

KV Cache

The Memory Bottleneck That Defines Inference Optimization

KV caching stores key and value tensors from previous token positions so they don't need to be recomputed during autoregressive generation. Without caching, generating n tokens requires $O(n^2)$ projection FLOPs.

With KV cache, the projection cost drops to $O(n)$ -- each step projects only the new token, appends K and V to the cache, and attends over the full cached history.

This demo covers:

1. Output equivalence: cached = uncached (bit-identical tokens)
2. Theoretical FLOP comparison at 7B model scale
3. Memory analysis: per-token cost, growth, 7B projections
4. Wall-clock timing benchmark
5. Prefill vs decode phase analysis
6. KV cache as THE memory bottleneck at production scale

Model config: V=256, d=64, layers=2, h=4, h_kv=2, d_ff=172

Random seed: 42

Generated by `demo.py`
Number of visualizations: 6

Examples: 6

Mathematical Foundation

Without KV Cache (Naive)

At step t , recompute all projections for $[0, \dots, t]$:

$$Q = XW_Q \in \mathbb{R}^{(t+1) \times d_k}, \quad K = XW_K, \quad V = XW_V$$

$$\text{scores} = \frac{QK^\top}{\sqrt{d_k}} \in \mathbb{R}^{(t+1) \times (t+1)}$$

$$\text{Projection FLOPs: } \sum_{i=1}^n i \cdot 3 \cdot d \cdot d_k = O(n^2 \cdot d)$$

With KV Cache

At step t , project only the new token x_t :

$$q_t = x_t W_Q \in \mathbb{R}^{1 \times d_k}, \quad k_t = x_t W_K, \quad v_t = x_t W_V$$

$$K_c = \text{concat}(K_c, k_t) \in \mathbb{R}^{(t+1) \times d_k}$$

$$\text{scores} = \frac{q_t K_c^\top}{\sqrt{d_k}} \in \mathbb{R}^{1 \times (t+1)}$$

$$\text{Projection FLOPs: } n \cdot 3 \cdot d \cdot d_k = O(n \cdot d)$$

Memory Analysis

$$\text{cache/layer} = 2 \cdot \text{seqlen} \cdot d_k \cdot \text{bytes}$$

$$\text{total} = n_{\text{layers}} \cdot n_{\text{heads}} \cdot 2 \cdot \text{seqlen} \cdot d_k \cdot \text{bytes}$$

$$7\text{B model (FP16): } 32 \times 32 \times 2 \times S \times 128 \times 2 = 524,288 \cdot S \text{ bytes}$$

$$\approx 0.5 \text{ MB/token} \Rightarrow 4\text{K ctx: } 2 \text{ GB, } 32\text{K ctx: } 16 \text{ GB, } 128\text{K ctx: } 64 \text{ GB}$$

Speedup

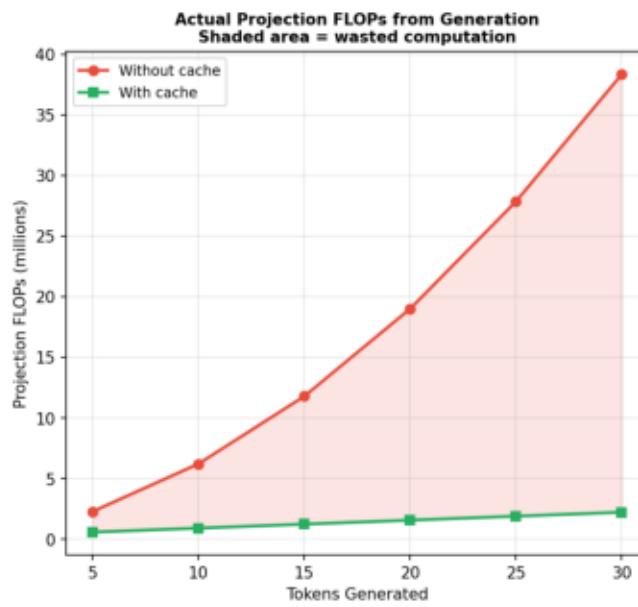
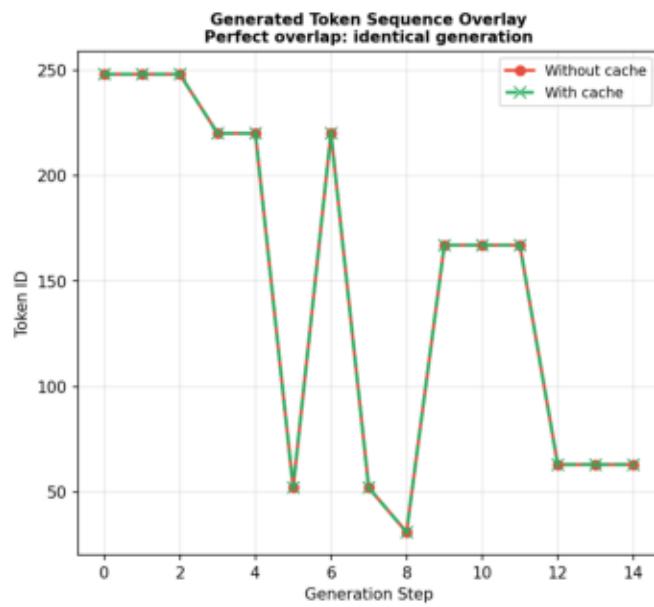
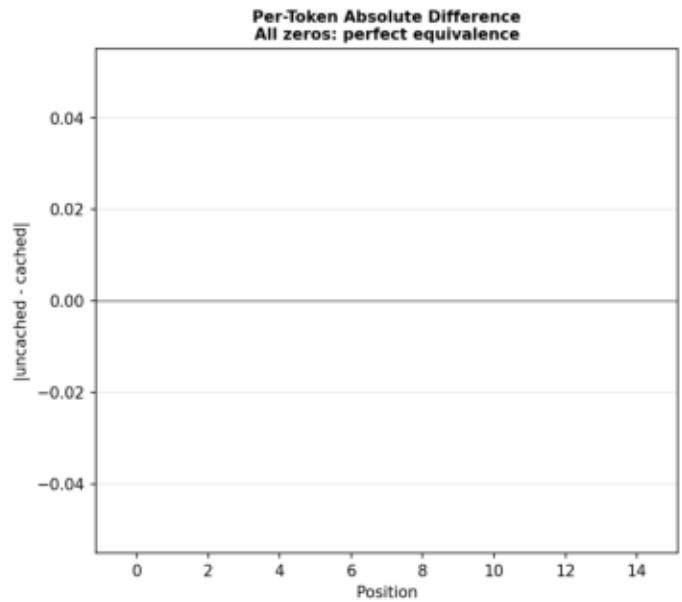
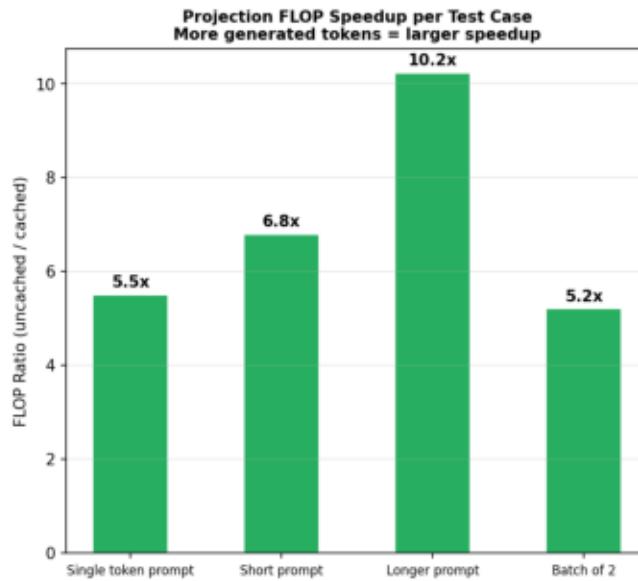
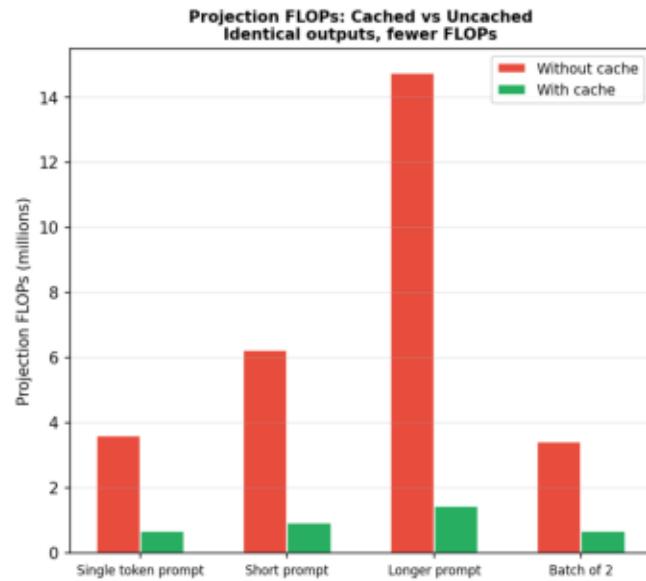
$$\text{Projection speedup} = \frac{\sum_{i=1}^n (P+i)}{P+n} \approx \frac{n}{2} \text{ for } n \gg P$$

Summary of Findings

1. Output Equivalence: Cached and uncached generation produce BIT-IDENTICAL token sequences across all tested configurations (single token, short/long prompts, batched). K and V for position i depend only on token i and the fixed weights -- caching avoids redundant recomputation without error.
2. FLOP Comparison: At 7B scale (32 layers, $d=4096$), projection FLOPs drop from $O(n^2 \cdot d^2)$ to $O(n \cdot d^2)$. For 4096 generated tokens with $P=512$ prompt, this yields $>1000x$ speedup in projection cost alone.
3. Memory Analysis: KV cache costs ~0.5 MB/token for a 7B model (FP16).
At 4K context: 2 GB. At 32K context: 16 GB. At 128K context: 64 GB.
With `batch_size=32` and 4K context: 64 GB -- exceeding model weights (14 GB).
4. Timing Benchmark: Wall-clock measurements on our small model confirm speedup that grows with generation length. Per-token latency is nearly constant with cache; grows linearly without cache.
5. Prefill vs Decode: Prefill is compute-bound (high arithmetic intensity -- batch matmul over all prompt tokens). Decode is memory-bound (reads entire cache for each single-token attention step). This two-regime nature drives architecture decisions in inference systems (separate prefill/decode GPUs).
6. Memory Bottleneck: For long contexts and large batches, KV cache EXCEEDS model weight memory. A 7B model in FP16 is ~14 GB, but KV cache for `batch_size=32` at 8K context is 128 GB. This motivates PagedAttention, GQA/MQA, KV cache quantization, and sliding window attention.

Example 1: Output Equivalence Verification

KV Cache: Output Equivalence Verification



OUTPUT EQUIVALENCE VERIFIED

Test cases: 4
All match: YES

What was verified:

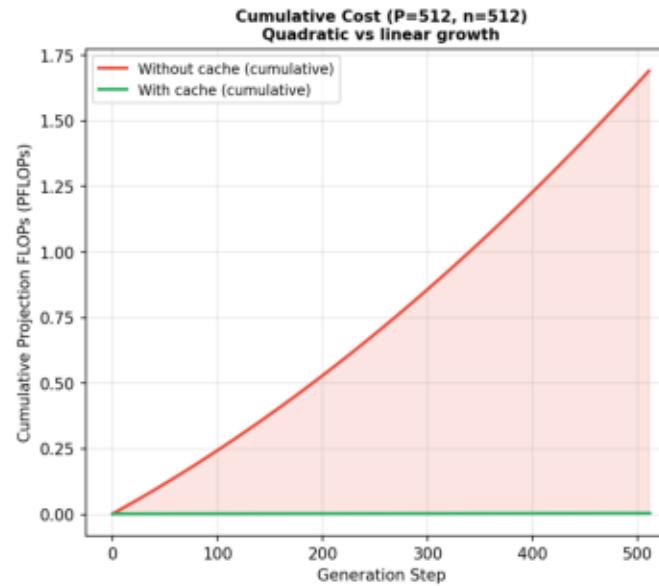
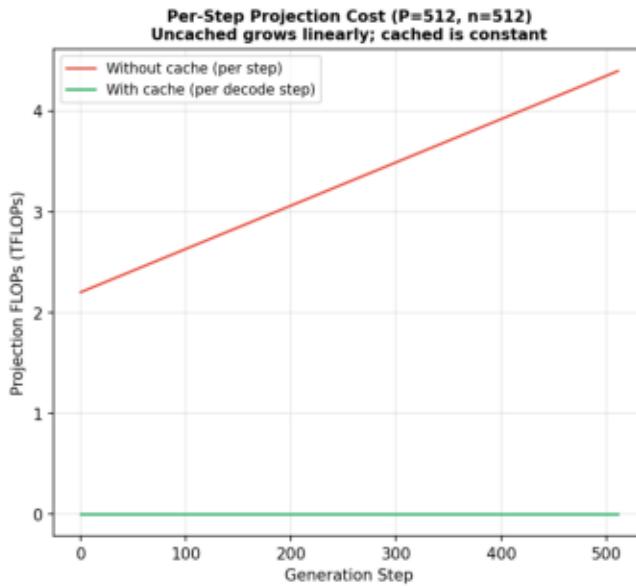
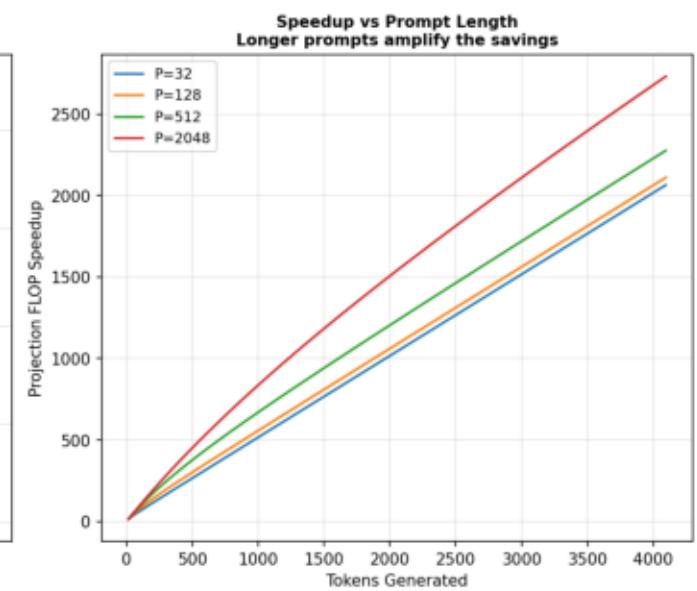
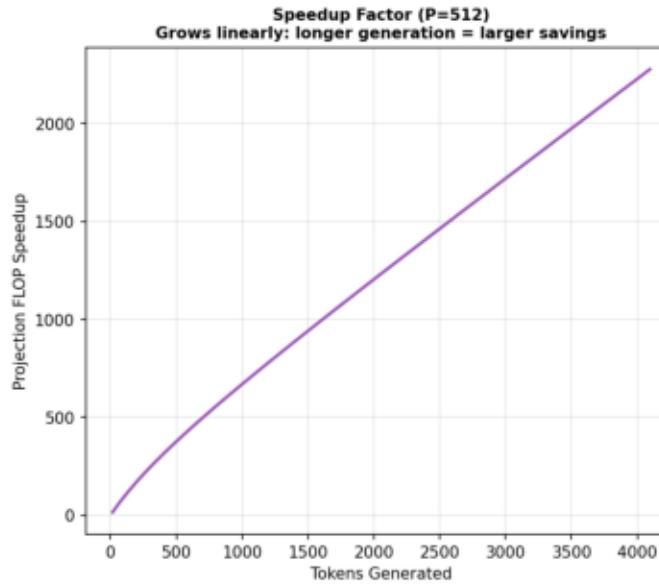
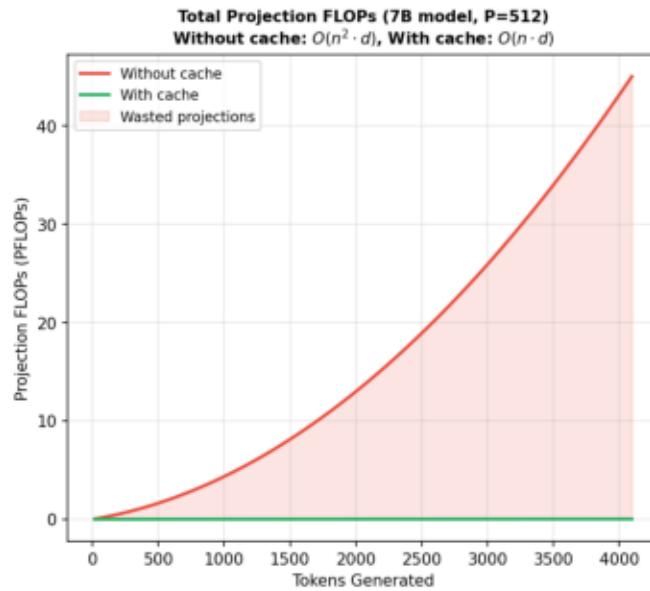
- Single token prompt
- Short prompt (5 tokens)
- Longer prompt (8 tokens)
- Batch of 2 prompts

Key insight:
K and V for position i depend only on token i and weights W, K, W, V. Future tokens don't change them. Caching avoids redundant recomputation without affecting the output.

generate_without_cache() and generate_with_cache() produce BIT-IDENTICAL token sequences.

Example 2: Theoretical FLOP Comparison

Theoretical Projection FLOP Comparison: 7B Model Scale



PROJECTION FLOP ANALYSIS

Without cache (step t):
Project ALL (P+t) tokens:
 $FLOPs = N \times 4 \times 2 \times (P+t) \times d^2$
Total = $N \times 4 \times 2 \times d^2 \times \sum(P+1) = O(n^2 \cdot d^2)$

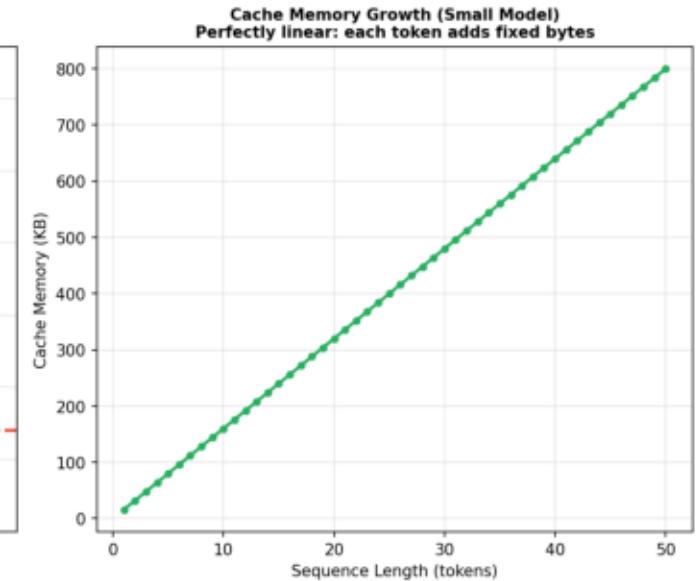
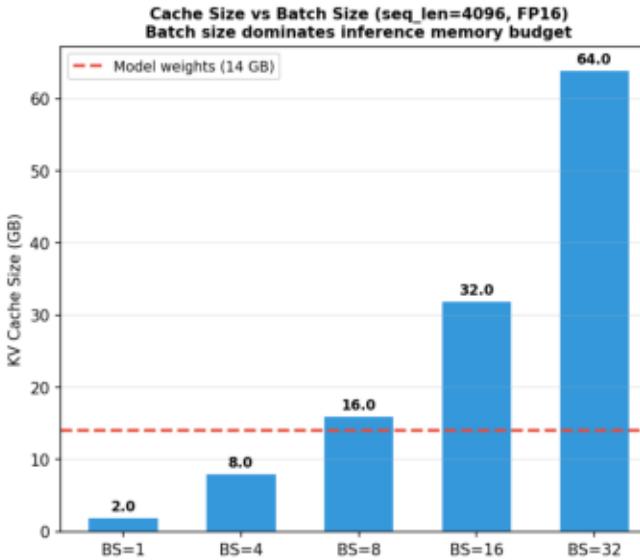
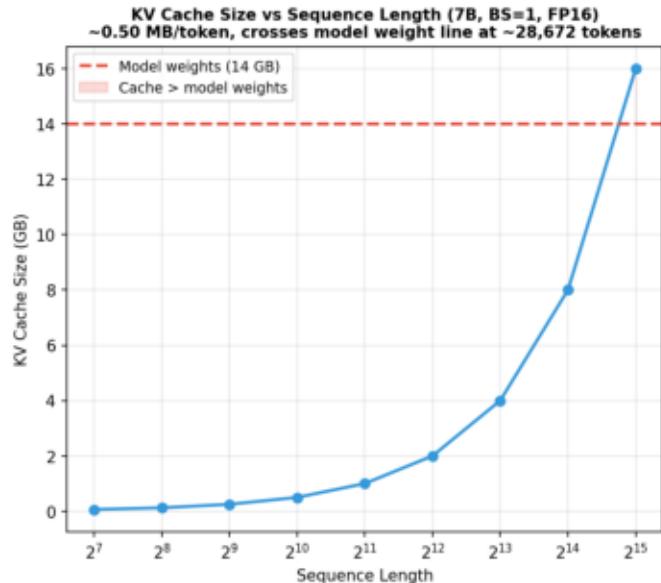
With cache (step t):
Project only 1 new token:
 $FLOPs = N \times 4 \times 2 \times 1 \times d^2$
Total = prefill + $n \times \text{const} = O(n \cdot d^2)$

Speedup for large n:
~ $n/2$ (for $n \gg P$)

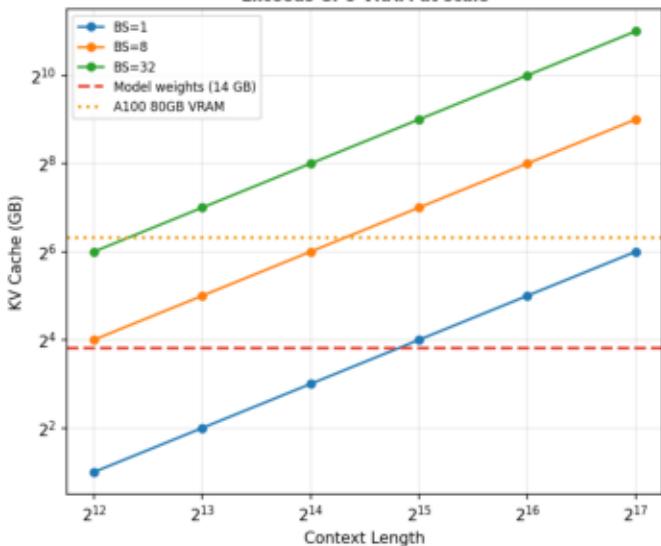
Note: Attention FLOPs (QK^T , AV) are $O(n^2)$ in both cases.
The cache saves PROJECTION cost, which is the dominant factor for $d_{\text{model}} \gg \text{seq_len}$.

Example 3: Memory Analysis

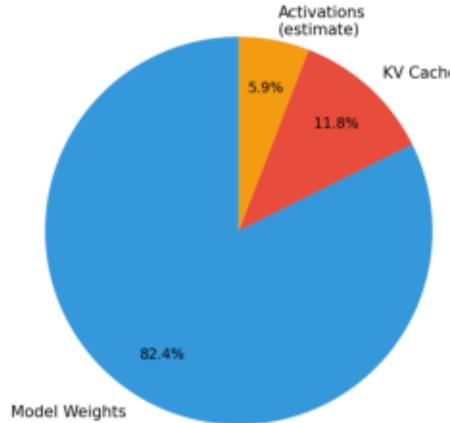
KV Cache Memory Analysis: The Inference Memory Bottleneck



7B Model KV Cache at Extended Context
Exceeds GPU VRAM at scale



GPU Memory Breakdown (7B, BS=1, ctx=4096)
Weights=14GB, Cache=2.0GB, Act~1GB



MEMORY ANALYSIS
=====

Per-token KV cache (7B, FP16):

$$2 * n_layers * n_heads * d_k * 28$$

$$= 2 * 32 * 32 * 128 * 2$$

$$= 524,288 \text{ bytes}$$

$$= 0.50 \text{ MB/token}$$

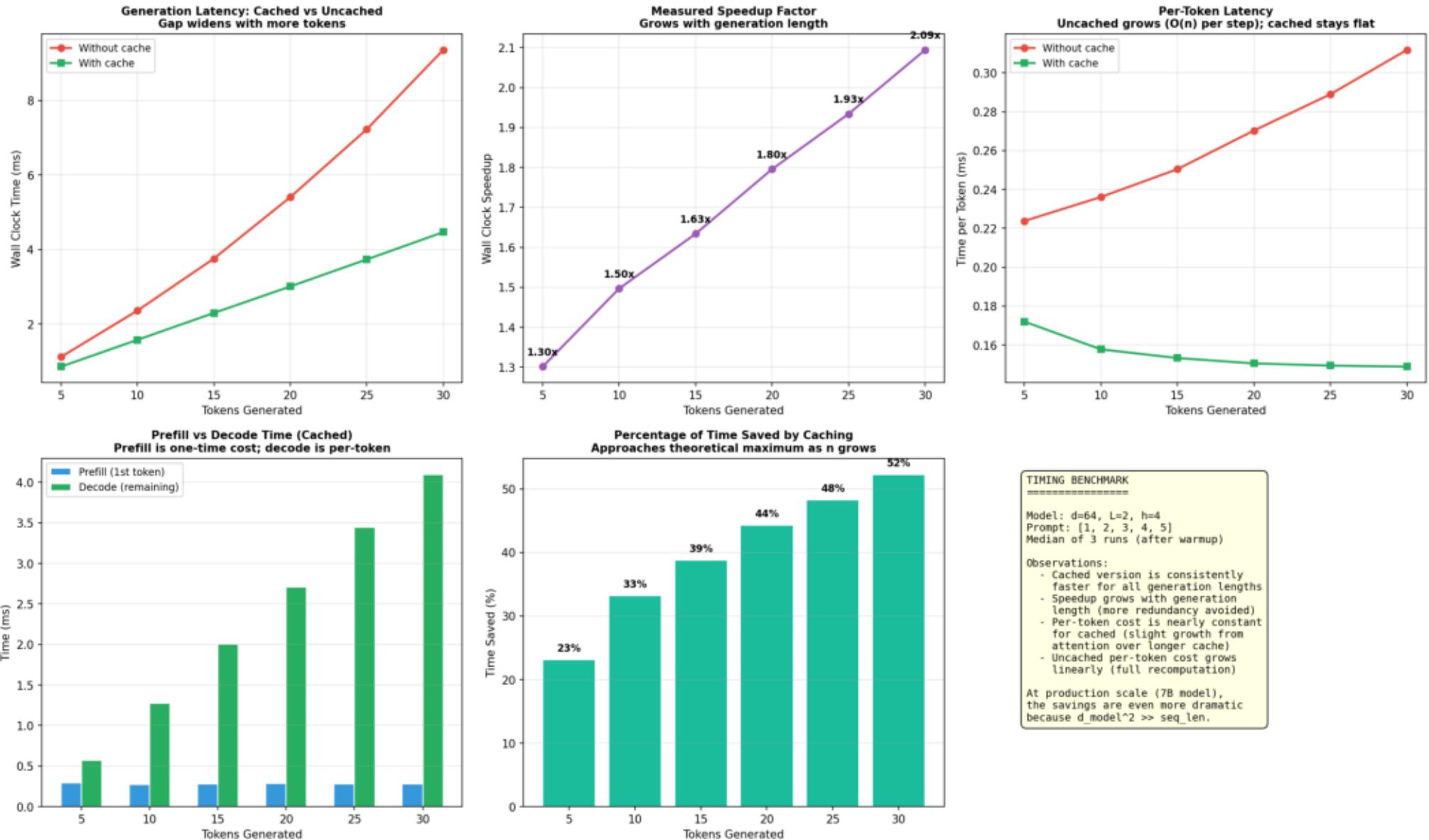
At key context lengths:
 4K tokens: 2.0 GB
 32K tokens: 16.0 GB
 128K tokens: 64.0 GB

With batching (ctx=4K):
 BS=8: 16.0 GB
 BS=32: 64.0 GB

THIS is why KV cache management is THE central challenge in LLM inference systems.

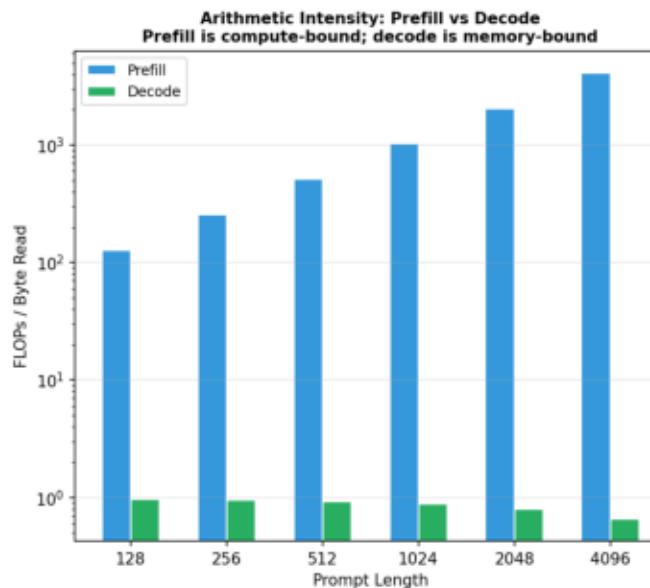
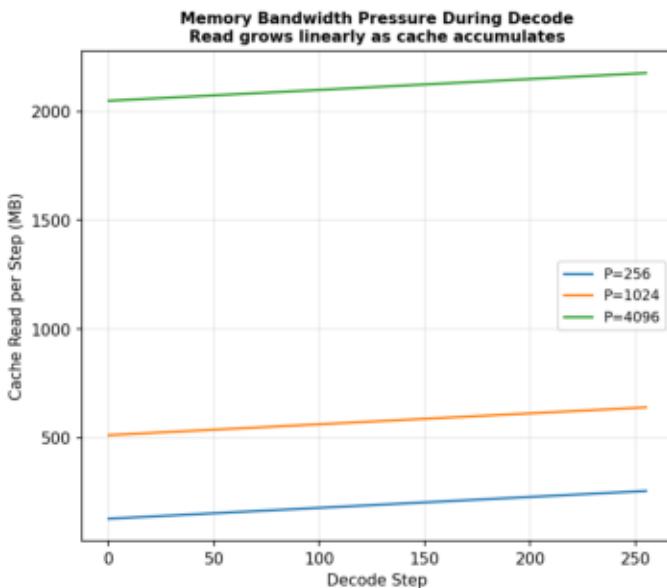
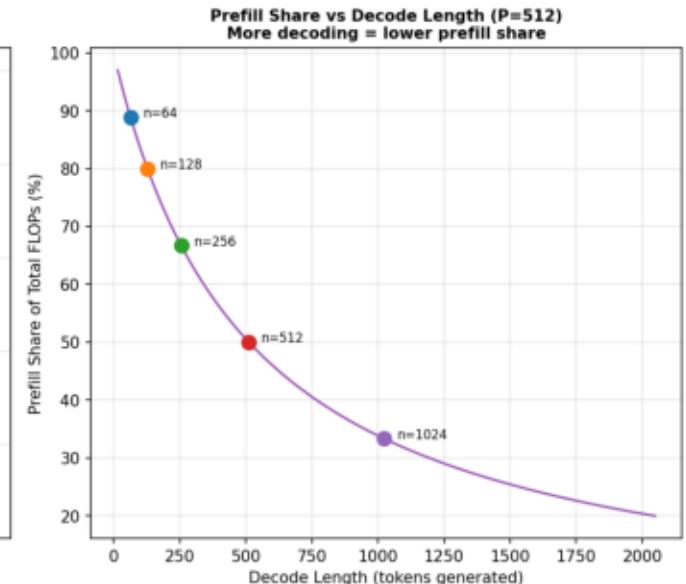
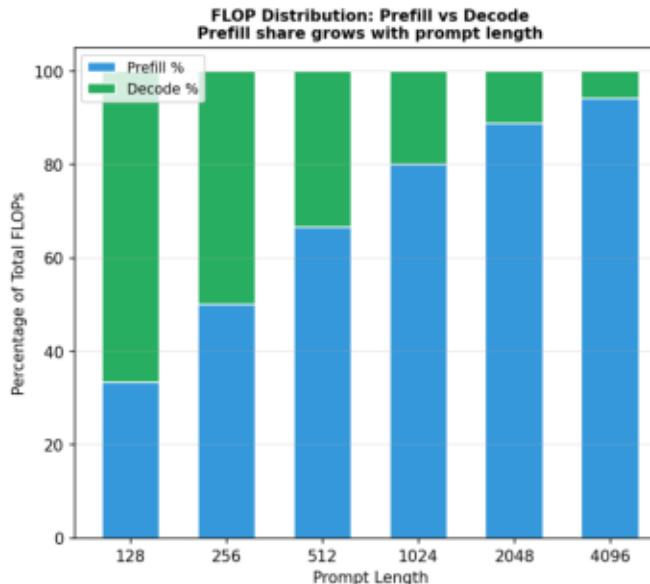
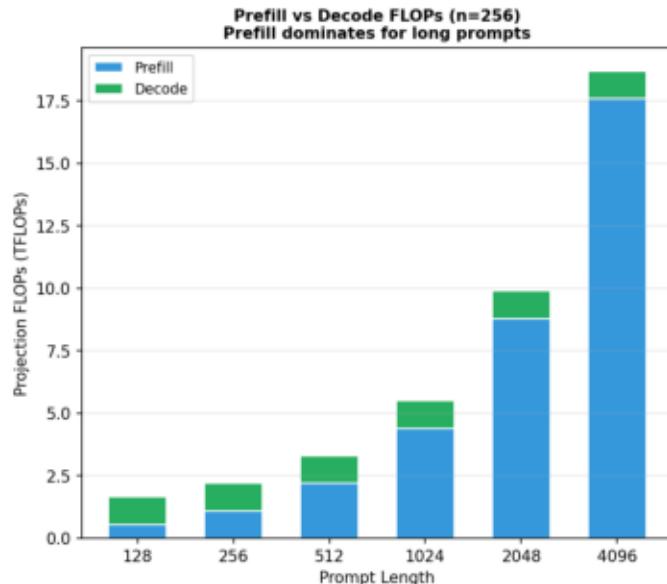
Example 4: Timing Benchmark

Wall-Clock Timing Benchmark: Cached vs Uncached Generation



Example 5: Prefill vs Decode Phase Analysis

Prefill vs Decode Phase Analysis: Two Regimes of Inference



PREFILL vs DECODE
=====

PREFILL PHASE:
Input: full prompt (B, P, d)
Operation: batch matmul for Q,K,V
FLOPs: $N * 4 * 2 * P * d^2$
Bound: COMPUTE (high intensity)
Output: first token + full KV cache

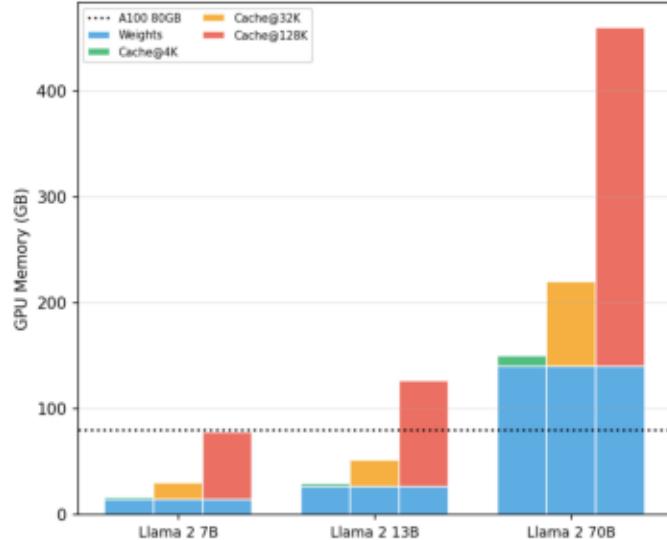
DECODE PHASE:
Input: single token ($B, 1, d$)
Operation: project 1 token,
attend over full cache
FLOPs: $N * 4 * 2 * d^2$ (proj)
+ $N * 2 * \text{seq} * d$ (attn)
Bound: MEMORY (reads full cache)
Output: next token

Implication: Prefill can be parallelized across tokens;
decode is inherently sequential.
This is why time-to-first-token
differs from inter-token latency.

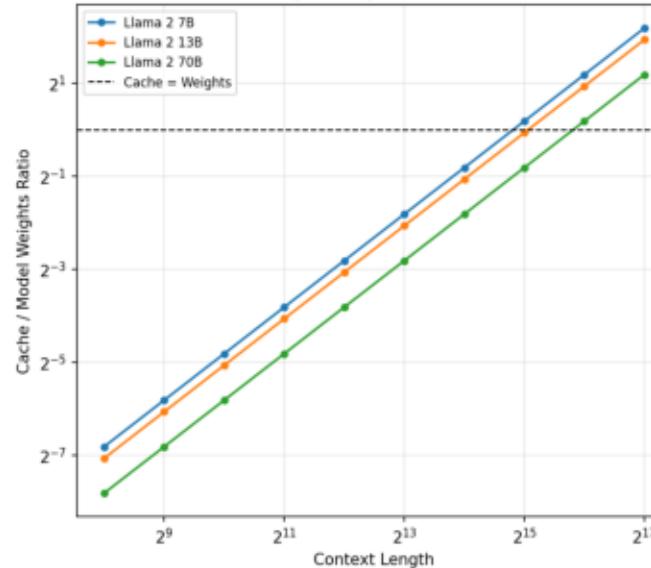
Example 6: KV Cache as THE Memory Bottleneck

KV Cache: The Memory Bottleneck That Defines Inference Systems

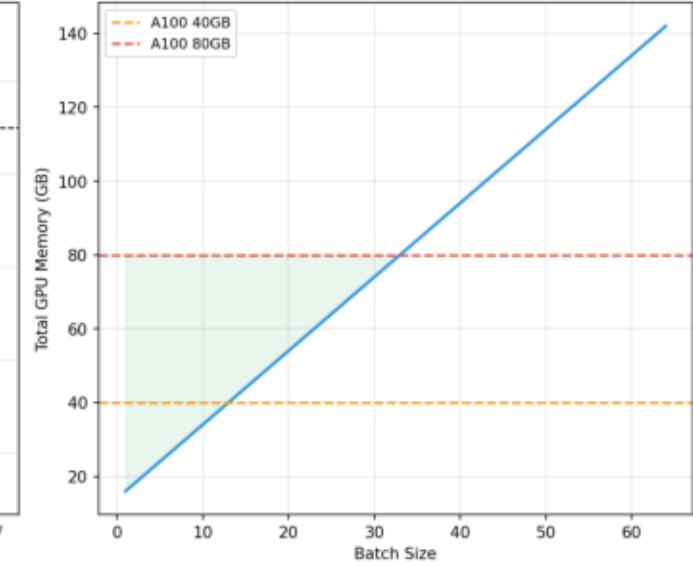
Model Weights + KV Cache (BS=1, FP16)
Cache dominates at long contexts



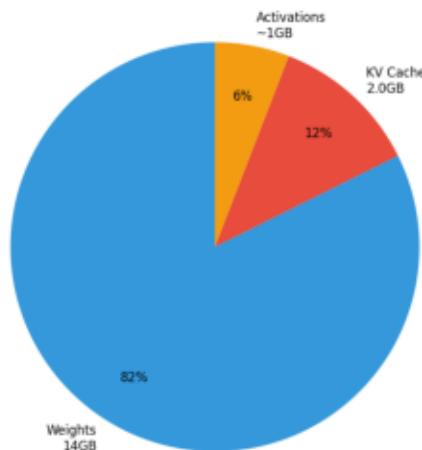
KV Cache / Model Weights Ratio (BS=1)
Crossover point depends on model size



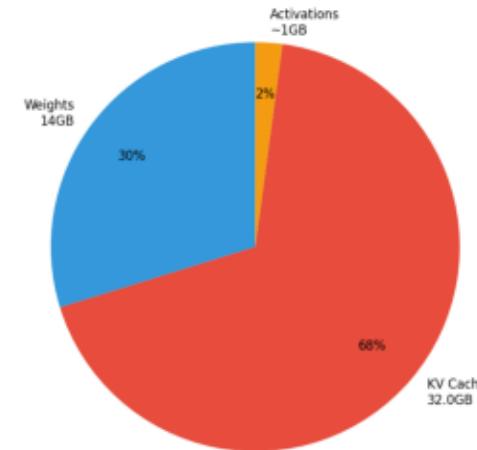
Total Memory vs Batch Size (7B, ctx=4096)
KV cache quickly fills available VRAM



7B, BS=1, 4K
Total: 17.0 GB



7B, BS=16, 4K
Total: 47.0 GB



7B, BS=1, 128K
Total: 79.0 GB

