CS265 Project: Building LLM Inference Like Clockwork (Part 2/2)

Spring 2026

1 Overview

In Part 1 of this project, you built the foundation of the inference pipeline. You downloaded the `Llama-3-8B-Instruct` weights, implemented a tokenizer to convert an input prompt into a sequence of token IDs, performed an embedding lookup to obtain the initial token embedding matrix $X \in R^{s \times d}$, and implemented your first CUDA kernel: a tiled matrix multiplication with shared-memory reuse and coalesced memory accesses.

In this part, you will use everything you built to complete the model. In Milestone 2, you implement RMSNorm and use your existing matmul to project the token embeddings into query, key, and value representations. In Milestone 3, you implement the remaining operators, assemble a full decoder block, run it through all 32 layers, and produce one new token from the model. Figure 1 shows the full architecture. By the end of this part, your system will accept a prompt and generate a token in response.

We use the same notation throughout: s for sequence length, $d = 4096$ for embedding dimension, $h = 32$ query heads, $h_k = 8$ key/value heads, $h_d = 128$ head dimension (with $d = h \cdot h_d$), $d_{\text{ff}} = 14336$ for the FFN intermediate size, and $V = 128256$ for vocabulary size. We also provide `reference.py` alongside this document, which contains PyTorch-level implementations of every operator so you can verify your CUDA kernels numerically.

2 Milestone 2: RMSNorm and Q, K, V Projections

2.1 Desired Functionality

In this milestone, you implement an RMSNorm CUDA kernel and use your existing matmul kernel to compute the query, key, and value matrices for the

attention mechanism. These are the first two operators applied to the token embedding matrix on the GPU.

RMSNorm. The input to each decoder block is a matrix $X \in R^{s\times d}$. Before computing Q, K, V, this input is normalized row-wise using RMSNorm. Applied independently to each row $x \in R^d$:

$$\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x)} \odot \gamma, \qquad \text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon}$$

where $\gamma \in R^d$ is a learned per-element scale loaded from the model weights (named `input_layernorm.weight` in the HuggingFace checkpoint), $\varepsilon = 10^{-5}$ is a small constant added inside the square root for numerical stability, and $\odot$ denotes elementwise multiplication. The output is $X_{\text{norm}} \in R^{s\times d}$.

Q, K, V Projections. Using X_{norm}, compute the query, key, and value matrices via three separate matmul calls with learned projection weights:

$$\begin{aligned}
Q &= X_{\text{norm}}\, W_Q^\top, & Q &\in R^{s\times(h\cdot h_d)}, & W_Q &\in R^{(h\cdot h_d)\times d}\\
K &= X_{\text{norm}}\, W_K^\top, & K &\in R^{s\times(h_k\cdot h_d)}, & W_K &\in R^{(h_k\cdot h_d)\times d}\\
V &= X_{\text{norm}}\, W_V^\top, & V &\in R^{s\times(h_k\cdot h_d)}, & W_V &\in R^{(h_k\cdot h_d)\times d}
\end{aligned}$$

Note that $h\cdot h_d = 4096 = d$ while $h_k\cdot h_d = 1024$. Q has more columns than K and V because of Grouped Query Attention (GQA), explained in Milestone 3. These projections are independent and each is a direct call to your matmul kernel.

1. **Step 1: RMSNorm kernel.** Implement a CUDA kernel that applies RMSNorm to each row of X using the scale weights γ and ε, producing $X_{\text{norm}} \in R^{s\times d}$.
2. **Step 2: Q projection.** Call your matmul kernel on X_{norm} and W_Q to produce $Q \in R^{s\times(h\cdot h_d)}$.
3. **Step 3: K projection.** Call your matmul kernel on X_{norm} and W_K to produce $K \in R^{s\times(h_k\cdot h_d)}$.
4. **Step 4: V projection.** Call your matmul kernel on X_{norm} and W_V to produce $V \in R^{s\times(h_k\cdot h_d)}$.

Assumption and Simplification: You are not required to implement batching or KV caching. Both are optional extensions. You may assume the input sequence length does not exceed 1000 tokens.

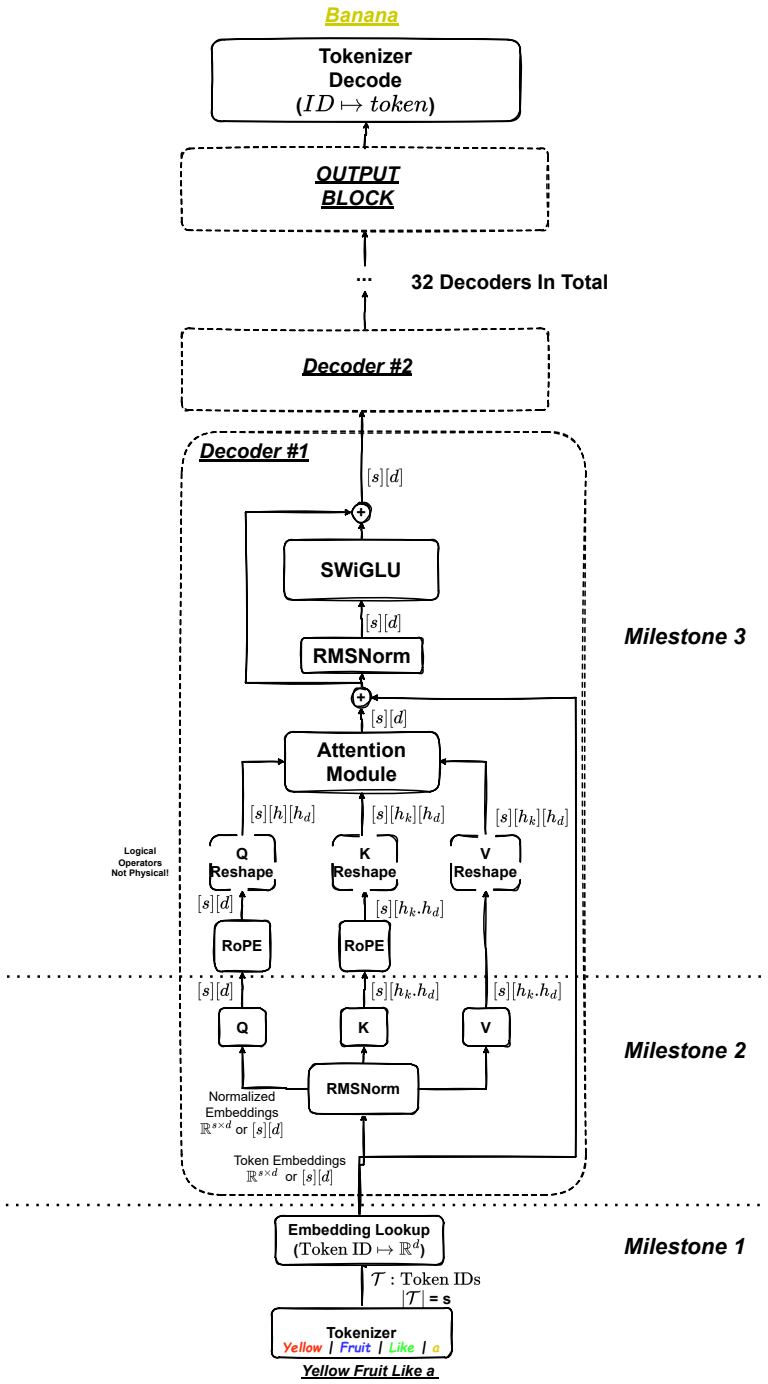

Figure 1: Architecture of `Llama-3-8B-Instruct`. Your implementation in Part 2 completes the full decoder block and produces one new token autoregressively.

2.2 Suggested Guideline

For the RMSNorm kernel, implement the row sum-of-squares accumulation using a parallel reduction, exactly as discussed in class. Assign one thread block per row of the input matrix. Each thread loads one or more elements, computes the squared values, and contributes to the block-level sum via shared memory. Once the reduction is complete, a single thread computes $\text{RMS}(x)$, broadcasts it through shared memory, and each thread writes its scaled and γ-weighted output. For a detailed walkthrough of this reduction pattern, see the Reduction chapter of Programming Massively Parallel Processors [1]. Note that ε is added inside the square root (not to the result) to prevent division by zero when the row has near-zero magnitude.

For Q, K, V, the weight matrices W_Q, W_K, W_V were already loaded in Milestone 1 as `q_proj.weight`, `k_proj.weight`, and `v_proj.weight` under each layer's `self_attn` module. Issue three sequential matmul calls. These shapes exercise your matmul at $(s \times d) \cdot (d \times h \cdot h_d)$ and its variants; make sure your kernel handles non-square operands correctly.

Verify each operator independently before moving to Milestone 3. Errors in RMSNorm or the projections will propagate silently into attention.

2.3 Bonus Thought Experiments & Questions[1]

- Before you run Nsight Compute (`ncu`), predict: is your RMSNorm kernel memory-bandwidth-bound or compute-bound? Estimate its arithmetic intensity in FLOPs per byte, assuming all data comes from HBM. Then profile it and compare your prediction to the measured roofline position. Do the same for your matmul kernel. Are the results what you expected?

- Your RMSNorm kernel reads each input row twice from HBM: once to accumulate the sum of squares and once to normalize and scale. What is the total HBM traffic per row in bytes? If you could restructure the kernel to keep the row in shared memory between both passes, how much HBM traffic would you save? What prevents a naive implementation from doing this for very long rows?

- The three Q, K, V projections read the same input matrix $X_{\text{norm}} \in R^{s \times d}$ from HBM in three separate kernel launches. What is the total HBM read traffic for X_{norm} across all three launches? If X_{norm} fit entirely in the L2 cache after the first read, how would the effective memory traffic change? Given the L2 size on your GPU, does X_{norm} actually fit?

3 Milestone 3: Attention, FFN, and End-to-End Inference

[1] Extra bonus of 5% of the project grade per milestone.

3.1 Desired Functionality

This milestone completes the decoder block and assembles the full inference pipeline. You will implement RoPE, Grouped Query Attention (GQA) with causal masking, the attention output projection, residual connections, the SwiGLU feed-forward network, the final output layer, and one step of autoregressive token generation.

Reshape Q, K, V. Before applying positional encodings or attention, reinterpret the flat token-dimension layout into per-head views:

$$Q \rightarrow R^{h\times s\times h_d}, \quad K \rightarrow R^{h_k\times s\times h_d}, \quad V \rightarrow R^{h_k\times s\times h_d}$$

This is a logical reshape of the last dimension into $(h,\, h_d)$ or $(h_k,\, h_d)$ respectively, and requires no data movement.

Rotary Positional Embeddings (RoPE). RoPE encodes position by rotating pairs of dimensions in Q and K in-place. For token position $p \in \{0, \ldots, s-1\}$ and dimension pair index $i \in \{0, \ldots, h_d/2 - 1\}$, define the angle:

$$\theta_i = \frac{1}{500000^{\,2i/h_d}}$$

Llama 3 uses base 500000, not 10000 as in the original RoPE paper. The rotation pairs dimension i with dimension $i + h_d/2$ (the first half of the head vector rotated with the second half, not interleaved even/odd indices):

$$\begin{pmatrix} q'_i \\ q'_{i+h_d/2} \end{pmatrix} = \begin{pmatrix} \cos(p\theta_i) & -\sin(p\theta_i) \\ \sin(p\theta_i) & \cos(p\theta_i) \end{pmatrix} \begin{pmatrix} q_i \\ q_{i+h_d/2} \end{pmatrix}$$

The same rotation is applied independently to every head of K.

Grouped Query Attention (GQA) with Causal Masking. In class we covered the standard Multi-Head Attention (MHA) formulation, where the number of Q, K, and V heads is identical. The model used in this project employs a different mechanism called Grouped Query Attention (GQA). In GQA, there are only $h_k = 8$ distinct K and V heads shared across the $h = 32$ query heads. The query heads are divided into h_k groups of $h/h_k = 4$ heads each, and all query heads in the same group share one K head and one V head. For query head i, the corresponding KV head index is:

$$g = \left\lfloor \frac{i}{h/h_k} \right\rfloor$$

Compared to MHA, GQA reduces the K and V memory by a factor of $h/h_k = 4$ while preserving full query expressiveness.

For each query head i with $Q_i \in R^{s\times h_d}$, $K_g \in R^{s\times h_d}$, $V_g \in R^{s\times h_d}$, compute scaled dot-product scores:

$$S_i = \frac{Q_i\, K_g^\top}{\sqrt{h_d}} \in R^{s\times s}$$

Apply a **causal mask**: for each position pair (p, q) with $q > p$, set $S_i[p, q] = -\infty$ so that each token only attends to itself and earlier tokens. Then compute attention weights with **numerically stable softmax**. Finding the row maximum first and subtracting it before exponentiating is *required*: it prevents floating-point overflow and is mathematically equivalent to standard softmax. For row p:

$$m_p = \max_{q\le p}\ S_i[p,\, q] \qquad \alpha_i[p,q] = \frac{\exp\big(S_i[p,q] - m_p\big)}{\sum_{k=0}^{p} \exp\big(S_i[p,k] - m_p\big)}$$

Compute the attended output for each head and concatenate:

$$O_i = \alpha_i\, V_g \in R^{s\times h_d}$$

$$O = \mathrm{concat}(O_0,\, \ldots,\, O_{h-1}) \in R^{s\times(h\cdot h_d)}$$

Output Projection and First Residual. Project the concatenated head outputs back to d using the output weight matrix, then add the pre-attention input as a residual:

$$\text{attn\_out} = O\, W_O^\top, \qquad \text{attn\_out} \in R^{s\times d}, \quad W_O \in R^{d\times(h\cdot h_d)}$$

$$X \ \leftarrow\ X + \text{attn\_out}$$

Feed-Forward Network (SwiGLU) and Second Residual. Apply a second RMSNorm (weight `post_attention_layernorm.weight`) to the updated X to obtain $X_{\text{norm}} \in R^{s\times d}$, then pass it through the SwiGLU feed-forward network to produce ffn_out $\in R^{s\times d}$ (full operator breakdown in the Suggested Guideline). Apply the second residual:

$$X \ \leftarrow\ X + \text{ffn\_out}$$

This completes one decoder block. Repeat for all 32 layers as shown in Figure 1, using each layer's own weight matrices.

Output Layer. After all 32 decoder blocks, apply a final RMSNorm (weight `model.norm.weight`):

$$X_{\text{out}} = \mathrm{RMSNorm}(X) \in R^{s\times d}$$

Since only the next token is needed, extract the last token's representation and compute logits over the vocabulary with the language model head (weight `lm_head.weight`, shared with the embedding table in Llama 3):

$$x_{\text{last}} = X_{\text{out}}[\, s-1, :\,] \in R^d$$

$$\text{logits} = x_{\text{last}}\, W_{\text{lm}}^{\top}, \qquad \text{logits} \in R^V, \quad W_{\text{lm}} \in R^{V \times d}$$

For greedy decoding, select the most probable token:

$$\text{token\_id}_{\text{new}} = \arg\max_v \ \text{logits}[v]$$

Decode this token ID to text using your tokenizer's `decode()` function. Since there is no KV caching, the entire sequence is reprocessed on each generation step; this is expected and correct for this project.

1. **Step 1: RoPE.** Implement a CUDA kernel that applies rotary positional embeddings in-place to Q and K using the rotation formula above.
2. **Step 2: Attention.** Implement the GQA attention computation: scaled dot-product scores, causal mask, numerically stable row-wise softmax, and weighted sum over V.
3. **Step 3: Output projection.** Use your matmul kernel to apply W_O to the concatenated attention output $O \in R^{s \times (h \cdot h_d)}$.
4. **Step 4: Residual add.** Implement an elementwise addition kernel for residual connections. This same kernel is reused for both residuals in the decoder block.
5. **Step 5: FFN, decoder block, and 32-layer loop.** Implement the SwiGLU feed-forward network (see Suggested Guideline for the full operator breakdown). Once the FFN is working, connect all components into a single decoder block in order: RMSNorm → QKV projections → RoPE → Attention → Output projection → Residual → RMSNorm → FFN → Residual. Then wrap the decoder block in a loop and execute it 32 times, using the correct weight matrices for each layer.
6. **Step 6: Output layer and token generation.** Apply the final RMSNorm, extract the last-token vector, and compute logits via W_{lm}. Select the argmax token, decode it with your tokenizer, and verify the result against `reference.py`.

3.2 Suggested Guideline

RoPE. Pre-compute $\cos(p\theta_i)$ and $\sin(p\theta_i)$ for all positions $p \in \{0, \ldots, s-1\}$ and all pairs $i \in \{0, \ldots, h_d/2 - 1\}$ before the forward pass and store them on the GPU. Your kernel then reads these tables directly. For each head vector

$q \in R^{h_d}$, the first half $q[:h_d/2]$ and second half $q[h_d/2:]$ form the rotation pair. This matches the `rotate_half` convention in the HuggingFace implementation (see `reference.py`).

Attention. A workable first approach is a loop over all $h = 32$ query heads in your C++ controller, launching CUDA kernels for each head's score computation, mask application, softmax, and weighted sum. To apply the causal mask without branching, add a large negative value such as -10^6 (acting as $-\infty$) to all positions where $q > p$, then proceed with softmax normally. In your softmax kernel, first find the row maximum, subtract it elementwise, exponentiate, then divide by the sum. This order is required. For GQA, compute $g = i \;/\; (h/h_k)$ (integer division) inside your loop to select the correct K and V slice for each query head i.

SwiGLU. The FFN decomposes into three matmuls and one elementwise kernel. Given $X_{\text{norm}} \in R^{s \times d}$:

$$\begin{aligned} \text{gate} &= X_{\text{norm}}\, W_{\text{gate}}^{\top} \in R^{s \times d_{\text{ff}}}, & W_{\text{gate}} &\in R^{d_{\text{ff}} \times d} \\ \text{up} &= X_{\text{norm}}\, W_{\text{up}}^{\top} \in R^{s \times d_{\text{ff}}}, & W_{\text{up}} &\in R^{d_{\text{ff}} \times d} \\ H &= \text{SiLU}(\text{gate}) \odot \text{up} \in R^{s \times d_{\text{ff}}} \\ \text{ffn\_out} &= H\, W_{\text{down}}^{\top} \in R^{s \times d}, & W_{\text{down}} &\in R^{d \times d_{\text{ff}}} \end{aligned}$$

where $\text{SiLU}(z) = z \cdot \sigma(z) = z\,(1+e^{-z})^{-1}$ is applied elementwise and $d_{\text{ff}} = 14336$. Implement the $\text{SiLU}(\text{gate}) \odot \text{up}$ step as a simple CUDA kernel with one thread per element. The three matmuls reuse your existing kernel. The corresponding weight names in the checkpoint are `mlp.gate_proj.weight`, `mlp.up_proj.weight`, and `mlp.down_proj.weight`.

Residual add. One thread per element is sufficient. Reuse this kernel for both residuals in the decoder block.

Output layer. $W_{\text{lm}} \in R^{V \times d}$ with $V = 128256$ is the largest weight in the model. Since you only need logits for the single last token, this reduces to a matrix-vector product $(1 \times d) \cdot (d \times V)$. Your existing matmul handles this; a dedicated GEMV kernel is an optional optimization.

The most common source of bugs in this milestone is confusing the flat layout $s \times (h \cdot h_d)$ with the per-head layout $h \times s \times h_d$. Add shape assertions and verify each sub-step against `reference.py` before assembling the full decoder block.

3.3 Bonus Thought Experiments & Questions[2]

- Profile one complete forward pass with NVIDIA Nsight Systems (`nsys`). Look at the kernel execution timeline across all 32 layers and identify which operator dominates wall time. Then double s and profile again. Which operator grows faster with sequence length and why? What does this tell you about the practical bottleneck of inference without KV caching?

[2] Extra bonus of 5% of the project grade per milestone.

- The attention score matrix $S_i \in R^{s \times s}$ is materialized in GPU memory for each of the $h = 32$ heads. What is the total memory required to hold all 32 score matrices at once for $s = 512$ in FP32? How does this scale with s? At what sequence length does the score matrix memory exceed the size of all model weights combined?
- Without KV caching, generating T new tokens from a prompt of length s_0 requires T full forward passes on sequences of length s_0+1, s_0+2, …, s_0+T. How does the total number of floating-point operations scale with T compared to a system that caches K and V and only processes one new token per step? What does this imply about the latency of your implementation as T grows large?

4 Reminders and Common Pitfalls

This section collects things that are easy to miss or get subtly wrong. Read it before you start debugging.

Softmax must be numerically stable. Do not implement softmax as $\exp(S)/\sum \exp(S)$ directly. For even moderately large scores, $\exp(S_{ij})$ overflows to `inf` and the result is `NaN`. Always subtract the row maximum first: compute $m_p = \max_q S_i[p, q]$, then exponentiate $S_i[p, q] - m_p$, then normalize. This is not optional! your attention will silently produce garbage without it.

RoPE pairs first half with second half, not even with odd. A common mistake is to rotate dimension pairs (q_0, q_1), (q_2, q_3), … (interleaved even/odd indices). Llama 3 pairs dimension i with dimension $i + h_d/2$, i.e., the first 64 elements of a head rotate with the last 64 elements. Using the wrong pairing produces outputs that look numerically reasonable but are silently wrong.

RoPE base is 500000, not 10000. The original RoPE paper uses base 10000. Llama 3 uses 500000. Using the wrong base shifts the frequency scale of positional encodings and will cause incorrect output even if everything else is correct.

Two separate RMSNorm weight vectors per layer. Each decoder block has two RMSNorm operations with two distinct learned weight vectors: `input_layernorm.weight` before the attention sub-block and `post_attention_layernorm.weight` before the FFN. Using the same weight for both is a common loading mistake.

Do not forget the γ scaling in RMSNorm. RMSNorm normalizes by the RMS and then multiplies elementwise by the learned scale γ. Stopping after normalization and skipping the γ multiplication gives a kernel that passes a rough sanity check but produces wrong outputs once integrated into the full model.

Weights are stored transposed in the checkpoint. The projection weights W_Q, W_K, W_V, W_O, and the FFN weights are stored in the HuggingFace checkpoint with shape $(\text{out\_dim}, \text{in\_dim})$, so your matmul always computes $XW^\top$, not XW. Forgetting the transpose is one of the most common shape bugs.

`lm_head` shares weights with the embedding table. In Llama 3, `lm_head.weight` and `model.embed_tokens.weight` are the same tensor. Load it once and reuse it; do not allocate a separate buffer for the output projection.

Extract only the last token before the lm_head. Compute the lm_head projection only for the last row of X_{out}, not the full $s \times d$ matrix. Projecting all s rows onto a $V = 128256$ vocabulary is wasteful and may cause memory issues for longer sequences.

Causal mask applies across all sequence positions. Your score matrix $S_i \in R^{s \times s}$ has a full upper triangle that must be masked to $-\infty$ (or a large negative approximation) before softmax. Do not apply the mask only to the diagonal or only to the last row; every row p must be masked at all columns $q > p$.

References

[1] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 4th edition, 2022.