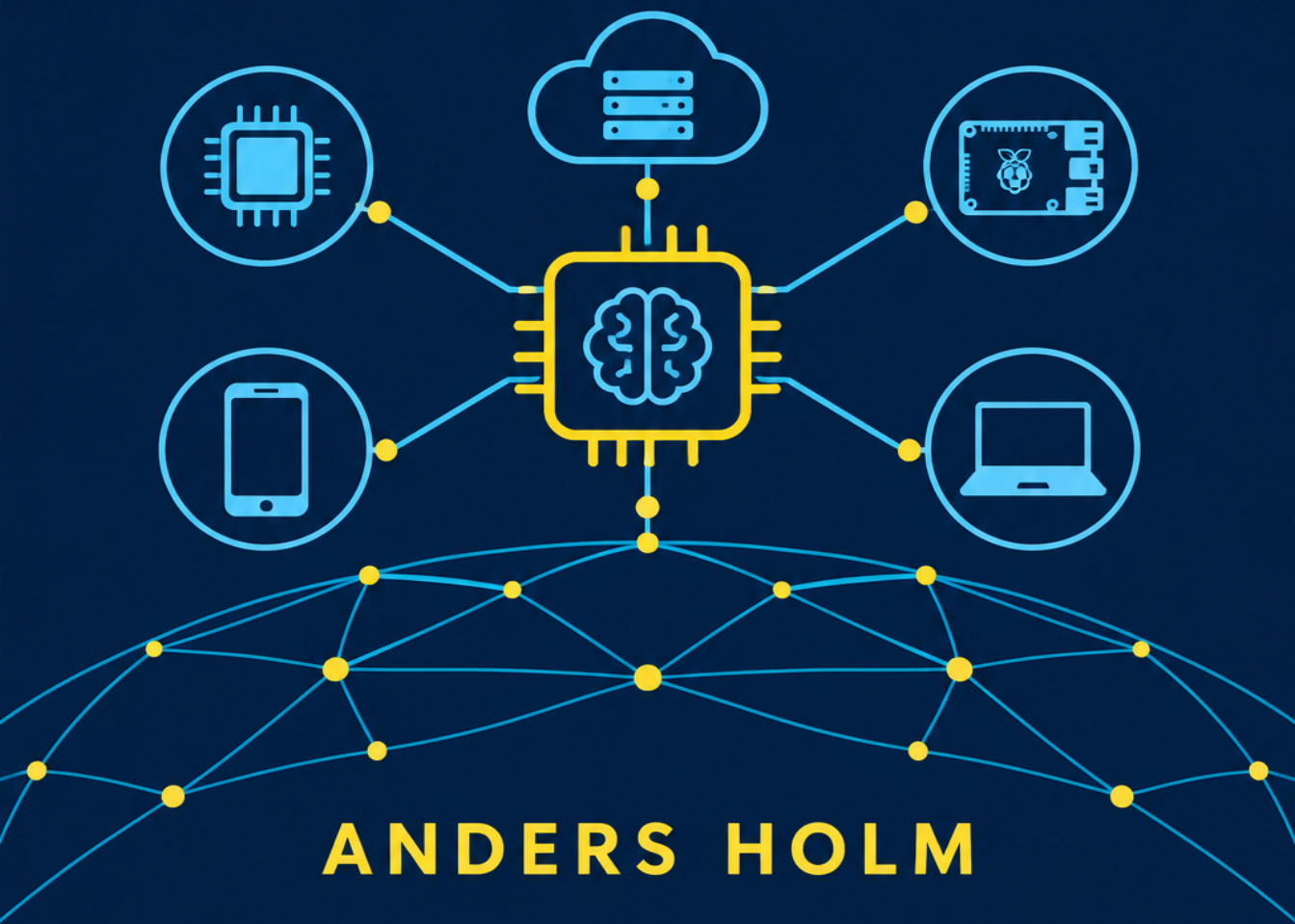


RUNNING **LLMs** **ON TINY HARDWARE**

A Systems Engineer's Guide to
Edge Inference on Resource-Constrained
Devices



ANDERS HOLM

Running LLMs on Tiny Hardware

A Systems Engineer's Guide to Edge Inference on Resource-Constrained Devices

Steve Publications

This book is available at <https://leanpub.com/runningllmsontinyhardware>

This version was published on 2026-09-29



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

A Systems Engineer’s Guide to Edge Inference on Resource-Constrained Devices	1
Introduction	2
What This Book Covers	2
How to Read This Book	3
Technology Context	4
The Central Thesis	5
Chapter 1: The Physics of Tiny	7
Why Memory Rules Everything	7
SRAM, DRAM, and Flash: Capacity, Speed, and Cost	8
Memory Hierarchy and Cache Behavior	9
Compute vs. Bandwidth: Arithmetic Intensity	10
Power Budgets and Thermal Walls	11
The Numbers: How Much Memory and Bandwidth Does an LLM Actually Need	12
Chapter 2: Inside an LLM Inference Step	14
Tokenization: Strings to Integers	14
The Embedding Layer and Positional Encoding	16
Transformer Blocks: Self-Attention and Feed-Forward	17
Multi-Head and Grouped-Query Attention	19
Layer Normalization and Residual Connections	20
Autoregressive Decoding and the KV Cache	21
The Complete Inference Dataflow	22
Chapter 3: Measuring Feasibility	25
Parameter Storage: Bits, Bytes, and Formats	25
Activation Memory: What You Cannot See Coming	25
KV Cache: The Hidden RAM Killer	25

Arithmetic Intensity and Compute Requirements	25
Bandwidth Calculations: The True Bottleneck	25
The Feasibility Checklist	25
Chapter 4: Model Architectures for Constrained Inference	27
Dense Decoder-Only Transformers: The Baseline	27
Mixture of Experts: Selective Activation	27
Recurrent and Hybrid Architectures	27
Compact Language Models: Design for Small	27
Architecture Choices That Matter for Embedded: Context Window, Heads, FFN Width	27
Selecting a Model Family for Your Target	27
Chapter 5: Quantization Fundamentals	29
Why Quantization Works: Redundancy in Floating-Point Weights . . .	29
Symmetric vs. Asymmetric Quantization: Scales and Zero-Points . . .	29
Per-Tensor vs. Per-Channel vs. Group-wise	29
Calibration Strategies	29
The Quantization-Error-Accuracy Tradeoff	29
Quantization Formats: FP16, BF16, INT8, and Lower	29
Chapter 6: Weight-Only Quantization	31
How Weight-Only Quantization Works	31
Dequantization on the Fly: The Compute Cost	31
Packing Schemes: 4-bit, 5-bit, 6-bit Weights	31
Weight-Only Quantization and Activation Outliers	31
Implementing Weight-Only Matrix Multiplication	31
When Weight-Only Is Enough (and When It Is Not)	31
Chapter 7: Activation Quantization and Mixed Precision	33
Why Activations Are Harder to Quantize	33
Activation Outliers and Their Impact	33
Mixed Precision Strategies	33
Quantization-Aware Training	33
Post-Training Quantization with Activation Calibration	33
INT4/INT3 Activation Quantization: The Cutting Edge	33
Chapter 8: Sparsity, Pruning and Structured Compression	35
Unstructured vs. Structured Sparsity	35
Pruning Methods and Magnitude Thresholding	35

CONTENTS

Compressed Sparse Row and Other Formats	35
The Problem of Sparsity on Small CPUs	35
Block-wise and Channel-wise Pruning	35
Low-Rank Factorization	35
When to Use Pruning and Sparsity	36
Chapter 9: Knowledge Distillation and Architecture Modification	37
Distillation: Making Small Models Smarter	37
Architecture Surgery: Head Merging, Layer Thinning	37
Vocabulary Reduction and Tokenizer Optimization	37
Context Window Management and Compression	37
Speculative Decoding: Drafting and Verifying	37
When to Modify the Model vs. When to Optimize the Runtime	37
Chapter 10: KV Cache Engineering	39
KV Cache Layout: Memory Mapping the Context	39
KV Cache Quantization and Compression	39
Eviction Policies and Sliding Windows	39
Activation Checkpointing and Recomputation	39
Streaming and Weight-Reuse Strategies	39
KV Cache for Long Context on Tiny Memory	39
Chapter 11: CPU, SIMD and Vector Optimization	41
Vector Units: NEON, SSE, AVX, and RISC-V V-Extensions	41
Writing Vectorized Matmul Kernels	41
Packing and Unpacking for SIMD	41
Instruction-Level Parallelism and Loop Unrolling	41
Memory Access Patterns and Cache Locality	41
Tuning for Specific CPU Microarchitectures	41
Chapter 12: GPU, NPU and DSP Acceleration	43
Mobile GPUs: Mali, Adreno, PowerVR	43
Neural Processing Units: NPU Architectures and Constraints	43
DSP Acceleration for Embedded Systems	43
Tensor Cores and Matrix Multiply Accelerators	43
Accelerator APIs and Vendor SDKs	43
When Accelerators Help (and When They Get in the Way)	43
Chapter 13: Inference Runtimes and Software Architecture	45
The Runtime Stack: From Model Files to Generated Tokens	45

CONTENTS

Graph Representations and IR Formats	45
Lightweight Transformer Runtimes	45
Vendor-Specific Runtimes vs. Portable Solutions	45
Kernel Selection and Code Generation	45
Designing Your Own Minimal Inference Runtime	45
Chapter 14: Model Conversion and Deployment Pipelines	47
Exporting Models: ONNX, GGUF, safetensors, and Others	47
Tokenizer Packaging and Conversion	47
Quantization Pipelines: Tools and Techniques	47
Cross-Compilation and Build Systems	47
Binary Size, Flash Constraints, and OTA Updates	47
Model Versioning and Update Strategies	47
Chapter 15: Performance Engineering and Benchmarking	49
Metrics That Matter: TTFT, ITL, Throughput, and Energy	49
Benchmarking Methodology: Warm-Up, Sampling, Reproducibility . .	49
Profiling: Finding the Real Bottleneck	49
Compute-Bound vs. Memory-Bound vs. Synchronization-Bound . . .	49
Compiler Optimization and Its Surprises	49
Performance Regression and Change Management	50
Chapter 16: Power, Energy and Thermal Management	51
Measuring Power and Energy in Real Deployments	51
Energy per Token: A Practical Metric	51
Dynamic Voltage and Frequency Scaling	51
Thermal Throttling and Sustained Performance	51
Power-Optimized Scheduling and Batching	51
Chapter 17: Debugging and Failure Analysis	52
Incorrect Outputs: Numerical Error vs. Model Behavior	52
Memory Exhaustion and Fragmentation	52
Stack Overflow and Heap Corruption	52
Quantization Bugs: Zero-Point and Scale Errors	52
Hardware-Specific Issues: Alignment, Endianness, FPU Modes	52
Silent Accuracy Degradation and Regression Testing	52
Chapter 18: Security and Privacy	54
The Security Model of Local Inference	54
Model Extraction and Protection	54

Supply Chain Risks: Malicious Models and Tooling	54
Input and Output Security: Prompt and Data Handling	54
Secure OTA Updates and Integrity Verification	54
Privacy Advantages and Their Limits	54
Chapter 19: Case Study : An LLM in 256 MB of RAM	56
The Hardware and Its Constraints	56
Choosing and Preparing the Model	56
Memory Budget: Weights, KV Cache, Runtime Overhead	56
Implementation Decisions and Trade-offs	56
Benchmarking Results	56
Lessons Learned	56
Chapter 20: Case Study : Offline Edge Assistant on an ARM SBC	58
System Requirements and Architecture	58
Model and Tokenizer Selection	58
Building the Inference Engine	58
Application Integration: Input, Output, Streaming	58
Robustness: Error Handling, Watchdogs, Recovery	58
Production Readiness Checklist	58
Chapter 21: Case Study : Pushing the Limits on MCU-Class Hardware	60
MCU Constraints: Kilobytes of RAM, Megahertz Clocks	60
Micro-Models: Can Transformers Run on an MCU?	60
Extreme Quantization and Architecture Reduction	60
Alternative Approaches: Recurrent and State-Space Models	60
Measured Performance and Practical Viability	60
The Real Limit: Where LLMs Cannot Go	60
Chapter 22: Production Deployment Patterns	62
Firmware Integration and Boot-Time Constraints	62
Multi-Model and Fallback Strategies	62
Handling Concurrent Requests on Constrained Systems	62
Observability: Logging, Metrics, Telemetry	62
A/B Testing and Model Rollout at the Edge	62
Maintenance and Lifecycle Management	62
Chapter 23: Future Directions	64
Architectural Innovations: Mamba, RWKV, and State-Space Models	64
Hardware Roadmaps: What Next-Gen Edge Silicon Will Bring	64

Algorithmic Advances: Better Quantization, Sparsity, Distillation . . .	64
Compiler and Runtime Innovations	64
The Economics of Edge Inference	64
The Next Frontier: What Comes After Tiny LLMs	65
Conclusion: The Engineer's Playbook	66
The Decision Tree: Choosing Your Optimization Strategy	66
A Checklist for Edge LLM Deployment	66
Accepting Trade-offs: No Solution Is Free	66
Final Thoughts: The Promise and Limits of Edge Intelligence	66
References	67

A Systems Engineer's Guide to Edge Inference on Resource-Constrained Devices

This book teaches you how to make modern language models run on extremely constrained devices: microcontrollers with kilobytes of RAM, single-board computers with gigabytes, and mobile chips with NPUs. It is a practical engineering handbook for systems programmers who want to deploy LLM inference entirely offline on real hardware. You will learn the physics of why LLM inference is hard on small devices, the mathematics of transformer computation, the techniques for shrinking and accelerating models, the architecture of inference runtimes, and the skills needed to benchmark, debug, and deploy working systems. Every chapter builds on the previous one, culminating in complete case studies that show you how to design and implement an on-device LLM system from scratch.

Introduction

There is a photograph that circulated widely after Meta released the LLaMA models in early 2023. It showed a developer sitting in front of a Raspberry Pi, a credit-card-sized computer with a few gigabytes of RAM, running an eight-billion-parameter language model in C++. The headline read something like “LLM on a \$35 computer,” and the internet was fascinated.

The fascination is understandable. Language models have become the default tool for working with text, but they have also become famously demanding. Training them requires clusters of specialized GPUs. Serving them in production requires dedicated clusters of servers. Running even a modest model “in the cloud” costs money per request, introduces latency and hands your data to someone else’s infrastructure.

Running a model entirely on-device changes everything. The model runs locally. The response is instantaneous for the user because there is no network round-trip. The data never leaves the device. The inference cost is fixed at purchase time instead of recurring per call. Offline operation is possible because the model lives on the device itself.

The photograph was not magic. It was a series of engineering decisions, each one rooted in understanding exactly what an LLM needs from hardware and exactly what a small device can provide. This book explains those decisions systematically. It is not a survey of existing tools or a collection of disconnected tutorials. It is a coherent development of the knowledge and skills needed to design, implement and deploy language model inference systems on tiny hardware.

What This Book Covers

The book progresses in a deliberate order, starting from physical constraints and building upward through model architecture, quantization, optimization, runtime design, deployment and finally complete case studies. The chapters fall into these groups:

The first section establishes why LLM inference is fundamentally a memory problem, not a compute problem. You will see the physical constraints of tiny hardware: the capacity and bandwidth of different memory types, the cache hierarchies, the power budgets, the thermal limits. You will learn to calculate whether a given model can fit on a given device before writing any code.

The second section explains transformer inference from first principles. Tokenization, embeddings, positional representations, attention, feed-forward networks, normalization, the KV cache, autoregressive decoding. You will understand exactly what is being computed at each step and why that computation has the properties it does. This understanding is necessary because you cannot optimize what you do not understand.

The third section covers model reduction and optimization: quantization from INT8 down to sub-4-bit formats, weight-only and activation quantization, pruning, knowledge distillation, architecture modification and KV cache compression. For each technique, the book explains the mechanism, the implementation strategy, the expected benefits and costs, the accuracy implications and the situations where it should or should not be used.

The fourth section treats hardware as a first-class topic. CPUs and their SIMD/vector units, GPUs, NPUs, DSPs, accelerators. You will learn what instruction sets are available, how memory hierarchies behave, what numerical formats accelerators support, and how to map inference operations onto real silicon efficiently.

The fifth section covers the software stack: inference runtimes, graph representations, kernel selection, model conversion, build systems, deployment pipelines, performance engineering, power management, debugging and security. You will learn how systems like llama.cpp, MLC-LLM and ONNX Runtime fit together, and when to build your own minimal runtime.

The final section presents detailed case studies that work from realistic hardware constraints backward to complete working deployments: fitting a model into 256 MB of RAM, building an offline edge assistant on an ARM SBC and exploring what is possible on MCU-class hardware. Each case study shows the calculations, design decisions, compromises, implementation, benchmarks and lessons learned.

How to Read This Book

This book assumes you are an experienced programmer or systems engineer. You should be comfortable reading C and C++ code, understanding how memory is laid out, knowing what a cache miss is and being willing to learn mathematics where needed. You do not need to already understand language models or deep learning internals, but you should be willing to engage with the material at a technical level.

The book is designed so that each chapter builds knowledge required by the next. Reading it in order is strongly recommended. If you are already familiar with transformer architectures and want to jump to a specific topic, you can do so, but you will find that chapters reference concepts from earlier material.

Code examples throughout the book are written to be complete, compilable, and reproducible. Every example specifies its target environment, build commands, dependencies, and expected behavior. Educational examples prioritize clarity and correctness over raw performance; production-quality examples prioritize efficiency and robustness. The distinction is always made explicit.

All claims about performance, accuracy, or hardware behavior are either derived from first principles, supported by cited sources, or based on direct measurements described in the text. Where sources disagree or the literature is uncertain, the book presents the different positions and explains the reasoning behind each.

Technology Context

The field of edge LLM inference changes rapidly. New models, new quantization techniques, new runtimes, and new hardware are introduced constantly. This book uses the following context:

- The primary tooling ecosystem discussed is based on versions available from late 2024 through mid 2026. Specific version numbers are called out where they matter.
- llama.cpp is used as the reference implementation throughout because it is the most mature, widely deployed, and thoroughly documented system for CPU-first LLM inference. Its design decisions reveal the fundamental trade-offs more clearly than any higher-level framework.

- Models discussed are drawn from the families that dominate practical edge deployment: LLaMA and Llama-style models, Mistral, Gemma, Phi, and Qwen. These models are chosen because they have well-documented architectures, strong community support, and variants specifically designed for smaller-scale deployment.
- Hardware examples use devices that are actually available and commonly used for edge deployment: Raspberry Pi models, Rockchip RK3588-based boards, ARM Cortex-M microcontrollers, mobile SoCs. The results from one device are never presented as universal; every measurement is contextualized.

Where the state of the art has moved past what this book covers, the notes in the relevant chapters will point you toward the new material. The goal is to give you the foundational knowledge and the practical skills to evaluate and adopt new techniques as they emerge.

The Central Thesis

The central claim of this book is simple: running LLMs on tiny hardware is not magic and it is not impossible. It is a tractable systems engineering problem.

The gap between a model that runs on a datacenter GPU cluster and a model that runs on a single-board computer is not a fundamental barrier. It is a set of trade-offs: precision for size, latency for accuracy, complexity for performance. Understanding those trade-offs and making them consciously is what separates working edge deployments from failed experiments.

The techniques needed to bridge that gap : quantization, pruning, architectural modification, memory optimization, kernel tuning : have been developed and refined over years of research and engineering. They are not secret. They are documented in papers, in source code, in blog posts, in discussion forums. The problem is not a lack of information. The problem is that the information is scattered, inconsistent and often written for a different audience than the systems engineer trying to deploy a model on a real device tomorrow.

This book gathers that information into a single, coherent narrative. It connects the algorithms to the architectures to the hardware to the software stacks to the deployment patterns. It shows you how to calculate whether a model will fit, how to make it fit, how to make it run fast enough to be useful and how to keep it running reliably in production.

By the time you finish this book, you will be able to look at any language model and any constrained device and determine whether running one on the other is feasible, what it will cost in terms of memory, latency and accuracy, and what engineering work is required to make it work. You will have the knowledge and the skills to design and build such systems independently.

That is the promise of this book. The next chapter begins by examining the physical constraints that define what “tiny hardware” means and why they matter.

Chapter 1: The Physics of Tiny

Before you can make an LLM run on constrained hardware, you must understand what constrains it. The constraints are not abstract. They are physical properties of silicon, of memory cells, of interconnect wires, of power delivery networks. Every optimization technique in this book exists to work around some constraint listed in this chapter.

Why Memory Rules Everything

The most important insight for edge LLM inference is this: inference is a memory bandwidth problem, not a compute problem.

This statement is counterintuitive. We measure chips in teraflops. We brag about how many floating-point operations a GPU can perform per second. We design algorithms that maximize computational throughput. But for autoregressive LLM decoding, the arithmetic intensity : the ratio of operations performed to bytes moved from memory : is so low that compute units sit idle waiting for data.

Consider what happens during token generation. The model receives one token at a time and produces one token at a time. To generate each new token, it must:

1. Load all model weights from memory (billions of parameters)
2. Load the KV cache for all previous tokens (growing with context length)
3. Perform the forward pass computation (attention, feed-forward, normalization)
4. Write the output logits to memory

The computation in step 3 is significant in absolute terms but small relative to the data movement in steps 1, 2 and 4. For a 7 billion parameter model at FP16 precision, the weights alone occupy 14 gigabytes. For every single token generated, the system must read those 14 gigabytes from memory. On a device

with 25 gigabytes per second of memory bandwidth, that is 560 milliseconds just moving weights, before any computation happens.

This is why the community has shifted toward quantization so aggressively. The primary benefit of reducing precision from FP16 to INT4 is not that integer multiplication is faster. It is that you move four times less data through the memory bottleneck.

To quantify this claim, we use the roofline model. The roofline model plots performance against arithmetic intensity. A hardware platform has two limits: a peak memory bandwidth and a peak compute throughput. The intersection point is the ridge point. If your workload's arithmetic intensity falls below the ridge, you are memory-bound. Above it, you are compute-bound.

For a modern GPU like the NVIDIA H100, the ridge point is approximately 295 operations per byte, calculated as 989 teraflops divided by 3.35 terabytes per second. For LLM decoding at batch size 1, the arithmetic intensity falls as low as 1 to 2 operations per byte : hundreds of times below the ridge point. The GPU is spending most of its time waiting for memory.

On tiny hardware, the gap is even more extreme. A Raspberry Pi 5 with a quad-core Cortex-A76 has roughly 30 gigabytes per second of DDR4 bandwidth and a total CPU throughput on the order of tens of gigaflops of FP32 operations. The ratio is even worse than on a GPU. LLM inference on such a device is overwhelmingly memory-bandwidth-limited.

SRAM, DRAM, and Flash: Capacity, Speed, and Cost

Understanding memory types is essential for understanding why LLMs are hard to run on small devices. Different memory technologies make fundamentally different trade-offs between capacity, bandwidth, latency and power.

SRAM : Static RAM : is the fastest memory in the system. It has nanosecond access times, byte-level addressability, and no refresh overhead. It is also the most expensive and power-hungry per bit. SRAM is used for CPU caches: L1, L2, and sometimes L3. On a typical edge CPU, L1 cache might be 64 kilobytes per core. L2 might be 512 kilobytes per core. L3, if present, might be 2 megabytes shared. These numbers are tiny compared to LLM weights.

DRAM : Dynamic RAM : is slower than SRAM but much cheaper per bit. It requires periodic refresh because each cell is a capacitor that leaks charge.

Latency is in the tens of nanoseconds. Bandwidth is in the tens to hundreds of gigabytes per second depending on the interface. DRAM is the main memory in most edge devices: the gigabytes of RAM in your Raspberry Pi, your phone, your laptop. This is where LLM weights typically live during inference.

Flash : NAND flash memory : is much slower and has limited write endurance. Latency is in the hundreds of microseconds to milliseconds. Bandwidth is in the gigabytes per second for modern SSDs but much lower for embedded NAND. Flash has the advantage of being non-volatile and much cheaper per bit than DRAM. This is where model weights are stored when not in use, and increasingly, where they are streamed from during inference when DRAM is insufficient.

The implications for LLM inference are stark. A 7B parameter model at INT4 precision occupies approximately 3.5 gigabytes. On a device with 4 GB of DRAM, that leaves no room for activations, the KV cache, or the operating system. On a device with 1 GB of DRAM, it is impossible to hold the weights in DRAM at all. You must either use a smaller model, quantize more aggressively, or stream weights from flash.

Streaming from flash is feasible but dramatically slower. An NVMe SSD might deliver 2 gigabytes per second. Loading 3.5 gigabytes of weights takes 1.75 seconds. If you stream layer-by-layer, you can overlap computation and I/O to some extent, but the memory hierarchy has fundamentally changed the timing.

Table below summarizes typical characteristics:

Memory Type	Latency	Bandwidth (typical)	Cost (per GB)	Volatile
SRAM (cache)	< 10 ns	100+ GB/s	\$\$\$\$	Yes
DRAM	50-100 ns	20-500 GB/s	\$\$	Yes
NAND Flash	100 μs - 1 ms	0.5-4 GB/s	\$	No

These numbers vary by generation and implementation, but the orders of magnitude are stable. A cache hit is roughly 1000 times faster than a flash read. Understanding these gaps is essential for designing inference systems.

Memory Hierarchy and Cache Behavior

Modern processors use a hierarchy of caches to hide the latency of DRAM. When a core needs data, it first checks L1 cache. If not found, it checks L2, then L3, then DRAM, then storage. Each level up the hierarchy is faster but smaller.

For LLM inference, cache behavior is critical but subtle. The model weights are read sequentially and repeatedly during inference. If the weights fit in cache, inference can be dramatically faster. They rarely do. A 7B INT4 model is 3.5 GB. Even an aggressive L3 cache of 32 MB covers less than 1% of the weights.

What this means is that most weight accesses are DRAM accesses. On systems with separate caches per core, the same weight may be loaded independently by multiple cores, wasting bandwidth. Techniques like cache-aware tiling, prefetching and coordinated multi-core scheduling attempt to mitigate this, but the fundamental constraint remains: the weights are too large for the caches.

Activations : the intermediate values computed during the forward pass : have better locality. They are computed and consumed within a small window of time. If the activation tensors are small enough, they can fit in cache or even registers. Optimizing activation layout to maximize reuse is a key kernel optimization.

The KV cache is special. It grows with context length. Each token adds a key and value vector to the cache. On a system with limited DRAM, the KV cache can be the memory constraint before the weights are. Managing the KV cache efficiently : choosing the right data layout, quantizing it when possible, evicting stale entries : is a central concern for edge inference.

Compute vs. Bandwidth: Arithmetic Intensity

Arithmetic intensity is the most important single metric for understanding LLM inference performance. It is defined as the number of arithmetic operations performed per byte of memory accessed.

For transformer inference, we can decompose the arithmetic intensity into components. The forward pass of a transformer layer involves matrix multiplications in the attention and feed-forward blocks. A matrix multiply of

shape (M, K) times (K, N) performs $2MKN$ operations and reads $MK + KN$ bytes of weights plus M bytes of input. At large matrix sizes, this is roughly $2N$ operations per byte of weights : high arithmetic intensity.

During the prefill phase, when the model processes the entire input prompt in parallel, the matrix multiplies are large and the arithmetic intensity is high. Prefill can be compute-bound on some hardware.

During decode, when the model generates one token at a time, the picture changes dramatically. Each token generates a single column of output. The matrix multiply becomes a matrix-vector product: (M, K) times $(K, 1)$. The operations are $2MK$. The bytes read are $M \cdot K$ for weights plus K for input. At large M and K , this is approximately 2 operations per byte : very low arithmetic intensity.

This is the fundamental asymmetry. Prefill is compute-heavy and parallelizable. Decode is memory-heavy and sequential. Decode is also where the user waits. Most of the wall-clock time of an LLM interaction is spent in decode.

The practical implication: optimizing decode throughput is more important than optimizing prefill speed. And decode throughput is limited by memory bandwidth, not compute throughput. Increasing clock speed, adding cores, or using a fancier math library will not help if the bottleneck is moving weights through the memory bus. Quantization, which reduces the data size, is the primary lever for improving decode speed.

Power Budgets and Thermal Walls

On edge devices, power is not just a cost. It is a hard constraint.

Battery-powered devices have finite energy. A smartphone battery might hold 15 watt-hours. Sustained inference at 10 watts would drain that battery in 90 minutes. Active cooling has limits. Passive cooling has stricter limits.

Thermal constraints are often overlooked in early deployments. A device might achieve excellent throughput in a short benchmark, but sustained inference generates continuous heat. When the thermal limit is reached, the system throttles: reducing clock frequency, reducing voltage, reducing active cores. Throughput drops. Latency increases. In some cases, the system becomes unusable.

Research on sustained edge inference demonstrates the severity of this problem. In a 2026 benchmark of a quantized 1.5B model on four platforms, mobile devices showed dramatic thermal throttling. The iPhone 16 Pro lost 44% of its throughput within two iterations. The Samsung S24 Ultra terminated inference entirely after five iterations because the OS enforced a hard GPU frequency floor. By contrast, a dedicated edge NPU (Hailo-10H) sustained 6.9 tokens per second at under 2 watts with near-zero variance over the entire benchmark.

The lesson is clear: peak performance metrics are misleading for edge inference. You must measure sustained performance under realistic workloads with realistic thermal conditions. Active cooling, thermal paste, larger heatsinks and case design matter. Software techniques that reduce power consumption : quantization, selective layer offloading, dynamic frequency adjustment : are not optional optimizations. They are necessary for reliability.

The Numbers: How Much Memory and Bandwidth Does an LLM Actually Need

Let us work through concrete numbers for a realistic scenario. Consider the Llama 3.1 8B model at INT4 precision, deployed on a device with 8 GB of DDR4 RAM at 25 GB/s bandwidth.

Weight storage. The model has approximately 8 billion parameters. At 4 bits per parameter, the weights occupy 4 gigabytes. The tokenizer, special tokens, and metadata add a few megabytes. Total: approximately 4 GB for the model file.

Activations. During inference, each transformer layer computes activations. For an 8B model with hidden dimension approximately 4096, the activation memory per token per layer is on the order of tens of kilobytes. Across 32 layers, this is manageable: perhaps 1-2 megabytes for activations at any point during decode.

KV cache. This is the hidden cost. The KV cache stores key and value vectors for every token in the context. The formula is:

$$\text{KV cache bytes} = 2 \times \text{num_layers} \times \text{num_kv_heads} \times \text{context_length} \times \text{head_dim} \times \text{bytes_per_element}$$

For Llama 3.1 8B with GQA (8 KV heads, 32 query heads), hidden dimension 4096, head dimension 128, 32 layers, and context length 2048:

$$\text{KV cache} = 2 \times 32 \times 8 \times 2048 \times 128 \times 2 \text{ bytes (FP16)} = 2.68 \text{ GB}$$

At 4096 tokens of context, the KV cache grows to 5.37 GB. At 8192 tokens, it is 10.7 GB : larger than the weights.

Total memory budget. Weights (4 GB) + KV cache at 2048 context (2.68 GB) + activations (0.02 GB) + OS and runtime overhead (0.5 GB) = approximately 7.2 GB. This fits in 8 GB of RAM but barely. At 4096 context, it exceeds the available RAM entirely.

Bandwidth requirement. Each decode step reads all weights plus the KV cache plus activations. Roughly $4 + 2.68 + 0.02 = 6.7$ GB per token. At 25 GB/s bandwidth, the theoretical maximum decode rate is $25 / 6.7 = 3.7$ tokens per second. In practice, memory accesses are not perfectly contiguous, caches do not fully hit, and the OS and other tasks consume bandwidth. Realistic throughput is 2-3 tokens per second.

These numbers show why edge LLM inference is challenging. An 8B model is already pushing the limits of an 8 GB device. A 70B model is impossible on such hardware without aggressive quantization and/or weight streaming. The KV cache grows with context and can dominate the memory budget. Bandwidth limits throughput to rates that feel slow for interactive use.

The next chapters explain how to work within these constraints. But the constraints themselves do not go away. Every optimization technique trades something: precision, accuracy, flexibility, or implementation complexity. Understanding the trade-offs starts with understanding the numbers.

Chapter 2: Inside an LLM Inference Step

To optimize LLM inference on tiny hardware, you must know exactly what is being computed. This chapter explains the complete inference dataflow of a decoder-only transformer: from raw text input to generated token output. We will examine each component, its computational cost, its memory requirements and its properties relevant to optimization.

This chapter assumes familiarity with basic linear algebra (matrix multiplication, vector addition, norms) but does not assume prior knowledge of transformers. Every concept is defined rigorously.

Tokenization: Strings to Integers

A language model does not operate on characters or words. It operates on integers called tokens. Tokenization is the process of converting a string of text into a sequence of token IDs, and detokenization is the reverse.

Tokenizers use vocabulary tables. The vocabulary is a mapping from token strings to integer IDs, typically ranging from 0 to approximately 32000 for modern models. Common words map to single tokens: “the”, “and”, “is”. Rare words or multi-character sequences map to longer token strings. Uncommon words may be split into subword tokens: “unbelievable” might become [“un”, “believ”, “able”].

The most common tokenizer algorithms for LLMs are:

Byte Pair Encoding (BPE): Starts with a base vocabulary of individual bytes. Iteratively merges the most frequent adjacent token pairs into single tokens until the target vocabulary size is reached. This produces a vocabulary optimized for the training distribution. Models like GPT-2, LLaMA and Mistral use BPE or variants.

SentencePiece: A subword tokenization method that operates on pretokenized text. It supports BPE, unigram and word-piece algorithms. Models like T5 and some variants use SentencePiece.

Tokenization matters for edge deployment for several reasons:

1. **Vocabulary size determines embedding memory.** A vocabulary of 32000 tokens with embedding dimension 4096 requires $32000 \times 4096 \times 4$ bytes = 512 megabytes for the embedding table at FP32 precision. Reducing vocabulary size reduces this memory.
2. **Tokenization efficiency affects decode speed.** Each generated token ID must be decoded to text. Some tokenizers are computationally expensive, especially when handling UTF-8 and special cases. On a tiny device, tokenizer overhead can be significant relative to model inference time.
3. **Token count affects KV cache size.** The KV cache grows with the number of tokens, not characters. A verbose tokenizer that splits text into many small tokens will increase cache requirements.

A minimal BPE tokenizer implementation might look like this:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define VOCAB_SIZE 1000
6  #define MAX_TOKEN_LEN 100
7  #define MAX_SEQUENCE 1024
8
9  typedef struct {
10     char token[MAX_TOKEN_LEN];
11     int rank;
12     float score;
13 } vocab_entry_t;
14
15 static vocab_entry_t vocab[VOCAB_SIZE];
16
17 int tokenize(const char *input, int *tokens, int max_tokens) {
18     // Simplified: convert input to byte sequence, then apply merges
19     // This is educational pseudocode, not production-quality
20     int len = strlen(input);
21     int num_tokens = 0;
22
23     // Initial encoding: each byte becomes a token
24     for (int i = 0; i < len && num_tokens < max_tokens - 1; i++) {
25         tokens[num_tokens++] = (unsigned char)input[i] + 256; // offset for
26         ↪ single bytes
27     }
28     tokens[num_tokens] = 0; // end marker

```

```

29     // Apply merges repeatedly (simplified)
30     // In real BPE, you maintain a priority queue of merge candidates
31     // and apply the highest-priority merge until vocabulary exhausted
32
33     return num_tokens;
34 }
35
36 int main() {
37     int tokens[MAX_SEQUENCE];
38     int n = tokenize("hello world", tokens, MAX_SEQUENCE);
39     printf("Tokens (%d): ", n);
40     for (int i = 0; i < n; i++) {
41         printf("%d ", tokens[i]);
42     }
43     printf("\n");
44     return 0;
45 }

```

Production tokenizers like the one in llama.cpp or SentencePiece are more sophisticated, handling Unicode normalization, special tokens, merge priorities and caching. For edge deployment, you typically use a precompiled vocabulary from the model repository.

The Embedding Layer and Positional Encoding

The first computational layer of a transformer converts token IDs into dense vectors called embeddings. The embedding layer is a learned lookup table of shape (vocab_size, hidden_dim).

For token ID t , the embedding is simply `embed[t]`, a vector of dimension `hidden_dim`. For Llama 3.1 8B, the hidden dimension is 4096. Each token is mapped to a 4096-dimensional vector. This is a single memory read per token: very cheap in compute, but the embedding table itself is large.

Embeddings alone do not encode position. The transformer architecture is permutation-invariant: if you reorder the input tokens, the output embeddings are reordered but unchanged in value. To give the model position information, positional encodings are added.

Absolute positional encodings add a fixed or learned vector to each embedding, depending on the token's position in the sequence. The original Transformer paper used sinusoidal functions:

$\text{position_encoding}[\text{pos}, 2i] = \sin(\text{pos} / 10000^{(2i/d_model)})$ $\text{position_encoding}[\text{pos}, 2i+1] = \cos(\text{pos} / 10000^{(2i/d_model)})$

Rotary Position Embeddings (RoPE) are used by LLaMA and many modern models. RoPE encodes position through rotation matrices applied to the embedding. Each dimension pair of the embedding is rotated by an angle proportional to the token's position. The key property of RoPE is that the relative position between any two tokens can be computed directly from their individual rotations, enabling efficient attention computation.

RoPE is computationally slightly more expensive than sinusoidal encodings but provides better extrapolation to longer sequences. On edge hardware, the extra cost is negligible compared to the transformer blocks.

The embedding layer output is a tensor of shape (sequence_length, hidden_dim). For a sequence of 1024 tokens and hidden dimension 4096, this is $1024 \times 4096 \times 4 = 16$ megabytes at FP32 precision. This memory is reused through the forward pass and released after inference.

Transformer Blocks: Self-Attention and Feed-Forward

The core of a transformer is a stack of identical blocks. Each block performs two main operations: multi-head self-attention and a feed-forward network (FFN). Modern transformers have 30-40 blocks.

Self-attention computes how each token should “attend” to every other token in the sequence. For each token i , attention produces a weighted combination of all token values, where the weights depend on the similarity between token i 's query and each token j 's key.

The computation proceeds in steps:

1. **Linear projections.** For each token, compute query (Q), key (K) and value (V) vectors by multiplying the embedding by learned weight matrices:

- $Q = X \cdot W_Q$
- $K = X \cdot W_K$
- $V = X \cdot W_V$ where X is the input tensor of shape (seq_len, hidden_dim), and each weight matrix is (hidden_dim, hidden_dim).

2. **Scaled dot-product attention.** Compute attention scores:

- $\text{scores} = (Q \cdot K^T) / \sqrt{d_k}$ where d_k is the key dimension. The scaling prevents the softmax from saturating for large dot products.
3. **Softmax.** Apply softmax over the key dimension to get attention weights:
 - $\text{weights} = \text{softmax}(\text{scores})$
 4. **Weighted combination.** Compute the output:
 - $\text{output} = \text{weights} \cdot V$
 5. **Output projection.** Multiply by a learned weight matrix:
 - $\text{output} = \text{output} \cdot W_O$

Each step involves matrix operations. The attention scores matrix has shape $(\text{seq_len}, \text{seq_len})$. For a sequence of 4096 tokens, this is 16 million entries : 64 megabytes at FP32. This quadratic scaling is why long context is expensive.

Feed-forward network (FFN) is applied independently to each token. A standard FFN has two linear layers with a non-linearity between them:

$$\text{FFN}(x) = \max(0, x \cdot W_1 + b_1) \cdot W_2 + b_2$$

where W_1 projects from hidden_dim to intermediate_dim (typically $4 \times \text{hidden_dim}$), and W_2 projects back from intermediate_dim to hidden_dim .

The FFN is simpler than attention because it does not mix information across tokens. It is also typically where the most parameters are stored. For a model with hidden_dim 4096 and intermediate_dim 14336, each FFN layer has:

- W_1 : $4096 \times 14336 \times 4 \text{ bytes} = 234 \text{ MB (FP32)}$
- W_2 : $14336 \times 4096 \times 4 \text{ bytes} = 234 \text{ MB (FP32)}$

Across 32 layers, the FFN weights alone are approximately 15 gigabytes at FP32 precision.

Residual connections add the input to the output of each sub-layer:

- After attention: $x = x + \text{attention}(x)$
- After FFN: $x = x + \text{FFN}(x)$

This allows gradients to flow directly through the network and helps train deeper models. During inference, residual addition is cheap: one vector addition per token per sub-layer.

Layer normalization is applied before or after each sub-layer (modern models use pre-normalization):

- $x = \text{LayerNorm}(x) = \frac{\mu \cdot (x - \mu)}{\sigma + \epsilon}$

where μ and σ are the mean and standard deviation across the feature dimension, and ϵ and μ are learned parameters.

Layer normalization is numerically critical. It stabilizes training and helps inference behave correctly. On edge hardware, it requires computing mean and variance across the hidden dimension for each token, followed by a division and multiplication. This is memory-bandwidth-heavy because each element of the input is read and written multiple times.

Multi-Head and Grouped-Query Attention

Standard self-attention uses a single “head”: one set of Q, K, V, O projections. Multi-head attention splits the hidden dimension into multiple heads, each with its own projections:

- $\text{hidden_dim} = \text{num_heads} \times \text{head_dim}$
- Each head independently computes Q_i, K_i, V_i
- Attention is computed within each head
- Results are concatenated and projected

Multi-head attention allows the model to attend in multiple ways simultaneously: one head might capture syntactic relationships, another semantic relationships, another positional relationships. The number of heads is typically 32-64 for modern models.

However, multi-head attention has a cost. Each head requires its own key and value vectors in the KV cache. With 32 heads and head dimension 128, each token requires $32 \times 128 \times 4 = 16$ kilobytes for keys and values at FP32. Across 32 layers, that is 512 kilobytes per token.

Grouped-Query Attention (GQA) reduces this cost by sharing keys and values across multiple query heads. Instead of `num_heads` key-value heads, GQA uses `num_kv_heads` where `num_kv_heads < num_heads`. Each group of query heads shares one set of key-value heads.

For Llama 3.1 8B: `num_heads = 32`, `num_kv_heads = 8`, so each KV head is shared by 4 query heads. The KV cache is reduced by a factor of 4 compared to full multi-head attention.

Multi-Query Attention (MQA) is the extreme case where all query heads share a single KV head. MQA minimizes the KV cache but can degrade model quality because all queries must attend to the same keys and values.

GQA is the preferred choice for edge deployment. It significantly reduces KV cache memory with minimal quality impact. When selecting models for edge use, preferring GQA over full MHA is one of the easiest and most effective optimizations.

Layer Normalization and Residual Connections

We mentioned normalization and residuals above, but they deserve closer examination because their implementation matters for performance.

RMSNorm is used by LLaMA and many modern models instead of standard LayerNorm. RMSNorm omits the mean centering:

$$\bullet \text{ RMSNorm}(x) = \frac{1}{\sigma} \cdot x / \sqrt{\text{mean}(x^2) + \epsilon}$$

This is simpler to compute (one fewer pass over the data) and empirically just as effective. The difference in compute is small but matters in a tight inner loop.

Residual connections are trivial during inference but have implications for memory management. Each block's input must be preserved so it can be added to the block's output. In a naive implementation, this means allocating memory for all inputs and all outputs simultaneously. Clever implementations reuse memory in-place to minimize allocation.

On tiny hardware, memory reuse is not optional. A 32-layer transformer with hidden dimension 4096 has activation memory of $32 \times 4096 \times 4$ bytes = 512 kilobytes per token. For a context of 2048 tokens, that is approximately 1

gigabyte of activation memory if everything is kept simultaneously. In reality, activations can be reused as they flow through the network, but the memory manager must be designed carefully.

Autoregressive Decoding and the KV Cache

The KV cache is the central optimization for autoregressive generation. Without it, generating a sequence of N tokens requires recomputing attention over all previous tokens at each step, giving $O(N^2)$ total computation. With the KV cache, each new token's attention computation reuses previously computed keys and values, giving $O(N)$ total computation.

The mechanism is simple:

1. During prefill, process the entire input prompt. For each token, compute Q , K , V . Store K and V in the cache. Use Q , K , V to compute attention and propagate through the network.
2. During decode, for each new token:
 - Compute Q from the input embedding
 - Read all cached K and V from memory
 - Compute attention between new Q and cached K , V
 - Generate the output token
 - Compute K and V for the new token and append to cache

The cache grows with context. This is why long conversations and long context windows are memory-intensive.

A minimal KV cache structure:

```

1  #include <stdlib.h>
2  #include <string.h>
3
4  typedef struct {
5      float *keys;    // Shape: [num_layers][num_kv_heads][max_seq][head_dim]
6      float *values;  // Shape: [num_layers][num_kv_heads][max_seq][head_dim]
7      int max_seq;
8      int num_layers;
9      int num_kv_heads;
10     int head_dim;
11     int current_seq_len;
12 } kv_cache_t;
13
14 void kv_cache_init(kv_cache_t *cache, int max_seq, int num_layers,
15                   int num_kv_heads, int head_dim) {
16     cache->max_seq = max_seq;
17     cache->num_layers = num_layers;
18     cache->num_kv_heads = num_kv_heads;
19     cache->head_dim = head_dim;
20     cache->current_seq_len = 0;
21
22     size_t kv_size = num_layers * num_kv_heads * max_seq * head_dim *
23     ↪ sizeof(float);
24     cache->keys = (float*)malloc(kv_size);
25     cache->values = (float*)malloc(kv_size);
26
27     if (!cache->keys || !cache->values) {
28         // Handle allocation failure
29         exit(1);
30     }
31 }
32 void kv_cache_store(kv_cache_t *cache, int layer, const float *k, const float
33 ↪ *v) {
34     int offset = (layer * cache->num_kv_heads * cache->max_seq *
35     ↪ cache->head_dim)
36     + (cache->current_seq_len * cache->num_kv_heads *
37     ↪ cache->head_dim);
38
39     memcpy(cache->keys + offset, k, cache->num_kv_heads * cache->head_dim *
40     ↪ sizeof(float));
41     memcpy(cache->values + offset, v, cache->num_kv_heads * cache->head_dim *
42     ↪ sizeof(float));
43 }

```

In production code, the KV cache is optimized for memory layout (contiguous allocation, cache-friendly ordering), quantization (INT8 or INT4 storage) and eviction (when context exceeds available memory). We will cover these optimizations in later chapters.

The Complete Inference Dataflow

Let us trace through a complete inference step for a decoder-only transformer:

Prefill phase:

1. Tokenize the input prompt into a sequence of token IDs.
2. Look up embeddings for each token ID from the embedding table.
3. Add positional encodings to each embedding.
4. For each transformer block in sequence:
 - a. Apply layer normalization.
 - b. Project to Q, K, V using attention weight matrices.
 - c. Store K and V in the KV cache.
 - d. Compute scaled dot-product attention between Q and all cached K.
 - e. Apply softmax to get attention weights.
 - f. Compute weighted sum of cached V.
 - g. Project attention output using W_O .
 - h. Add residual connection from block input.
 - i. Apply layer normalization.
 - j. Apply FFN (two linear layers with non-linearity).
 - k. Add residual connection from attention output.
5. Apply final layer normalization.
6. Project to logits using the output projection (often tied to embedding table).
7. Select the next token from the logits distribution using a sampling strategy.

Decode phase (repeated for each generated token):

1. Embed the newly selected token.
2. Add positional encoding.
3. For each transformer block:
 - a. Layer normalization.
 - b. Project to Q (K and V also computed but not used for attention on the current step).
 - c. Store new K and V in cache.
 - d. Compute attention between current Q and all cached K.
 - e. Softmax.
 - f. Weighted sum of cached V.
 - g. Project attention output.
 - h. Residual add.
 - i. Layer normalization.
 - j. FFN.
 - k. Residual add.
4. Final normalization.
5. Project to logits.
6. Sample next token.
7. If stop condition met (EOS token, max length), stop. Otherwise, repeat.

The total inference cost is dominated by:

- Prefill: attention computation over the full prompt (quadratic in prompt length)
- Decode: reading weights and KV cache for each generated token (linear in context length)

For edge deployment, decode is the critical path because it determines interactive responsiveness. Prefill is important for initial response time but less impactful for overall user experience.

The next chapter uses this understanding of inference to show you how to calculate whether a specific model can run on a specific device.

Chapter 3: Measuring Feasibility

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

Parameter Storage: Bits, Bytes, and Formats

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

Activation Memory: What You Cannot See Coming

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

KV Cache: The Hidden RAM Killer

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

Arithmetic Intensity and Compute Requirements

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

Bandwidth Calculations: The True Bottleneck

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

The Feasibility Checklist

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

Chapter 4: Model Architectures for Constrained Inference

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

Dense Decoder-Only Transformers: The Baseline

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

Mixture of Experts: Selective Activation

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

Recurrent and Hybrid Architectures

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

Compact Language Models: Design for Small

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

Architecture Choices That Matter for Embedded: Context Window, Heads, FFN Width

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

Selecting a Model Family for Your Target

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

Chapter 5: Quantization Fundamentals

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

Why Quantization Works: Redundancy in Floating-Point Weights

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

Symmetric vs. Asymmetric Quantization: Scales and Zero-Points

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

Per-Tensor vs. Per-Channel vs. Group-wise

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

Calibration Strategies

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

The Quantization-Error-Accuracy Tradeoff

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

Quantization Formats: FP16, BF16, INT8, and Lower

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

Chapter 6: Weight-Only Quantization

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

How Weight-Only Quantization Works

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

Dequantization on the Fly: The Compute Cost

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

Packing Schemes: 4-bit, 5-bit, 6-bit Weights

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

Weight-Only Quantization and Activation Outliers

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

Implementing Weight-Only Matrix Multiplication

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

When Weight-Only Is Enough (and When It Is Not)

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

Chapter 7: Activation Quantization and Mixed Precision

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

Why Activations Are Harder to Quantize

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

Activation Outliers and Their Impact

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

Mixed Precision Strategies

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

Quantization-Aware Training

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

Post-Training Quantization with Activation Calibration

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

INT4/INT3 Activation Quantization: The Cutting Edge

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

Chapter 8: Sparsity, Pruning and Structured Compression

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

Unstructured vs. Structured Sparsity

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

Pruning Methods and Magnitude Thresholding

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

Compressed Sparse Row and Other Formats

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

The Problem of Sparsity on Small CPUs

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

Block-wise and Channel-wise Pruning

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

Low-Rank Factorization

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

When to Use Pruning and Sparsity

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

Chapter 9: Knowledge Distillation and Architecture Modification

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

Distillation: Making Small Models Smarter

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

Architecture Surgery: Head Merging, Layer Thinning

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

Vocabulary Reduction and Tokenizer Optimization

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

Context Window Management and Compression

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

Speculative Decoding: Drafting and Verifying

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

When to Modify the Model vs. When to Optimize the Runtime

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

Chapter 10: KV Cache Engineering

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

KV Cache Layout: Memory Mapping the Context

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

KV Cache Quantization and Compression

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

Eviction Policies and Sliding Windows

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

Activation Checkpointing and Recomputation

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

Streaming and Weight-Reuse Strategies

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

KV Cache for Long Context on Tiny Memory

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

Chapter 11: CPU, SIMD and Vector Optimization

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

Vector Units: NEON, SSE, AVX, and RISC-V V-Extensions

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

Writing Vectorized Matmul Kernels

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

Packing and Unpacking for SIMD

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

Instruction-Level Parallelism and Loop Unrolling

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

Memory Access Patterns and Cache Locality

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

Tuning for Specific CPU Microarchitectures

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

Chapter 12: GPU, NPU and DSP Acceleration

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

Mobile GPUs: Mali, Adreno, PowerVR

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

Neural Processing Units: NPU Architectures and Constraints

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

DSP Acceleration for Embedded Systems

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

Tensor Cores and Matrix Multiply Accelerators

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

Accelerator APIs and Vendor SDKs

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

When Accelerators Help (and When They Get in the Way)

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

Chapter 13: Inference Runtimes and Software Architecture

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

The Runtime Stack: From Model Files to Generated Tokens

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

Graph Representations and IR Formats

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

Lightweight Transformer Runtimes

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

Vendor-Specific Runtimes vs. Portable Solutions

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

Kernel Selection and Code Generation

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

Designing Your Own Minimal Inference Runtime

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

Chapter 14: Model Conversion and Deployment Pipelines

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

Exporting Models: ONNX, GGUF, safetensors, and Others

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

Tokenizer Packaging and Conversion

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

Quantization Pipelines: Tools and Techniques

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

Cross-Compilation and Build Systems

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

Binary Size, Flash Constraints, and OTA Updates

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

Model Versioning and Update Strategies

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

Chapter 15: Performance Engineering and Benchmarking

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

Metrics That Matter: TTFT, ITL, Throughput, and Energy

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

Benchmarking Methodology: Warm-Up, Sampling, Reproducibility

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

Profiling: Finding the Real Bottleneck

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

Compute-Bound vs. Memory-Bound vs. Synchronization-Bound

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

Compiler Optimization and Its Surprises

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

Performance Regression and Change Management

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

Chapter 16: Power, Energy and Thermal Management

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

Measuring Power and Energy in Real Deployments

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

Energy per Token: A Practical Metric

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

Dynamic Voltage and Frequency Scaling

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

Thermal Throttling and Sustained Performance

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

Power-Optimized Scheduling and Batching

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

Chapter 17: Debugging and Failure Analysis

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

Incorrect Outputs: Numerical Error vs. Model Behavior

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

Memory Exhaustion and Fragmentation

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

Stack Overflow and Heap Corruption

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

Quantization Bugs: Zero-Point and Scale Errors

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

Hardware-Specific Issues: Alignment, Endianness, FPU Modes

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

Silent Accuracy Degradation and Regression Testing

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

Chapter 18: Security and Privacy

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

The Security Model of Local Inference

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

Model Extraction and Protection

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

Supply Chain Risks: Malicious Models and Tooling

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

Input and Output Security: Prompt and Data Handling

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

Secure OTA Updates and Integrity Verification

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

Privacy Advantages and Their Limits

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

Chapter 19: Case Study : An LLM in 256 MB of RAM

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

The Hardware and Its Constraints

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

Choosing and Preparing the Model

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

Memory Budget: Weights, KV Cache, Runtime Overhead

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

Implementation Decisions and Trade-offs

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

Benchmarking Results

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

Lessons Learned

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

Chapter 20: Case Study : Offline Edge Assistant on an ARM SBC

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

System Requirements and Architecture

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

Model and Tokenizer Selection

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

Building the Inference Engine

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

Application Integration: Input, Output, Streaming

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

Robustness: Error Handling, Watchdogs, Recovery

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

Production Readiness Checklist

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

Chapter 21: Case Study : Pushing the Limits on MCU-Class Hardware

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

MCU Constraints: Kilobytes of RAM, Megahertz Clocks

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

Micro-Models: Can Transformers Run on an MCU?

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

Extreme Quantization and Architecture Reduction

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

Alternative Approaches: Recurrent and State-Space Models

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

Measured Performance and Practical Viability

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

The Real Limit: Where LLMs Cannot Go

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

Chapter 22: Production Deployment Patterns

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

Firmware Integration and Boot-Time Constraints

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

Multi-Model and Fallback Strategies

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

Handling Concurrent Requests on Constrained Systems

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

Observability: Logging, Metrics, Telemetry

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

A/B Testing and Model Rollout at the Edge

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

Maintenance and Lifecycle Management

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

Chapter 23: Future Directions

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

Architectural Innovations: Mamba, RWKV, and State-Space Models

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

Hardware Roadmaps: What Next-Gen Edge Silicon Will Bring

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

Algorithmic Advances: Better Quantization, Sparsity, Distillation

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

Compiler and Runtime Innovations

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

The Economics of Edge Inference

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

The Next Frontier: What Comes After Tiny LLMs

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

Conclusion: The Engineer's Playbook

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

The Decision Tree: Choosing Your Optimization Strategy

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

A Checklist for Edge LLM Deployment

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

Accepting Trade-offs: No Solution Is Free

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

Final Thoughts: The Promise and Limits of Edge Intelligence

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

References

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