ChatGPT

Building a Rust LLM Inference Engine (CPU-only, GGUF)

This guide outlines how to implement a basic transformer LLM inference engine in Rust, supporting the GGUF model format and token-by-token (streamed) generation. We break it down into key stages, highlight helpful crates, and link to relevant resources.

1. Loading and Parsing GGUF Files

- **GGUF overview:** GGUF is a binary "unified" format (by @ggerganov of llama.cpp) for LLM weights and metadata. It stores key-value metadata (model hyperparameters, tokenizer info, etc.) and weight tensors in one file [1] [2]. GGUF is designed for fast loading in inference and is backward-compatible: new metadata can be added without breaking older tools [3] [1]. For example, a GGUF file might list the model's architecture, context length, head count, tokenizer merges, etc. (see the **gguf-rs** README for a sample metadata dump) [1].

- **Parser crate:** Use an existing Rust GGUF parser. For instance, the `gguf-rs` crate provides a `get_gguf_container(path)` function that reads a GGUF file. Calling `.decode()` returns a model object with typed access to metadata and tensors [4]. As shown below, you can then inspect fields like `model.model_family()` or `model.num_tensor()` after decoding [4]:

```
let mut container = gguf_rs::get_gguf_container("model.gguf")?;
let model = container.decode()?;
println!("Family: {}, Params: {}",
         model.model_family(), model.model_parameters());
```

- **Example usage:** The `gguf-rs` README shows how to list metadata and tensors (via a CLI or programmatically) [5] [6]. Alternatively, crates like ggus or gguf can be used. In practice, you'll load all weight tensors (e.g. embeddings, projection matrices, layer norms) into Rust data structures (e.g. contiguous `Vec<f32>` or `Tensor` objects) keyed by name.

- **Integration with llama.cpp:** Many converters (e.g. Hugging Face scripts) produce GGUF for Llama, Mistral, Phi, etc. Note that **llama.cpp** requires GGUF format by default [7], and most Rust LLM tools assume GGUF weights. Once parsed, you have access to all model parameters and can begin arranging them in memory.

2. Model Parameters and Memory Layout

- **Weight storage:** After parsing, store each tensor's data contiguously for efficient access. For example, weights like the token embedding matrix or attention projection matrices should be stored in row-major order matching the model's expected layout. Using Rust crates like `ndarray` or `burn` (or even raw `Vec<f32>`) is fine, but ensure alignment and memory order match your math routines. Often models use 2D matrices of shape (in_features, out_features) or 4D tensors for convolutional blocks – flatten these into Rust arrays as needed.

- **Memory mapping:** For large models, consider memory-mapping the GGUF file instead of copying all data. The `memmap2` crate can map the file into memory so you only touch pages on demand. In fact, the llama2.rs example shows that using mmap reduced peak memory from ~480 MB to 59 MB (an 88% reduction) [8]. In code, you'd open the GGUF file and create an `Mmap`; then use unsafe Rust to interpret byte slices as `&[f32]` (careful about endianness, though GGUF files encode byte-order in their header).

- **Sharing memory:** To save RAM, you can map weights as `Arc<[f32]>` or reuse slices for shared projections (e.g. QKV weights). Using multithreading (Rust's Rayon or raw threads) can speed up layer computations, as seen in llama2.rs (which achieved a 46% speedup via parallelism) [9]. Plan a memory layout so that each layer's weights are sequentially stored (or grouped by layer) to benefit from caching.

- **Key–Value cache (streaming):** Prepare buffers for caching each layer's keys/values per token (see section 6). For example, you might allocate `Vec<f32>` of shape (num_layers, num_heads, max_seq_len, head_dim) for keys and similarly for values. Initially fill these with zeros or unused.

3. Rotary Embeddings and Multi-Head Attention

- **Rotary positional embeddings:** LLaMA, Mistral, and many modern LLMs use **Rotary Positional Embeddings (RoPE)**. Instead of adding fixed position vectors, RoPE *rotates* each query/key vector by a position-dependent complex phase. Concretely, split each query/key vector into even/odd pairs and multiply by [cos, sin] rotation matrices that depend on the token position. This encodes both absolute and relative position information efficiently [10]. In practice, precompute sin/cos tables up to your context length and apply them to each layer's Q and K before computing attention. (The Llama.cpp tutorial notes that "rotary positional embeddings (RoPE) was added at each layer" [10].)

- **Attention mechanics:** For each layer, compute Queries, Keys, and Values by linear projections of the hidden state:
  $$Q = XW_Q + b_Q,\quad K = XW_K + b_K,\quad V = XW_V + b_V.$$
  Then apply RoPE to Q and K. Next, form the attention scores:
  $$\text{scores} = \frac{Q K^T}{\sqrt{d_k}}\,$$
  where the division by $\sqrt{d_k}$ stabilizes gradients. Apply a causal mask so tokens cannot attend to future positions. Softmax the scores to get weights, and compute the weighted sum over values:
  $$\text{head_out} = \text{softmax(scores)}\, V.$$

- **Multi-head processing:** Do the above for each attention head in parallel. In Rust, you can use a crate like ndarray or burn to compute batched matrix multiplications. For example, `ndarray` supports efficient broadcasted dot-products and can call BLAS under the hood [11]. Alternatively, split the matmul into independent threads per head. After computing all heads, concatenate and project back (via another linear layer) with `W_O`. Don't forget to add residual connections and layer normalization around the attention and FFN blocks.

- **Feed-forward (FFN):** Each transformer layer also has a two-layer feed-forward network: a linear + activation (e.g. SwiGLU/ReLU) followed by another linear. For example, if the hidden size is 4096 and inter size is 11008 (LLaMA-7B), then implement `FFN(X) = SwiGLU(X W_1) W_2`. For CPU, simple `ndarray` ops or manual loops suffice.

4. Tokenization and Detokenization

- **Tokenizer libraries:** Use a robust tokenizer crate to convert user text to token IDs and back. Common choices include Hugging Face's tokenizers (which is a Rust library) and tiktoken-rs (a Rust port of OpenAI's GPT tokenizers). These handle Byte-Pair Encoding (BPE) or SentencePiece rules as needed. Tokenizers in Rust are **extremely fast** – for example, the Hugging Face Tokenizers crate can process ~1 GB of text in <20 seconds on CPU [12].

- **Integration:** Load the model's tokenizer vocab and merges (often included as metadata in GGUF) into your chosen tokenizer. Then you can do `ids = tokenizer.encode(text)` at inference start, and after generation use `tokenizer.decode(tokens)` to join tokens into text. Ensure to handle special tokens (BOS/EOS) according to the model.

- **Detokenization:** After each generated token (ID), convert it back to a subword string and append to the output. (For streaming, you can decode token-by-token or buffer and decode in chunks.) Both HuggingFace's tokenizers and `tiktoken-rs` can decode lists of IDs to strings.

5. Inference Loop and Sampling Logic

- **Loop structure:** The core loop feeds tokens into the model and samples next tokens. For each step (or batch of steps), you run a forward pass: embed the tokens, run all transformer layers, and compute logits over the vocabulary from the final output.

- **Sampling:** Convert logits to probabilities (via softmax). Then apply a sampling strategy. The simplest is *greedy* (pick argmax), but LLMs often use *top-k* and *top-p* sampling with temperature. You can implement these manually with the `rand` crate (sorting logits, drawing from distribution), or use a dedicated sampler crate. For example, `llm-samplers` provides a modular framework to chain samplers: it advertises "Rusty samplers for large language models" [13]. You can build a `SamplerChain` with steps like biasing, repetition-penalty, top-k, top-p, temperature, etc.

```
let mut sc = SamplerChain::new()
    + SampleTemperature::new(0.8)
    + SampleTopK::new(50)
    + SampleTopP::new(0.95);
let next_token = sc.sample_token(&mut resources, &mut logits)?;
```

(Here `resources` can hold an RNG and recently generated tokens for repetition penalty. See the `llm-samplers` examples [13] for details.)

- **Llama sampling reference:** The llama.cpp library uses a sampling chain similar to above (top-K, top-P, temperature, repeat penalty) to produce diversity. The `llm-samplers` crate's docs even list suggested ordering mimicking llama.cpp [13].

6. Token-by-Token (Streaming) Generation

- **Streaming mode:** To generate text one token at a time, maintain a loop where after each new token you update the model's state and output that token immediately. This requires using a

key–value cache: once you compute Keys/Values for a token in each layer, store them. On the next generation step, you feed only the *new* token (plus cached context) so you don't recompute earlier tokens.

- **Implementation:** Initialize your KV cache (zeroed). After feeding the initial prompt (all past tokens), enter a loop: compute one more forward pass for the current token, append the new token to the output, and repeat. Many Rust LLM APIs illustrate this. For example, the Rust binding of llama.cpp uses `ctx.start_completing_with(...)` to return an iterator of tokens; inside it simply loops over generation and yields each token [14]. The key insight: print or stream the output token as soon as it's produced, flush stdout, then continue the loop [14].

- **Example (llama_cpp):**

```
let mut ctx = llama_model.create_session(...)?;
ctx.advance_context(&input_ids)?;
let mut stream = ctx.start_completing_with(StandardSampler::default(),
max_tokens).into_strings();
for token_str in stream {
    print!("{}", token_str);
    std::io::stdout().flush().unwrap();
}
```

Here each `token_str` is generated and flushed immediately [14].

- **Caveat:** Streaming costs an initial "warm-up" as the KV cache fills, but thereafter each token is cheap. Ensure your loop checks for EOS and stops at the desired length.

Rust Libraries and Crates

- **Matrix and tensor math:**
- `ndarray` – N-dimensional arrays and matrix ops in Rust. It supports sliced views and can use BLAS for large matmuls [11]. Good for quick implementation.
- `candle` – a *minimalist ML framework for Rust* with GPU support. Candle is designed for inference and supports Transformers out of the box [15]. It has tensor types and built-in linear layers, and includes examples for LLMs.
- `burn` – a Rust deep-learning framework focusing on flexibility and performance [16]. Burn can handle model building and training, though for inference you might use its tensor routines.
- `matrixmultiply` – low-level BLAS-like matmul for `f32` / `f64`. Used by ndarray internally for fast GEMM.

- `tch` – Rust bindings for PyTorch's libtorch, useful if you want to load TorchScript models (not needed for GGUF).

- **File I/O & utilities:**

- `memmap2` for memory-mapping large weight files.
- [anyhow][18†L139-142] and [`thiserror`] for error handling and parsing code.

- Standard `std::fs` or `tokio::fs` for file reading if not using mmap.

- **Tokenization:**

- Hugging Face Tokenizers crate – very fast Rust implementation for BPE/WordPiece/etc. As noted: *"Extremely fast (both training and tokenization)"* thanks to Rust [12].
- `tiktoken-rs` – a Rust port of OpenAI's GPT tokenizers (works for models like GPT-3/4). It's still evolving but can be used if you target GPT-family models.

- **Sampling:**

- `llm-samplers` – chainable token samplers for LLM logits (temperature, top-k, top-p, repetition, etc.) [13].
- Or implement sampling manually using `rand` (for random draws) and sorting the logits array.

- **Examples & frameworks:**

- **Candle Examples:** The Hugging Face Candle repo includes many CLI examples for LLMs: LLaMA v1/v2/v3, Falcon, CodeGeex, GLM, Mistral7B, etc. [17]. Exploring `candle/examples/` or `candle-transformers` shows how to load GGUF models via Candle.
- **llama_cpp-rs:** The `llama_cpp` crate bundles llama.cpp and provides a Rust API (shown above) [14]. It's useful for learning streaming and sampler usage.
- **Mistral.rs:** mistral.rs is a full-featured Rust LLM runtime (supporting quantization, multi-GPU, etc.). Its repo and docs explain advanced inference techniques and serve as a reference.
- **Rust LLM Ecosystem:** A community list catalogs many Rust ML projects. For example, Candle (Hugging Face) and Mistral.rs are highlighted as active inference engines [18] [19].

Documentation and Guides

- **GGUF format:**
- Hugging Face's docs describe GGUF as a fast binary format for model weights [2].
- IBM's blog *"GGUF versus GGML"* explains its purpose (fast loading, flexible metadata) and quantum support [3] [20].
- The `gguf-rs` README (GitHub) shows how to decode metadata and tensor info [4].

- A recent blog *"LLM GGUF Guide"* (May 2025) also details GGUF internals (not linked here).

- **Transformer inference:**

- Many tutorials cover transformer/LLM inference. For example, a DataCamp blog "Llama.cpp Tutorial" mentions features like RoPE being added in each layer [10].
- General Rust ML articles (e.g. "Building Your First AI Inference Engine in Rust" [21]) overview frameworks like Tract, ONNX Runtime, and Burn.

- llama.cpp and similar projects have documentation on their core algorithms (see [candle-transformers docs] or [llama.cpp docs]).

- **Tokenization:**

- The Hugging Face Tokenizers repo provides documentation and examples for using BPE, WordPiece, etc. It emphasizes speed and flexibility (e.g. *"Extremely fast, thanks to Rust"* [12]).

- **Community resources:**

- The Rust ML/LLM community maintains guides (HackMD, GitHub lists) on crates and examples. These are useful to find up-to-date tips (e.g. on llama-index integrations, Rust Wasm inference, etc.).

Example Implementations

- **llama2.rs:** A pure-Rust port of Meta's `llama2.c`. It demonstrates loading a GGUF (tiny LLaMA) model and running inference. Notably, it achieves ~40 tokens/sec on a MacBook by using OpenMP-like threading, outperforming the C version (27 tok/s) by ~46% [9]. It also has a `--is_mmap` mode that slashes memory usage from ~480 MB to ~59 MB [9]. Check its repo for concrete code on model loading, multithreading, and caching.

- **Candle examples:** The Candle repo's examples (and `candle_transformers`) show how to load GGUF LLaMA/Mistral models and run inference, including quantized (`Q4`) variants. The README lists command-line examples for LLaMA v1/v2/v3, Falcon, Codegeex, Mistral, etc. [17].

- **llama_cpp-rs:** The `llama_cpp` crate (Rust wrapper around llama.cpp) is a good reference for streaming API usage. Its documentation snippet (above) shows how to feed input and start a completion iterator [14].

- **Mistral.rs & others:** For more advanced features (GPU, LoRA, paged attention, etc.), see mistral.rs or candle-vllm. These are complex engines; for a minimal build-from-scratch, focus on the essentials above.

Common Pitfalls and Performance Tips

- **CPU optimizations:** Build in release mode. Use optimized BLAS (or `matrixmultiply`) for large matmuls. Enable multithreading on independent heads or batches. For example, llama2.rs used parallelism to achieve ~46% higher throughput [9].

- **Memory layout:** Ensure tensors are contiguous and aligned for SIMD. Consider fusing QKV computation (some implementations pack Q/K/V matrices together for cache efficiency). Use memory mapping (`memmap2`) to avoid duplicating data [8].

- **Key-Value cache:** Always cache intermediate K/V states to avoid recomputing from scratch each token. This is crucial for streaming performance. Llama.cpp-based tools rely on this; if omitted, generation becomes $O(N^2)$ instead of $O(N)$.

- **Numeric stability:** When implementing softmax, subtract the max logit per row to avoid overflow. Verify correctness on small models first. Watch for underflow/overflow in float; consider layer normalization epsilon tweaks as per model.

- **Quantization:** While initial engine can use `f32`, GGUF supports quantized weights (2–8 bit, GPTQ/AWQ/AFQ) [22]. You can plan your design to allow future quantized arithmetic (e.g. 8-bit ops). However, implementing quant math (INT8 or FP4) adds complexity; start with full precision.

- **Thread-safety:** If using Rayon or threads, ensure read-only access to weight data (immutable references) during inference. Use thread-safe RNG for sampling.

- **Testing:** Validate each component. Compare logits on a known small model (like tiny LLaMA) against a reference (e.g. llama.cpp or PyTorch) to confirm correct tensor ordering, mask logic, and RoPE application.

References

- GGUF format (IBM and Hugging Face docs) [3] [2]; `gguf-rs` crate usage [1] [4].
- Rust ML libraries: Candle (Hugging Face) [15] [18], Burn [16], ndarray [11].
- Tokenizers (Hugging Face) [12]; `tiktoken-rs`.
- Sampling in Rust: `llm-samplers` crate [13].
- Streaming example (llama_cpp) [14].
- Inference projects: llama2.rs (Rust port of llama2.c) [9], mistral.rs (Rust LLM runtime), Candle examples [17].
- Community guides/tutorials: DataCamp llama.cpp tutorial (RoPE mention) [10]; Rust inference overview [21].

---

[1] [4] [5] [6] GitHub - zackshen/gguf: a GGUF file parser
https://github.com/zackshen/gguf

[2] GGUF
https://huggingface.co/docs/hub/en/gguf

[3] [20] [22] GGUF versus GGML | IBM
https://www.ibm.com/think/topics/gguf-versus-ggml

[7] GitHub - ggml-org/llama.cpp: LLM inference in C/C++
https://github.com/ggml-org/llama.cpp

[8] [9] GitHub - lintian06/llama2.rs: Inference Llama 2: A Rust port of llama2.c
https://github.com/lintian06/llama2.rs

[10] Llama.cpp Tutorial: A Complete Guide to Efficient LLM Inference and Implementation | DataCamp
https://www.datacamp.com/tutorial/llama-cpp-tutorial

[11] ndarray - Rust
https://docs.rs/ndarray/

[12] GitHub - huggingface/tokenizers: Fast State-of-the-Art Tokenizers optimized for Research and Production
https://github.com/huggingface/tokenizers

[13] llm_samplers - Rust
https://docs.rs/llm-samplers

[14] llama_cpp - Rust
https://docs.rs/llama_cpp

[15] [17] GitHub - huggingface/candle: Minimalist ML framework for Rust
https://github.com/huggingface/candle

[16] Burn
https://burn.dev/

18 19 Rust Ecosystem for AI & LLMs - HackMD
https://hackmd.io/@Hamze/Hy5LiRV1gg

21 Building Your First AI Model Inference Engine in Rust | Nerds Support, Inc.
https://nerdssupport.com/building-your-first-ai-model-inference-engine-in-rust/