

PagedAttention & vLLM

Efficient Memory Management for Large Language Model Serving with PagedAttention
Kwon, Li, Zhuang, Sheng, Zheng, Yu, Gonzalez, Zhang, Stoica · SOSP '23 · arXiv:2309.06180 · Study sheet for L5 system design

Problem

LLM decoding is memory-bound, not compute-bound. The decode phase emits one token per step, so it runs matrix-vector products instead of matrix-matrix. GPU FLOPs sit idle; the only lever for throughput is a bigger batch. Batch size is capped by the KV cache, and the KV cache is managed badly.

The capacity math (memorize this — it's the opening move in the interview)

- KV bytes per token = $2 \times (K,V) \times \text{hidden} \times \text{layers} \times \text{dtype_bytes}$. For OPT-13B: $2 \times 5120 \times 40 \times 2 = 800$ KB/token.
- At a 2048-token max length, **one request can demand 1.6 GB of KV cache.**
- On a 40 GB A100: weights 26 GB (65%), KV cache budget $\approx$ 12 GB, rest is activations. That's **$\sim$ 7 concurrent requests** if you provision for max length.
- Trend makes it worse: A100→H100 more than doubles FLOPs, but HBM stays at 80 GB. Memory is the widening bottleneck.

Why prior systems (FasterTransformer, Orca) waste it

DL frameworks require tensors in contiguous memory, so they pre-allocate a contiguous chunk sized to the request's *maximum* length. Output length is unknown a priori, so that chunk is a worst-case guess. Three distinct wastes:

Waste type	Cause	Known when?
Internal frag.	Reserved 2048 slots, sequence ended at 10. Those slots are never touched.	Only after the request finishes
Reservation	Slots that <i>will</i> be used later, but are held idle for the request's whole lifetime — blocking other requests <i>now</i> .	N/A — pure opportunity cost
External frag.	Buddy allocator gaps between variable-sized chunks. Never usable at all.	Before serving even starts

Measured result: **only 20.4%–38.2% of KV memory actually holds token state.** vLLM reaches 96.3%.

Second, orthogonal problem: no sharing

Parallel sampling, beam search, and shared system prompts all produce sequences with *byte-identical prefix KV*. Contiguous per-sequence buffers make sharing structurally impossible — every beam candidate holds its own full copy, and divergence forces bulk memory copies.

Intuition

The unlock: contiguity was never a requirement of the attention *math*. Attention is a sum over positions, so it decomposes cleanly into a sum over *groups* of positions. Contiguity was a requirement of the *kernel implementation*. Rewrite the kernel to walk an indirection table, and the memory manager is suddenly free to do whatever it wants.

Once you're free, this is the 1962 operating-systems problem verbatim. The paper's mapping:

OS virtual memory	vLLM	What it buys
Page	KV block (B tokens)	Fixed size $\Rightarrow$ external fragmentation is mathematically zero (all blocks interchangeable)
Byte	Token's K/V vectors	Small B $\Rightarrow$ internal waste bounded by < 1 block per sequence
Process	Request / sequence	Independent address spaces
Page table	Block table	The indirection layer — sharing and CoW fall out for free
Copy-on-write on fork	CoW on beam divergence	Diverging beams copy one block , not the whole prefix
Swap to disk	Swap to CPU RAM	Eviction under pressure
Shared library pages	Cached system-prompt prefix	Skip prefill for the shared part

Interview framing: "This is virtual memory for the KV cache." Then immediately state the tradeoff you accepted: **you trade a slower attention kernel for a much larger batch.** That trade is the whole paper, and stating it unprompted is what separates a senior answer from a recital.

Solution

1. PagedAttention — the kernel

Partition each sequence's KV into blocks of B tokens. Key block K_j , value block V_j . Attention becomes blockwise:

```
A_ij = softmax_over_j( q_i · K_j / sqrt(d) )    # score row-vector vs block j
o_i = sum over j of V_j · A_ij^T              # j = 1 .. ceil(i/B)
```

The kernel fetches each block separately via the block table. Blocks live wherever there is free physical space.

2. KV cache manager — the allocator

- Logical blocks (per sequence, dense, left-to-right) → **block table** → physical blocks (scattered in a pre-carved GPU DRAM arena).
- Block table entry = `(physical_block_id, num_filled)`.
- **Allocate one block at a time, only when the previous one is full.** No max-length reservation, ever. Total waste per sequence $\leq B-1$ slots.
- Freed on completion, immediately reusable by any other request (uniform size).

3. Sharing — reference counts + copy-on-write

- **Parallel sampling:** all n samples map their prompt logical blocks to the *same* physical blocks; refcount = n . On first divergent write, allocate a fresh block, copy, decrement. Savings: 6.1–9.8% (Alpaca), 16.2–30.5% (ShareGPT).
- **Beam search:** sharing pattern is a *tree that mutates every step* — like compound forks. Dropped candidates just decrement refcounts; blocks hitting 0 are freed. Savings: 37.6–55.2% (Alpaca), 44.3–66.3% (ShareGPT). Prior systems needed bulk KV copies here; vLLM copies at most one block.
- **Shared prefix (system prompt / few-shot):** provider pre-registers physical blocks for the prefix; incoming requests just map to them and prefill only their own suffix. **1.67× throughput at 1-shot (80 tok prefix), 3.58× at 5-shot (341 tok).**
- **Mixed decoding in one batch:** the kernel only ever sees a list of physical block IDs. All sharing complexity is hidden in the mapping layer, so requests using different decoding algorithms batch together — something prior systems couldn't do.

4. Scheduling & preemption

- **FCFS**, latest-arrived preempted first (fairness, no starvation).
- **All-or-nothing eviction.** No LRU heuristic needed: every block of a sequence is accessed on every step, so partial eviction is worthless. Evict the whole sequence or none.
- **Gang scheduling:** beam candidates form a "sequence group" and are preempted/resumed together, because they share blocks.
- **Two recovery paths:**
 - *Swap* to CPU RAM. Swap space is bounded by the GPU KV arena size (nice property — it can never blow up).
 - *Recompute*. Cheaper than it sounds: prompt + already-generated tokens concatenate into **one prefill pass**, not T sequential decode steps.

Recompute wins at small block sizes (small PCIe transfers waste bandwidth); swap wins at large. They're comparable for $B = 16-64$, and recompute overhead is never worse than 20% of swap's latency.

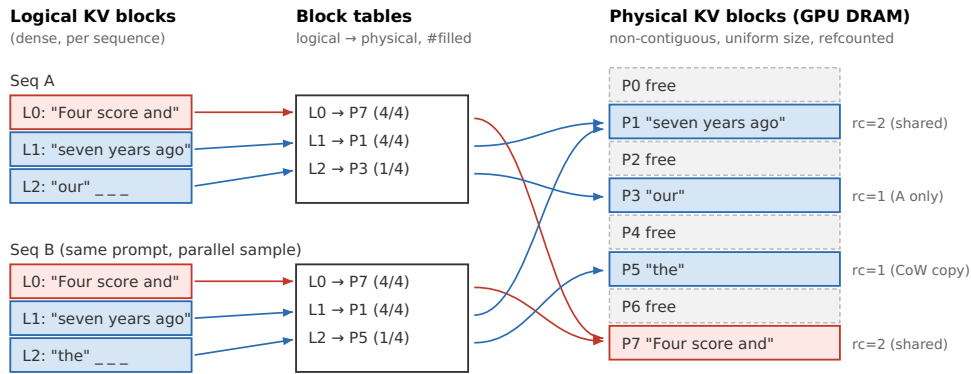
5. Distributed execution

Megatron-style tensor parallelism, attention split on the head dimension. Key observation: **every shard processes the same token positions, so one block table serves all workers.** A single centralized KV manager broadcasts `(token_ids, block_tables)` to workers each step; workers all-reduce among themselves without touching the scheduler. Workers never synchronize on memory management.

6. Kernel-level work (the part people forget to mention)

- Fused reshape + block write (avoid per-layer launch overhead).
- Fused block read + attention; **one GPU warp per block** to keep memory access coalesced.
- Fused block copy — CoW touches discontinuous blocks, so batch all copies into one launch instead of many `cudaMemcpyAsync` calls.

Solution Diagram



Read this as: both sequences' prompt blocks point at the SAME physical blocks (red, rc=2). Zero duplication. B's divergent token hit shared block L2. refcount > 1 triggered copy-on-write: allocate P5, copy, decrement. One block was copied — not the prefix. Free blocks (dashed) are interchangeable, so external fragmentation is zero.

Logical → block table → physical mapping, with refcounting and copy-on-write.

Code

The block manager is the part worth being able to write on a whiteboard. It is ~40 lines and it *is* the design.

```
# --- Physical arena: one flat tensor, carved into uniform blocks ---
# kv_cache[layer] shape: (num_blocks, block_size, num_heads, head_dim) x2 (K,V)

class BlockManager:
    def __init__(self, num_blocks, block_size):
        self.block_size = block_size
        self.free = list(range(num_blocks))  # 0(1) alloc/free: uniform size
        self.refcount = [0] * num_blocks
        self.tables = {}  # seq_id -> [physical_block_ids]
        self.filled = {}  # seq_id -> tokens in LAST block

    def _alloc(self):
        if not self.free: raise OutOfMemory  # -> scheduler preempts a sequence
        b = self.free.pop(); self.refcount[b] = 1; return b

    def _release(self, b):
        self.refcount[b] -= 1
        if self.refcount[b] == 0: self.free.append(b)

    def append_token(self, seq_id):
        """Called once per decode step. Grows by AT MOST one block."""
        tbl, n = self.tables[seq_id], self.filled[seq_id]
        if n == self.block_size:  # last block full -> new block
            tbl.append(self._alloc()); self.filled[seq_id] = 1
        elif self.refcount[tbl[-1]] > 1:  # --- COPY ON WRITE ---
            old = tbl[-1]; new = self._alloc()
            copy_block(new, old)  # fused CUDA kernel, 1 launch
            tbl[-1] = new; self._release(old)
            self.filled[seq_id] = n + 1
        else:
            self.filled[seq_id] = n + 1
        return tbl[-1], self.filled[seq_id] - 1  # (physical block, slot) to write

    def fork(self, parent, child):
        """Parallel sampling / beam branch: share everything, copy nothing."""
        self.tables[child] = list(self.tables[parent])
        self.filled[child] = self.filled[parent]
        for b in self.tables[child]: self.refcount[b] += 1

    def free_seq(self, seq_id):
        for b in self.tables.pop(seq_id): self._release(b)
        self.filled.pop(seq_id)
```

```
# --- PagedAttention: gather via block table instead of slicing a contiguous tensor ---
# Real version is a CUDA kernel (one warp per block, coalesced). Semantics:
```

```
def paged_attention(q_i, block_table, ctx_len, k_cache, v_cache, block_size):
    scores = []
    for j, phys in enumerate(block_table):
        valid = min(block_size, ctx_len - j * block_size)
        K_j = k_cache[phys, :valid] # gather, NOT a contiguous slice
        scores.append(q_i @ K_j.T / sqrt(d))
    A = softmax(concat(scores)) # softmax spans all blocks
    o = 0
    for j, phys in enumerate(block_table):
        valid = min(block_size, ctx_len - j * block_size)
        A_j = A[j * block_size : j * block_size + valid]
        o += A_j @ v_cache[phys, :valid]
    return o
```

The one line that matters: `k_cache[phys]` — an indexed gather — replaces `k_cache[seq_start : seq_start + len]` — a contiguous slice. Everything else in the system is a consequence of that substitution.

Results — and the tradeoff to volunteer

Measurement	Number
Throughput vs Orca (state of the art), same latency	2-4×
Throughput vs FasterTransformer (no fine-grained scheduling)	up to 22×
Batched requests, OPT-13B/ShareGPT (Orca-Oracle → vLLM)	13.6 → 30.4
KV memory actually holding token state	20-38% → 96.3%
Shared 5-shot prefix, translation workload	3.58×
Attention kernel latency vs FasterTransformer	20-26% SLOWER
Default block size (sweet spot: GPU parallelism vs internal frag.)	16

Say this out loud in the interview: the attention kernel got *worse* — block-table lookups, extra branches, variable lengths. They shipped it anyway because attention is one operator among many, and a 2-4× larger batch dominates a 25% regression on one kernel. That's the correct way to reason about a system-level optimization, and it's an argument that transfers to almost any infra design question.

When NOT to apply this (the paper says so, and interviewers probe it)

- **DNN training:** tensor shapes are static, so memory can be planned ahead of time. Paging adds indirection for nothing.
- **Non-LLM inference:** typically compute-bound. Freeing memory buys no throughput, and you pay the indirection cost.
- The preconditions are specifically: **dynamic, unpredictable allocation lifetime + memory-capacity-bound performance**. State those two conditions and you've shown you understand the idea rather than the artifact.

What's not covered

Being able to name the gaps is what makes this paper usable as a *design* answer rather than a summary. Every item below became a follow-up paper or a vLLM feature after publication.

Inside the serving loop

- **No automatic prefix caching.** Shared prefixes must be *pre-registered by the operator*. Automatic, content-hash-based prefix reuse across arbitrary requests came later (and is now vLLM's APC).
- **No cross-turn KV reuse.** The chatbot experiment explicitly *discards* KV between conversation rounds, which is why it re-prefills 1024 tokens each turn. Multi-turn cache retention / KV offloading tiers (LMCache-style) are outside scope.
- **Prefill and decode share one batch.** A long prompt's prefill blocks every decode in that step — head-of-line blocking on TTFT. **Chunked prefill** and **prefill/decode disaggregation** (Sarathi, DistServe, Splitwise) are the follow-ups.
- **Scheduling is FCFS only.** No priorities, no SLO-awareness, no per-tenant fairness, no admission control. Preemption is triggered by memory pressure, never by latency targets.
- **Metric is normalized latency** (end-to-end ÷ output length). No TTFT/TPOT split, no p99 tail analysis, no goodput-under-SLO — the metrics an interviewer will actually ask you to define.

Orthogonal techniques it does not address

- **Making the KV cache smaller** rather than better-packed: GQA/MQA, MLA, KV quantization (FP8/INT4), and eviction policies (H2O, StreamingLLM). Note the tension worth raising: GQA cuts KV by ~8×, which partly *relieves* the pressure this paper attacks. Both are still needed; they compose.

- **Speculative decoding** — attacks the sequential-decode bottleneck itself, not memory.
- **LoRA multi-tenancy** (S-LoRA), **MoE** expert placement, weight quantization, weight offloading.
- **FlashAttention integration.** The 2023 kernel is adapted from FasterTransformer; the FlashAttention-compatible paged kernel came afterward.

Architectural scope limits

- **Tensor parallelism only.** No pipeline parallelism, no multi-node KV transfer, no autoscaling, no replica placement, no cross-GPU cache migration.
- **Centralized scheduler.** Single point of coordination and failure; the paper doesn't analyze its scaling limit or failure recovery.
- **No fault tolerance** — a preempted-and-swapped sequence lives only in that node's CPU RAM.
- **Swap-out stalls admission:** once preemption begins, vLLM *stops accepting new requests* until preempted sequences drain. That's a real availability cliff and a great thing to poke at.

Follow-up reading, in order: Orca (iteration-level scheduling — the complementary half of vLLM's win) → Sarathi-Serve (chunked prefill) → DistServe / Splitwise (prefill-decode disaggregation) → S-LoRA (multi-tenant adapters) → vAttention (2024, argues you can get paging via CUDA virtual-memory APIs and keep contiguity, avoiding the kernel rewrite entirely — the strongest published counterargument to PagedAttention's design choice).