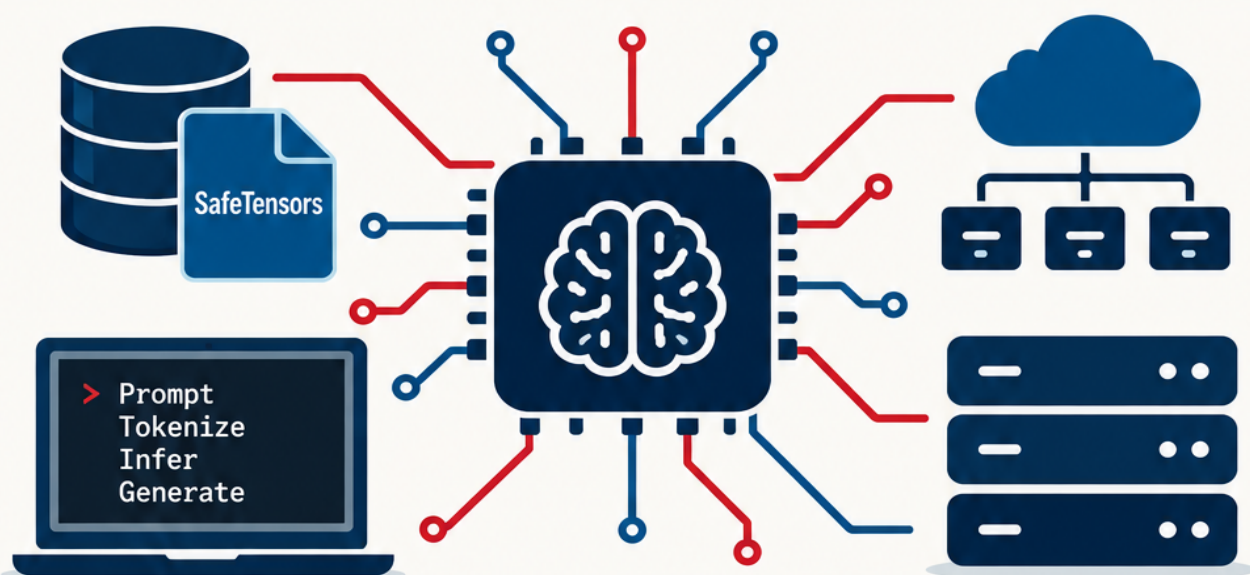


BUILDING **HIGH-PERFORMANCE** **LLM** **INFERENCE ENGINES** **IN C**

From Transformer Fundamentals to a Complete
SafeTensors Inference Runtime



JULIAN VOGEL

Building High-Performance LLM Inference Engines in C

From Transformer Fundamentals to a Complete
SafeTensors Inference Runtime

Steve Publications

This book is available at

<https://leanpub.com/buildinghigh-performancellminferenceenginesinc>

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

From Transformer Fundamentals to a Complete SafeTensors Inference Runtime	1
Introduction: How LLMs Actually Generate Text	2
Chapter 1: How LLMs Think: The Transformer Inference Journey	5
The Inference Journey from Prompt to Response	5
Why Inference Is Different from Training	6
The Transformer Blueprint at a Glance	7
What This Book Builds	9
A First Look: Loading, Running, and Generating	10
Chapter 2: Tokens, Vocabularies, and the Language of Models	12
What a Token Really Is	12
Byte Pair Encoding and Subword Tokenization	13
Loading and Interpreting Tokenizer Files	14
Implementing Token Encoding in C	16
Implementing Token Decoding and Special Tokens	21
Chapter 3: Tensors, Shapes, and Memory Layouts	24
The Tensor as a Multi-Dimensional Array	24
Row-Major vs Column-Major and Why It Matters	24
Shapes, Strides, and Offsets	24
Float16, BFloat16, and Float32 Representations	24
Contiguous vs Strided vs Packed Tensors	24
Chapter 4: Building the Tensor Core Library	26
Tensor Allocation and Lifetime Management	26
Views, Reshaping, and Slicing Without Copies	26
Element-Wise Operations and Broadcasting	26
Matrix Multiplication from First Principles	26

Reductions, Sums, and Normalization Primitives	26
Chapter 5: The Transformer Layer in Depth	28
Layer Normalization and RMSNorm	28
Multi-Head Self-Attention Mechanism	28
Feed-Forward Networks and Activation Functions	28
Residual Connections and Gate Linear Units	28
Putting It All Together: One Transformer Block	28
Chapter 6: Positional Encodings and RoPE	30
Why Positional Information Matters	30
Learnable vs Absolute vs Rotary Encodings	30
The Mathematics of RoPE	30
Implementing RoPE in C	30
RoPE Scaling and Long-Context Strategies	30
Chapter 7: Softmax, Logits, and Sampling	32
From Hidden States to Logits	32
Numerically Stable Softmax	32
Temperature Scaling and Exploration	32
Top-K Filtering and Nucleus Sampling	32
Deterministic and Reproducible Generation	32
Chapter 8: The KV Cache and Autoregressive Generation	34
The Autoregressive Generation Loop	34
Key-Value Caching: The Core Optimization	34
KV Cache Layouts and Memory Planning	34
Prefill vs Decode Phases	34
Managing Cache Growth and Limits	34
Chapter 9: The SafeTensors File Format	36
Why SafeTensors and What Problem It Solves	36
Header Structure and Serialization	36
Tensor Descriptors, Data Types, and Offsets	36
Endianness, Alignment, and File Boundaries	36
Supported Data Types and Shape Encoding	36
Chapter 10: Implementing the SafeTensors Loader in C	38
Header Reading and Length Prefix Parsing	38
JSON Metadata Parsing in C	38

CONTENTS

Tensor Descriptor Extraction and Validation	38
Mapping and Reading Tensor Data Blocks	38
Error Handling, Boundaries, and Integrity Checks	38
Chapter 11: Model Configuration and Weight Mapping	40
The config.json: Architecture Parameters	40
Weight Naming Conventions and Patterns	40
Mapping Transformer Layer Weights	40
Handling Embeddings and Output Projections	40
Validation: Checking Loaded Weight Shapes	40
Chapter 12: Building the Model Forward Pass	42
Token Embedding Lookup	42
The Layer Loop: Attention, FFN, and Residuals	42
Positional Encoding at Each Position	42
Final Layer Normalization	42
Computing Output Logits	42
The Complete Attention Implementation	43
Putting It All Together: The Forward Pass	43
Chapter 13: End-to-End Inference: From Prompt to Text	44
Runtime State and Context Management	44
The Generation Loop Implementation	44
Integration: Tokenizer, Model, and Sampler	44
Streaming Output and Callbacks	44
Command-Line Interface and First Run	44
Chapter 14: Verifying Correctness Before Optimizing	46
Reference Implementations and Ground Truth	46
Comparing Embeddings and Intermediate Activations	46
Logit Comparison and Numerical Tolerance	46
Generating Test Cases from Known Outputs	46
Assertions, Sanitizers, and Debugging Techniques	46
Chapter 15: Performance Profiling and Bottleneck Analysis	48
Latency, Throughput, and Tokens Per Second	48
Profiling Tools: Perf, Valgrind, and Flame Graphs	48
Compute-Bound vs Memory-Bound Analysis	48
Cache Misses, Branch Mispredictions, and Latency	48
Identifying the Critical Path	48

Chapter 16: SIMD and Vectorization 50
 Understanding SIMD Architectures: AVX2, AVX-512, NEON 50
 Autovectorization vs Hand-Written Intrinsics 50
 Vectorized Element-Wise Operations 50
 Vectorized Matrix Multiplication 50
 Cross-Platform SIMD Portability 50

Chapter 17: Memory and Threading Optimizations 52
 Cache Locality and Data Layout Optimization 52
 Loop Tiling and Blocking for MatMul 52
 Multithreading with OpenMP and Native Threads 52
 False Sharing, Synchronization, and Contention 52
 NUMA Awareness and Large Memory Models 52
 Memory Bandwidth, Compute-Bound vs Memory-Bound Workloads . 53

Chapter 18: Advanced Optimization Techniques 54
 Weight Quantization and Low-Bit Representations 54
 Packed Weights and Efficient Memory Use 54
 Batching and Continuous Batching 54
 Speculative Decoding and Draft Models 54
 Asynchronous Execution and Pipeline Parallelism 54
 Model-Size-Dependent Optimization Strategies 55

Conclusion: What We Built and Why It Matters 56

References 57

From Transformer Fundamentals to a Complete SafeTensors Inference Runtime

This book takes you from the mathematical foundations of transformer inference all the way to a complete, working large language model inference engine written in portable C. You will understand exactly how transformers generate text one token at a time, how weights are stored and loaded from SafeTensors format, how KV caching makes inference practical, and how to optimize every component for performance. By the final chapter you will have a genuinely executable program that loads a real TinyLlama-1.1B model from HuggingFace, tokenizes your prompt, runs the full transformer forward pass on your CPU, maintains the KV cache, and streams generated text to your terminal. No pseudocode, no placeholders, no exercises left for you to implement on your own: every line of code is provided, explained, and tested as a complete end-to-end system.

Introduction: How LLMs Actually Generate Text

When you type a prompt into a chat interface and receive a response, a massive computation is happening behind the scenes. The model does not think, reason, or understand in any human sense. Instead it executes a precise sequence of linear algebra operations on arrays of floating-point numbers. The same mathematical operations run whether the model is answering a question about Python programming or writing a poem about rain. Understanding inference means understanding these operations, the data structures that hold them together, and the engineering trade-offs that determine whether a response appears in half a second or thirty seconds.

Most developers who use large language models interact with them through high-level APIs: you send a string and receive a string. Libraries like HuggingFace Transformers abstract away the complexity with elegant Python interfaces. Production systems like vLLM, TensorRT-LLM, and llama.cpp hide the complexity under layers of optimization and abstraction. This abstraction is necessary and valuable, but it also creates a black box. When inference is slow, when memory runs out, when you need to run a model on unusual hardware, when you want to understand why one model behaves differently from another, the black box becomes a barrier.

This book opens the box. We will build an inference engine from the ground up in C, one component at a time. By the end, you will know exactly what happens to your prompt from the moment it enters the system until the final token appears on your screen. You will be able to read profiler output and tell whether the bottleneck is memory bandwidth or compute. You will understand why quantization reduces accuracy in some dimensions but not others. You will know why KV caching is not just an optimization but a fundamental necessity for autoregressive generation.

The implementation we build targets a specific, real model: TinyLlama-1.1B, a 1.1 billion parameter model trained on 3 trillion tokens using the exact Llama-2 architecture. It is small enough to run on most modern machines yet large enough to produce coherent, useful text. We load it from the standard

SafeTensors format used by HuggingFace, which means our implementation is compatible with thousands of real models. Every tensor name, every dimension, every data type corresponds to actual model weights you can download and run.

We will not stop at a toy implementation that demonstrates concepts in isolation. The code we write in Chapter 3 for matrix multiplication is the same code that processes attention in Chapter 12. The SafeTensors loader we implement in Chapter 10 is the same loader that reads the model weights used for generation in Chapter 13. The KV cache we design in Chapter 8 is the same structure that grows as we generate each token. This is not a collection of examples; it is a single, evolving system that you can build, run, measure, and extend.

The journey is organized into three phases. First we establish the foundation: what a transformer is, how it processes text, what tokens and tensors actually are, and how the mathematics of attention, normalization, and positional encoding work. This is the understanding phase, where we build the mental model that makes the rest of the book comprehensible. Second we implement: we write C code for tensor operations, for loading model weights, for running the transformer forward pass, for managing the KV cache, and for sampling tokens. This is the construction phase, where understanding becomes a working system. Third we optimize: we profile the system, identify bottlenecks, apply SIMD vectorization, add multithreading, explore quantization and advanced techniques used in production engines. This is the refinement phase, where a correct implementation becomes a performant one.

To get the most from this book you should be comfortable with C programming, have basic familiarity with linear algebra concepts like vectors, matrices, and matrix multiplication, and be willing to engage with numerical details like floating-point representation and memory layout. You do not need deep expertise in machine learning or prior experience with neural networks. Every mathematical concept is introduced when it becomes necessary and explained in terms that connect directly to the code you are writing.

By the final chapter you will have something tangible: a compiled C program on your machine that loads a real neural network model from a file, processes your input, and generates text. More importantly you will have something less tangible but more valuable: a deep, practical understanding of how large language models actually work at the systems level, and the ability to reason about performance, memory, and correctness when building or deploying

inference systems of your own.

Chapter 1: How LLMs Think: The Transformer Inference Journey

The Inference Journey from Prompt to Response

Let us trace what happens when you ask a language model to answer a question. Consider the prompt: “What is the capital of France?” This sequence of characters travels through a series of transformations before the model produces its response. Each step is deterministic, each operation is a well-defined computation, and the entire process can be understood by following the data.

The journey begins with tokenization. The raw string “What is the capital of France?” is not fed into the model as text. Instead the tokenizer breaks it into tokens, which are integer indices into a vocabulary of about thirty thousand subword units. For our example the tokenizer might produce something like: [908, 478, 464, 2787, 338, 2690, 30]. These integers are the actual input to the neural network. The vocabulary size is a fixed property of the model; TinyLlama-1.1B uses a vocabulary of 32000 tokens, so every input token is an integer between 0 and 31999.

Next comes the embedding layer. Each token integer is looked up in a massive weight matrix of shape [32000, 2048], mapping every possible token to a 2048-dimensional vector of floating-point numbers. These vectors are not random; they encode semantic information learned during training. The token for “capital” has a vector representation that places it close to vectors for “city” and “government” in the geometric space of the embedding. After this lookup our sequence of 7 tokens becomes a matrix of shape [7, 2048]: seven rows, each a 2048-dimensional vector.

This matrix then flows through a stack of 22 identical transformer layers. Each layer transforms the vectors through two main operations: self-attention and a feed-forward network. Self-attention allows each position in the sequence to attend to every other position, computing how much each token should influence every other token. The feed-forward network applies

a nonlinear transformation to each position independently. Between these operations the layer applies normalization and residual connections to stabilize the computation. After passing through all 22 layers we still have a matrix of shape [7, 2048], but the vectors have been deeply transformed to encode the model's understanding of the input.

The final operation maps the output vectors back to the vocabulary. The last row of the matrix, corresponding to the last input token "?", is multiplied by a weight matrix of shape [2048, 32000] to produce 32000 raw scores called logits. Each logit corresponds to a token in the vocabulary and represents how likely that token is to be the next token in the sequence. The logit for the token "Paris" will be among the highest. We select one token from this distribution, append it to our sequence, and repeat the entire process.

This repetition is the essence of autoregressive generation. The model always predicts only the next token. After generating "Paris" we now have the sequence "What is the capital of France? Paris" and we run inference again to predict the token after "Paris". This continues token by token until the model generates a special end-of-sequence token or we stop it. Each step runs the full transformer forward pass, which is why generation speed is measured in tokens per second.

Why Inference Is Different from Training

Understanding inference requires understanding how it differs from training, because the differences have profound consequences for system design.

During training the model sees vast amounts of text, one sequence at a time. For each sequence the model tries to predict every token given all the preceding tokens. If the training example is "The capital of France is Paris", the model learns to predict "The" from nothing, "capital" from "The", "of" from "The capital", and so on. The training process computes all these predictions in parallel: for a sequence of length 2048 tokens the model predicts all 2048 next tokens in a single forward pass. This is efficient because the computation is highly parallelizable and can fully utilize GPU compute resources.

During inference the situation is fundamentally different. For the initial prompt we do have a parallel phase: we can process all the prompt tokens at once to build the context. This is called the prefill phase, and it works the same way as a training forward pass over the prompt sequence. But after the

prompt we are stuck generating one token at a time. The model cannot predict the second generated token until it has predicted the first, because the first token is part of the input for the second prediction. This is called the decode phase, and it is sequential by nature.

This asymmetry between prefill and decode creates very different performance characteristics. Prefill is compute-bound: we have many tokens to process, we are doing a lot of matrix multiplication, and the bottleneck is how fast we can perform arithmetic. Decode is memory-bound: we are processing only one new token at a time but we must load the entire model's weights and read all the cached key and value vectors from previous tokens. The arithmetic is trivial compared to the memory traffic. This is why inference optimization is so different from training optimization: during decode the critical metric is not floating-point operations per second but bytes per second moved through memory.

Another key difference is that inference is typically latency-sensitive. A user waiting for a response does not care about throughput across many users; they care about how quickly their own response starts appearing. This drives design choices like KV caching to minimize per-token latency, speculative decoding to hide latency behind predictions, and streaming output so the user sees text as soon as it is generated rather than waiting for the complete response.

Training also requires storing and computing gradients for backpropagation, which multiplies memory requirements by roughly four times compared to the forward pass alone. Inference needs only the forward pass and the cached activations, which simplifies memory management but creates different pressure points.

The Transformer Blueprint at a Glance

The transformer architecture, introduced in the landmark paper “Attention Is All You Need” in 2017, has become the foundation of virtually all modern large language models. While specific implementations vary, the core components are consistent across models like Llama, GPT, Mistral, and their derivatives. Our implementation targets the Llama-2 style architecture, which TinyLlama-1.1B uses verbatim.

A transformer is a stack of identical layers, each performing the same computation pattern. In TinyLlama-1.1B there are 22 such layers, stacked in sequence. Each layer has the following structure:

First the input vectors pass through an input layer normalization called RMSNorm, which rescales each vector so its components have a consistent magnitude. Normalization stabilizes training and inference by preventing activations from growing too large or too small as they propagate through the network.

Next comes the self-attention block. The input vectors are projected into three sets of vectors: queries, keys, and values. Each position produces a query vector that asks “what information do I need?” and every position produces key and value vectors that represent “what information am I offering?” The attention mechanism computes how well each query matches each key, producing attention weights that determine how much of each value to contribute to each position. This allows information to flow between positions in the sequence based on semantic relationships rather than just proximity.

The attention output is added back to the original input via a residual connection, and the result passes through a second RMSNorm. Then the feed-forward network processes each position independently: it expands the vector to a larger dimension, applies a nonlinear activation function called SwiGLU, and projects it back to the original dimension. Another residual connection adds this output to the previous result.

The output of the 22nd layer is the final hidden state for each position. For generation we use only the hidden state of the last position, which encodes the model’s understanding of the entire sequence up to that point.

Several architectural details distinguish Llama-style models from the original transformer. Instead of standard multi-head attention they use full multi-head attention where each head operates independently. They use rotary positional embeddings (RoPE) instead of additive positional encodings, encoding position directly into the attention computation through rotation matrices. The feed-forward network uses a gated structure (SwiGLU) rather than a simple ReLU activation, which improves capacity. Layer normalization uses RMSNorm rather than standard LayerNorm, omitting the mean centering step. The model ties its output projection weights to its input embedding weights, sharing the same matrix for both encoding tokens to vectors and decoding vectors to token probabilities.

These details matter for implementation. Each architectural choice maps to specific tensor operations, specific weight matrices, and specific data layouts. We will explore each component in depth and implement it exactly as the

architecture specifies.

What This Book Builds

The system we are building is an inference engine for decoder-only transformer models in the Llama family. Given a model stored in HuggingFace SafeTensors format and a corresponding tokenizer, the engine:

- Loads the model configuration to determine architecture parameters like hidden size, number of layers, and vocabulary size
- Loads all model weights from SafeTensors files, parsing the binary format and mapping tensors to the internal representation
- Tokenizes user input into integer sequences using the model's tokenizer
- Runs the transformer forward pass: embedding lookup, through all transformer layers with attention and feed-forward operations, producing logits for the next token
- Samples a token from the logits using configurable strategies like greedy selection, temperature scaling, or top-p nucleus sampling
- Caches key and value vectors for all processed tokens to avoid redundant recomputation
- Repeats the forward pass and sampling step token by token until generation ends
- Decodes the generated token sequence back into text and streams it to the output

The implementation is organized into clean, independent modules. A tensor library provides the fundamental data structures and operations: multi-dimensional arrays, matrix multiplication, element-wise operations, and reductions. A SafeTensors reader parses the file format and loads weights. A tokenizer module handles text encoding and decoding. The model module implements the transformer architecture: embedding lookup, layer normalization, attention, RoPE, feed-forward networks, and the layer loop. A KV cache manager stores and retrieves cached activations. A sampling module selects tokens from logits. A runtime ties everything together in the generation loop.

Each module is implemented in a single C source file with a corresponding header. The code is portable: it compiles on Linux, macOS, and Windows with any standard C compiler, requires no external dependencies for the

base implementation, and is designed from the start to be extensible with optimizations like SIMD intrinsics, multithreading, and hardware accelerators.

The base implementation prioritizes correctness and clarity over raw performance. Every operation is implemented straightforwardly so that you can read the code and immediately understand what it does. Once we verify correctness by comparing outputs against a reference implementation, we progressively add optimizations: vectorized kernels, cache-aware data layouts, tiled matrix multiplication, parallel execution, and more. At each stage you can measure the improvement and understand exactly what changed to produce it.

A First Look: Loading, Running, and Generating

Before we dive into the mathematics and implementation details, let us look at what the final system will look like from the user's perspective. This will give you a concrete target to aim for as we build each component.

You download the TinyLlama-1.1B model files from HuggingFace:

```
1 $ huggingface-cli download TinyLlama/TinyLlama-1.1B-Chat-v1.0 \
2   --local-dir ./tinylama \
3   --include "config.json" "tokenizer.json" "model.safetensors"
```

You compile the inference engine:

```
1 $ mkdir build && cd build
2 $ cmake -DCMAKE_BUILD_TYPE=Release ..
3 $ make -j$(nproc)
```

You run inference from the command line:

```
1 $ ./llm_infer --model ./tinylama/model.safetensors \
2   --config ./tinylama/config.json \
3   --tokenizer ./tinylama/tokenizer.json \
4   --prompt "What is the capital of France?" \
5   --max-tokens 50 --temperature 0.7
```

The output shows:

```
1 Loading model configuration... done
2 Loading tokenizer... done (32000 tokens)
3 Loading model weights... done (22 layers, hidden_size=2048)
4 Model size: 4.4 GB (BF16)
5 Starting inference...
6 What is the capital of France? Paris is the capital of France.
7
8 Tokens generated: 12
9 Generation time: 0.42s
10 Tokens per second: 28.6
```

This is not a simulated example or a simplified demonstration. This is exactly what the complete implementation will do. The 28.6 tokens per second is a reasonable baseline for an unoptimized implementation on a modern CPU running TinyLlama in 32-bit floating-point arithmetic. After optimization in later chapters we can push this higher through SIMD vectorization and multithreading.

The command-line interface is simple on purpose. The core of the system is the inference loop, not the user interface. Later chapters will show how to add features like chat templates, streaming output callbacks, HTTP server mode, and configuration options, but the fundamental flow remains the same: load, tokenize, run, generate.

Understanding this end-to-end flow is the foundation for everything that follows. Each chapter builds a piece of this system. By the time we reach the final generation implementation, every component will be in place and the puzzle will snap together. Let us start building.

Chapter 2: Tokens, Vocabularies, and the Language of Models

What a Token Really Is

Neural networks operate on numbers, not text. When you feed “Hello world” into a language model, the model never sees those characters directly. It sees integers. The job of the tokenizer is to convert text into these integers and convert the integers back into text. Understanding tokenization is not optional; it is the interface between the real world of natural language and the mathematical machinery of the model.

A token is a unit of text that the model processes as a single atomic item. Tokens can be whole words, parts of words, punctuation marks, or whitespace. The specific tokens for a given text depend entirely on the tokenizer used. For example the string “unbelievable” might be one token in one tokenizer, or it might be split into [“un”, “believ”, “able”] in another. This is not arbitrary: the choice of tokenization directly affects model performance, memory requirements, and generation behavior.

Models like TinyLlama use a vocabulary of about 32000 tokens. This vocabulary is learned during training by analyzing a massive corpus of text and finding the most efficient way to represent it. Every token in the vocabulary has a unique integer ID from 0 to `vocab_size` minus 1. When we tokenize text, we are essentially compressing the string into a sequence of these IDs. When we decode the model’s output, we reverse the process: we look up each ID in the vocabulary and concatenate the corresponding strings.

The vocabulary size is a fundamental architectural parameter. It determines the size of the embedding matrix, which maps tokens to vectors. For TinyLlama with vocabulary size 32000 and hidden dimension 2048, the embedding matrix has $32000 * 2048 = 65,536,000$ elements. In BF16 precision at 2 bytes per element that is about 128 MB just for the embedding layer. A larger vocabulary means a larger embedding matrix and more memory, but potentially fewer tokens per sequence. A smaller vocabulary saves memory but requires more

tokens to represent the same text, increasing the sequence length and computation. The vocabulary size is a trade-off decided during training and fixed for inference.

Beyond the basic vocabulary, tokenizers define special tokens that serve structural purposes. The beginning-of-sequence token marks the start of a new sequence. The end-of-sequence token signals the model to stop generating. The padding token fills out variable-length sequences to a fixed length in batch processing. The unknown token represents words or strings that cannot be decomposed into known subwords. For TinyLlama the special tokens are: BOS (ID 1), EOS (ID 2), PAD (ID 0). Chat models also define role-specific tokens for formatting conversations, like the system prompt markers and assistant response markers.

Byte Pair Encoding and Subword Tokenization

The tokenizer used by TinyLlama, like all modern Llama-family models, implements Byte Pair Encoding, an algorithm that builds a subword vocabulary by iteratively merging the most frequently co-occurring pairs of symbols. Understanding how BPE works illuminates why tokenization behaves the way it does and helps explain phenomena like why some words are one token and others are five.

BPE starts with the basic alphabet of all characters that appear in the training corpus. In practice this includes not just ASCII letters but also Unicode characters, whitespace, and punctuation. The algorithm then iteratively finds the most common adjacent pair of symbols and merges them into a new symbol. This new symbol is added to the vocabulary, and the process repeats until the vocabulary reaches the desired size.

Consider a simplified example. Suppose our training corpus consists of these words: “low”, “lower”, “newest”, “wider”. Starting with individual characters we count adjacent pairs: “l” and “o” appear together twice in “low” and “lower”, “o” and “w” also appear twice, “e” and “r” appear three times, and so on. The pair “e” and “r” is most frequent, so we merge them into a new token “er”. Now “lower” becomes “lo” + “w” + “er” and “newest” becomes “new” + “es” + “t”. We repeat the process with the updated tokenization until we reach our target vocabulary size.

The resulting vocabulary has a hierarchical structure. Common character sequences become single tokens, rare sequences remain as multiple tokens.

This is why “hello” might be one token but “hellooo” becomes “hello” + “o” + “o”: the exact sequence “hellooo” never appeared frequently enough during training to become its own token. This subword approach gives tokenizers the ability to handle any text, including words never seen during training, by decomposing them into known subword units.

Modern implementations add refinements to this basic idea. Byte-level BPE, used by GPT-2 and some other models, starts with all 256 possible byte values rather than Unicode characters. This guarantees that any byte sequence can be tokenized, even if it contains characters outside the training vocabulary. SentencePiece, the library used to train Llama’s tokenizer, uses a similar approach but operates on Unicode code points directly and uses a special symbol to represent whitespace: U+2581, a lower one eighth block displayed as ▯. This means “hello world” is not tokenized as “hello” + “world” but potentially as “hello” + “▯world”, with the space being part of the second token.

The BPE training process outputs two artifacts: a vocabulary mapping tokens to IDs, and a merge table listing the merge operations in the order they were applied. During inference the encoder uses both: it first breaks the input into characters, then repeatedly applies the merge operations in order, replacing adjacent pairs that match the next merge in the table. The decoder uses only the vocabulary: it looks up each ID to get the token string and concatenates them.

The merge order is critical. It determines the priority with which pairs are combined. If the merge table first merges “h” and “e” into “he”, and later merges “h” and “i” into “hi”, then “hi” will only merge if no “he” pairs remain that could be merged first. This means tokenization is deterministic: the same input always produces the same tokens.

Loading and Interpreting Tokenizer Files

HuggingFace models package their tokenizers in a JSON file called `tokenizer.json` that fully describes the tokenizer’s behavior. This file contains everything needed to encode and decode text: the vocabulary, the merge operations, the normalization rules, the special tokens, and the processing pipeline. For TinyLlama this file is about 1.8 MB, containing the complete description of the BPE tokenizer with 32000 tokens and approximately 32000 merge operations.

The `tokenizer.json` structure is hierarchical. At the top level it defines the tokenizer type (“BPE”), the model vocabulary and merges, the normalizer to apply before tokenization, the pre-tokenizer to split text into words, the post-processor to add special tokens, and a decoder for converting tokens back to text. Each component is a JSON object with specific fields.

The vocabulary is a JSON object mapping token strings to integer IDs:

```

1  {
2    "model": {
3      "type": "BPE",
4      "vocab": {
5        "<s>": 1,
6        "</s>": 2,
7        "_": 3,
8        "a": 4,
9        "b": 5,
10       "...": "...",
11       "_world": 1438,
12       "hello": 15043,
13       "...": "...",
14     },
15     "merges": [
16       "_ t",
17       "i n",
18       "a t",
19       "...",
20       "h e"
21     ]
22   }
23 }
```

The vocabulary object contains every token the model knows, mapped to its ID. The token strings may include Unicode characters, special symbols, and the SentencePiece whitespace marker. Some tokens are single characters, some are whole words, some are parts of words. The IDs are not ordered alphabetically or by frequency; they are assigned during training and the assignment is essentially arbitrary.

The merges array lists each merge operation as a string containing the two tokens to merge, separated by a space. For example “_ t” means that whenever the token “_” is followed by the token “t”, they should be merged into a single token “_t”. The merges are listed in the order they should be applied: first merge is highest priority, last merge is lowest priority. In practice the implementation does not literally apply merges sequentially; it builds a lookup

table or tree to efficiently find the applicable merge for any pair of adjacent tokens.

The normalizer describes text transformations applied before tokenization. Common normalizations include NFKC Unicode normalization to canonicalize character forms, stripping control characters, and lowercasing (though Llama-style tokenizers do not lowercase). For TinyLlama the normalizer is an NFKC normalizer, which handles Unicode compatibility decomposition and composition.

The pre-tokenizer describes how to split the input text into units before applying BPE merges. For Llama tokenizers this is typically a sequence of steps: first split on whitespace, then split on punctuation boundaries using a regex pattern. The default pattern used by Llama is similar to the GPT-4 tokenizer pattern: it splits around spaces, punctuation marks, and Unicode character boundaries. This preprocessing ensures that BPE merges only happen within meaningful word-like units.

The decoder describes how to convert token strings back to text. The most common decoder for Llama is the “ByteLevel” decoder, which removes the leading whitespace marker from tokens and handles unrepresented bytes by converting them to Unicode replacement characters.

Implementing a complete tokenizer in C requires parsing this JSON structure and implementing each component. For the purposes of our inference engine we need two primary functions: encode, which converts a string to token IDs, and decode, which converts token IDs back to a string. We will implement both.

Implementing Token Encoding in C

Our tokenizer implementation needs to load the vocabulary and merges from `tokenizer.json`, then provide efficient encode and decode functions. We will structure this as a self-contained module with the following interface:

```

1  typedef struct {
2      // Vocabulary: token_id -> token_bytes
3      size_t vocab_size;
4      const char **tokens;          // tokens[token_id] = token string
5      const size_t *token_lens;     // token_lens[token_id] = length
6
7      // Reverse lookup: token_bytes -> token_id
8      // Implemented as a hash table for O(1) lookup
9      HashTable *id_lookup;
10
11     // Merges: pairs of (left_id, right_id) -> merged_id
12     size_t merge_count;
13     uint32_t *merge_left;          // merge_left[i] = left token ID of merge i
14     uint32_t *merge_right;         // merge_right[i] = right token ID of merge i
15     uint32_t *merge_result;        // merge_result[i] = resulting token ID
16
17     // Special token IDs
18     uint32_t bos_token_id;
19     uint32_t eos_token_id;
20     uint32_t pad_token_id;
21     uint32_t unk_token_id;
22
23     // Internal merge priority table
24     // merge_priority[a * vocab_size + b] = merge index if pair (a,b) can merge
25     int32_t *merge_priority;
26
27     // Whether we have loaded successfully
28     int loaded;
29 } Tokenizer;
30
31 int tokenizer_load(Tokenizer *tk, const char *path);
32 void tokenizer_free(Tokenizer *tk);
33 int tokenizer_encode(const Tokenizer *tk, const char *text, int text_len,
34                     uint32_t *tokens, int max_tokens);
35 int tokenizer_decode(const Tokenizer *tk, const uint32_t *tokens, int n_tokens,
36                     char *out, int max_out);

```

The `tokenizer_load` function reads `tokenizer.json`, parses the JSON structure, builds the vocabulary arrays, constructs the reverse lookup table, parses the merges array, and builds the merge priority table. The merge priority table is the key optimization that makes encoding efficient: it maps each possible pair of adjacent token IDs to the index of the merge operation that applies to them, or -1 if no merge applies. With this table we can find applicable merges in O(1) time instead of scanning through thousands of merge rules.

The encoding algorithm works as follows. First normalize the input text using NFKC normalization. For a C implementation we can use the ICU library, `libunistring`, or a simplified approximation. Then split the normalized text into

initial tokens using the pre-tokenizer's regex pattern. Each initial character sequence is broken into its constituent characters or byte-level tokens and looked up in the vocabulary. This produces an initial sequence of token IDs, one per character.

Next apply merges iteratively. Scan through the token sequence, and for each adjacent pair of tokens check the merge priority table. Record all applicable merges and their priorities. Apply the highest-priority merge first (the one with the lowest merge index). Repeat until no more merges apply. This process is equivalent to repeatedly applying the merge rules in priority order, but implemented more efficiently by scanning the current sequence and finding all applicable merges at once.

A more efficient implementation uses a priority queue of merge candidates. Start with the character-level tokenization, then for each adjacent pair compute the applicable merge priority. Insert all candidates into a priority queue. Repeatedly extract the highest-priority candidate and apply the merge, which may create new merge candidates with neighboring tokens. Continue until the queue is empty. This approach is $O(n \log n)$ for a sequence of n initial tokens.

For TinyLlama with vocabulary size 32000, the merge priority table requires $32000 * 32000 * 4$ bytes = about 4 GB if implemented as a dense 2D array, which is impractical. Instead we implement it as a hash table mapping pairs to merge indices, or as a tree-based lookup structure. The llama.cpp project uses a trie-based approach that builds a suffix tree from the vocabulary tokens and walks it during encoding.

For our educational implementation we will use a simpler approach: we build a hash table mapping merge pairs to their priorities. The hash key is a combination of the two token IDs, computed as a perfect or near-perfect hash. This gives us $O(1)$ average lookup with reasonable memory usage.

Here is the core encoding loop:

```

1  #include <stdlib.h> // malloc, free
2
3  int tokenizer_encode(const Tokenizer *tk, const char *text, int text_len,
4                      uint32_t *tokens, int max_tokens) {
5      // Step 1: normalize text (simplified: NFKC approximation)
6      char *normalized = normalize_nkcf(text, text_len);
7      if (!normalized) return -1;
8
9      // Step 2: break into initial character tokens
10     int n_initial = 0;
11     uint32_t *initial_tokens = malloc(text_len * sizeof(uint32_t));
12     const char *p = normalized;
13     while (*p && n_initial < text_len) {
14         // Get next UTF-8 character
15         uint32_t codepoint;
16         int consumed = utf8_decode(p, &codepoint);
17         // Look up character as token
18         char buf[5];
19         int len = utf8_encode(buf, codepoint);
20         buf[len] = 0;
21         uint32_t id = hash_lookup(tk->id_lookup, buf, len);
22         if (id == HASH_NOT_FOUND) id = tk->unk_token_id;
23         initial_tokens[n_initial++] = id;
24         p += consumed;
25     }
26     free(normalized);
27
28     // Step 3: apply merges using priority queue
29     struct merge_candidate {
30         int pos;           // position in token sequence
31         int priority;      // merge index (lower = higher priority)
32     };
33     // Build initial candidate list
34     int n_candidates = 0;
35     struct merge_candidate *candidates = NULL;
36     if (n_initial > 1)
37         candidates = malloc((size_t)(n_initial - 1) * sizeof(*candidates));
38     for (int i = 0; i + 1 < n_initial; i++) {
39         int idx = get_merge_priority(tk, initial_tokens[i],
40                                     ↪ initial_tokens[i+1]);
41         if (idx >= 0) {
42             candidates[n_candidates].pos = i;
43             candidates[n_candidates].priority = idx;
44             n_candidates++;
45         }
46     }
47
48     // Use simple selection to find min priority each iteration
49     int n_tokens = n_initial;
50     while (n_candidates > 0 && n_tokens <= max_tokens) {

```

```

50 // Find highest priority (lowest index) candidate
51 int best = 0;
52 for (int i = 1; i < n_candidates; i++) {
53     if (candidates[i].priority < candidates[best].priority) best = i;
54 }
55
56 // Apply this merge
57 int pos = candidates[best].pos;
58 int merge_idx = candidates[best].priority;
59 initial_tokens[pos] = tk->merge_result[merge_idx];
60
61 // Remove the second token of the merged pair (shift starts at
62 // pos + 2 so the merged token written at pos is preserved)
63 for (int i = pos + 2; i < n_tokens; i++) {
64     initial_tokens[i - 1] = initial_tokens[i];
65 }
66 n_tokens--;
67
68 // Remove invalid candidates (positions past merge)
69 int new_candidates = 0;
70 for (int i = 0; i < n_candidates; i++) {
71     if (candidates[i].pos == pos || candidates[i].pos == pos + 1)
72         ↪ continue;
73     if (candidates[i].pos > pos + 1) candidates[i].pos--;
74     candidates[new_candidates++] = candidates[i];
75 }
76 n_candidates = new_candidates;
77
78 // Add new candidates around merge position. Any entry already
79 // present at these positions refers to a pair the merge just
80 // invalidated, so it must be updated or removed, never duplicated:
81 // a stale entry would apply an old merge to the new token.
82 for (int side = 0; side < 2; side++) {
83     int p = (side == 0) ? pos - 1 : pos;
84     if (p < 0 || p + 1 >= n_tokens) continue;
85     int idx = get_merge_priority(tk, initial_tokens[p],
86     ↪ initial_tokens[p+1]);
87     int found = -1;
88     for (int i = 0; i < n_candidates; i++) {
89         if (candidates[i].pos == p) { found = i; break; }
90     }
91     if (found >= 0) {
92         if (idx >= 0) {
93             candidates[found].priority = idx;
94         } else {
95             candidates[found] = candidates[--n_candidates];
96         }
97     } else if (idx >= 0) {
98         candidates[n_candidates].pos = p;
99         candidates[n_candidates].priority = idx;

```

```

98         n_candidates++;
99     }
100 }
101 }
102
103 // Copy result
104 int count = 0;
105 for (int i = 0; i < n_tokens && count < max_tokens; i++) {
106     tokens[count++] = initial_tokens[i];
107 }
108
109 free(initial_tokens);
110 free(candidates);
111 return count;
112 }

```

This implementation is not optimal but is correct and illustrates the algorithm clearly. Production implementations use more sophisticated data structures: a trie-based encoder that builds tokens incrementally by scanning the input text, or a priority queue that maintains candidates more efficiently. The llama.cpp project uses a trie-based approach that runs in approximately linear time with respect to the input length.

Implementing Token Decoding and Special Tokens

Decoding is significantly simpler than encoding because it requires only the vocabulary, not the merge table. Given a sequence of token IDs, we look up each ID in the vocabulary to get the corresponding token string, concatenate them, and apply any post-processing.

```

1  #include <stddef.h> // size_t
2
3  int tokenizer_decode(const Tokenizer *tk, const uint32_t *tokens, int n_tokens,
4                      char *out, int max_out) {
5      int pos = 0;
6      for (int i = 0; i < n_tokens && pos < max_out - 1; i++) {
7          uint32_t id = tokens[i];
8          if (id >= tk->vocab_size) continue; // skip out-of-range IDs
9
10         const char *token = tk->tokens[id];
11         size_t len = tk->token_lens[id];
12
13         // For BPE tokenizers with SentencePiece whitespace marker,
14         // convert the marker to a real space

```

```

15     for (size_t j = 0; j < len && pos < max_out - 1; j++) {
16         char c = token[j];
17         if (c == '\xe2' && j + 2 < len && token[j+1] == '\x96' && token[j+2]
18             ↪ == '\x81') {
19             // SentencePiece whitespace marker U+2581 -> ' '
20             out[pos++] = ' ';
21             j += 2;
22         } else {
23             out[pos++] = c;
24         }
25     }
26     out[pos] = '\0';
27     return pos;
28 }

```

Special tokens require careful handling. The BOS token should be prepended to the input sequence before running inference, because the model expects sequences to start with it. The EOS token should be checked after each generated token; if the model generates EOS, generation should stop. During chat completion the system may also insert special tokens to separate user messages from assistant messages, following a template like:

```
1 <s>[INST] {system_prompt} {user_message} [/INST] {assistant_response}</s>
```

Our tokenizer module will provide helper functions for these operations:

```

1 // Add BOS token to the beginning of a token sequence
2 int tokenizer_add_bos(const Tokenizer *tk, const uint32_t *input, int n_input,
3                     uint32_t *output, int max_output) {
4     if (n_input >= max_output) return -1;
5     output[0] = tk->bos_token_id;
6     for (int i = 0; i < n_input; i++) output[i + 1] = input[i];
7     return n_input + 1;
8 }
9
10 // Check if a token is the EOS token
11 int tokenizer_is_eos(const Tokenizer *tk, uint32_t token_id) {
12     return token_id == tk->eos_token_id;
13 }

```

The loading function `tokenizer_load` must parse `tokenizer.json`. Since we are writing in C without external JSON libraries, we implement a minimal JSON parser that handles the subset of JSON used by tokenizer files: objects with

string keys, arrays of strings or numbers, and string values. We do not need full JSON parsing; we need only to extract specific fields from a known structure. The parser reads character by character, skips whitespace, and recognizes the delimiters (braces, brackets, colons, commas, quotes). For string values it handles UTF-8 bytes directly, since JSON strings are UTF-8 encoded.

The vocabulary loading step iterates through the “vocab” object, reading each key-value pair. The key is the token string (possibly containing Unicode), the value is the integer ID. We store both the token string and its length, and insert an entry into the reverse lookup hash table. The merges loading step iterates through the “merges” array, reading each merge string, splitting it on whitespace to get the two tokens being merged, looking up their IDs, and recording the merge operation. We also build the merge priority lookup table at this stage.

Complete tokenizer implementation, including the JSON parser and hash table, forms a substantial module of several hundred lines of C code. We will provide the full implementation in the project files, but the essential algorithm is as described above: parse the file, build the data structures, implement the merge-based encoding loop, and implement vocabulary lookup for decoding.

With the tokenizer complete, we have the interface between text and the model’s numerical representation. When the user provides a prompt we convert it to token IDs and feed those IDs to the embedding layer. When the model outputs token IDs we convert them back to text and display them. The quality of tokenization directly affects the model’s behavior: unusual tokenization of user input can confuse the model, and the model’s ability to generate coherent text depends on its vocabulary.

Chapter 3: Tensors, Shapes, and Memory Layouts

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

The Tensor as a Multi-Dimensional Array

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

Row-Major vs Column-Major and Why It Matters

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

Shapes, Strides, and Offsets

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

Float16, BFloat16, and Float32 Representations

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

Contiguous vs Strided vs Packed Tensors

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

Chapter 4: Building the Tensor Core Library

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

Tensor Allocation and Lifetime Management

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

Views, Reshaping, and Slicing Without Copies

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

Element-Wise Operations and Broadcasting

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

Matrix Multiplication from First Principles

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

Reductions, Sums, and Normalization Primitives

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

Chapter 5: The Transformer Layer in Depth

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

Layer Normalization and RMSNorm

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

Multi-Head Self-Attention Mechanism

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

Feed-Forward Networks and Activation Functions

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

Residual Connections and Gate Linear Units

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

Putting It All Together: One Transformer Block

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

Chapter 6: Positional Encodings and RoPE

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

Why Positional Information Matters

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

Learnable vs Absolute vs Rotary Encodings

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

The Mathematics of RoPE

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

Implementing RoPE in C

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

RoPE Scaling and Long-Context Strategies

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

Chapter 7: Softmax, Logits, and Sampling

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

From Hidden States to Logits

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

Numerically Stable Softmax

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

Temperature Scaling and Exploration

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

Top-K Filtering and Nucleus Sampling

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

Deterministic and Reproducible Generation

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

Chapter 8: The KV Cache and Autoregressive Generation

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

The Autoregressive Generation Loop

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

Key-Value Caching: The Core Optimization

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

KV Cache Layouts and Memory Planning

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

Prefill vs Decode Phases

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

Managing Cache Growth and Limits

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

Chapter 9: The SafeTensors File Format

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

Why SafeTensors and What Problem It Solves

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

Header Structure and Serialization

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

Tensor Descriptors, Data Types, and Offsets

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

Endianness, Alignment, and File Boundaries

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

Supported Data Types and Shape Encoding

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

Chapter 10: Implementing the SafeTensors Loader in C

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

Header Reading and Length Prefix Parsing

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

JSON Metadata Parsing in C

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

Tensor Descriptor Extraction and Validation

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

Mapping and Reading Tensor Data Blocks

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

Error Handling, Boundaries, and Integrity Checks

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

Chapter 11: Model Configuration and Weight Mapping

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

The config.json: Architecture Parameters

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

Weight Naming Conventions and Patterns

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

Mapping Transformer Layer Weights

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

Handling Embeddings and Output Projections

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

Validation: Checking Loaded Weight Shapes

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

Chapter 12: Building the Model

Forward Pass

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

Token Embedding Lookup

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

The Layer Loop: Attention, FFN, and Residuals

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

Positional Encoding at Each Position

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

Final Layer Normalization

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

Computing Output Logits

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

The Complete Attention Implementation

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

Putting It All Together: The Forward Pass

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

Chapter 13: End-to-End Inference: From Prompt to Text

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

Runtime State and Context Management

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

The Generation Loop Implementation

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

Integration: Tokenizer, Model, and Sampler

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

Streaming Output and Callbacks

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

Command-Line Interface and First Run

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

Chapter 14: Verifying Correctness Before Optimizing

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

Reference Implementations and Ground Truth

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

Comparing Embeddings and Intermediate Activations

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

Logit Comparison and Numerical Tolerance

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

Generating Test Cases from Known Outputs

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

Assertions, Sanitizers, and Debugging Techniques

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

Chapter 15: Performance Profiling and Bottleneck Analysis

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

Latency, Throughput, and Tokens Per Second

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

Profiling Tools: Perf, Valgrind, and Flame Graphs

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

Compute-Bound vs Memory-Bound Analysis

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

Cache Misses, Branch Mispredictions, and Latency

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

Identifying the Critical Path

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

Chapter 16: SIMD and Vectorization

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

Understanding SIMD Architectures: AVX2, AVX-512, NEON

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

Autovectorization vs Hand-Written Intrinsics

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

Vectorized Element-Wise Operations

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

Vectorized Matrix Multiplication

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

Cross-Platform SIMD Portability

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

Chapter 17: Memory and Threading Optimizations

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

Cache Locality and Data Layout Optimization

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

Loop Tiling and Blocking for MatMul

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

Multithreading with OpenMP and Native Threads

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

False Sharing, Synchronization, and Contention

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

NUMA Awareness and Large Memory Models

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

Memory Bandwidth, Compute-Bound vs Memory-Bound Workloads

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

Chapter 18: Advanced Optimization Techniques

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

Weight Quantization and Low-Bit Representations

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

Packed Weights and Efficient Memory Use

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

Batching and Continuous Batching

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

Speculative Decoding and Draft Models

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

Asynchronous Execution and Pipeline Parallelism

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

Model-Size-Dependent Optimization Strategies

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

Conclusion: What We Built and Why It Matters

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

References

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