

AHMED ADAWY TECH CAPSULES

LLM INFERENCE ENGINE ARCHITECTURE

Custom Kernels, PagedAttention,
and Memory-Efficient Serving



AHMED ADAWY

LLM INFERENCE ENGINE ARCHITECTURE

Custom Kernels, PagedAttention, and Memory-Efficient Serving

By Ahmed Adawy

2026 Edition

Introduction

The LLM landscape has shifted. Training massive models is only the beginning. The real battleground for performance and scalability now lies in Inference. Running these models instantly and cost-effectively for millions of users demands engineering mastery of the software-hardware stack.

This book moves beyond theory to the absolute metal of Inference Engineering. It is a rigorous, practical guide on optimizing VRAM footprint, reducing latency, and maximizing throughput using Custom CUDA & Triton Kernels, PagedAttention, and advanced Quantization techniques. If you are building high-performance serving systems, this is your engineering manual.

Table of Contents

Part 1: Foundations and the Physics of LLM Inference

Chapter 1: The Inference Lifecycle

- An Inference-Centric View of the Transformer
- Prefill Phase vs. Decode Phase
- Data Flow: From Input Tokens to Output Logits

Chapter 2: Hardware Landscape and Bottlenecks

- The Roofline Model
- Compute-Bound vs. Memory-Bandwidth Bound
- Why LLM Inference is Primarily a Memory Bandwidth Problem

Part 2: The VRAM Problem - The Main Bottleneck

Chapter 3: Anatomy of the KV Cache

- What is the KV Cache and Why Does It Grow Exponentially?
- Context Length Impact on Memory Footprint
- Memory Fragmentation in VRAM

Chapter 4: Traditional vs. Modern Memory Management

- Naive Contiguous Memory Allocation
- Why Traditional Memory Managers Fail at Scale

Part 3: Memory-Efficient Solutions and PagedAttention Architecture

Chapter 5: PagedAttention in Depth

- Operating System Paging Inspiration
- Building the Block Manager
- Non-Contiguous KV Cache Allocation
- Memory Sharing: Beam Search and Parallel Sampling

Chapter 6: Efficient Attention Kernels

- The Mathematics of FlashAttention (1, 2, 3)
- Why Fused Kernels Drive High Performance
- Integrating PagedAttention with FlashAttention

Part 4: Quantization and Memory-Efficient Data Transfer

Chapter 7: Quantization Systems for Serving

- The Accuracy-Latency Trade-Off
- INT8 Quantization (W8A8 / W8A16)
- Quantizing the KV Cache
- FP8 (E4M3 & E5M2) in Modern Serving Engines

Chapter 8: Advanced Quantization Techniques

- INT4 AWQ and GPTQ Engines
- Symmetric vs. Asymmetric Quantization
- Calibrating Scales and Zeros

Part 5: Serving Engine Architecture and Scheduling

Chapter 9: Request Scheduling and Continuous Batching

- The Limits of Static Batching
- Continuous Batching Under the Hood
- Scheduler Design for Simultaneous Prefill and Decode
- Managing Variable Sequence Lengths

Chapter 10: Multi-GPU Inference

- Tensor Parallelism (TP)
- Pipeline Parallelism (PP)
- Inter-GPU Communication (NVLink & NCCL)
- Efficient Model Loading Strategies

Part 6: Building Custom Kernels

Chapter 11: GPU Programming (CUDA vs. Triton)

- CUDA Fundamentals: Threads, Blocks, and Shared Memory
- Why Triton? Rapid Python GPU Kernel Development
- Authoring Fused Kernels in Triton

Chapter 12: Practical Case Study: Building a Custom Kernel

- Fused Quantized-GEMM Implementation
- Benchmarking vs. CuBLAS and Hardware Limits

Part 7: Production and The Future

Chapter 13: Deployment and Advanced MLOps

- Containerizing the Serving Engine
- Performance Profiling: Measuring Throughput and Latency
- Edge LLM Serving: Challenges and Limits

Chapter 14: Future Trends

- Speculative Decoding Mechanics
- Mixture of Experts (MoE) Inference Strategies
- Hardware-Software Co-Design

CHAPTER 1: THE INFERENCE LIFECYCLE

1.1 An Inference-Centric View of the Transformer Architecture

When studied in academic research or machine learning courses, the Transformer architecture is almost exclusively analyzed through the lens of training. Training focuses on compute-intensive backward passes, loss function optimization, gradient updates, distributed data parallelism, and teacher forcing across static, fixed-length sequence batches. However, serving Large Language Models (LLMs) in production environments requires a fundamentally different paradigm: an inference-centric view.

In an inference environment, model weights are entirely frozen. There are no backward passes, no optimizer states (such as Adam m and v vectors), no activation caching for backpropagation, and no gradient communications across node clusters. The primary operational objective shifts entirely from maximizing statistical convergence per epoch to executing deterministic forward passes with minimal Time-to-First-Token (TTFT), minimal Time-Per-Output-Token (TPOT), maximum VRAM capacity utilization, and maximum request throughput (tokens per second per GPU).

To engineer high-performance serving software, one must dissect the Transformer block not as an abstract deep learning architecture, but as a deterministic series of hardware-bound matrix operations and VRAM access patterns. A modern Decoder-Only Transformer (such as the Llama 3, Mistral, or Qwen architecture) consists of L stacked Transformer layers. Each layer contains two primary computational blocks wrapped in residual connections: Multi-Head / Grouped-Query Attention and a Feed-Forward Network (FFN).

Linear Projection Operations and Attention Variations

At layer l , the input hidden representation vector sequence X in $\mathbb{R}^{(B \times S \times d_{\text{model}})}$ (where B is Batch Size, S is Sequence Length, and d_{model} is the Model Hidden Dimension) undergoes linear transformations to generate Query (Q), Key (K), and Value (V) projections.

In standard Multi-Head Attention (MHA): $Q = X * W_Q$, where W_Q in $\mathbb{R}^{(d_{\text{model}} \times (n_{\text{heads}} * d_{\text{head}}))}$ $K = X * W_K$, where W_K in $\mathbb{R}^{(d_{\text{model}} \times (n_{\text{heads}} * d_{\text{head}}))}$ $V = X * W_V$, where W_V in $\mathbb{R}^{(d_{\text{model}} \times (n_{\text{heads}} * d_{\text{head}}))}$

Here, n_{heads} represents the number of attention heads, and d_{head} is the dimension per head, where $d_{\text{model}} = n_{\text{heads}} * d_{\text{head}}$. In classic MHA, every attention head maintains its own dedicated set of Key and Value vectors. While optimal for model quality, this creates a severe memory bottleneck during inference because the VRAM required to store Key-Value states scales linearly with n_{heads} .

To mitigate this VRAM footprint during serving, modern architectures utilize Multi-Query Attention (MQA) or Grouped-Query Attention (GQA):

- Multi-Query Attention (MQA): All n_{heads} Query heads share a single Key head and a single Value head ($n_{\text{kv_heads}} = 1$). This reduces KV Cache VRAM consumption by a factor of n_{heads} (typically 32x to 64x), but can occasionally degrade reasoning capabilities in complex models.
- Grouped-Query Attention (GQA): Query heads are divided into $n_{\text{kv_heads}}$ groups, where each group of $(n_{\text{heads}} / n_{\text{kv_heads}})$ Query heads shares a single Key head and Value head. For instance, Llama-3-70B uses 64 Query heads and 8 KV heads (an 8:1 ratio). GQA achieves near-MHA statistical accuracy while reducing KV Cache VRAM bandwidth and memory capacity requirements by 8x.

The Mathematical Formulation of Attention Mechanics

Once projected, the Attention calculation per head follows the scaled dot-product formula:

$$\text{Attention}(Q, K, V) = \text{Softmax} \left(\frac{(Q * K^T)}{\sqrt{d_{\text{head}}}} + M \right) * V$$

Where:

- Q in $\mathbb{R}^{(B \times S_q \times d_{\text{head}})}$ represents the Query sequence.
- K in $\mathbb{R}^{(B \times S_{\text{kv}} \times d_{\text{head}})}$ represents the total cached Key sequence.
- V in $\mathbb{R}^{(B \times S_{\text{kv}} \times d_{\text{head}})}$ represents the total cached Value sequence.
- M is the Causal Attention Mask matrix in $\mathbb{R}^{(S_q \times S_{\text{kv}})}$, which enforces autoregressive causality by assigning -infinity to future sequence positions, ensuring token i cannot attend to token j where $j > i$.
- $\sqrt{d_{\text{head}}}$ is the scaling factor that prevents dot products from growing excessively large in high dimensions, which would otherwise drive the Softmax function into regions with vanishingly small gradients.

After computing attention outputs across all heads, the concatenated attention output is projected back to the hidden dimension using the Output Projection matrix W_O in $R^{(d_{\text{model}} \times d_{\text{model}})}$:

$$\text{Output} = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_n) * W_O$$

Feed-Forward Networks (FFN) and SwiGLU Activations

The output of the attention block passes through a normalization layer (typically RMSNorm) and enters the Feed-Forward Network (FFN). Modern open-weights models predominantly use the SwiGLU (Swish Gated Linear Unit) FFN architecture rather than standard GELU MLPs.

The SwiGLU block utilizes three distinct projection matrices per layer:

- Gate Projection (W_{gate} in $R^{(d_{\text{model}} \times d_{\text{ffn}})}$)
- Up Projection (W_{up} in $R^{(d_{\text{model}} \times d_{\text{ffn}})}$)
- Down Projection (W_{down} in $R^{(d_{\text{ffn}} \times d_{\text{model}})}$)

Where d_{ffn} is typically set to $(8/3) * d_{\text{model}}$ (or rounded to the nearest multiple of 256 for optimal GPU memory alignment).

The mathematical formulation for SwiGLU is:

$$\text{FFN}(X) = (\text{SiLU}(X * W_{\text{gate}}) \text{ (elementwise)} (X * W_{\text{up}})) * W_{\text{down}}$$

Where $\text{SiLU}(x) = x * \text{sigmoid}(x)$.

Computational Profile of FFN vs. Attention: The FFN block contains approximately 66.6% (two-thirds) of the Transformer's total parameter count. From an engineering system perspective, FFN layers perform strictly point-wise (token-independent) matrix operations. Unlike Attention, FFN execution requires no cross-token memory retrieval and maintains no historical state. Consequently, FFN execution speed depends purely on weight-streaming bandwidth or raw Matrix Multiplication (GEMM) performance.

Layer Normalization: RMSNorm vs. LayerNorm

Standard Transformers historically utilized Layer Normalization, which computes both mean and variance across the hidden dimension:

$$\text{LayerNorm}(x) = ((x - \mu) / \sqrt{\sigma^2 + \epsilon}) * \gamma + \beta$$

Modern high-performance LLMs replace LayerNorm with Root Mean Square Normalization (RMSNorm). RMSNorm assumes a zero-mean distribution, eliminating the need to calculate μ , thereby reducing memory reads and arithmetic overhead:

$$\text{RMSNorm}(x) = (x / \text{RMS}(x)) * \gamma, \text{ where } \text{RMS}(x) = \sqrt{(1 / d_{\text{model}}) * \sum(x_i^2) + \epsilon)}$$

From a GPU kernel optimization perspective, RMSNorm is an element-wise, reduction-heavy operation. Because it is Memory-Bandwidth Bound, high-performance inference engines always fuse RMSNorm kernels directly into the input projections of the subsequent Linear layers (QKV projection or SwiGLU projection) to prevent redundant round-trips to off-chip VRAM.

1.2 Prefill Phase vs. Decode Phase Deep Dive

The fundamental operational challenge of LLM serving stems from the duality of its workload execution phases. A request submitted to an inference engine does not execute as a static, homogeneous computation. Instead, every single inference request undergoes two distinct operational regimes: the Prefill Phase (Prompt Processing) and the Decode Phase (Autoregressive Token Generation).

The Prefill Phase (Prompt Processing)

When a prompt is submitted to the engine (e.g., a system instruction and user query total 2,048 tokens), the model must process all 2,048 tokens simultaneously to construct the initial context understanding.

1. **Parallel Execution Mechanics:** Because all 2,048 input tokens are known prior to execution, the GPU can compute the activations for all tokens concurrently across its parallel SIMT (Single Instruction, Multiple Threads) compute architecture.
2. **Matrix Multiplication Profile (GEMM):** The projections for Q, K, V, and FFN transformations operate on a 2D activation matrix of shape $[S_{\text{prompt}}, d_{\text{model}}]$. For $S_{\text{prompt}} = 2048$ and $d_{\text{model}} = 4096$, the operations translate to dense General Matrix Multiplications (GEMM):

Operation: $[2048 \times 4096] * [4096 \times 4096] \rightarrow [2048 \times 4096]$

1. **High Arithmetic Intensity:** The Operational (Arithmetic) Intensity of a GEMM operation scales with the sequence length S . For a matrix multiplication of shapes $[M \times K]$ and $[K \times N]$, the total floating-point operations are $2 * M * N * K$, while memory transfers scale with $(MK + KN + M*N)$. When M (Sequence Length S) is large, the ratio of FLOPs to bytes read from VRAM is extremely high.
2. **Hardware Bottleneck: Compute-Bound.** The execution time of the Prefill phase is bounded by the raw arithmetic TFLOPS capacity of the GPU's Tensor Cores. During Prefill, GPU core utilization approaches 80%–90% of theoretical maximum capacity.
3. **Key System Outputs:** The Prefill phase calculates the Key and Value vectors for all 2,048 input tokens across all L layers and writes them into VRAM (the KV Cache). It also computes the Logits for the 2,048th token to sample and predict the very first output token (Token 2,049).

The Decode Phase (Autoregressive Generation)

Once the first output token is generated, the inference engine transitions into the Decode Phase. The model generates subsequent tokens sequentially, one token per iteration step. To generate Token N, the model must consume Token N-1 alongside the previously computed KV Cache vectors of tokens 1 through N-1.

1. **Sequential Execution Mechanics:** Token generation is strictly autoregressive and non-parallelizable across time. Token N cannot be calculated until Token N-1 has fully traversed all L layers of the network and passed through the sampling pipeline.
2. **Matrix-Vector Profile (GEMV):** Because only a single query token is being processed per active sequence at step t, the input activation matrix collapses from a 2D matrix [S, d_model] to a 1D vector [1, d_model] (or [B, d_model] for a batch of B sequences). The linear projections degrade from General Matrix Multiplications (GEMM) to General Matrix-Vector Multiplications (GEMV):

Operation: $[1 \times 4096] * [4096 \times 4096] \rightarrow [1 \times 4096]$

1. **Low Arithmetic Intensity:** To process this single token, the GPU must stream the entire 140 GB of model weights (for a 70B FP16 model) from off-chip High Bandwidth Memory (HBM) into on-chip SRAM registers, solely to perform 2 FLOPs of calculation per parameter byte. The Arithmetic Intensity drops to approximately 1.0 FLOP/Byte.
2. **Hardware Bottleneck: Memory-Bandwidth Bound.** The GPU compute units complete the mathematical calculations almost instantaneously and spend over 95% of their execution clock cycles completely idle (stalled), waiting for parameter bytes and KV Cache vectors to arrive across the memory bus.
3. **Key System Outputs:** For every step, the engine calculates 1 new Key vector and 1 new Value vector per layer, appends them to the expanding KV Cache, outputs 1 token, and loops back for the next iteration step.

System Latency Metrics: TTFT vs. TPOT

Because of this dual-phase lifecycle, LLM inference performance cannot be measured using a single unified latency metric. Serving systems evaluate two primary latency targets:

1. **Time-to-First-Token (TTFT):** The elapsed time between when a user submits a prompt and when the client receives the first generated token. TTFT is dominated entirely by the execution latency of the Prefill Phase. It reflects how fast the serving engine can ingest long context windows and compute initial KV Cache states.
2. **Time-Per-Output-Token (TPOT):** The average time required to generate each subsequent token during the streaming response (also referred to as Inter-Token Latency or ITL). TPOT is dominated entirely by the Decode Phase. A low TPOT ensures a smooth, human-readable real-time text generation stream.

3. End-to-End Latency (E2E): $E2E\ Latency = TTFT + (TPOT * N_{generated_tokens})$

Optimization Conflict: Optimizing for TTFT requires maximizing Tensor Core compute throughput (e.g., larger prefill batching, higher precision GEMM kernels). Optimizing for TPOT requires maximizing memory bus throughput and minimizing VRAM bandwidth overhead (e.g., quantization, continuous batching, PagedAttention). High-performance serving engines must dynamically balance hardware resources between these two competing phases.

Comparative Matrix: Prefill Phase vs. Decode Phase

Dimension: Workload Pattern Prefill Phase: Parallel execution across all prompt tokens simultaneously. Decode Phase: Autoregressive, step-by-step sequential execution (1 token at a time).

Dimension: Mathematical Operation Prefill Phase: General Matrix Multiplication (GEMM). Decode Phase: General Matrix-Vector Multiplication (GEMV).

Dimension: Arithmetic Intensity Prefill Phase: High (Typically > 100 FLOPs/Byte, scaling with sequence length). Decode Phase: Extremely Low (Typically 1 to 5 FLOPs/Byte at small batch sizes).

Dimension: Hardware Bottleneck Prefill Phase: Compute-Bound (Tensor Core TFLOPS ceiling). Decode Phase: Memory-Bandwidth Bound (HBM Bus Bandwidth ceiling).

Dimension: GPU Core Utilization Prefill Phase: High (80% – 95% ALU saturation). Decode Phase: Extremely Low ($< 5\%$ ALU saturation; memory bus is 100% saturated).

Dimension: VRAM Access Focus Prefill Phase: Writes new Key-Value tensors to VRAM. Decode Phase: Reads all past Key-Value tensors + streams full Model Weights per step.

Dimension: Causal Masking Prefill Phase: Requires full Lower-Triangular Causal Mask matrix $[S \times S]$. Decode Phase: No mask matrix needed; Query vector cross-attends to 1D KV sequences.

Dimension: Latency Impact Prefill Phase: Dictates Time-to-First-Token (TTFT). Decode Phase: Dictates Time-Per-Output-Token (TPOT) / Inter-Token Latency.

1.3 End-to-End Data Flow & Memory Lifecycle

To build or optimize an inference engine, one must trace the precise hardware-level lifecycle of data as it transitions from raw string requests on the host network down to register-level calculations on the GPU silicon, and back out to streaming client sockets.

Phase 1: Ingestion, Tokenization, and Host-to-Device Memory Transfer

1. Network Ingestion: The user prompt arrives over HTTP/gRPC as an uncompressed UTF-8 string at the serving framework's host process (Python/C++ server running on the CPU).
2. Tokenization: The host CPU executes tokenization using Byte-Pair Encoding (BPE) or WordPiece algorithms. The UTF-8 string is converted into a 1D tensor of integer Token IDs in host system RAM (CPU memory). Example: "Optimize CUDA kernels" -> [1204, 45201, 31822]
3. Host-to-Device Transfer (PCIe Overhead): The Token ID integer array must be transferred from host system RAM to GPU VRAM across the PCIe bus (PCIe Gen 4 x16 provides ~31.5 GB/s; PCIe Gen 5 provides ~63 GB/s). Engineering Constraint: Synchronous transfers (e.g., standard host-to-device memory copies) block CPU execution threads and introduce severe microsecond stalls. High-performance engines allocate Pinned Host Memory (Page-Locked Memory via `cudaHostAlloc`) and launch asynchronous transfers (`cudaMemcpyAsync`) over dedicated CUDA Streams to completely overlap host tokenization with GPU execution cycles.

Phase 2: Embedding Lookup and Rotary Position Embedding (RoPE)

1. Embedding Lookup Kernel: Once the Token ID tensor resides in VRAM, the GPU launches an Embedding Lookup CUDA kernel. The token integers serve as index pointers into a large embedding matrix W_{embed} in $R^{(V \times d_{\text{model}})}$, where V is the Vocabulary Size (e.g., 128,256 for Llama 3). The kernel fetches the corresponding d_{model} float16/bfloat16 vectors from VRAM and constructs the activation tensor X_0 in $R^{(B \times S \times d_{\text{model}})}$.
2. Rotary Position Embeddings (RoPE): Unlike early Transformer models that added static positional vectors to input embeddings, modern models apply Rotary Position Embeddings (RoPE) dynamically within every attention layer. RoPE encodes absolute position by rotating the Query and Key vector pairs in a 2D complex plane using position-dependent rotation matrices.

For a 2D vector slice $[x_1, x_2]$ at sequence position m and frequency ω_i :

$$R_{\text{Theta},m}^2 * [x_1, x_2]^T = [x_1 * \cos(m\omega_i) - x_2 * \sin(m\omega_i), x_1 * \sin(m\omega_i) + x_2 * \cos(m\omega_i)]$$

Hardware Implementation Note: Executing RoPE as an isolated GPU kernel requires reading Q and K matrices from VRAM, modifying them in registers, and writing them back to VRAM. Because this is Memory-Bandwidth Bound, elite engines fuse RoPE directly into the QKV projection GEMM output kernel or the Attention kernel, eliminating two full round-trips to off-chip VRAM.

Phase 3: The Deep-Dive Layer Execution Loop

The activation tensor X passes sequentially through all L Transformer layers. Within layer l :

Step A: RMSNorm Activations are normalized in SRAM registers. The normalized activations are retained for linear projection.

Step B: QKV Linear Projection The fused GEMM kernel computes Q, K, and V projections simultaneously: $[Q, K, V] = X_{\text{norm}} * W_{\text{QKV}}$ Where W_{QKV} fuses W_Q , W_K , and W_V into a single contiguous weight array in VRAM to maximize GEMM operational efficiency.

Step C: KV Cache Allocation & Append The newly calculated Key and Value vectors for the current step are written directly into designated KV Cache memory buffers in VRAM.

Step D: Attention Score Computation & Softmax

1. Dot Product: Q is multiplied by all past cached K vectors: $\text{Score} = Q * K^T / \sqrt{d_{\text{head}}}$.
2. Masking: During Prefill, causal masking sets upper-triangular elements to -infinity.
3. Online Softmax: Softmax converts raw scores to attention probabilities. To avoid numerical overflow and eliminate multi-pass VRAM reads, inference engines use Online Softmax (computing running max and running sum-exp within a single kernel pass).
4. Context Aggregation: Probability matrix multiplied by cached Value vectors: $\text{Context} = \text{Softmax_Scores} * V$.

Step E: Output Projection & Residual Connection The context vectors pass through the Output Linear Projection W_O . The output is added back to the original input activation tensor via an element-wise residual addition in VRAM: $X_{l+1} = X_l + (\text{Context} * W_O)$

Step F: SwiGLU FFN Block X_{l+1} undergoes RMSNorm, followed by the SwiGLU gate/up projections, SiLU activation, down projection, and a final residual connection addition: $X_{\text{layer_out}} = X_{l+1} + \text{SwiGLU}(\text{RMSNorm}(X_{l+1}))$

Phase 4: Unembedding, Logit Generation, and Sampling

1. Final Normalization: After exiting layer L, the final activations X_L in $R^{(B \times 1 \times d_{\text{model}})}$ pass through a final RMSNorm layer.
2. Unembedding Projection (LM Head): The activations are multiplied by the Unembedding Weight Matrix W_{unembed} in $R^{(d_{\text{model}} \times V)}$: $\text{Logits} = \text{RMSNorm}(X_L) * W_{\text{unembed}}$

The output Logits tensor in $R^{(B \times V)}$ contains unnormalized log-probabilities for every token in the vocabulary. For a model with $V = 128,256$, this single matrix multiplication requires projecting 4,096 dimensions to 128,256 dimensions, generating a 256 KB logit vector per sequence.

1. Logit Processing & Transformations: Before sampling, the raw logits undergo sequential transformations on the GPU:

- Temperature Scaling: $\text{Logits} = \text{Logits} / T$. (Higher T increases entropy/randomness; lower T sharpens peak probabilities).
- Repetition Penalty: Applies a scaling factor to logits of tokens that already appear in the prompt/generated sequence context to prevent infinite looping.
- Top-K Filtering: Sets logits below the K highest values to $-\infty$.
- Top-P (Nucleus) Filtering: Sorts logits in descending order, computes cumulative probabilities, and zeroes out all logits beyond the cumulative probability threshold P (e.g., $P = 0.9$).
- Min-P Filtering: Filters out tokens whose scaled probability is less than $\text{Min_P} * \text{Max_Probability}$.

1. Token Sampling (Greedy vs. Categorical):

- Greedy Search: Executes a argmax operation on the logit vector to select the token ID with the absolute highest probability. Extremely fast, fully deterministic.
 - Categorical / Stochastic Sampling: Converts processed logits to a multinomial probability distribution via Softmax, samples a random floating-point value u in $[0, 1)$ from a GPU Random Number Generator (cuRAND), and selects the corresponding token interval.
1. Host Notification and Streaming Output: The newly sampled Token ID (e.g., Token ID 9412) is copied back to CPU host memory. The host CPU tokenizer decodes the integer to its corresponding text string (e.g., "kernel") and streams the text chunk over gRPC/WebSocket to the client.

Simultaneously, without waiting for client ACK, the selected Token ID is instantly appended to the GPU input buffer to initiate the next Decode Phase iteration.

1.4 Concrete Numerical Case Study & Memory Budget Analysis

To consolidate these concepts, we construct an exact mathematical memory and compute profile for serving two real-world model architectures on modern hardware: Llama-3-8B and Llama-3-70B.

Model Architectural Parameters

Parameter: Parameter Count (P) Llama-3-8B: 8.03 Billion Llama-3-70B: 70.6 Billion

Parameter: Hidden Dimension (d_{model}) Llama-3-8B: 4,096 Llama-3-70B: 8,192

Parameter: Number of Layers (L) Llama-3-8B: 32 Llama-3-70B: 80

Parameter: Query Attention Heads (n_{heads}) Llama-3-8B: 32 Llama-3-70B: 64

Parameter: KV Attention Heads ($n_{\text{kv_heads}}$) Llama-3-8B: 8 (GQA 4:1 ratio) Llama-3-70B: 8 (GQA 8:1 ratio)

Parameter: Head Dimension (d_{head}) Llama-3-8B: 128 Llama-3-70B: 128

Parameter: FFN Intermediate Dimension (d_{ffn}) Llama-3-8B: 14,336 Llama-3-70B: 28,672

Parameter: Vocabulary Size (V) Llama-3-8B: 128,256 Llama-3-70B: 128,256

Calculation 1: Static Weight Memory Footprint

Model weights must reside permanently in VRAM. The required static memory depends on parameter count and numerical precision (Bytes per parameter):

- FP16 / BF16 (16-bit): 2 Bytes / parameter
- INT8 / FP8 (8-bit): 1 Byte / parameter
- INT4 (4-bit): 0.5 Bytes / parameter

For Llama-3-8B in FP16 precision: $\text{Memory_Weights} = 8.03 \times 10^9 \text{ params} * 2 \text{ Bytes} = 16.06 \text{ GB}$

For Llama-3-70B in FP16 precision: $\text{Memory_Weights} = 70.6 \times 10^9 \text{ params} * 2 \text{ Bytes} = 141.2 \text{ GB}$

System Takeaway: A 70B model in FP16 cannot fit on a single NVIDIA A100/H100 GPU (80 GB VRAM). It requires a multi-GPU Tensor Parallel setup across at least 2 GPUs (or downcasting weights to INT8/INT4 to fit on 1 GPU).

Calculation 2: KV Cache Memory Footprint Per Token

For a single token in a single active sequence, the engine must store Key and Value vectors across all L layers.

Formula for KV Cache size per token (in Bytes): $\text{Bytes_per_token} = 2 (\text{Layers } L) * (n_{\text{kv_heads}} * d_{\text{head}}) * \text{Bytes_per_element}$

For Llama-3-8B (FP16 precision, 2 bytes/element):

- $n_{\text{kv_heads}} = 8$
- $d_{\text{head}} = 128$
- Element size = $8 * 128 = 1,024$ elements per layer for Keys, and 1,024 for Values.
- Total elements per layer = 2,048 elements.
- Bytes per layer = $2,048 * 2 \text{ Bytes} = 4,096 \text{ Bytes (4 KB)}$.
- Across all 32 layers = $32 * 4 \text{ KB} = 128 \text{ KB per token}$.

For Llama-3-70B (FP16 precision, 2 bytes/element):

- $n_{\text{kv_heads}} = 8$
- $d_{\text{head}} = 128$
- Bytes per layer = $2 * (8 * 128) * 2 = 4,096 \text{ Bytes (4 KB)}$.

- Across all 80 layers = $80 * 4 \text{ KB} = 320 \text{ KB}$ per token.

Scaling to Batch Sizes and Context Lengths: Suppose an inference server handles a concurrent batch of $B = 64$ requests, each processing a context length of $S = 4,096$ tokens using Llama-3-70B in FP16:

$$\begin{aligned} \text{Total_KV_Cache_Size} &= \text{Batch_Size} * \text{Sequence_Length} * \text{Bytes_per_token} \\ \text{Total_KV_Cache_Size} &= 64 * 4,096 * 320 \text{ KB} \\ \text{Total_KV_Cache_Size} &= 262,144 * 320 \text{ KB} = 83,886,080 \text{ KB} \approx 83.88 \text{ GB} \end{aligned}$$

System Takeaway: At scale, the dynamic KV Cache memory requirement (83.88 GB) exceeds the entire static weight footprint of the model itself! This proves why managing KV Cache memory is the central engineering bottleneck in LLM serving.

Calculation 3: Floating-Point Operations (FLOPs) Per Token

To compute the arithmetic workload during inference:

1. Prefill Phase (Prompt of length S tokens): $\text{FLOPs}_{\text{prefill}} \approx 2 * \text{Parameter_Count} * S + 2 * L * n_{\text{heads}} * d_{\text{head}} * S^2$

For Llama-3-8B with a 4,096-token prompt:

- Linear Layer FLOPs = $2 * (8.03 * 10^9) * 4,096 \approx 65.78 \text{ TeraFLOPs}$.
 - Attention Score FLOPs = $2 * 32 * 32 * 128 * (4,096)^2 \approx 0.88 \text{ TeraFLOPs}$.
 - Total Prefill FLOPs $\approx 66.66 \text{ TFLOPs}$.
1. Decode Phase (Generating 1 token step): $\text{FLOPs}_{\text{decode}} \approx 2 * \text{Parameter_Count} * 1$ (Attention matrix-vector FLOPs are negligible per step).

For Llama-3-8B per decode step:

- Decode FLOPs = $2 * (8.03 * 10^9) = 16.06 \text{ GigaFLOPs}$.

Hardware Throughput Comparison: An NVIDIA H100 GPU provides 989 TFLOPS of FP16 compute capacity and 3.35 TB/s memory bandwidth.

- Theoretical Prefill Execution Time ($66.66 \text{ TFLOPs} / 989 \text{ TFLOPs/s}$) ≈ 67.4 milliseconds. (Compute-Bound)
- Theoretical Decode Step Time ($16.06 \text{ GB weight transfer} / 3,350 \text{ GB/s bandwidth}$) ≈ 4.79 milliseconds per token (~ 208 tokens/sec). (Memory-Bandwidth Bound)

1.5 PyTorch Reference Implementation: Analyzing Inefficiencies

To ground these theoretical concepts in code, the following Python reference implementation demonstrates a naive, unoptimized Transformer generation loop. This

script highlights explicitly where traditional Python/PyTorch code breaks down in production serving environments.

Naive PyTorch Generation Loop Code:

```
import torch
import torch.nn as nn
import time

class NaiveTransformerDecodeStep(nn.Module):
    def __init__(self, num_layers, d_model, num_heads, num_kv_heads, head_dim, vocab_size):
        super().__init__()
        self.num_layers = num_layers
        self.d_model = d_model
        self.num_heads = num_heads
        self.num_kv_heads = num_kv_heads
        self.head_dim = head_dim

    # Linear layers for Q, K, V projections and FFN
    self.q_proj = nn.Linear(d_model, num_heads * head_dim, bias=False)
    self.k_proj = nn.Linear(d_model, num_kv_heads * head_dim, bias=False)
    self.v_proj = nn.Linear(d_model, num_kv_heads * head_dim, bias=False)
    self.out_proj = nn.Linear(num_heads * head_dim, d_model, bias=False)
    self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

    def forward_decode_step(self, x_single_token, layer_kv_caches):
        """ Executes a single decode step for 1 token across layers.
        x_single_token shape: [Batch, 1, d_model] """
        batch_size = x_single_token.shape[0]
        current_activations = x_single_token

        for layer_idx in range(self.num_layers):
            # 1. Project Query, Key, Value for single token
            q = self.q_proj(current_activations) # [B, 1, n_heads * d_head]
            k = self.k_proj(current_activations) # [B, 1, n_kv_heads * d_head]
            v = self.v_proj(current_activations) # [B, 1, n_kv_heads * d_head]

            # 2. Reshape projections for multi-head attention
            q = q.view(batch_size, 1, self.num_heads, self.head_dim).transpose(1, 2)
            k = k.view(batch_size, 1, self.num_kv_heads, self.head_dim).transpose(1, 2)
            v = v.view(batch_size, 1, self.num_kv_heads, self.head_dim).transpose(1, 2)

            # 3. Dynamic Tensor Concatenation for KV Cache (HIGHLY INEFFICIENT)
            past_k, past_v = layer_kv_caches[layer_idx]
            if past_k is not None:
                # Allocates NEW VRAM memory every single token step!
                k_cat = torch.cat([past_k, k], dim=2)
                v_cat = torch.cat([past_v, v], dim=2)
            else:
                k_cat, v_cat = k, v

            # Update KV cache handle
            layer_kv_caches[layer_idx] = (k_cat, v_cat)

            # 4. Expand KV heads for Grouped-Query Attention (GQA)
            repeat_factor = self.num_heads // self.num_kv_heads
            k_expanded = k_cat.repeat_interleave(repeat_factor, dim=1)
            v_expanded = v_cat.repeat_interleave(repeat_factor, dim=1)

            # 5. Scaled Dot-Product Attention
            scores = torch.matmul(q, k_expanded.transpose(-2, -1)) / (self.head_dim ** 0.5)
            attn_weights = torch.softmax(scores, dim=-1)
            context = torch.matmul(attn_weights, v_expanded)
```

```
# 6. Reshape and project output context = context.transpose(1,
2).contiguous().view(batch_size, 1, -1) layer_output = self.out_proj(context)

# Residual connection current_activations = current_activations + layer_output

# Final Unembedding to generate Logits logits = self.lm_head(current_activations)
return logits, layer_kv_caches
```

Detailed Engineering Critique of the Naive Implementation

While the code above is functionally correct in PyTorch, executing this implementation in a production serving environment results in catastrophic performance degradation due to six major system flaws:

1. **Flaw 1: VRAM Re-allocation via `torch.cat`** Line 33 uses `torch.cat([past_k, k], dim=2)` to append new Key-Value tokens to the historical cache. In PyTorch, CUDA tensors are immutable in size. Calling `torch.cat` forces the CUDA memory allocator to allocate an entirely new contiguous VRAM block, copy all past historical KV vectors plus the new vector to the new address, and free the old block. This causes massive VRAM memory fragmentation, severe CUDA memory allocator locks, and latency spikes that scale quadratically with sequence length. Solution: Pre-allocate static VRAM buffers or use PagedAttention with virtual page tables.
2. **Flaw 2: Redundant Memory Duplication via `repeat_interleave`** Line 41 expands GQA Key/Value heads using `repeat_interleave`. This physically duplicates Key and Value tensors in VRAM by a factor of 8x, completely destroying the memory-saving benefits of GQA. Solution: Use custom CUDA/Triton kernels that handle strided indexing directly over grouped KV heads without physical VRAM duplication.
3. **Flaw 3: Individual Kernel Launch Overhead** Each operation (`q_proj`, `k_proj`, `v_proj`, `view`, `transpose`, `softmax`) launches an independent CUDA kernel from the CPU to the GPU queue. For a 80-layer model, a single decode step launches over 800 separate CUDA kernels. The CPU driver overhead required to enqueue these small kernels exceeds the actual execution time on the GPU. Solution: Implement Kernel Fusion (combining RMSNorm + QKV Projection + RoPE into a single kernel) and use CUDA Graphs to launch the entire layer sequence in a single GPU call.
4. **Flaw 4: Separate Q, K, and V Projections** Launching three distinct linear layers (`q_proj`, `k_proj`, `v_proj`) streams the input activation tensor from VRAM three separate times. Solution: Merge weights into a single fused `qkv_proj` weight matrix to fetch activations exactly once.
5. **Flaw 5: Unfused Softmax Computation** Standard Softmax requires three separate passes over the attention matrix in VRAM: finding the maximum element, subtracting max and exponentiating, and summing exponents for normalization. Solution: Use Fused FlashAttention / FlashDecoding

kernels that execute Online Softmax entirely within on-chip SRAM registers.

6. Flaw 6: Synchronous Python Interpreter Loop Running the autoregressive generation loop in native Python introduces Python GIL (Global Interpreter Lock) contention and overhead between token steps. Solution: Rewrite the core engine scheduling and execution orchestration loop in C++ / Rust.

CHAPTER 18: PRODUCTION LLM SERVING ENGINES, METRICS, & SCALABLE DEPLOYMENT ARCHITECTURE

18.1 Serving Engine Internals: vLLM, TensorRT-LLM, & TGI Execution Loops

Deploying Large Language Models at production scale requires high-performance serving frameworks designed to maximize GPU hardware saturation, minimize memory bandwidth bottlenecks, and handle concurrency without incurring severe latency degradation. Modern inference engines—such as **vLLM**, **Hugging Face Text Generation Inference (TGI)**, and **NVIDIA TensorRT-LLM**—replace traditional synchronous execution pipelines with asynchronous, event-driven architectures.

1. Asynchronous Engine Architecture

- **AsyncLLM / Motor Core:** Decouples incoming client HTTP/gRPC requests from the underlying GPU execution loop using Python `asyncio` or C++ event loops.

- **Tokenizers & Batch Builders:** Requests are tokenized asynchronously and fed into an intermediate waiting queue. The scheduler queries the queue at every iteration step to form dynamic micro-batches based on available physical memory blocks (e.g., PagedAttention block availability).
1. **TensorRT-LLM C++ Runtime & In-Flight Batching** TensorRT-LLM compiles model layers into optimized CUDA graphs and custom fused kernels (e.g., FlashAttention-2, fused RMSNorm, quantized GEMM kernels). The C++ runtime manages execution flow entirely on the device, eliminating Python interpreter overhead during token generation steps.

18.2 Production Latency & Throughput Performance Metrics

Evaluating production LLM deployments requires tracking granular performance metrics across both prefill and decode phases.

Key Performance Indicators (KPIs)

1. **Time to First Token (TTFT):** The duration from when a client submits a prompt until the first generated token is returned. TTFT is dominated by the **prefill phase** (compute-bound matrix multiplications over prompt length S).
2. **Time per Output Token (TPOT) / Inter-Token Latency (ITL):** The time elapsed between consecutive generated tokens during the decode phase. ITL is strictly **memory-bandwidth bound** (loading all model parameters M from HBM to SRAM per token step).
3. **Goodput:** The number of requests completed per second that satisfy strict Service Level Agreements (SLAs)—such as maintaining an ITL below a maximum threshold (e.g., $< 50 \text{ ms/token}$).
4. **GPU Memory Utilization:** The ratio of allocated VRAM (weights + KV cache + activations) to total physical GPU VRAM capacity.

Performance Metric	Primary Phase	Limiting Bottleneck	Optimization Lever	Target Production SLA
TTFT	Prefill	Tensor Core Compute FLOPs	FlashAttention-2, Prefill/Decode Disaggregation	$< 200 \text{ ms}$ (for 2^K tokens)
ITL / TPOT	Decode	High-Bandwidth Memory (HBM GB/s)	PagedAttention, Speculative Decoding, INT4/FP8	$< 35 \text{ ms/token}$

Goodput	System-Wide	Queue Congestion & Memory Limits	Continuous Batching, Radix Prefix Caching	Maximum under SLA constraints
----------------	-------------	----------------------------------	---	-------------------------------

18.3 Request Scheduling, Load Shedding, & Priority Queuing

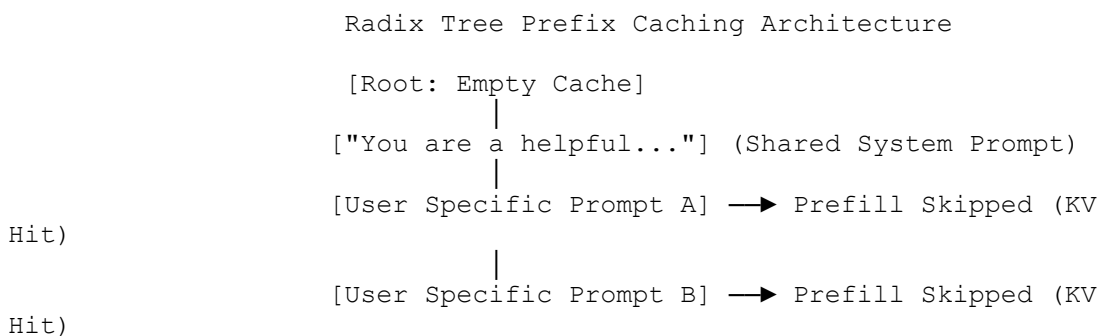
When request rates exceed GPU compute capacity, incoming queries are managed via advanced scheduling policies.

1. **Radix Tree Prefix Caching** Many production prompts share common system prompts, few-shot examples, or document contexts. Modern serving engines organize active KV cache blocks into a **Radix Tree (Prefix Tree)**:

- When a new request arrives, its prompt tokens are matched against the Radix tree.
- If a matching prefix KV block is found in GPU memory, the prefill phase for those tokens is skipped entirely, reducing TTFT by up to 10\times.

1. **Priority Queuing & Load Shedding**

- **Tiered Queues:** Prioritizes latency-sensitive interactive chat requests over offline batch summarization tasks.
- **Load Shedding:** Rejects incoming requests with HTTP 429 when the waiting queue exceeds maximum memory thresholds, preventing cascading out-of-memory (OOM) failures across worker pools.



18.4 Production Implementation: Asynchronous Token Serving Simulator & Metrics Collector

The self-contained Python script below simulates an asynchronous LLM serving engine. It manages a continuous batching request queue, tracks TTFT and ITL metrics, and evaluates system throughput and goodput.

```

import asyncio
import time
import random
from typing import List, Dict, Any

```

```

class ServingRequest:
    """Represents an incoming client inference request."""
    def __init__(self, request_id: str, prompt_len: int,
target_gen_len: int):
        self.request_id = request_id
        self.prompt_len = prompt_len
        self.target_gen_len = target_gen_len
        self.tokens_generated = 0
        self.arrival_time = time.perf_counter()
        self.first_token_time = 0.0
        self.completion_time = 0.0
        self.latencies: List[float] = []

class MockAsyncServingEngine:
    """
    Simulates an asynchronous continuous batching LLM serving engine
    tracking TTFT, ITL, and system throughput.
    """
    def __init__(self, max_batch_size: int = 4, step_delay_ms: float
= 10.0):
        self.max_batch_size = max_batch_size
        self.step_delay_ms = step_delay_ms
        self.waiting_queue: asyncio.Queue[ServingRequest] =
asyncio.Queue()
        self.active_batch: List[ServingRequest] = []
        self.completed_requests: List[ServingRequest] = []

        async def add_request(self, req: ServingRequest):
            await self.waiting_queue.put(req)

        async def execution_loop(self, total_steps: int):
            """Simulates continuous batching iteration steps."""
            for step in range(total_steps):
                # 1. Admit new requests from waiting queue into active
batch
                while len(self.active_batch) < self.max_batch_size and
not self.waiting_queue.empty():
                    req = self.waiting_queue.get_nowait()
                    self.active_batch.append(req)

                if not self.active_batch:
                    await asyncio.sleep(self.step_delay_ms / 1000.0)
                    continue

                # 2. Simulate hardware forward step for active batch
                await asyncio.sleep(self.step_delay_ms / 1000.0)
                current_time = time.perf_counter()

                # 3. Process token generation step for each active
request
                finished_indices = []
                for idx, req in enumerate(self.active_batch):
                    if req.tokens_generated == 0:
                        # First token generated (Prefill phase complete)
                        req.first_token_time = current_time
                    else:
                        # Subsequent decode token step (ITL tracking)

```

```

        req.latencies.append(current_time -
last_step_time if 'last_step_time' in locals() else 0.01)

        req.tokens_generated += 1

        # Check if request has completed generation
        if req.tokens_generated >= req.target_gen_len:
            req.completion_time = current_time
            self.completed_requests.append(req)
            finished_indices.append(idx)

        last_step_time = current_time

        # Evict completed requests from active batch
        for idx in sorted(finished_indices, reverse=True):
            self.active_batch.pop(idx)

# --- Verification and Execution Test Harness ---
async def main():
    random.seed(42)
    print("=== Asynchronous LLM Serving Engine Simulation ===")

    engine = MockAsyncServingEngine(max_batch_size=3,
step_delay_ms=15.0)

    # Simulate client requests arriving asynchronously
    num_requests = 6
    requests = [
        ServingRequest(request_id=f"req_{i}",
prompt_len=random.randint(128, 512),
target_gen_len=random.randint(10, 20))
        for i in range(num_requests)
    ]

    # Enqueue requests with slight staggering delays
    for req in requests:
        await engine.add_request(req)

    # Run execution loop for 40 steps
    engine_task =
asyncio.create_task(engine.execution_loop(total_steps=40))

    # Wait for completion simulation
    while len(engine.completed_requests) < num_requests:
        await asyncio.sleep(0.05)

    engine_task.cancel()

    print(f"\nSuccessfully Served Requests :
{len(engine.completed_requests)} / {num_requests}")
    print("-" * 60)
    print(f"{'Request ID':<12} | {'Prompt Len':<10} | {'Gen
Tokens':<10} | {'TTFT (ms)':<10} | {'Mean ITL (ms)':<12}")
    print("-" * 60)

    ttfts = []
    itls = []
    for req in engine.completed_requests:
        ttft_ms = (req.first_token_time - req.arrival_time) * 1000.0

```

```

        mean_itl_ms = (sum(req.latencies) / len(req.latencies) *
1000.0) if req.latencies else 0.0
        ttfts.append(ttft_ms)
        itls.extend([l * 1000.0 for l in req.latencies])
        print(f"{req.request_id:<12} | {req.prompt_len:<10} |
{req.target_gen_len:<10} | {ttft_ms:<10.2f} | {mean_itl_ms:<12.2f}")

        print("-" * 60)
        print(f"Average TTFT (Time to First Token) :
{sum(ttfts)/len(ttfts):.2f} ms")
        print(f"Average ITL (Inter-Token Latency) :
{sum(itls)/len(itls):.2f} ms/token")
        print("=" * 60)

if __name__ == "__main__":
    asyncio.run(main())

```