total_FLOPs / bytes_loaded_from_memory [23]

This metric determines whether a workload is compute-bound or memory-bound on a given hardware platform. Hardware has its own ops-per-byte ratio determined by peak FLOPS divided by peak memory bandwidth. If the workload arithmetic intensity exceeds the hardware ratio, the system is compute-bound: processors are saturated performing calculations while waiting for more work to arrive. If workload intensity falls below the hardware ratio, the system is memory-bound: processors sit idle waiting for data [24].

Modern GPUs have extremely high ops-per-byte ratios due to massive parallel compute arrays paired with limited (albeit fast) memory bandwidth. An NVIDIA A100 delivers 19.5 TFLOPS FP16 throughput with 2 TB/s HBM bandwidth, yielding an ops-per-byte ratio of approximately 9.8 [25]. This means any workload performing fewer than 9.8 operations per byte loaded cannot fully utilize the GPU's compute capacity regardless of how many cores are available.

Transformer training has high arithmetic intensity because weights are reused extensively across large batches and long sequences. Each weight participates in thousands of matrix multiplications before being reloaded,

creating an ops-per-byte ratio well above hardware limits. Training saturates GPU compute units and benefits directly from higher FLOPS ratings.

Transformer inference during decode has extremely low arithmetic intensity. Consider a 7B parameter model at FP16 precision (14 GB of weights). Generating one token requires approximately 14 billion FLOPS but must load all 14 GB of weights plus KV-cache entries from memory [26]. The arithmetic intensity is:

14,000,000,000 FLOPs / 14,000,000,000 bytes = approximately 1 operation per byte

This is far below the A100's 9.8 ops-per-byte ratio, confirming that decode-phase inference is memory-bound. The GPU cores spend most cycles waiting for data rather than performing calculations [27].

This analysis explains why quantization improves inference performance so dramatically. Reducing weights from FP16 (2 bytes per parameter) to INT4 (0.5 bytes per parameter) reduces memory traffic by a factor of four while reducing computation only slightly. The arithmetic intensity increases proportionally, bringing the workload closer to hardware limits and allowing more compute utilization [28]. For memory-bound workloads like LLM decode, performance scales roughly linearly with available bandwidth, so halving data movement approximately halves latency.

The prefill phase tells a different story. Processing a long prompt in parallel creates large matrix multiplications where each weight is reused across many token positions within the batch. Arithmetic intensity rises significantly and can exceed hardware ratios for sufficiently long prompts, making prefill compute-bound rather than memory-bound [29]. This asymmetry between phases means optimal inference systems use different strategies for each: aggressive batching and compute acceleration for prefill, quantization and bandwidth optimization for decode.

Understanding arithmetic intensity is essential because it tells you what hardware improvements actually matter. For single-user local inference dominated by decode-phase latency, buying a GPU with more FLOPS but similar bandwidth yields diminishing returns. Adding memory bandwidth through faster memory types or wider buses has direct impact. Running multiple concurrent requests increases batch size and arithmetic intensity, eventually saturating compute units and making higher-FLOPS hardware worthwhile again. The right optimization depends entirely on your workload character-

istics.

Chapter 2: Tokenization, Prefill, and Decode

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

How Tokenizers Actually Work

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

The Prefill Phase: Parallel Computation

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

The Decode Phase: Sequential Bottleneck

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

Context Length and Its Real Cost

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

Why These Phases Demand Different Optimizations

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

Chapter 3: Model Weights, Formats, and GGUF Architecture

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

From Checkpoints to Inference: Weight Formats

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

The GGML Story and Design Goals

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

GGUF Header Structure and Metadata

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

Tensor Storage Layout and Quantization Types

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

Reading a GGUF File: A Practical Walkthrough

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

Chapter 4: Quantization: Mathematics, Methods, and Tradeoffs

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

Why Quantization Works on Transformers

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

Symmetric vs Asymmetric Quantization

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

Per-Tensor, Per-Channel, Per-Group Schemes

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

GGUF Quantization Types Decoded

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

Measuring Quality Loss: Benchmarks and Perception

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

Choosing a Quant Level for Your Use Case

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

Chapter 5: Memory, Bandwidth, and Hardware Constraints

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

Computing Model Size: The Exact Formula

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

KV-Cache Memory Requirements

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

Memory Bandwidth vs Compute Throughput

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

CPU Architectures for Inference

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

GPU Architectures and VRAM Realities

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

Hardware Selection Decision Framework

Chapter 6: llama.cpp: Installation, Models, and Core Usage

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

Building llama.cpp from Source

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

Installing Prebuilt Binaries

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

Acquiring Models: Hugging Face and Beyond

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

Converting Weights to GGUF

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

Quantizing Models with llama.cpp

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

Your First Inference Run

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

Chapter 7: GPU Acceleration and Offloading Strategies

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

How Offloading Works Under the Hood

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

Full vs Partial Offloading: When Each Wins

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

CUDA Backend Configuration

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

Vulkan and Cross-Vendor GPU Support

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

Apple Silicon and Metal Acceleration

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

Diagnosing Offloading Failures

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

Chapter 8: KV-Cache, Context Windows, and Memory Management

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

How the KV-Cache Works

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

Sizing Your KV-Cache Correctly

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

Context Overflow and Handling Strategies

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

RoPE Scaling and Long-Context Techniques

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

Cache Eviction and Sliding Windows

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

Memory Fragmentation and Allocation Patterns

Chapter 9: Throughput, Batching, and Concurrency

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

Single Request vs Concurrent Serving

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

Static Batching Mechanics

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

Continuous Batching and Scheduling Strategies

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

Queue Management Under Load

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

Configuring llama.cpp Server for Throughput

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

Capacity Planning and Dimensioning

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

Chapter 10: Benchmarking, Profiling, and Bottleneck Diagnosis

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

Metrics That Matter: Definitions and Formulas

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

Designing Valid Benchmark Experiments

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

Profiling CPU Utilization and Vector Units

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

GPU Profiling and Occupancy Analysis

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

Memory Bandwidth Measurement Techniques

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

Systematic Bottleneck Diagnosis Workflow

Chapter 11: Production Systems: APIs, RAG, and Deployment

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

Building a Private Local Assistant

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

Coding Assistant Integration Patterns

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

RAG Service Architecture

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

Multi-User Inference API Design

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

Containerization and Orchestration

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

Observability, Security, and Reliability

Chapter 12: Troubleshooting, Comparisons, and the Road Ahead

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

Common Failure Modes and Diagnostics

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

Performance Troubleshooting Decision Tree

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

llama.cpp vs Other Inference Runtimes

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

What Is Durable vs Rapidly Changing

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

Emerging Trends and Open Problems

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

Conclusion: Engineering Judgment Over Magic Numbers

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

References

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