Lecture 10: Efficient LLM Inference

Marshall Meng and Dylan Fang

Stanford CS336 - Spring 2025

1 The Inference Bottleneck

1.1 Inference Metrics

Unlike training, which is a one-time cost, inference is a recurring cost. The efficiency of inference is measured by three key metrics:

- **Time-to-first-token (TTFT):** The latency before generation begins. This is determined by the "prefill" phase (processing the prompt). Crucial for interactive applications.
- **Latency (seconds/token):** The speed at which subsequent tokens are generated. Determined by the "generation" phase.
- **Throughput (tokens/second):** The total volume of tokens processed across all concurrent requests. Important for batch processing.

Toy Example: Latency vs. Throughput

Scenario: You have 10 users, and you need to generate **1 token** for each of them.

Case A: Serial Processing (Batch Size = 1)

- The GPU loads weights (10ms) and computes 1 token (negligible).
- **Time per step:** 10ms.
- **Latency (User perspective):** User 1 waits 10ms. (Fast!)
- **Total Time (System perspective):** 10 users $\times$ 10ms = 100ms.
- **Throughput:** 10 tokens/0.1s = **100** tokens/sec.

Case B: Batched Processing (Batch Size = 10)

- The GPU loads weights (10ms) and computes 10 tokens in parallel (adds slight overhead, say 2ms).
- **Time per step:** 12ms.
- **Latency (User perspective):** User 1 waits 12ms. (Slower than Case A).
- **Total Time (System perspective):** All 10 users are served in that single 12ms step.
- **Throughput:** 10 tokens/0.012s $\approx$ **833** tokens/sec.

Conclusion: By increasing the batch size, we made the individual user wait slightly longer (worse latency: 10ms $\rightarrow$ 12ms), but the system became 8$\times$ more efficient (better throughput: 100 $\rightarrow$ 833 tok/s).

1.2 Arithmetic Intensity vs. Accelerator Intensity

The fundamental bottleneck in inference is determined by the ratio of computation to memory transfer.

Definition 1 (Arithmetic Intensity) *The ratio of floating-point operations (FLOPs) performed to the number of bytes transferred from High Bandwidth Memory (HBM).*

Definition 2 (Accelerator Intensity) *The ratio of a hardware accelerator's peak compute throughput to its memory bandwidth.*

Calculating FLOPs and Memory Traffic Consider a linear layer operation (Matrix Multiplication) $Y = XW$, where we multiply input X by weights W.

- **Input** X: shape $[B, D]$ (Batch, Hidden Dimension)
- **Weights** W: shape $[D, F]$ (Hidden Dimension, Feedforward Dimension)
- **Output** Y: shape $[B, F]$

1.2.1 1. FLOPs Calculation (Computational Work)

For every element in the output matrix Y, we perform a dot product between a row of size D and a column of size D. Each element in the dot product requires 1 multiplication and 1 addition (accumulate), totaling 2 operations.

$$\text{FLOPs} = \underbrace{(B \times F)}_{\text{Elements in } Y} \times \underbrace{(2 \times D)}_{\text{Ops per element}} = 2 \cdot B \cdot D \cdot F \tag{1}$$

1.2.2 2. Bytes Transferred (Memory Cost)

Assuming the model uses `bfloat16` (16-bit) precision, each parameter requires **2 bytes**. We must read the inputs/weights and write the output.

$$\text{Bytes} = \underbrace{2BD}_{\text{Read } X} + \underbrace{2DF}_{\text{Read } W} + \underbrace{2BF}_{\text{Write } Y} \tag{2}$$

1.3 Arithmetic Intensity vs. Accelerator Intensity

1.3.1 Algorithm Intensity

We define Arithmetic Intensity as the ratio of work done to data moved.

$$\mathcal{I}_{algo} = \frac{\text{FLOPs}}{\text{Bytes}} = \frac{2BDF}{2(BD + DF + BF)} \tag{3}$$

If we assume dimensions D and F are large (typical in LLMs), the term DF (weights) dominates the denominator when B is small. As we increase B, we amortize the cost of reading W. In the limit:

$$\lim_{D,F \to \infty} \mathcal{I}_{algo} \approx B \tag{4}$$

This implies that for Linear Layers, **Arithmetic Intensity scales linearly with Batch Size.**

1.3.2 Accelerator Intensity (Hardware Roofline)

The hardware has a fixed ratio determined by its physical specs. For an H100 GPU:

- Peak Compute: 989×10^{12} FLOPs/s
- Memory Bandwidth: 3.35×10^{12} Bytes/s

$$\mathcal{I}_{hw} = \frac{989}{3.35} \approx 295 \text{ FLOPs/Byte} \tag{5}$$

1.3.3 The Comparison (Intuition)

We compare $\mathcal{I}_{algo}$ with $\mathcal{I}_{hw}$ to determine the bottleneck:

1. **Memory-Limited ($\mathcal{I}_{algo} < 295$):** The chip can calculate faster than memory can provide data. The compute cores sit idle waiting for bytes. (Typical for low batch sizes, e.g., $B = 1$).
2. **Compute-Limited ($\mathcal{I}_{algo} > 295$):** The data is loaded, and the chip is fully utilized processing it. The memory bandwidth is sufficient. (Achieved at high batch sizes).

1.4 Architecture: The SwiGLU MLP

Modern Transformers (e.g., Llama 2, Mistral) utilize a Gated Linear Unit (GLU) variant for the MLP layer, requiring three distinct weight matrices instead of the traditional two.

Let $x \in \mathbb{R}^D$ be the input vector. The layer is defined by:

$$W_{up} \in \mathbb{R}^{D \times F} \quad \text{(Up-Projection)} \tag{6}$$

$$W_{gate} \in \mathbb{R}^{D \times F} \quad \text{(Gate-Projection)} \tag{7}$$

$$W_{down} \in \mathbb{R}^{F \times D} \quad \text{(Down-Projection)} \tag{8}$$

The forward pass computes:

$$\text{MLP}(x) = (\sigma(xW_{gate}) \odot (xW_{up}))\, W_{down} \tag{9}$$

where σ is the activation function (typically SiLU) and $\odot$ denotes element-wise multiplication.

Computational Cost:

- The introduction of the third matrix (W_{gate}) increases the parameter count and FLOPs by approximately 50% compared to standard ReLU MLPs, but typically yields better training convergence and performance.
- Total parameters: $3 \times D \times F$.

1.5 The Role of Batch Size (B) in MLP Layers

To understand the bottleneck, we perform a FLOPs vs. Memory accounting for the Multi-Layer Perceptron (MLP) layers. Modern LLMs (like Llama) typically use three matrices: Up (W_{up}), Gate (W_{gate}), and Down (W_{down}).

Let Input $X \in \mathbb{R}^{B \times T \times D}$ and Weights $W \in \mathbb{R}^{D \times F}$.

- **FLOPs:** We perform 3 GEMMs (General Matrix Multiplications), resulting in $\approx 6BTDF$ operations.
- **Bytes:** We read the input X ($2BTD$), read three weight matrices ($6DF$), and write the output ($2BTD$).

The arithmetic intensity is derived as:

$$\text{Intensity}_{MLP} = \frac{6BTDF}{4BTD + 4BTF + 6DF} \tag{10}$$

Assuming D and F are large constants (weights dominate parameters), and taking the limit as they approach infinity:

$$\lim_{D,F \to \infty} \text{Intensity}_{MLP} \approx B \cdot T \tag{11}$$

Implication:

- During **Prefill** (T is large), intensity is high (Compute-Limited).
- During **Generation** ($T = 1$), intensity becomes $\approx B$.

To saturate an H100 (requires Intensity > 295), we must use a batch size $B > 295$. Thus, **batching is the primary optimization for MLP layers**.

1.6 Inference Phases and Attention Bottlenecks

Inference consists of two phases with distinct arithmetic intensities, driven largely by the Attention mechanism's dependency on the KV Cache.

1.6.1 1. Prefill Phase (Processing Prompt)

- **Operation:** We process the entire prompt history S at once, so $T = S$.
- **Attention Intensity:**

$$\text{Intensity} \approx \frac{S \cdot T}{S + T} \xrightarrow{T=S} \frac{S^2}{2S} = \frac{S}{2} \tag{12}$$

- **Status:** Since S is usually large (e.g., 1024), $S/2 \gg 295$. Prefill is **Compute-Limited** (Efficient).

1.6.2 2. Generation Phase (Autoregression)

- **Operation:** We generate 1 token at a time, so $T = 1$.
- **Attention Intensity:** We must read the inputs Q (current token) and the entire history K, V (KV Cache) from HBM.

$$\text{Intensity} \approx \frac{S \cdot T}{S + T} \xrightarrow{T=1} \frac{S}{S + 1} < 1 \tag{13}$$

- **Status: Severely Memory-Limited**. The intensity is < 1 FLOP/Byte, while H100 requires 295.
- **The Batching Problem:** Unlike MLPs, **batching does not improve Attention intensity**.

$$\text{Intensity} = \frac{\text{Compute}(B)}{\text{Memory}(B)} = \frac{4 \cdot B \cdot S \cdot D}{4 \cdot B \cdot S \cdot D} \approx 1 \tag{14}$$

  Because every request in the batch has its own unique history (KV Cache), adding more batch items increases memory reads linearly, canceling out the compute gains. This makes the Attention layer the fundamental bottleneck of generative inference.

1.7 Throughput vs. Latency Trade-off

The efficiency of inference is governed by the interplay between fixed costs (model weights) and variable costs (KV cache).

Let M_{params} be the memory size of the model weights and M_{KV} be the size of the KV cache per sequence.

- **Total Memory Read:** $M(B) = M_{params} + B \cdot M_{KV}$
- **Latency (Time per token):** $L(B) = \frac{M_{params} + B \cdot M_{KV}}{\text{Bandwidth}}$
- **Throughput (Tokens per second):** $T(B) = \frac{B}{L(B)} = \text{Bandwidth} \cdot \frac{B}{M_{params} + B \cdot M_{KV}}$

1.7.1 Regimes of Operation

- **Small Batch Size ($B \approx 1$):**
  - $M_{params} \gg B \cdot M_{KV}$.
  - **Latency:** Lowest (best). We primarily read weights.
  - **Throughput:** Low. We waste bandwidth reading the massive M_{params} just to generate 1 token.
  - **Behavior:** Throughput scales linearly with B (easy gains).
- **Large Batch Size ($B \uparrow$):**
  - The $B \cdot M_{KV}$ term becomes significant.
  - **Latency:** Increases (worse). The overhead of reading the KV cache for many users slows down the step time.
  - **Throughput:** High, but with diminishing returns. We amortize the cost of reading M_{params} across many users.
  - **Limit:** Eventually, $M(B)$ exceeds physical HBM capacity (OOM), as seen with $B = 256$ on Llama 2 13B.

Optimization Strategy:

- **Prefill:** Use smaller batches to minimize TTFT (latency sensitive).
- **Generation:** Use larger batches to maximize system Throughput (cost sensitive).

2 Approximation Techniques (Lossy)

To overcome the memory bandwidth bottleneck, we employ "lossy" optimizations. Unlike exact system optimizations (like FlashAttention), these techniques trade a negligible amount of model accuracy for significant gains in arithmetic intensity and memory efficiency.

2.1 Reducing KV Cache Size

The Key-Value (KV) cache grows linearly with sequence length T ($O(T)$), creating the primary memory bottleneck. We use four main approximations to compress this cache.

2.1.1 Grouped-Query Attention (GQA)

GQA interpolates between Multi-Head Attention (MHA) and Multi-Query Attention (MQA) to balance memory efficiency and accuracy.

- **Mechanism:** Instead of every Query head having a unique KV head (MHA), or all Query heads sharing a single KV head (MQA), GQA divides Query heads into G groups. All queries in a group share a single KV head.
- **Reduction Factor:** The KV cache size is reduced by a factor of N_{heads}/G.
- **Throughput Gain:** By reducing the memory footprint of the cache, we can fit a larger batch size B into VRAM (e.g., increasing B from 64 to 256), significantly improving system throughput.
- **Accuracy:** GQA maintains accuracy much better than MQA while offering similar speedups.

2.1.2 Multi-Head Latent Attention (MLA)

Introduced in DeepSeek-V2, MLA addresses the memory bottleneck by compressing the KV vectors into a low-rank latent space.

- **Mechanism:** Rather than storing the full $N \cdot H$ dimensional vectors for Keys and Values, MLA projects them down to a compressed latent vector of dimension C.
- **DeepSeek Example:** Reduces the effective dimension from $16,384$ ($N \cdot H$) down to 512 (C) + 64 (for RoPE compatibility) = 576.
- **Benefit:** Achieves massive compression ratios (approx. $28\times$ smaller) while outperforming GQA in accuracy benchmarks.

2.1.3 Cross-Layer Attention (CLA)

CLA applies the logic of sharing KV vectors "vertically" across the network depth.

- **Mechanism:** A KV cache computed at Layer i is reused by subsequent layer(s) (e.g., Layer $i + 1$).
- **Benefit:** Reduces the total number of KV caches stored by the model (e.g., 50% reduction if shared pairwise) without modifying the head structure within a layer.

2.1.4 Local (Sliding Window) Attention

To break the $O(T)$ linear memory dependency, Local Attention restricts the context window.

- **Mechanism:** The model only attends to the most recent W tokens. Older tokens are evicted from the cache.
- **Memory Cost:** $O(W)$ (Constant), allowing for theoretically infinite sequence generation.
- **Hybrid Approach:** To prevent the loss of global context, architectures (e.g., Mistral, Character.ai) interleave many Local Attention layers with occasional **Global Attention layers** (e.g., 1 Global per 6 Local) to maintain long-range dependencies.

2.2 Alternatives to the Transformer

The fundamental bottleneck of the Transformer is the Attention mechanism's reliance on the KV Cache, which scales linearly with sequence length ($O(T)$). To achieve $O(1)$ inference memory, recent research explores replacing Attention with alternative primitives.

2.2.1 State-Space Models (SSMs)

SSMs map a 1-D input signal $u(t)$ to an output $y(t)$ via a latent state $x(t)$.

1. **Continuous Representation (The Math):** Defined by the differential equation $\dot{x}(t) = Ax(t) + Bu(t)$. This continuous formulation allows the model to theoretically handle infinite context.
2. **Structured State Matrix (The Memory):** The matrix A controls memory retention. In models like **S4**, A is initialized with specific "HiPPO" structures (High-order Polynomial Projection Operators).

- *Visual Interpretation:* The lower-triangular structure of A prevents the memory state from vanishing or exploding, allowing the model to compress long-term history faithfully.

3. **Discrete Representation (The Computation):** For implementation on GPUs, the continuous system is discretized (step size Δ). This grants SSMs a "Dual Nature":
   - **Training (Convolutional):** The entire sequence is processed in parallel using a long convolution kernel ($\bar{K}$). Fast like a Transformer.
   - **Inference (Recurrent):** The model operates step-by-step ($x_t = \bar{A}x_{t-1} + \bar{B}u_t$), requiring only a fixed-size state x. Fast like an RNN.

2.2.2 Evolution of SSM Architectures

- **S4 (2021):** Solved the long-range dependency problem but failed at **Associative Recall** (e.g., Input: `A 4 ... A ?` → Output: `4`). The static matrices caused the model to "blur" specific details.
- **Mamba (2023):** Introduced a **Selection Mechanism**. By making the matrices B and C *input-dependent* (changing based on the current token), Mamba can selectively "remember" or "ignore" information. This solves the associative recall problem while retaining efficient inference.
- **Jamba (Hybrid Architecture):** Visualized as a stack of interleaved blocks, Jamba combines the strengths of both families:
  - **Ratio 1:7:** Uses 1 Transformer layer for every 7 Mamba layers.
  - **Role:** Mamba layers handle bulk context compression ($O(1)$ memory), while sparse Transformer layers provide sharp retrieval capabilities for critical tokens.
  - **MoE:** Incorporates Mixture-of-Experts to scale parameters (52B) while keeping active compute low (12B).

2.2.3 Diffusion Models for Text

While Transformers are autoregressive (generating tokens sequentially $t_1 \to t_2 \to \dots$), Diffusion models are non-autoregressive.

- **Mechanism:** The model initializes a sequence of random noise vectors (x_T) and iteratively "denoises" them in parallel to reveal the text.
- **Continuous-Discrete Bridging (Graph Interpretation):** Since text (w) is discrete but diffusion operates on continuous Gaussian noise, the architecture requires a specific bridging step:
  - **Embedding/Un-rounding:** The discrete text w is mapped to a continuous latent space x_0.
  - **Denoising Process (p_θ):** The model refines the latent vectors from pure noise $x_T \to \dots \to x_0$.
  - **Rounding Step:** Finally, the clean continuous vectors x_0 are "rounded" (projected) back to the nearest discrete vocabulary tokens to produce the output text.

- **Efficiency Gain:** Although the model may run multiple refinement steps (e.g., 20 steps), it generates the entire sequence length T simultaneously. This increases Arithmetic Intensity by amortizing weight loading over T tokens, shifting the workload from memory-bound to compute-bound.
- **Performance:** Recent benchmarks (e.g., Mercury Coder) show Diffusion models significantly outperforming autoregressive models in tokens-per-second for code generation tasks.

2.3 Quantization

Quantization reduces the precision of weights and activations to decrease the memory footprint (bytes transferred), thereby increasing arithmetic intensity.

2.3.1 Number Formats

- **bf16 (2 bytes):** The standard for inference. No accuracy loss.
- **int8 (1 byte):** Range $[-128, 127]$. Doubles intensity but introduces quantization noise.
- **int4 (0.5 bytes):** Range $[-8, 7]$. Quadruples intensity but requires advanced calibration to prevent model collapse.

2.3.2 Addressing Quantization Challenges

Simple "Round-to-Nearest" (RTN) quantization often fails for Large Language Models due to two specific phenomena:

1. The Outlier Problem (LLM.int8()) In large models, certain feature dimensions (columns) exhibit systematic "outliers" with magnitudes far larger than the rest (e.g., 50.0 vs 1.0). Standard scaling based on the max value causes the small (but informative) values to be quantized to zero.

- **Solution (Mixed-Precision Decomposition):** The matrix multiplication is split into two parallel paths:
  1. **Outlier Path:** Columns with values exceeding a threshold are extracted and computed in **FP16**.
  2. **Main Path:** The remaining 99.9% of values are quantized and computed in **INT8**.
- **Result:** Preserves accuracy by protecting outliers while retaining INT8 speed for the majority of parameters.

2. The Salience Problem (AWQ) Standard quantization assumes that small weights are unimportant. However, **Salience** is determined by the product of the weight and its input activation.

$$\text{Impact} = \text{Weight} \times \text{Input} \tag{15}$$

If a small weight (e.g., 0.001) is multiplied by a massive outlier input (e.g., 100), its impact is significant (0.1). Rounding this weight to 0 destroys the signal, causing the PPL to explode.

- W **vs.** $Q(W)$**:** W represents the original high-precision weights (FP16), while $Q(W)$ represents the compressed weights (e.g., INT3, INT4).

- **PPL (Perplexity):** A metric measuring the model's uncertainty (lower is better).
  - PPL ≈ 13.0: Good (Original Llama performance).
  - PPL ≈ 43.0: Bad (Model generates gibberish).

We compare three approaches to quantizing weights while handling salient features.

1. RTN (Round-to-Nearest)

- **Mechanism:** Naively rounds all weights to the nearest quantized integer representation.
- **Failure Mode:** Small but salient weights (those multiplied by large inputs) are often rounded to zero, destroying the signal.
- **Result:** PPL explodes to 43.2. The model fails.

2. Mixed-Precision (Salient Columns)

- **Mechanism:** Identify the top 1% of weights that receive large inputs (salient columns). Keep these in high precision (FP16) and quantize the rest to INT3/INT4.
- **Result:** PPL recovers to 13.0 (Good).
- **Drawback: Bad Hardware Efficiency**. Computing matrix multiplications with mixed data types (FP16 + INT) prevents the use of optimized integer Tensor Cores, slowing down inference.

3. AWQ (Activation-aware Weight Quantization)

- **Mechanism:** Protects salient weights by scaling them up before quantization.
- **Mathematical Trick:** Relies on the invariance of matrix multiplication:

$$X \cdot W = (X \cdot s^{-1}) \cdot (W \cdot s) \tag{16}$$

- **Step 1:** Multiply salient weight channels by $s > 1$. This increases their magnitude, making them robust to quantization noise (prevents rounding to zero).
- **Step 2:** Divide input activations by s to mathematically cancel the scaling.
- **Result:** We achieve PPL 13.0 using **uniform integer formats** (hardware efficient), as the scaling is baked into the quantization parameters.

2.3.3 Comparative Toy Example

Consider a dot product $y = X \cdot W$ where we have a massive outlier input.

- **Input X:** $[100, 1]$ (100 is the outlier/salient feature).
- **Weights W:** $[0.03, 0.03]$ (Small weights).
- **Target Output:** $100(0.03) + 1(0.03) = \mathbf{3.03}$.

- **Constraint:** Assume our quantized grid only has resolution steps of 0.1 (rounds to $0.0, 0.1, \dots$).

1. RTN Approach:

- Weights 0.03 are closer to 0.0 than 0.1. Both round to 0.
- Calculation: $100(0) + 1(0) = \mathbf{0}$.
- **Error:** Massive loss of signal.

2. Mixed-Precision Approach:

- We identify column 1 as salient (Input 100). Keep first weight as FP16. Quantize second.
- Calculation: $100(0.03)_{\text{FP16}} + 1(0)_{\text{INT}} = \mathbf{3.0}$.
- **Error:** Negligible. Accuracy saved, but hardware inefficient.

3. AWQ Approach (Scaling Trick):

- Choose scale factor $s = 4$.
- **Step 1 (Scale Weights):** $0.03 \times 4 = 0.12$. This now rounds to **0.1** (preserved!).
- **Step 2 (Inverse Input):** $100/4 = 25$.
- Calculation: $25(0.1) = \mathbf{2.5}$.
- **Result:** Signal is preserved ($2.5 \gg 0$) using purely integer math. In real scenarios with INT4 grids, this result approaches the target 3.03 closely.

2.4 Structural Pruning (The Minitron Approach)

Idea: Instead of training a small model from scratch, we "compress" a pre-trained large model by permanently removing unimportant components. This leverages the computational investment already made in the large model.

2.4.1 The Pruning Algorithm

The process involves a "Prune and Heal" cycle, as demonstrated by NVIDIA's Minitron research:

1. **Estimate Importance:** Using a small calibration dataset (e.g., 1024 samples), calculate sensitivity scores for specific architectural dimensions:
    - **Depth:** Number of layers.
    - **Width:** Hidden dimension size (D) and MLP intermediate size (F).
    - **Heads:** Number of attention heads (N).
2. **Trim (Rank & Remove):** Rank components by importance and physically excise the lowest-ranked ones. This results in a smaller, dense model architecture (not a sparse mask).
3. **Distillation (Healing):** The trimmed model initially suffers performance degradation. We "heal" it by identifying the original large model as the **Teacher** and the pruned model as the **Student**. Retraining uses Knowledge Distillation (KD) to recover accuracy, which converges significantly faster than training from random initialization.

2.4.2 Economic Impact

Structural pruning shifts the Pareto frontier of Training Cost vs. Accuracy.

- **Training Efficiency:** Pruning a 15B model down to 8B is up to **40x cheaper** than training an 8B model from scratch to the same accuracy.
- **Result:** We obtain models (e.g., Minitron 8B/4B) that outperform baselines (e.g., Llama-3 8B, Phi-2) while requiring a fraction of the training compute.

3 Algorithmic Speedups (Lossless)

Unlike quantization or pruning, algorithmic speedups like **Speculative Sampling** optimize the inference process without changing the model weights or the output distribution. They trade excess compute for reduced memory bandwidth latency.

3.1 Speculative Sampling (SpS)

Core Intuition: Since inference is memory-bound, the massive compute capability of GPUs (like H100s) is idle while waiting for weights to load. We can utilize this idle compute to verify the work of a smaller, faster model.

Speculative Sampling is a **lossless** algorithmic speedup that converts the memory-bound inference process into a compute-bound process.

3.1.1 Algorithm: Draft and Verify

The method utilizes two models: a small *Draft Model* (M_p) and a large *Target Model* (M_q).

Algorithm 1 Speculative Sampling with Auto-Regressive Target and Draft

Input: Prompt sequence x, Lookahead K
while $n < T$ **do**
  // Step 1: Draft Phase (Cheap, Serial)
  for $t = 1$ to K **do**
    Sample draft token $\tilde{x}_t \sim p(x|x_1, \ldots, \tilde{x}_{t-1})$
  end for
  // Step 2: Verification Phase (Expensive, Parallel)
  Compute $K + 1$ sets of logits from M_q in parallel for the sequence:

$$q(x|x_1, \ldots), \ldots, q(x|x_1, \ldots, \tilde{x}_K)$$

  // Step 3: Acceptance Loop
  for $t = 1$ to K **do**
    Sample $r \sim U[0, 1]$
    if $r < \min\left(1, \frac{q(\tilde{x}_t)}{p(\tilde{x}_t)}\right)$ **then**
      Accept $\tilde{x}_t$: set $x_{n+t} \leftarrow \tilde{x}_t$
    else
      Reject $\tilde{x}_t$.
      Sample correct token from residual: $x_{n+t} \sim \max(0, q(x) - p(x))$
      Exit For Loop
    end if
  end for
end while

3.1.2 Operational Phases

The algorithm proceeds through five distinct phases to convert the memory-bound inference process into a compute-bound one.

Phase 1: Drafting (The Loop) The small model p (e.g., 7B parameters) runs autoregressively for K steps to generate a "best guess" sequence (e.g., "The", "cat", "sat", "on").

- **Cost:** Low. The draft model is small and cheap to run serially.

Phase 2: Parallel Verification (The Speedup) The entire draft sequence $\tilde{x}_1, \ldots, \tilde{x}_K$ is fed into the large target model q in a single batch.

- **Mechanism:** Because Transformers use causal masking, the probability for token t only depends on tokens $1 \ldots t-1$. We do not need to wait for serial generation; we can calculate distributions for all K positions simultaneously.
- **Source of Speedup:** The speedup does *not* come from skipping computation (we still compute full distributions for every token). It comes from **amortizing memory bandwidth**.
  - *Serial:* Load 70B weights → Compute Token 1 → Load 70B weights → Compute Token 2. (Penalty paid K times).
  - *Parallel:* Load 70B weights → Compute [Token 1 ... Token K] at once. (Penalty paid 1 time).

- **The $K+1$ Logic:** Why do we get $K+1$ distributions from K draft tokens?
  - Input x_1 predicts x_2.
  - Input x_K predicts x_{K+1} (The Bonus Token).
  - Total: K checks + 1 future prediction = $K+1$ distributions.

Phase 3: The Acceptance Loop We verify tokens sequentially using a modified Rejection Sampling criterion: $r < \min\left(1, \frac{q(x)}{p(x)}\right)$.

- **Case A ($q(x) \geq p(x)$):** The Target model thinks the token is *more* likely than the Draft model did (Draft was "under-confident").
  - **Result:** Always Accept (ratio ≥ 1).
- **Case B ($q(x) < p(x)$):** The Draft model was "over-confident" (assigned higher probability than Target).
  - **Result:** Accept probabilistically based on the ratio $\frac{q}{p}$.
  - *Example:* If Draft $p = 0.5$ and Target $q = 0.25$, we accept with 50% chance.

Phase 4: Rejection and Correction

- **If Accepted:** Keep $\tilde{x}_t$ and check the next token.
- **If Rejected:** Stop immediately. Discard $\tilde{x}_t$ and all subsequent draft tokens.
- **The Fix (Residual Distribution):** Sample a new token from the difference between the distributions.
- **Intuition:** If we reject a token proposed by the Draft, we cannot just pick any other token randomly. We must pick a token that the Target Model likes more than the Draft Model did. We are sampling from the "gap" between the two distributions.

Phase 5: The Bonus Token If all K draft tokens are accepted, we have fully validated the history. The parallel verification step (which computed $K+1$ logits) has already computed the valid prediction for the *next* token (x_{n+K+1}).

- **Benefit:** We sample this extra token for free without an additional forward pass.
- **Index Update Note:** The algorithm increments n inside the acceptance loop (Step 3). Therefore, at this final step, n already points to the end of the draft ($n_{start} + K$). The final operation $n \leftarrow n+1$ accounts solely for this bonus token.

3.1.3 SpS Example

Let us trace the algorithm with a lookahead of $K = 2$ tokens given the prompt "The".

- **Phase 1: Drafting (Serial)**
  - M_p predicts x_1 = "quick" with probability $p = 0.6$.
  - M_p predicts x_2 = "brown" with probability $p = 0.8$.

- **Phase 2: Verification (Parallel)**
  - M_q processes "The quick brown" in one forward pass.
  - M_q assigns $q(\text{"quick"}) = 0.7$.
  - M_q assigns $q(\text{"brown"}) = 0.4$.
- **Phase 3: Acceptance Loop**
  1. **Check x_1 ("quick"):**
  $$\alpha = \min\left(1, \frac{q}{p}\right) = \min\left(1, \frac{0.7}{0.6}\right) = 1$$
  Result: Accept. (Draft was correct/under-confident).
  2. **Check x_2 ("brown"):**
  $$\alpha = \min\left(1, \frac{q}{p}\right) = \min\left(1, \frac{0.4}{0.8}\right) = 0.5$$
  Result: We accept with 50% probability. Suppose random $r = 0.7$. Since $0.7 > 0.5$, we **Reject** "brown".
- **Phase 4: Correction** Since x_2 was rejected, we discard it and sample a new token x_2' from the residual distribution:
$$x_2' \sim \max(0, q(x) - p(x))$$
Suppose we sample x_2' = "blue".
- **Outcome:** We effectively generated 2 tokens ("quick", "blue") using only a single forward pass of the expensive Target model M_q, effectively doubling the speed compared to standard autoregressive generation.

3.1.4 Proof of Exactness

Despite relying on a potentially inaccurate draft model $p(x)$, SpS guarantees that the output distribution follows the target $q(x)$ exactly.

Consider a binary vocabulary $\{A, B\}$ where the draft model oversamples A ($p(A) > q(A)$).

Case 1: Sampling A Token A is sampled if drafted and accepted.

$$P(\text{out} = A) = \underbrace{p(A)}_{\text{Draft A}} \times \underbrace{\frac{q(A)}{p(A)}}_{\text{Accept Ratio}} = q(A) \tag{17}$$

Case 2: Sampling B Token B is sampled if drafted (Scenario 1) OR if A was drafted but rejected (Scenario 2).

$$P(\text{out} = B) = \underbrace{p(B) \cdot 1}_{\text{Draft B (Always Accept)}} + \underbrace{p(A) \cdot \left(1 - \frac{q(A)}{p(A)}\right) \cdot 1}_{\text{Draft A, Reject, Resample B}} \tag{18}$$

$$= p(B) + p(A) - q(A) \tag{19}$$

$$= q(B) \quad (\text{via } p(A) + p(B) = 1) \tag{20}$$

Thus, the algorithm is theoretically lossless.

3.1.5 Extensions to Draft Models

Standard Speculative Sampling draft generation is still autoregressive (serial). Recent methods attempt to parallelize or enrich this drafting phase.

1. Medusa (Parallel Drafting)

- **Limitation of Standard SpS:** The draft model must run K times sequentially to generate K draft tokens ($t_1 \to t_2 \to \dots$).
- **Solution:** Medusa augments the draft model with multiple decoding heads (extra layers on top of the last hidden state).
- **Mechanism:** given a single history, Head 1 predicts token t_k, Head 2 predicts token t_{k+1}, and so on.
- **Benefit:** It generates all K draft candidates in a **single forward pass**, making the drafting phase parallel rather than serial.

2. EAGLE (Feature-Aware Drafting)

- **Limitation of Standard SpS:** The draft model typically only sees the discrete token history, discarding the rich semantic information (embeddings/hidden states) computed by the Target model.
- **Solution:** EAGLE (Extrapolation Algorithm for Greater Language-model Efficiency) operates on the **feature level**.
- **Mechanism:** The draft model takes the Target model's high-level feature vectors (f_t) as input. It uses a lightweight auto-regression head to predict the next feature vector (f_{t+1}) and the corresponding token.
- **Benefit:** Because features contain more semantic context than discrete tokens, EAGLE achieves higher draft accuracy (higher acceptance rate) than standard token-based draft models.

4 System-Level Optimization (Dynamic Workloads)

4.1 Continuous Batching and Selective Batching

During training, we usually work with a dense tensor of shape $[B, S, H]$ (Batch, Sequence length, Hidden size). All sequences in the batch have the same length S, so every step looks like a clean block of tokens.

Inference is different: user requests arrive and finish at different times.

- Some users send longer prompts than others.
- Some generations end early (the model outputs an `<END>` token).
- New users may arrive while we are still decoding old ones.

This creates a *ragged* layout instead of a nice $B \times S$ block: some rows are shorter, some are already finished, and new rows keep appearing over time.

Continuous Batching (Iteration-Level Scheduling)

- **Problem:** Standard batching assumes a fixed batch of sequences that all start and end together. In inference, if we wait for all current requests to finish before starting new ones, the GPU is under-utilized and latency becomes high for new arrivals.
- **Idea:** Switch from *request-level* scheduling to *iteration-level* scheduling.

Definition

Continuous batching keeps a single decoding "loop" running, and at each decoding step:

1. We decode *one token* for every active request in the batch.
2. We **add** new user requests into the batch immediately when they arrive.
3. We **remove** finished requests (those that emit `<END>`) from the batch.

The batch is updated continuously instead of being fixed.

Intuitively, we fill any empty "slots" in the batch with new requests as soon as they appear. This keeps the GPU busy and improves overall throughput (tokens/second) without having to wait for an entire batch to finish.

Example

Toy Picture: At time step T_1, we might be decoding for four users S_1, S_2, S_3, S_4. By T_4, request S_3 may have finished (generated `<END>`), so at T_5 we can insert a new request S_5 into that slot instead of leaving it idle.

Selective Batching for Variable-Length Sequences

Continuous batching raises a second challenge:

- **Problem:** Batching is easiest when all sequences have the same shape $[S, H]$. In practice, each active request may have a different current length (e.g., $[3, H], [9, H], [5, H]$).

Definition

Selective batching treats different parts of the model separately:

- **Attention computation:** Process each sequence independently, respecting its own length and mask.
- **Non-attention computation (e.g., MLP layers):** Concatenate all sequences along the sequence dimension into a single tensor of shape $[\sum_i S_i, H]$ and run one large batched matrix multiplication.

This design lets us:

- Preserve the correct attention behavior for each sequence (no mixing histories).
- Still enjoy large, efficient matrix multiplies for the MLP and other pointwise layers by treating all tokens from all sequences as one big batch.

Together, **continuous batching** (iteration-level scheduling) and **selective batching** allow an inference server to:

- Accept new user requests at any time,
- Recycle finished slots quickly,
- And keep the GPU well utilized even when sequences have very different lengths.

4.2 PagedAttention and KV Cache Management

So far we have treated the Key-Value (KV) cache as one long, continuous array for each request. In practice this leads to a lot of wasted memory, especially when requests have different lengths or when we generate fewer tokens than the maximum budget.

Naive KV Cache Allocation and Fragmentation

- **Previous status quo:**
  1. A request comes in (for example, Request A).
  2. We reserve one big, contiguous region of the KV cache large enough to hold:
     - The entire prompt,
     - Plus space for a maximum number of future generated tokens.
  3. We do the same for Request B, Request C, and so on.

In the picture, Request A's prompt is "`Four score and seven years ago our fathers brought forth ...`". We allocate a long segment of KV slots for A:

- The yellow cells hold the prompt tokens that are already processed.
- The blue cells hold tokens that have been generated so far.
- The grey cells on the right are **reserved** for future tokens, up to some maximum sequence length (they are empty now).

The problem is that we often do *not* use all of the reserved slots:

- **Internal fragmentation:** Inside the reserved big region for a single request, many slots are never used.
  - In the figure, Request A has thousands of reserved KV slots that the model never actually fills.
  - Similarly, Request B has its own reserved tail that stays empty.
- **External fragmentation:** There can also be gaps between these big reserved regions that are too small or awkwardly placed to be re-used efficiently by new requests.

This situation is similar to what happens on a hard drive when files are allocated in big chunks: over time, you end up with many small holes and partially used blocks.

Pitfall

Even though the logical sequence lengths may be short, the naive allocation strategy forces us to keep large continuous KV segments in memory, wasting valuable High Bandwidth Memory (HBM).

PagedAttention: Divide KV Cache into Blocks

PagedAttention takes an idea from operating systems: instead of allocating one huge continuous region, we **divide the KV cache into fixed-size blocks** and allow sequences to occupy *non-contiguous* blocks.

Definition

In **PagedAttention**, the KV cache for each sequence is represented as a list of *logical blocks*. Each logical block is mapped to some *physical block* in GPU memory, and these physical blocks do not need to be next to each other.

In the toy example:

- Block 0 might store the KV entries for "`Four score and seven`".
- Block 1 might store "`years ago our fathers`".
- Block 2 might store "`brought forth`".

When we do attention for the query token "`forth`", we simply follow the block table to find which blocks contain the previous tokens (e.g., Block 0, Block 1, Block 2) and read their KV vectors.

This block-based layout removes the need to reserve one giant continuous space per request. We can now pack blocks tightly in memory and reuse freed blocks for new requests.

Sharing KV Cache Across Requests

A block-based design also makes it easy to **share** KV cache between different requests.

Example: Two requests sharing a prefix. Consider Request A with prompt:

"`Four score and seven years ago our fathers brought ...`"

and Request B with prompt:

"`It was the best of times ...`"

In the figure:

- The middle grid shows all **physical** KV blocks in memory.
- On the left and right, each request has its own list of **logical** KV blocks (its own view of which blocks belong to its sequence).
- The logical blocks for Request A and Request B point into the same pool of physical blocks.

Because we only store the KV vectors once in the physical blocks, we can let many logical sequences refer to the same content.

Shared prefixes in practice. In real workloads, many prompts share the same prefix:

- A common pattern is a long system prompt or instruction (e.g., "`Translate English to French: sea otter` → `loutre de mer, ...`"), followed by a short task-specific query.
- Another pattern is sampling multiple responses for the same prompt (for example, for program synthesis or brainstorming).

PagedAttention lets us:

- Store the shared prefix (system prompt) *once* in KV blocks.
- Make multiple sequences (Sequence A, Sequence B, etc.) point to these same blocks as their prefix.

This reduces both memory usage and prefill time, because we do not recompute or re-store the same prefix KV cache over and over.

Copy-on-Write at the Block Level

What happens when two sequences that share a prefix start to diverge?

- Suppose we have two samples, A1 and A2, that both start from the same prompt:

  "`Four score and seven years ago our fathers ...`"

- At first, their logical KV blocks point to the same physical blocks (shared prefix).

Later, sample A2 generates a different token (for example, "`mothers`" instead of "`fathers`") in one of the blocks. We cannot modify the shared block in place, because A1 still needs the original version.

PagedAttention solves this using a simple **copy-on-write** rule:

1. Each physical block keeps a *reference count* (how many logical views are using it).
2. When a sequence wants to write into a block and the reference count is greater than 1, we:
   - Allocate a new physical block,
   - Copy the old contents into it,
   - Update the logical block pointer for this sequence to point to the new block,
   - Decrease the reference count of the old block.
3. Now the sequence can safely modify its private copy, while other sequences keep using the original data.

This is exactly the same trick used in operating systems when forking processes that share memory until one of them writes.

Other vLLM System Optimizations

PagedAttention is part of a larger system (vLLM) that adds several low-level optimizations:

- **Fused kernels:** Combine the block-read operation and the attention computation into one GPU kernel to reduce kernel launch overhead.
- **Modern kernels:** Use the latest highly optimized kernels for attention and decoding, such as FlashAttention and FlashDecoding, to speed up the math.
- **CUDA graphs:** Use CUDA graph execution to avoid per-step kernel launch overhead when running the same computation pattern many times.

Summary

- Naive, contiguous KV allocation wastes memory due to both internal and external fragmentation.
- PagedAttention divides each sequence's KV cache into reusable blocks, inspired by paging in operating systems.
- Block-level indirection lets multiple sequences share prefixes and system prompts efficiently.
- Copy-on-write at the block level lets shared prefixes diverge safely when needed.
- Together with other vLLM optimizations (kernel fusion, modern attention kernels, CUDA graphs), this design makes better use of GPU memory for dynamic, real-world inference workloads where requests start, end, and branch at unpredictable times.