CS336: Language Modeling from Scratch

Lecture 10: Inference

Stanford University - Spring 2025

Abstract

Abstract: This lecture provides a comprehensive analysis of inference in large language models, focusing on the computational challenges, optimization strategies, and architectural innovations that enable efficient text generation. We examine the fundamental differences between training and inference workloads, analyze memory bottlenecks, and explore various approaches to accelerate inference including KV cache optimization, alternative architectures, quantization, and speculative decoding. The content emphasizes practical engineering considerations for deploying language models at scale.

Contents

1 Introduction and Problem Definition

1.1 The Inference Challenge

Core Problem: Given a fixed, trained model, generate responses to prompts efficiently and at scale.

Inference appears in multiple contexts:

- **Interactive Applications:** Chatbots, code completion (Cursor)
- **Batch Processing:** Large-scale data processing jobs
- **Model Evaluation:** Instruction following benchmarks
- **Test-Time Compute:** Models "thinking" before responding
- **Training Support:** Reinforcement learning requires inference for sampling

1.2 Scale and Importance

Industry Statistics:

- OpenAI: 100 billion words generated daily
- Cursor: 1 billion lines of accepted code daily
- Inference costs increasingly dominate training costs due to repeated usage

1.3 Performance Metrics

Time-To-First-Token (TTFT): Latency until first token generation begins. Critical for interactive applications.

Token Latency: Speed of subsequent token generation. Affects user experience quality.

Throughput: Total tokens generated per second across all users. Key for batch processing and cost efficiency.

Key Insight

Training allows parallel processing over entire sequences, but inference requires sequential generation. This fundamental constraint makes inference memory-limited rather than compute-limited.

2 Computational Analysis of Transformer Inference

2.1 Notation and Setup

Standard Transformer Parameters:

- B: Batch size (number of sequences)
- L: Number of layers
- T: Sequence length (tokens to generate)

- S: Context length (prompt tokens)
- V: Vocabulary size
- D: Model dimension
- F: MLP hidden dimension (typically $4D$)
- H: Attention head dimension
- N: Number of query heads
- K: Number of key-value heads (for GQA)

FLOP Count: Feedforward pass requires $6 \times \text{tokens} \times \text{parameters} + O(T^2)$ for attention.

2.2 Arithmetic Intensity Analysis

Definition: Arithmetic intensity = FLOPs performed / bytes transferred

For matrix multiplication $X_{B\times D} \times W_{D\times F}$:

$$\text{FLOPs} = 2BDF \tag{1}$$

$$\text{Bytes} = 2BD + 2DF + 2BF \text{ (bf16)} \tag{2}$$

$$\text{Intensity} = \frac{2BDF}{2BD + 2DF + 2BF} \approx B \text{ (when } D, F \gg B) \tag{3}$$

H100 Specifications:

- Peak compute: 989 TFLOPs/s
- Memory bandwidth: 3.3 TB/s
- Accelerator intensity: $989/3.3 \approx 295$

Performance Analysis

Compute vs Memory Bound:

- Compute-bound: Arithmetic intensity > 295 (can saturate GPU)
- Memory-bound: Arithmetic intensity < 295 (limited by memory bandwidth)
- Batch size $B = 1$ gives intensity $= 1$ (severely memory-bound)

3 KV Cache and Two-Stage Inference

3.1 Naive vs Cached Inference

Naive Approach: Recompute entire transformer for each token. Complexity: $O(T^2)$ per token.

KV Cache Solution: Store key-value vectors from previous computations in HBM.

Cache Structure: For each sequence, token, layer, and head: store H-dimensional key and value vectors.

Memory Requirement: $2 \times B \times S \times L \times N \times H \times 2$ bytes (bf16)

3.2 Prefill vs Generation Stages

Prefill Stage:

- Process entire prompt in parallel
- Compute-limited (like training)
- Fill KV cache for prompt tokens
- Generate first token

Generation Stage:

- Generate one token at a time
- Memory-limited due to sequential nature
- Update KV cache incrementally
- Repeat until completion

4 Detailed Arithmetic Intensity Analysis

4.1 MLP Layers

For MLP computation with input $X_{T\times D}$:

Operations:

1. Up projection: $X \times W_{\text{up}}$
2. Gate projection: $X \times W_{\text{gate}}$
3. Nonlinearity and element-wise multiplication
4. Down projection: Result $\times W_{\text{down}}$

Analysis:

$$\text{FLOPs} = 6BTDF \tag{4}$$

$$\text{Bytes} = 2BTD + 6DF + 2BTF \tag{5}$$

$$\text{Intensity} \approx BT \text{ (when } DF \gg BT\text{)} \tag{6}$$

Implications:

- Prefill: BT can be large (good efficiency)
- Generation: $T = 1$, so intensity $= B$ (requires large batch sizes)

4.2 Attention Layers

Key Operations:

1. QKV projections
2. Attention computation: $Q \times K^T$
3. Value combination: Attention $\times V$
4. Output projection

Analysis:

$$\text{FLOPs} = 4BSTD \tag{7}$$
$$\text{Bytes} = B(ST + S + T)D \tag{8}$$
$$\text{Intensity} = \frac{4ST}{S+T} \tag{9}$$

Critical Point

Attention Bottleneck:

- Prefill: Intensity $\approx S$ (reasonable when S is large)
- Generation: Intensity ≈ 1 (always poor, independent of batch size)
- No batch size benefits due to sequence-specific KV cache

5 Throughput and Latency Analysis

5.1 Llama 2 13B Example

Model Configuration:

- Parameters: 13B
- Sequence length: 1,000
- Model dimension: 5,120
- MLP dimension: 13,824
- Heads: 40
- Layers: 40

Memory Requirements:

$$\text{Parameters} = 26 \text{ GB (bf16)} \tag{10}$$
$$\text{KV Cache per sequence} = 2 \times S \times L \times N \times H \times 2 \tag{11}$$
$$= 2 \times 1000 \times 40 \times 40 \times 128 \times 2 \text{ bytes} \tag{12}$$
$$= 1.28 \text{ GB per sequence} \tag{13}$$

Performance Scaling:

Batch Size	Memory (GB)	Latency (ms)	Throughput (tok/s)
1	27.3	8.0	124
16	46.5	38.4	416
64	108.1	146.9	436
256	354.3	587.8	436

Key Observations:

- Latency-throughput tradeoff
- Diminishing returns in throughput
- Memory constraints limit maximum batch size

5.2 Simple Parallelism

Model Replication: Deploy M copies of the model on separate GPUs.

- No communication required
- Latency unchanged
- Throughput scales linearly by M

Time-To-First-Token: Determined by prefill stage, typically compute-limited with little room for improvement given fixed architecture.

6 KV Cache Optimization Strategies

Key Insight

Since inference is memory-limited, reducing KV cache size directly improves latency and throughput while enabling larger batch sizes.

6.1 Grouped Query Attention (GQA)

Concept: Reduce number of key-value heads while maintaining number of query heads.
Standard Multi-Head Attention: N query heads, N key heads, N value heads
GQA: N query heads, K key heads, K value heads (where $K < N$)
Cache Reduction: Factor of N/K reduction in KV cache size
Llama 2 13B with GQA (1:5 ratio):

- Memory: 46.5 GB $\rightarrow$ 30.9 GB
- Throughput: 416 $\rightarrow$ 671 tokens/s
- Enables larger batch sizes due to memory savings

Accuracy Impact: Minimal degradation with proper ratios (Llama 3 adopted GQA)

6.2 Multi-Head Latent Attention (MLA)

Concept: Project key-value vectors to lower-dimensional latent space

Standard: Store $N \times H$ dimensions per token **MLA:** Store C dimensions per token (where $C \ll N \times H$)

DeepSeek V2 Example: Reduction from 16,000 to 512 dimensions

Limitation: Incompatible with RoPE (requires additional dimensions for positional encoding)

6.3 Cross-Layer Attention (CLA)

Concept: Share key-value projections across transformer layers

Implementation: Same KV vectors used in multiple layers, reducing total cache size

Benefits: Improves accuracy-cache size Pareto frontier

6.4 Local Attention with Global Layers

Local Attention: Attend only to past K tokens

- Constant KV cache size (independent of sequence length)
- Reduced expressiveness for long-range dependencies

Hybrid Approach: Interleave local and global attention layers

- Example: 1 global layer per 6 local layers
- Combine with cross-layer KV sharing
- Maintains long-range modeling with reduced cache

7 Alternative Architectures for Inference

7.1 State Space Models

Motivation: Avoid quadratic attention complexity while maintaining long-context modeling

Evolution:

1. **S4:** Linear dynamical systems for long sequences
2. **Problem:** Poor performance on associative recall tasks
3. **Mamba:** Modified SSM handling associative tasks up to 1B scale
4. **Scaling:** 52B MoE models still require some transformer layers

Inference Advantage: Replace $O(T)$ KV cache with $O(1)$ state vector

7.2 Linear Attention

Core Idea: Replace exponential attention kernel with linear approximation

Mathematical Form:

$$\text{Standard:} \quad \text{softmax}(QK^T)V \tag{14}$$

$$\text{Linear:} \quad \phi(Q)(\phi(K)^T V) \tag{15}$$

Benefits:

- Linear complexity in sequence length
- RNN-like computation enables constant memory
- Can be mixed with occasional full attention layers

Recent Success: MiniMax 456B parameter MoE models using primarily linear attention

7.3 Diffusion Models for Text

Paradigm Shift: Generate all tokens in parallel, then iteratively refine

Advantages:

- No autoregressive constraint
- Full parallelization across sequence length
- Potential for massive speedups

Challenges:

- Text diffusion more complex than image diffusion
- Quality-speed tradeoffs still being explored

Early Results: Inception Labs demonstrating 10x+ speedups on coding tasks

8 Quantization and Model Compression

8.1 Precision Reduction

Standard Precisions:

- FP32: Training standard (32 bits)
- BF16: Inference default (16 bits)
- FP8/INT8: Aggressive quantization (8 bits)
- INT4: Extreme quantization (4 bits)

Memory and Speed Benefits: Linear reduction in memory bandwidth requirements

8.2 LLM.int8() Method

Problem: Large models develop outlier features that break simple quantization

Solution:

1. Identify outlier values in weight matrices
2. Process outliers in FP16 precision
3. Quantize remaining weights to INT8
4. Mixed computation preserves accuracy

Tradeoff: Memory savings vs. computational complexity

8.3 Activation-Aware Weight Quantization (AWQ)

Key Insight: Quantization sensitivity depends on activation magnitudes

Method:

1. Analyze activation patterns on calibration data
2. Protect weights corresponding to important activations
3. Aggressively quantize less critical weights
4. Achieve INT4 with minimal accuracy loss

Results: 3x speedup with INT4 quantization

8.4 Structured Pruning

NVIDIA Approach:

1. Identify important layers/heads/dimensions using calibration
2. Remove unimportant components
3. Distill original model into pruned architecture
4. Start from pruned initialization rather than random

Example Results:

- 15B → 8B parameters: Minimal accuracy loss
- 15B → 4B parameters: Some degradation but major speedup

9 Speculative Decoding

Key Insight

Checking token probabilities (prefill) is faster than generating tokens. This asymmetry enables speculative approaches.

9.1 Core Algorithm

Setup:

- Draft model P: Small, fast model
- Target model Q: Large, accurate model
- Lookahead K: Number of speculative tokens

Process:

1. Draft model generates K tokens autoregressively
2. Target model evaluates all K tokens in parallel (prefill)
3. Accept/reject based on probability ratio: $\min(1, Q(x)/P(x))$
4. If rejected, sample corrected distribution: $(Q(x) - P(x))_+$

Guarantees: Exact sampling from target model distribution

9.2 Theoretical Foundation

Rejection Sampling: Use proposal distribution P to sample from target Q
Acceptance Probability: $\alpha = \min(1, Q(x)/P(x))$
Correction Sampling: When rejected, sample from $(Q(x) - P(x))_+/Z$
Key Property: Unbiased sampling regardless of draft model quality

9.3 Practical Implementation

Model Selection:

- Target: 70B parameter model
- Draft: 8B parameter model (similar architecture)
- Goal: Maximize acceptance rate while minimizing draft cost

Performance: Typically 2x speedup with well-matched draft models
Scaling Consideration: Returns to scaling laws for optimal draft model sizing

10 Dynamic Batching and Systems Considerations

10.1 Request Arrival Patterns

Challenge: Users arrive asynchronously with varying prompt lengths and generation requirements
Batching Strategies:

- Static batching: Wait for fixed batch size
- Dynamic batching: Continuous request processing
- Sequence completion: Handle variable-length generations

10.2 Memory Management

KV Cache Allocation: Pre-allocate vs. dynamic allocation tradeoffs
Fragmentation: Managing variable-length sequences in fixed memory pools
Eviction Policies: When memory is full, which sequences to preempt

11 Summary and Engineering Guidelines

Fundamental Insights:

1. Inference is memory-limited due to sequential generation
2. KV cache dominates memory usage and transfer costs
3. Attention layers provide no batch size benefits
4. Prefill is compute-limited, generation is memory-limited

Optimization Hierarchy:

1. **Architecture Changes:** GQA, MLA, alternative architectures
2. **Quantization:** Precision reduction with outlier handling
3. **Speculative Decoding:** Exact sampling with speedup
4. **System Optimization:** Kernels, memory management, batching

Design Principles:

- Minimize memory transfers over compute optimization
- Consider inference constraints during model design
- Balance accuracy vs. efficiency based on application needs
- Leverage parallelism where possible (model replication, prefill)

The future of efficient inference lies in co-designing architectures and systems with deployment constraints in mind, moving beyond post-hoc optimizations to fundamental algorithmic innovations.

12 Mind Map: LLM Inference Optimization

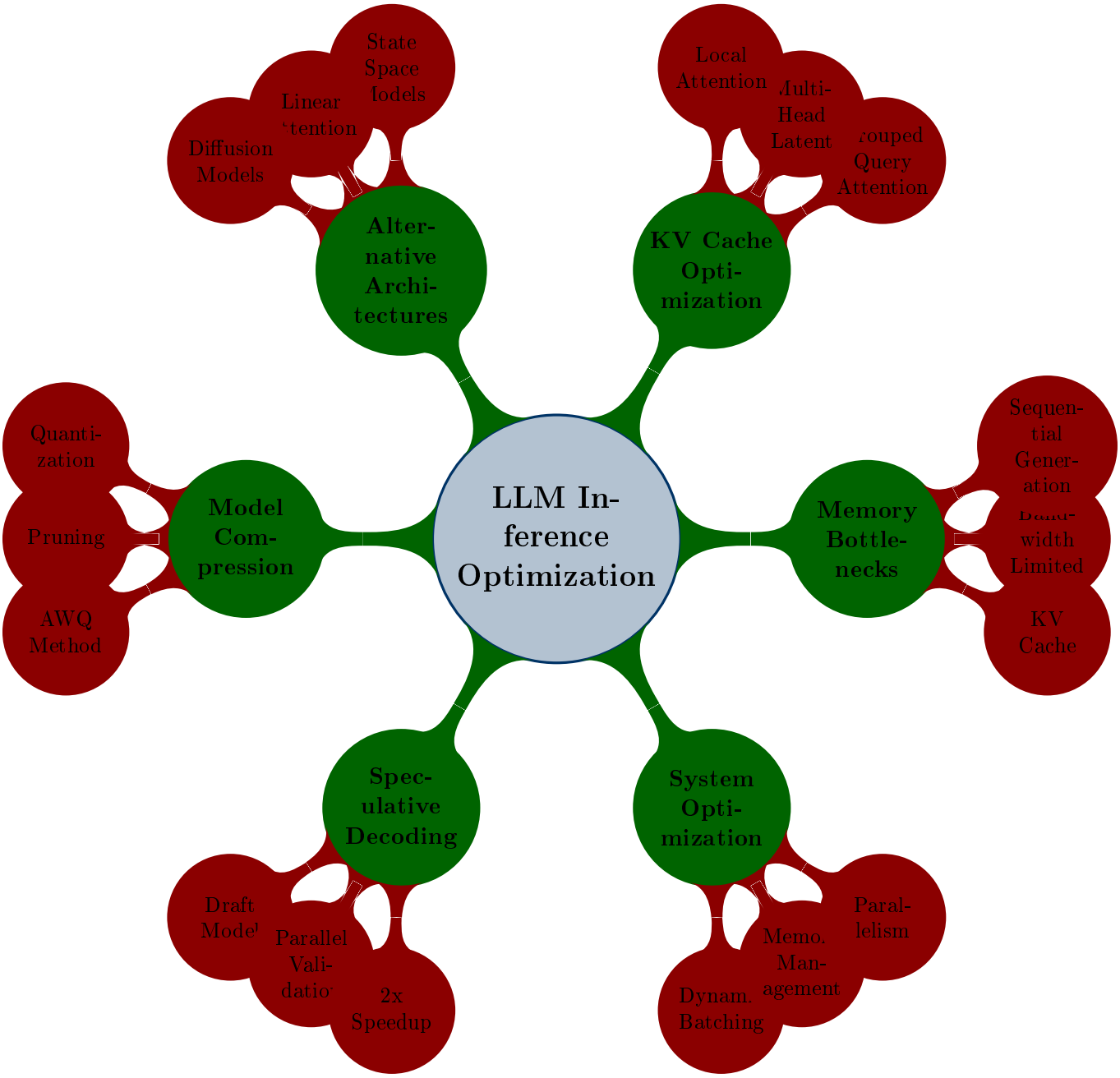

Figure 1: Comprehensive mind map of LLM inference optimization strategies, showing six major categories and their key components arranged to avoid overlap.

13 Detailed Mind Map Description

The mind map presented in Figure 1 provides a comprehensive visual overview of the six major categories of LLM inference optimization strategies discussed in this lecture:

- **Memory Bottlenecks (0°):** Positioned to emphasize this as the fundamental challenge
  - Sequential nature of autoregressive generation creates bandwidth limitations
  - KV cache is the primary memory consumer
  - Memory-bound operations dominate inference performance
- **KV Cache Optimization (60°):** Direct solutions to the memory problem through architectural modifications
  - Grouped Query Attention reduces the number of key-value heads
  - Multi-Head Latent Attention compresses to lower dimensions
  - Local Attention limits the attention window
  - Cross-layer sharing strategies
- **Alternative Architectures (120°):** Fundamental departures from transformer attention mechanisms
  - State Space Models offer constant memory complexity
  - Linear Attention provides linear scaling
  - Diffusion Models enable parallel generation
  - RNN-like approaches for sequence modeling
- **Model Compression (180°):** Techniques to reduce model size and memory requirements
  - Quantization reduces precision (FP16, INT8, INT4)
  - Pruning removes unnecessary components
  - Activation-Aware Weight Quantization (AWQ) provides intelligent bit allocation
  - Structured pruning with distillation
- **Speculative Decoding (240°):** Leverages the asymmetry between generation and validation costs
  - Uses a fast draft model for speculation
  - Parallel validation by the target model
  - Achieves typical 2x speedups with exact sampling guarantees
  - Rejection sampling with correction distributions
- **System Optimization (300°):** Infrastructure-level improvements
  - Dynamic batching for handling variable request patterns
  - Memory management for efficient allocation
  - Parallelism strategies for scaling throughput
  - Kernel optimization and hardware utilization

The circular arrangement emphasizes that these approaches are complementary and can be combined for maximum effectiveness in production deployments.