

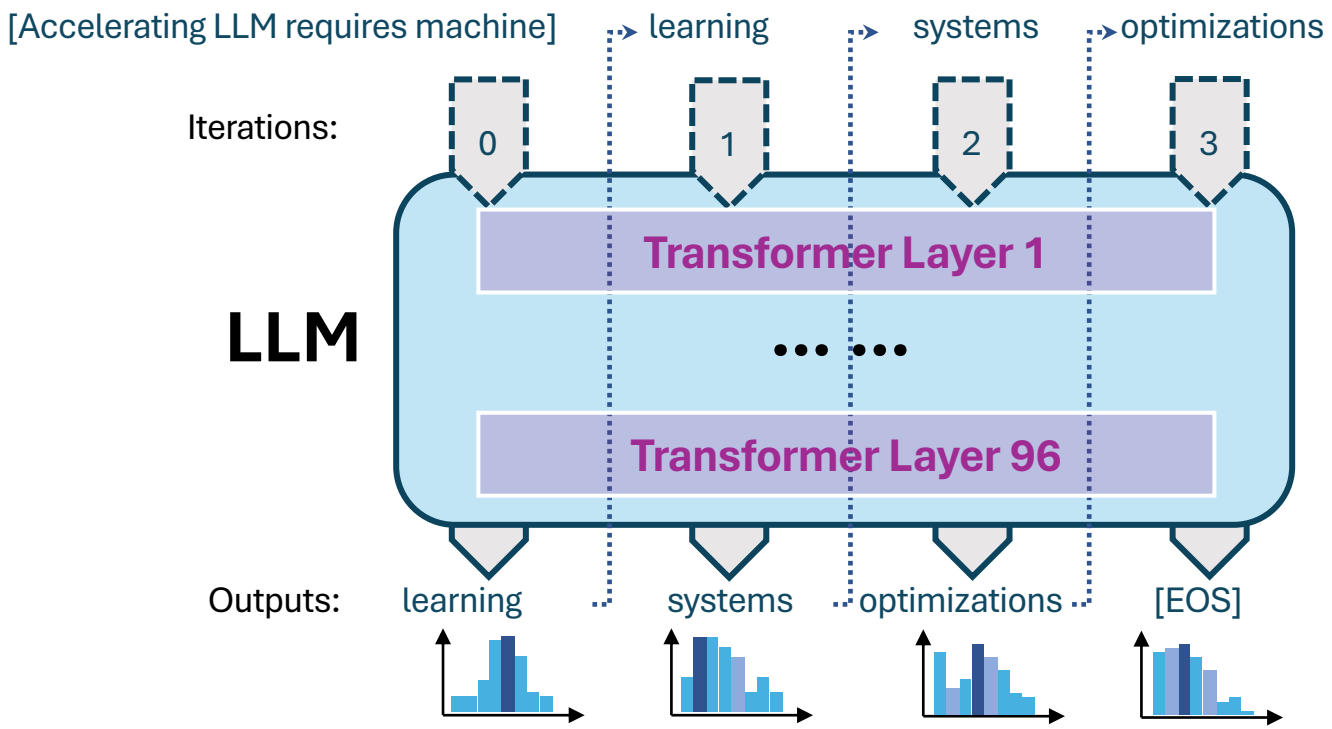
Inference Systems

CSCI 5942

Fall 2026

College of Engineering & Applied Science
UNIVERSITY OF COLORADO **BOULDER**

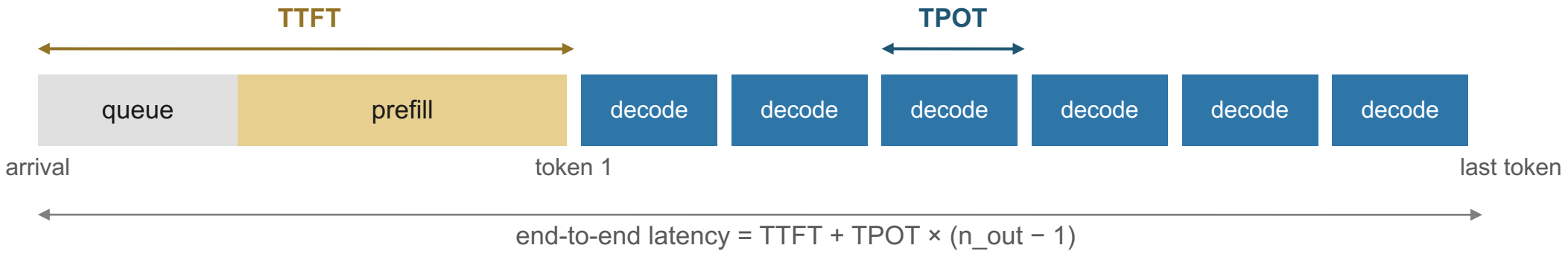
What is LLM inference?



Main systems issues compared to training?

Figure from Tianqi Chen and Zhihao Jia

Inference serving stages/metrics



TTFT

Time to first token: queueing + prefill.

(Interactive apps want hundreds of ms)

TPOT / ITL / TBT

Time per output token after the first.

(Often times, faster than human reading latency is okay)

Throughput

Tokens/s or requests/s across all users.

(Operators want to maximize this)

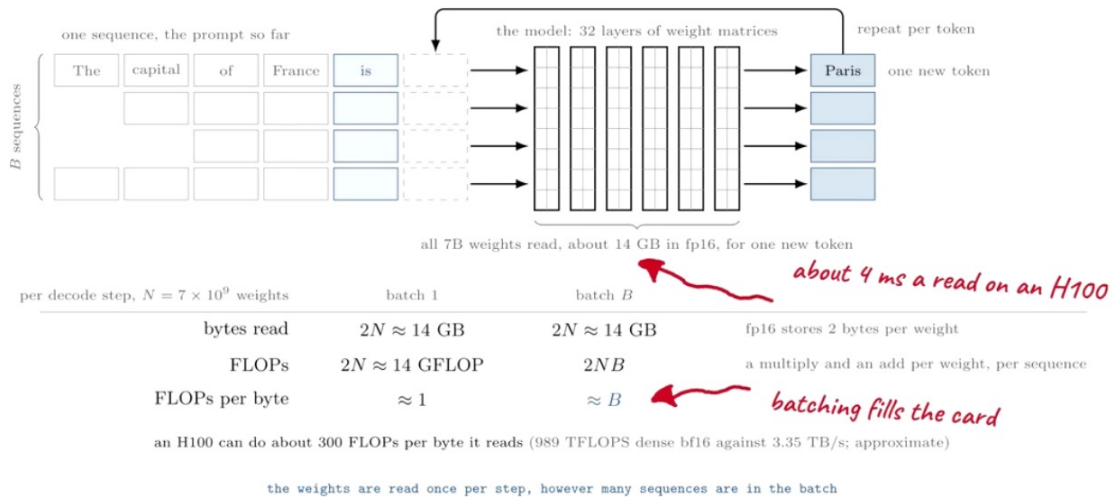
Goodput

Requests/s whose TTFT and TPOT both meet the SLO.

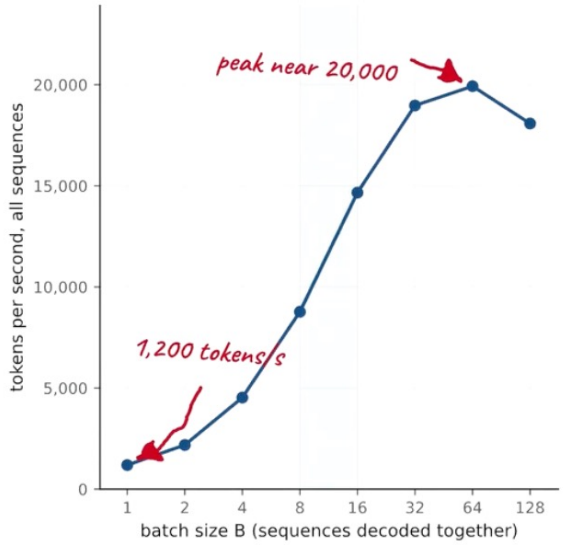
(SLO differs from app to app)

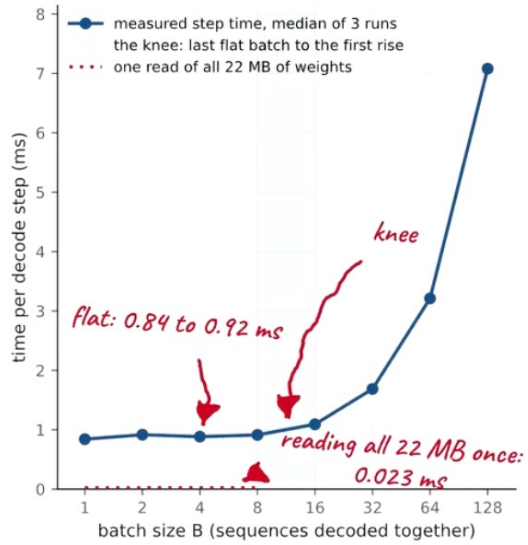
Recap from earlier lecture

What a Step Moves

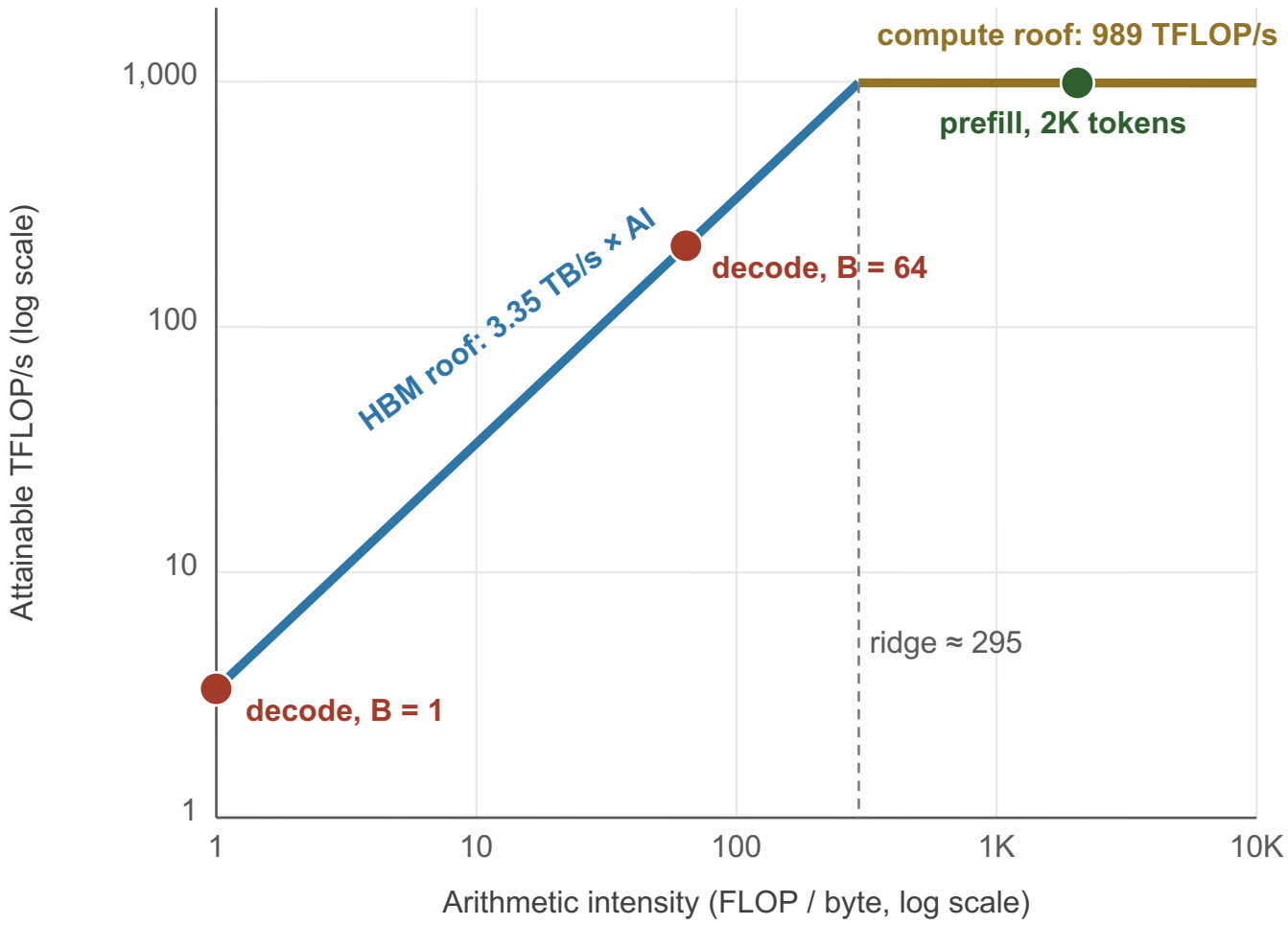


Time per Step





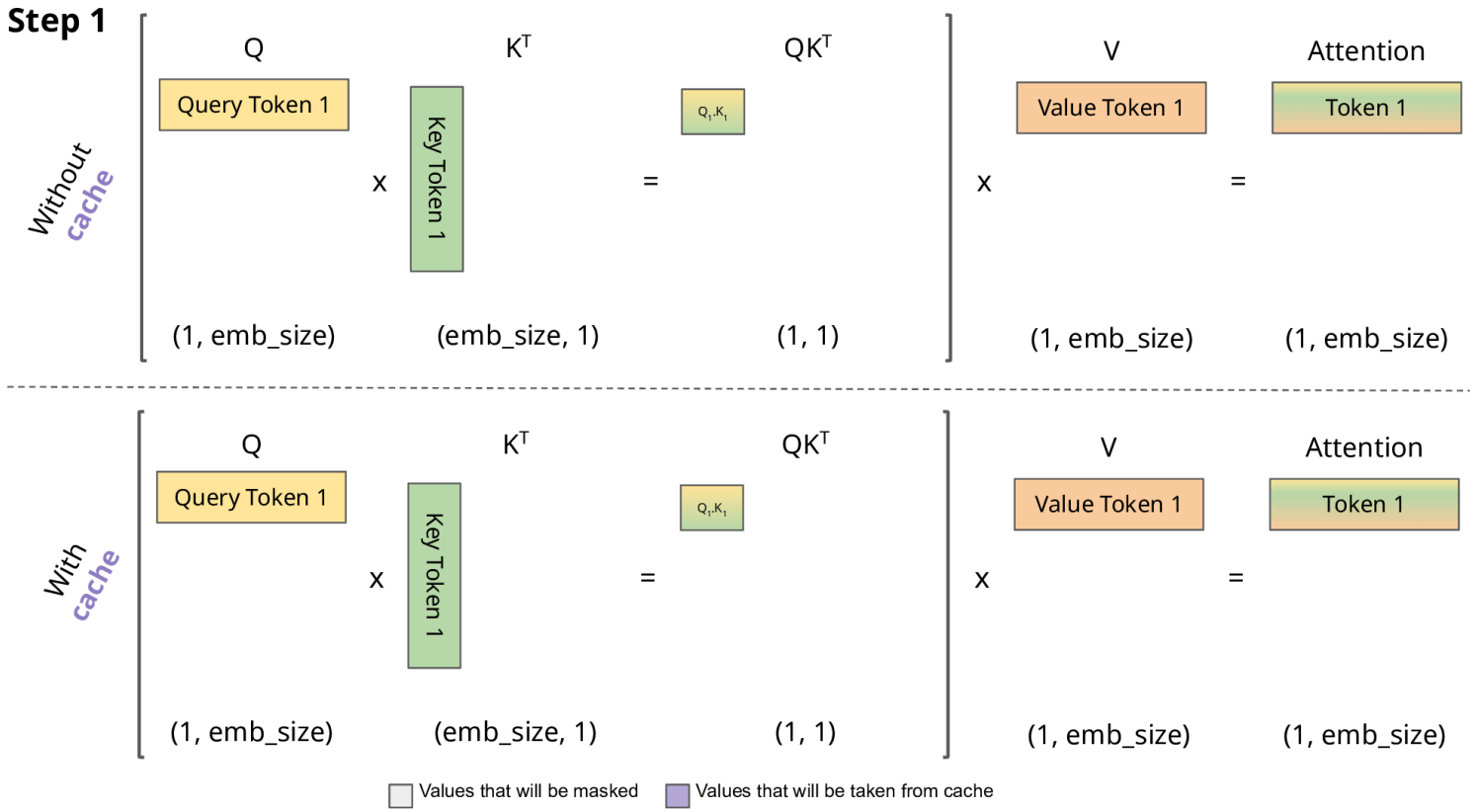
A closer look at the roofline



H100 Roofline

Recap: KV Cache

- Key Idea: *Cache* generated Key, Value vectors
  - No need to re-compute Q, K, V
  - Much smaller matrix multiplications



Animation from https://medium.com/@joaolages/kv-caching-explained-276520203249

What is the fundamental trade-off of KV Cache?

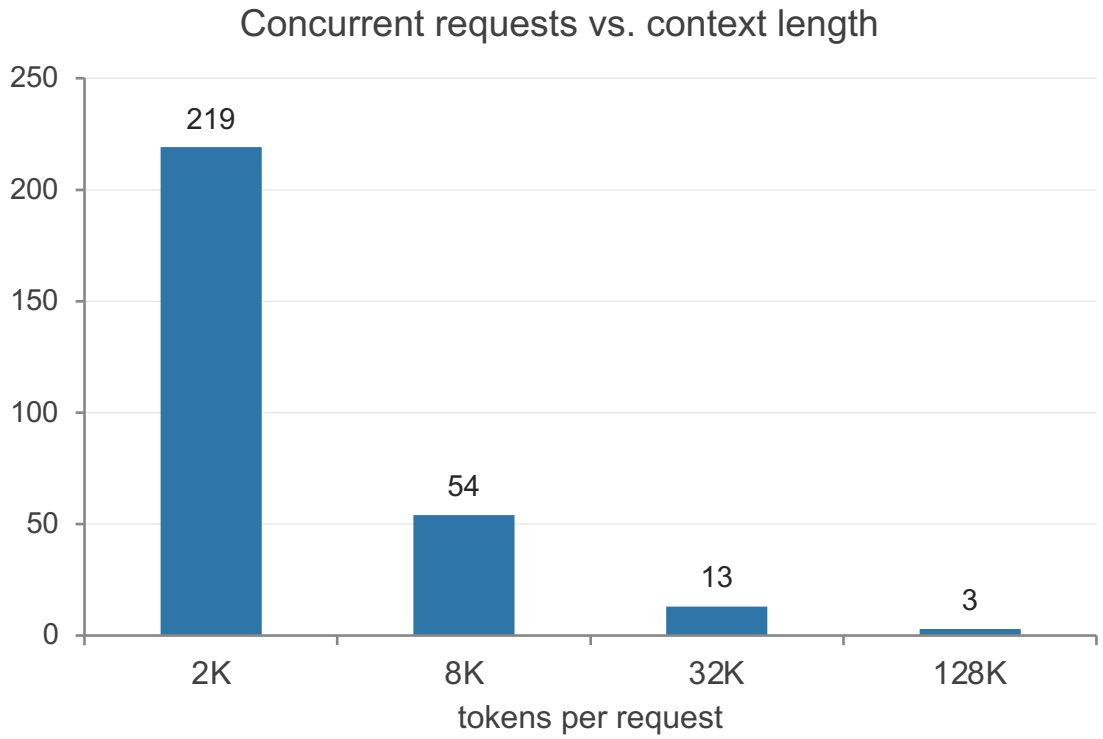
How big is the KV cache?

bytes per token = 2 (K and V) × n_layers × n_kv_heads × d_head × bytes per element

Model	Attention	Layers	KV heads × d_head	KV per token (BF16)
Llama-2-7B	MHA	32	32 × 128	**512 KB**
Llama-3.1-8B	GQA	32	8 × 128	**128 KB**
Llama-3.1-70B	GQA	80	8 × 128	**320 KB**
Llama-3.1-405B	GQA	126	8 × 128	**504 KB**
DeepSeek-V3 (671B MoE)	MLA	61	1 latent: 512 + 64 (RoPE key)	**≈ 69 KB**

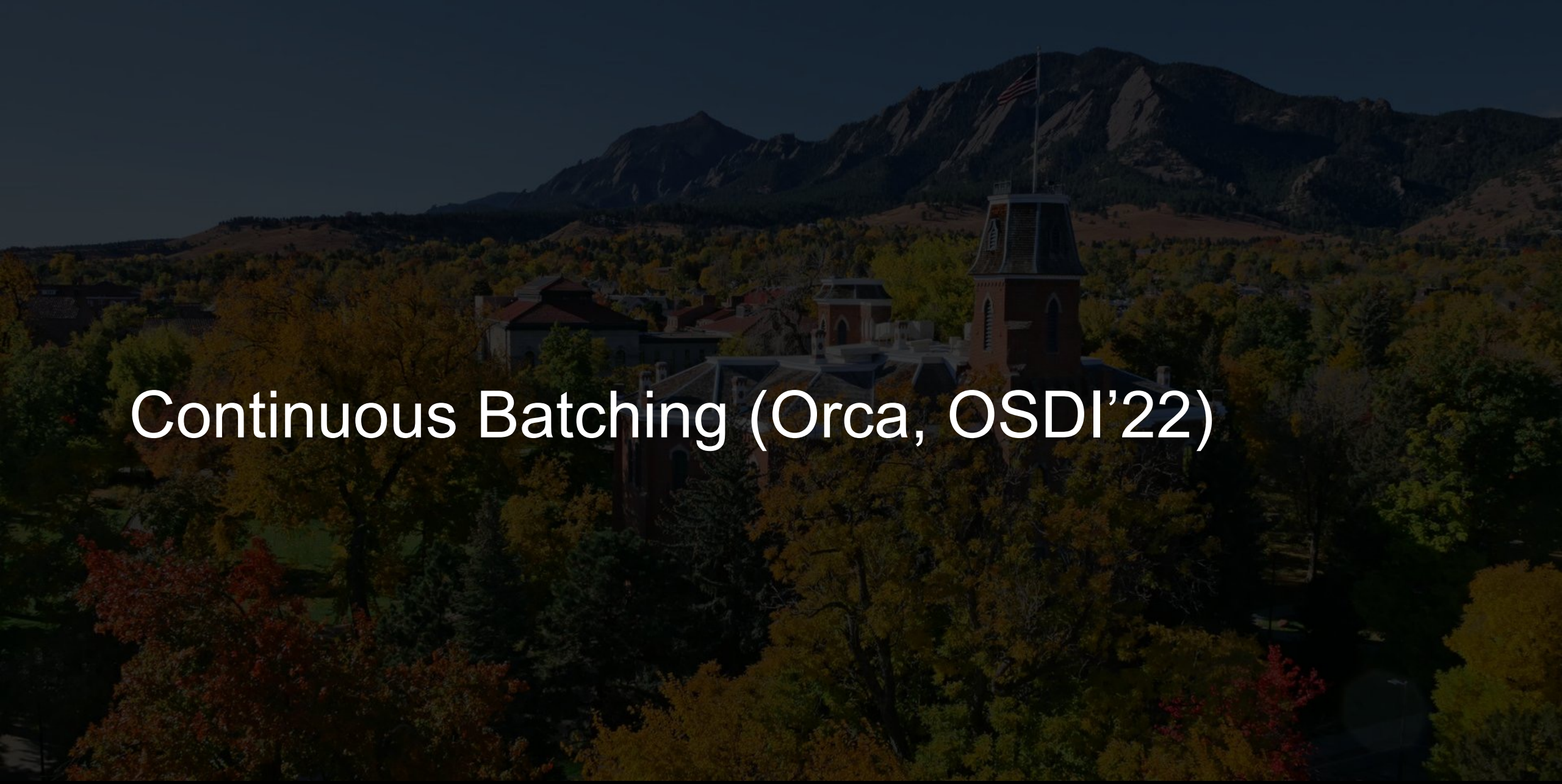
How many requests fit?

- Scenario:
  - Llama-3-70B: 320 KB/token
  - TP=4 across H100 GPUs (320 GB)
  - How many concurrent requests fit?



LLM Inference Systems

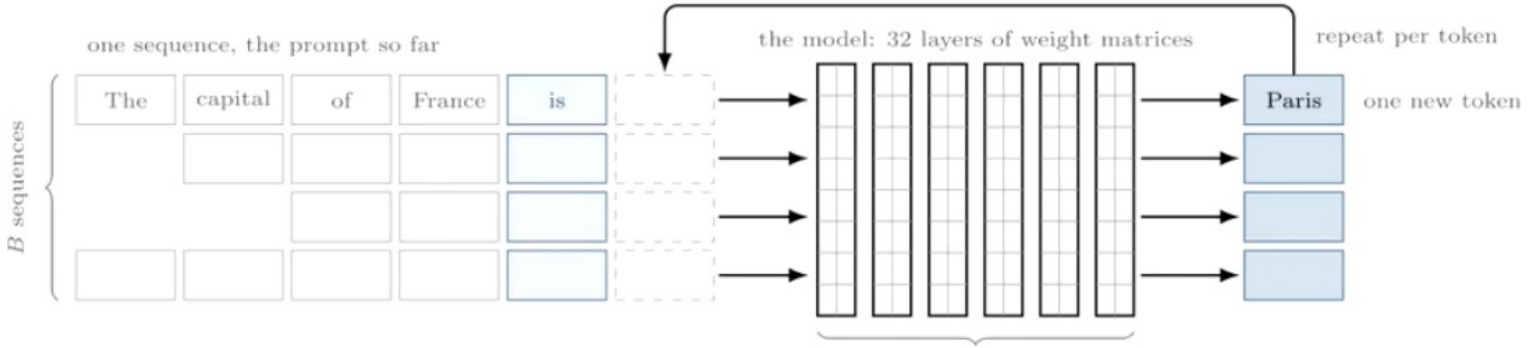
- Goal: Reduce TTFT / TBT, increase throughput/goodput
- Systems Challenges
  - Prefill and Decode are distinct stages of inference, exhibit completely different hardware requirements (memory vs compute)
  - Want high batch sizes to increase arithmetic intensity of decode
  - KV cache necessary to reduce compute complexity, but comes at the (massive) cost of linearly-increasing memory w.r.t. context length



Continuous Batching (Orca, OSDI'22)

Challenge 1: Batching

What a Step Moves



about 4 ms a read on an H100

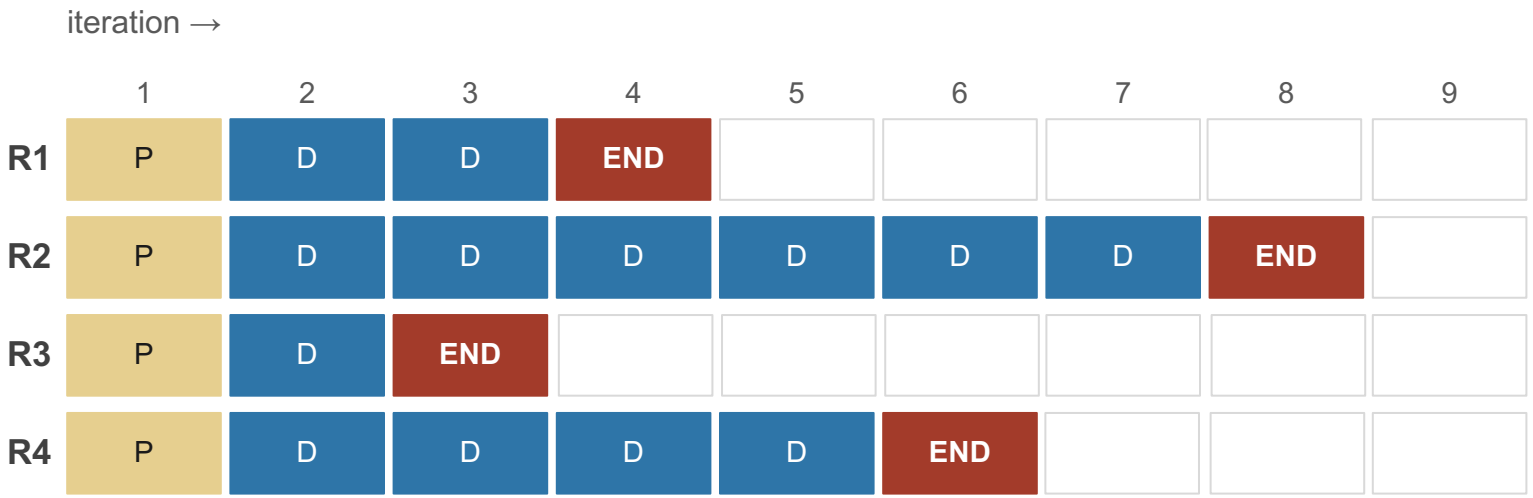
per decode step, $N = 7 \times 10^9$ weights	batch 1	batch B	
bytes read	$2N \approx 14$ GB	$2N \approx 14$ GB	fp16 stores 2 bytes per weight
FLOPs	$2N \approx 14$ GFLOP	$2NB$	a multiply and an add per weight, per sequence
FLOPs per byte	≈ 1	$\approx B$	

batching fills the card

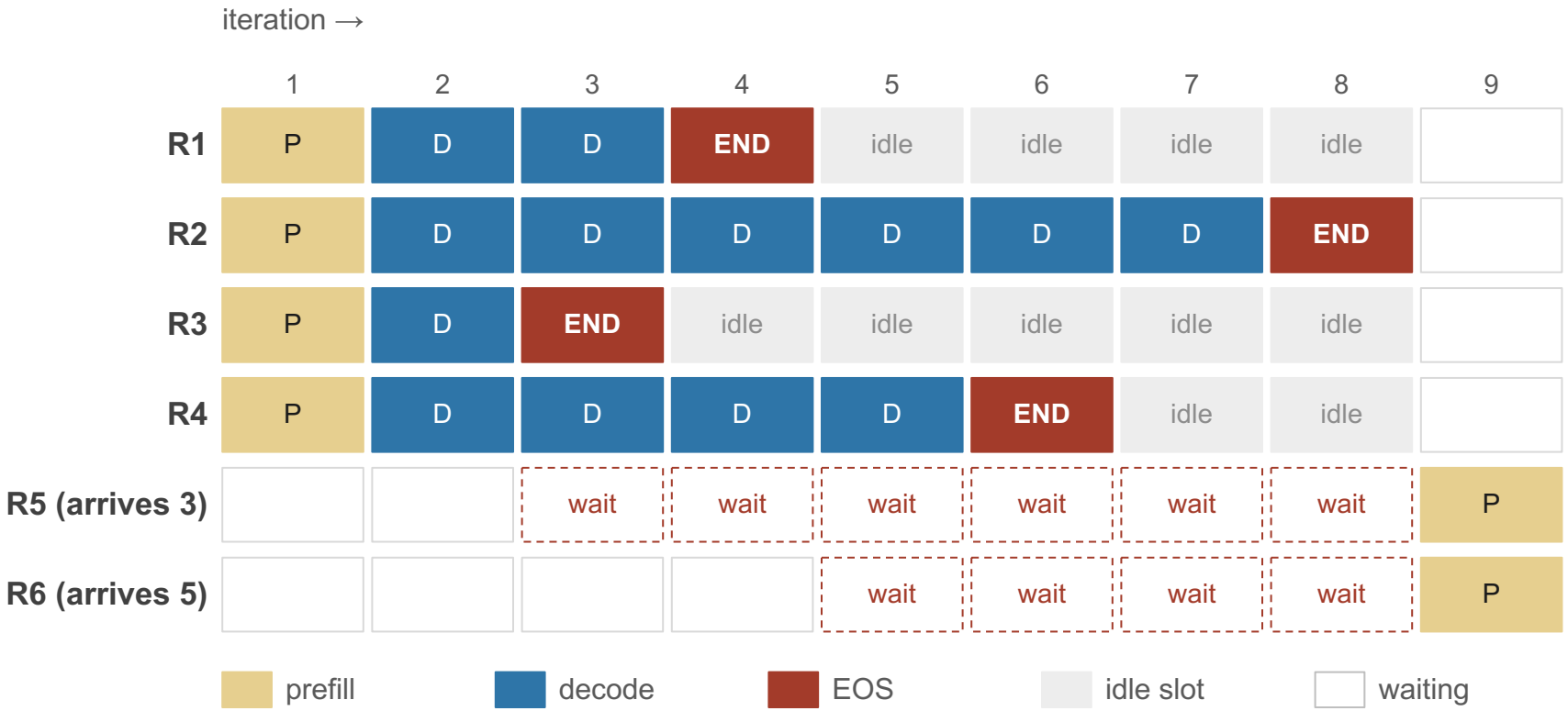
an H100 can do about 300 FLOPs per byte it reads (989 TFLOPS dense bf16 against 3.35 TB/s; approximate)

the weights are read once per step, however many sequences are in the batch

Variable length autoregressive decoding



Static batching: all requests are grouped



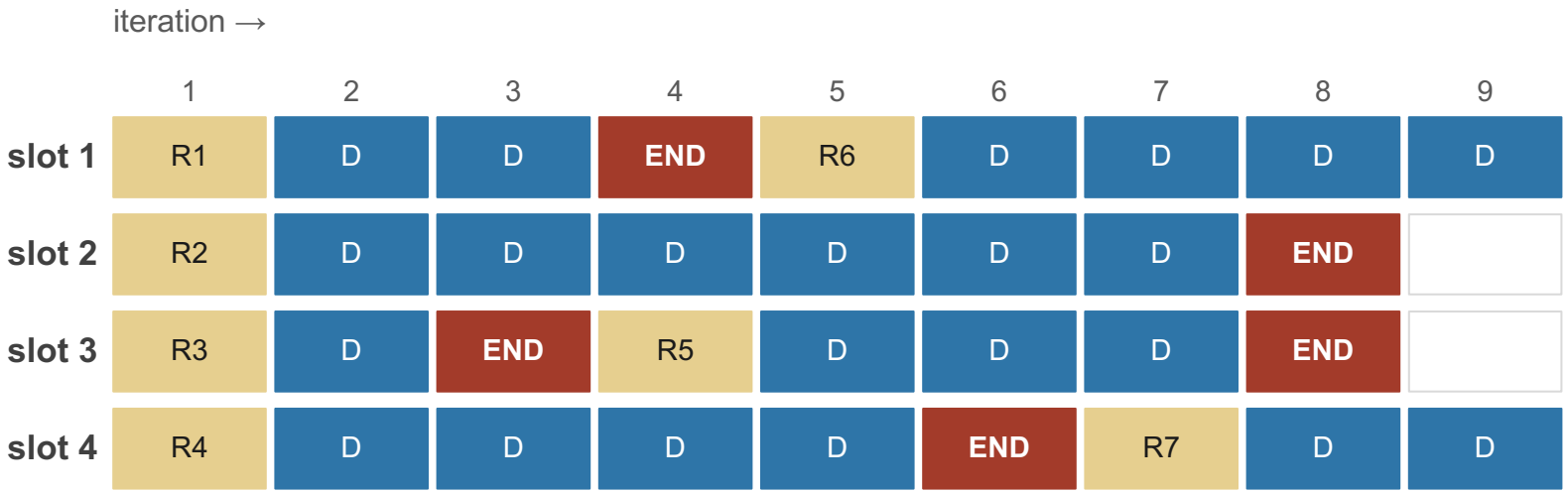
How much does static batching waste?

$W = 1 - \bar{S} / S_{max}$ **(S = output length; $\bar{S}$ = batch mean)**

Request	Output tokens	Done: static	Done: continuous
R1	50	4,000 ms	1,000 ms
R2	200	4,000 ms	4,000 ms
R3	100	4,000 ms	2,000 ms
R4	150	4,000 ms	3,000 ms

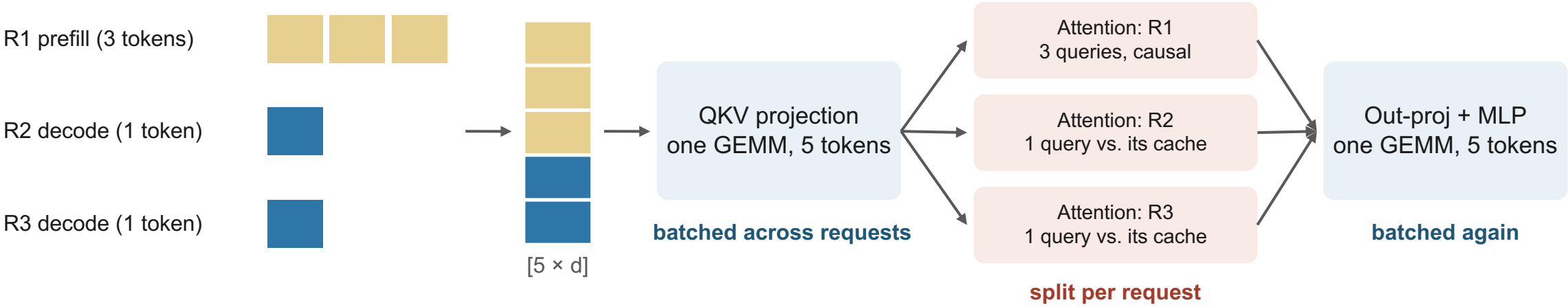
Ex: 20 ms per decode iteration; $\bar{S}$ = 125, S_{max} = 200.

Continuous batching: schedule every iteration



- At every iteration: retire finished requests, admit waiting ones, run one step
- Short requests leave as soon as they finish
- Potential issues with this approach?

Selective Batching



- **Issues:**
  - Continuous batching results in “ragged” tensors of different lengths (e.g., prefill and decode).
  - Requests may have different sequence lengths (and different attention matrix shapes)
- **Idea**: Selectively batch operations across different requests
- **Token-wise ops** (projections, MLP, norms) don’t care which request a token belongs to → concatenate all tokens
- **Attention** mixes tokens within a request → split by request

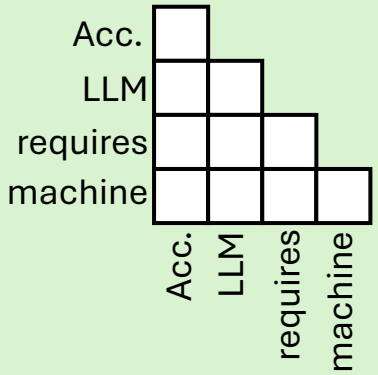


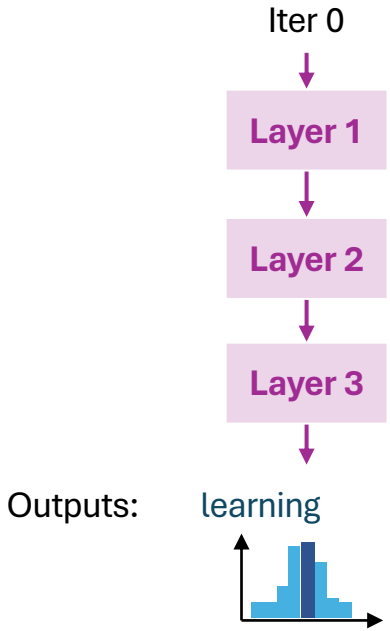
PagedAttention/vLLM (SOSP'23)

The KV cache dynamically grows and shrinks

[Accelerating LLM requires machine]

Attention Matrix



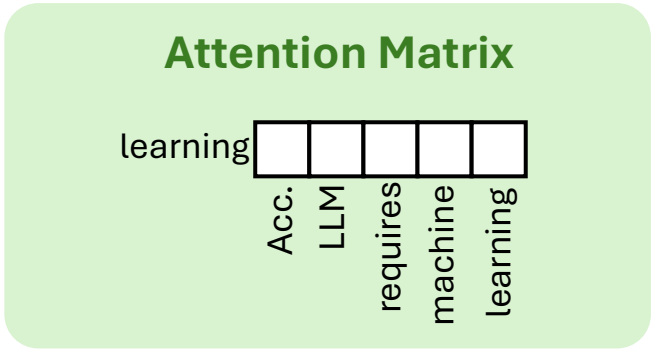


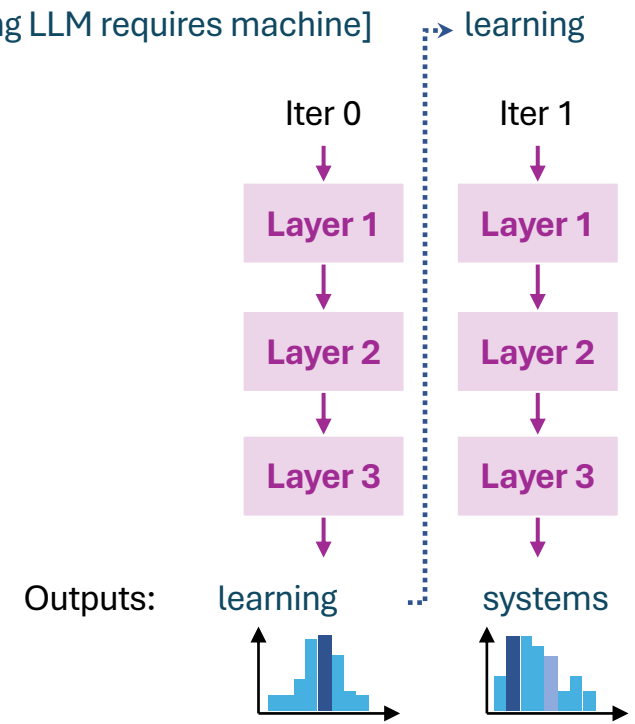
KV Cache

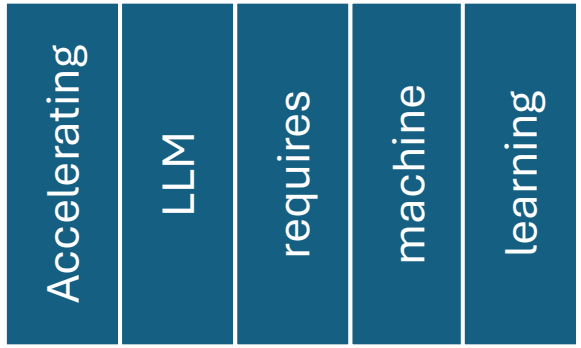


Slides from Zhihao Jia, Tianqi Chen

The KV cache dynamically grows and shrinks

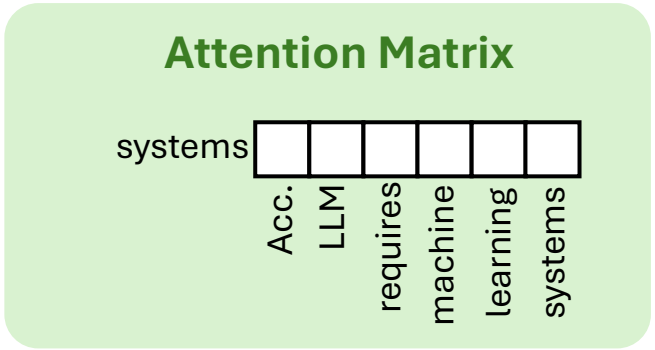


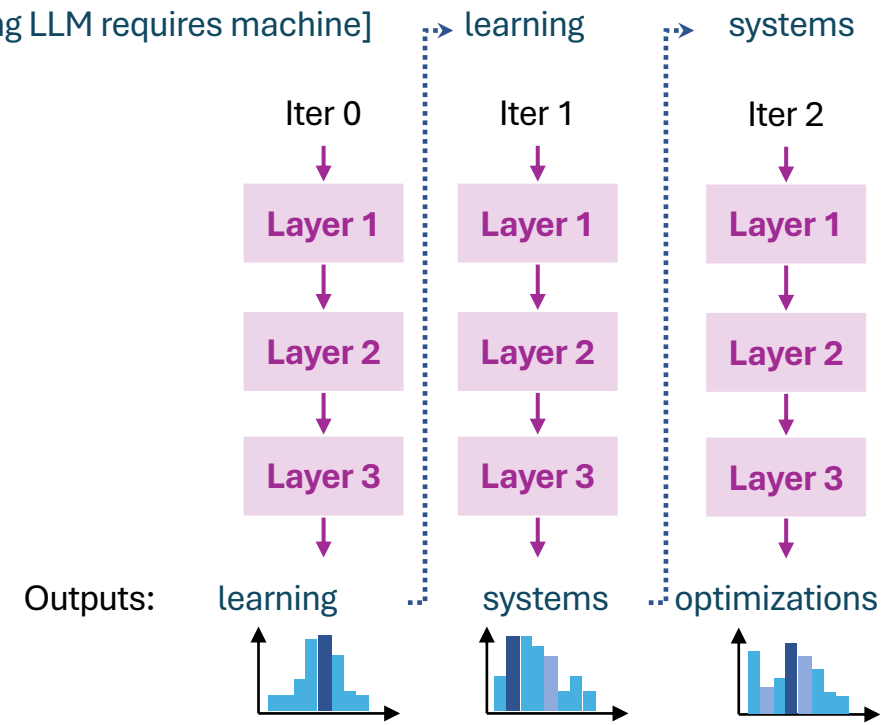


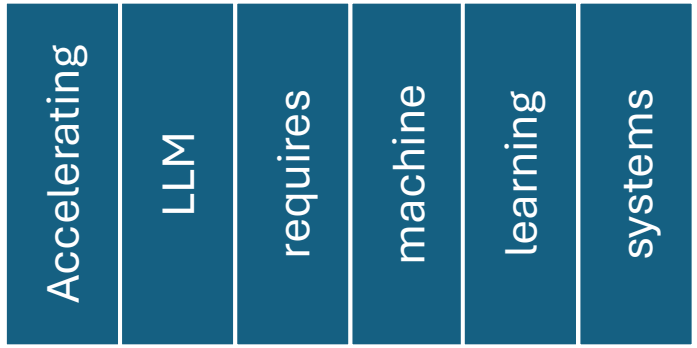


Slides from Zhihao Jia, Tianqi Chen

The KV cache dynamically grows and shrinks

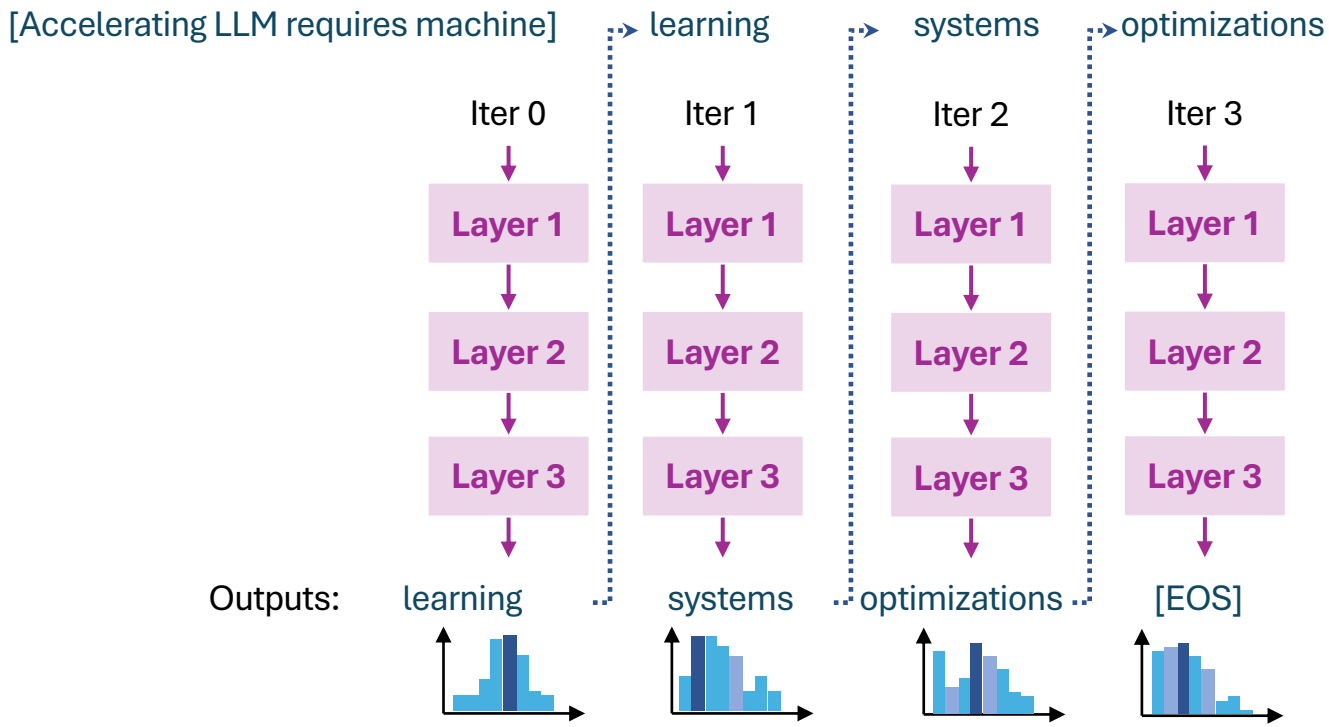






Slides from Zhihao Jia, Tianqi Chen

The KV cache dynamically grows and shrinks



KV Cache



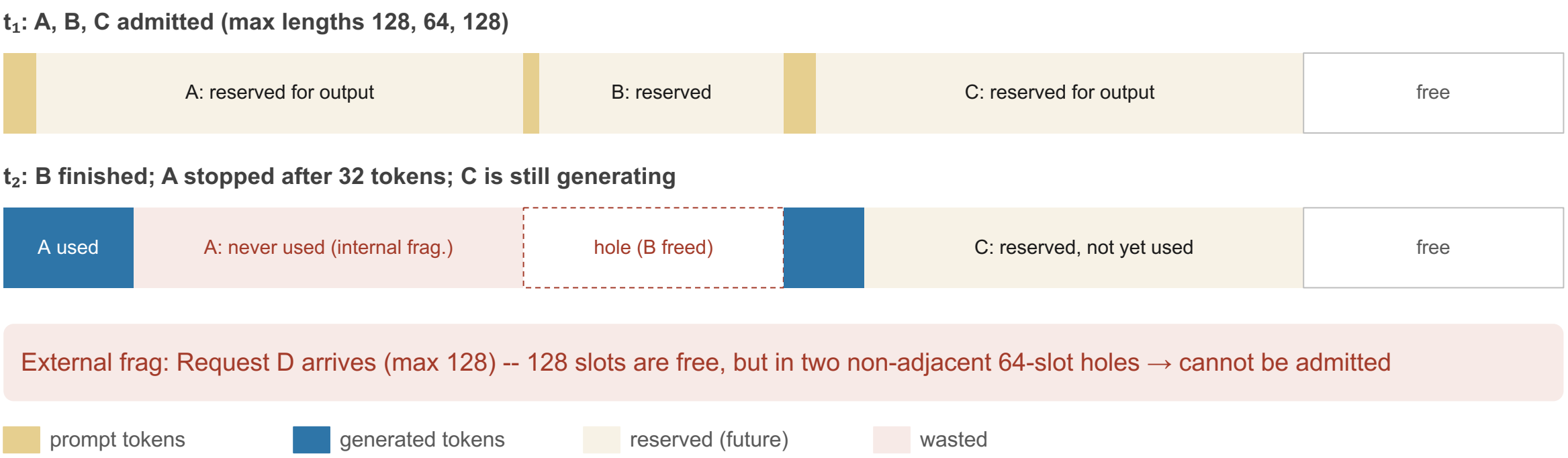
Slides from Zhihao Jia, Tianqi Chen

How do we reserve memory for KV cache?

- Example:
  - Three requests: A (maxlen=128), B (maxlen=64), C (maxlen=128)
- How do you make sure each request has enough space for KV cache?

- What are some issues with this?

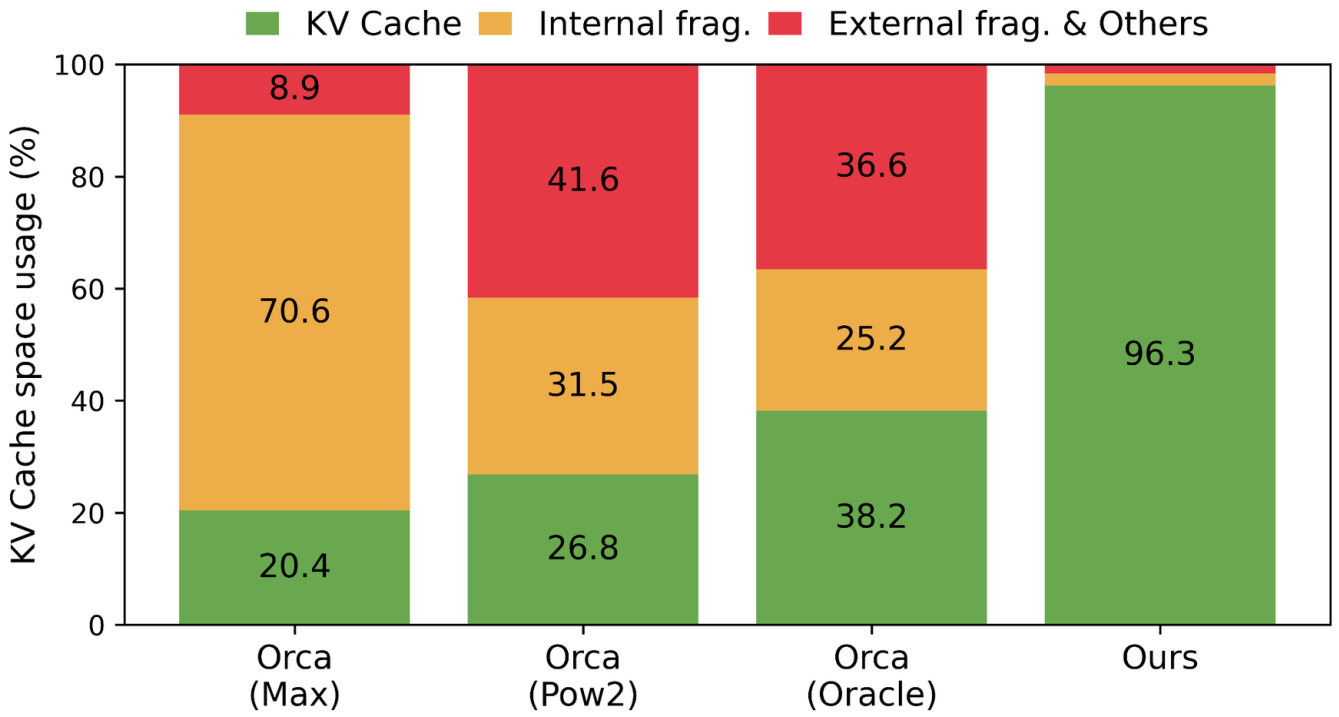
Contiguous pre-allocation reserves for the worst case



Wasted memory due to three reasons:

- **Reservation**: held for un-generated tokens, but unused now
- **Internal fragmentation**: Reserved for max length, but never used because the request stopped early
- **External frag**: Free memory split into holes too small for the next request's reservation

Orca: only 20–38% of KV memory holds token state



- **Orca (Max):** reserve the maximum length
- **Orca (Pow2):** reserve the next power of 2 above the true length
- **Orca (Oracle):** knows the true output length in advance
- **vLLM (“Ours”):** 96.3% of KV memory holds tokens

Even with an oracle, 62% is wasted: reservation plus external fragmentation.

Borrow from the OS: virtual memory for the KV cache

Operating system	PagedAttention (vLLM)
Process	Request (sequence)
Virtual page	Logical KV block
Physical frame	Physical KV block: KV for a fixed number of tokens (default 16)
Page table	Block table, one per request
Demand paging	Allocate a new block only when the last one fills
Swapping	Preemption: move a request's blocks to CPU memory, or drop and recompute
Shared pages + copy-on-write	Shared prompt blocks for parallel sampling and beam search

- KV block is a **fixed-size** contiguous chunk of memory that stores KV states from **left to right**
  - Logical: request allocates contiguous blocks
  - Physical: vLLM maps contiguous blocks to (potentially) discontiguous GPU memory
- **Benefit:**
  - Fixed-size blocks remove external fragmentation
  - Only allocate block-by-block as needed to bound internal fragmentation to <1 block

KV blocks and the block table

Request A: “Alan Turing is a computer scientist”

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist		
block 2				
block 3				

Block table

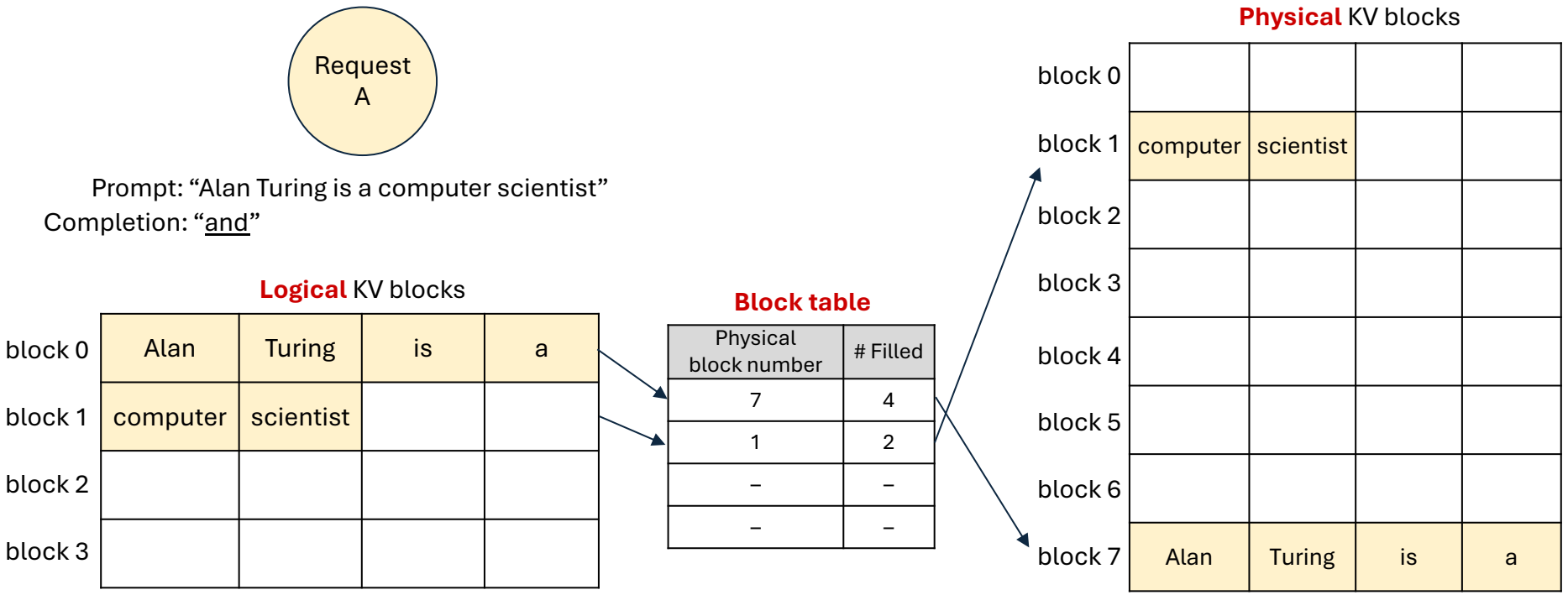
logical	physical	# filled
0	7	4
1	1	2
2	–	–
3	–	–

Physical KV blocks (GPU HBM)

block 0	req B			
block 1	computer	scientist		
block 2				
block 3	req B			
block 4	req B			
block 5				
block 6				
block 7	Alan	Turing	is	a

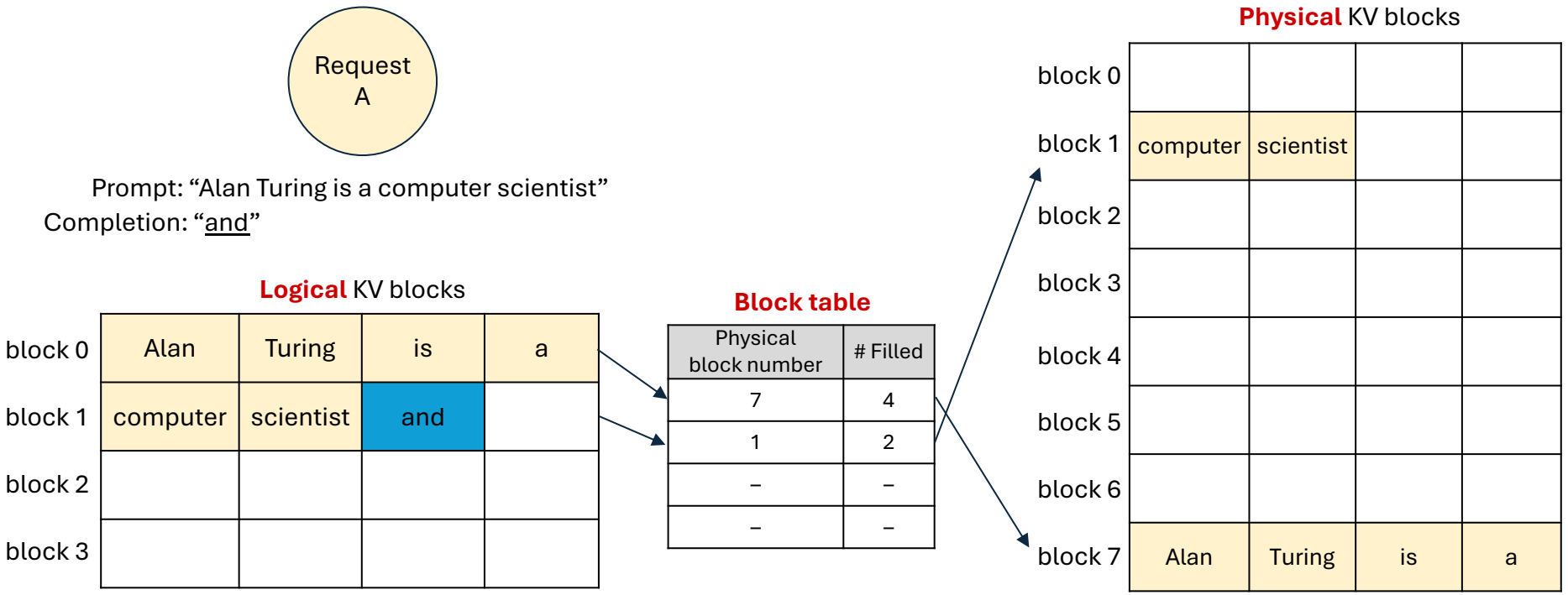
- Logical blocks are contiguous per request, physical blocks can live anywhere
- The block table maps logical → physical and records how full the last block is
- Any free physical block can serve any request → no external fragmentation

Memory Management with PagedAttention



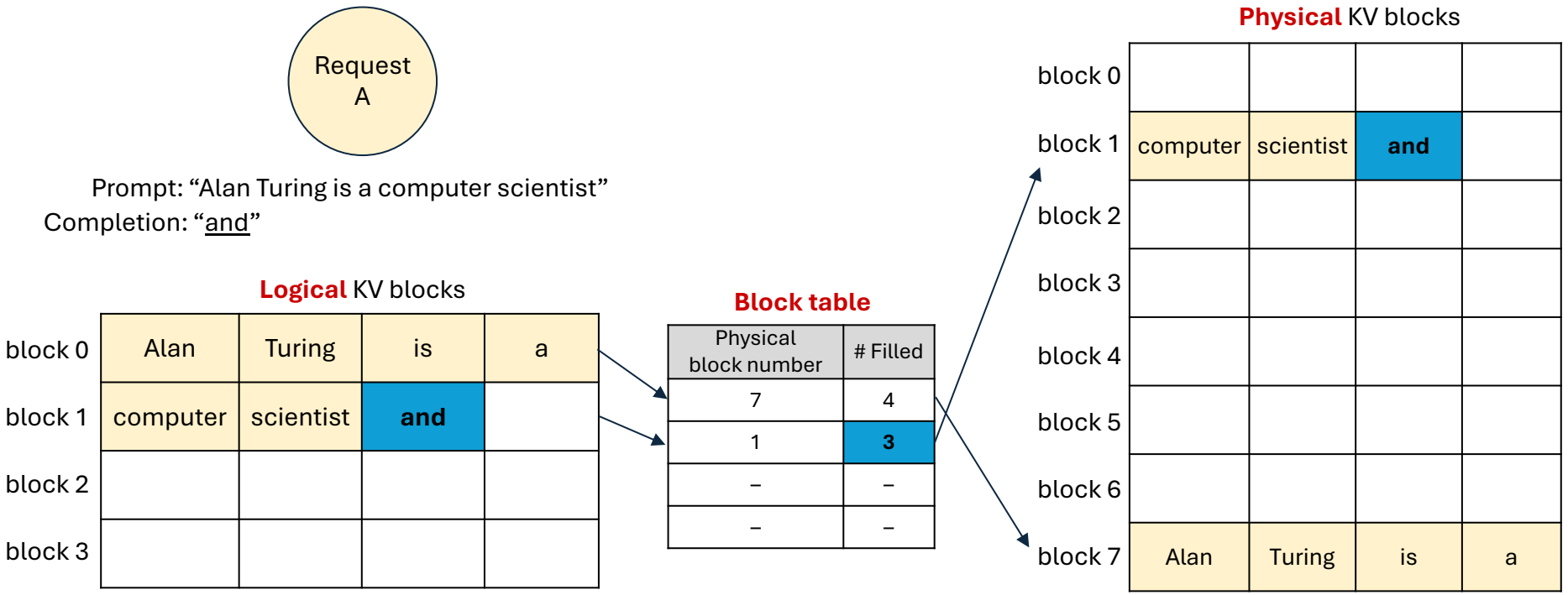
slides from vllm: Efficient Memory Management for Large Language Model Serving with PagedAttention

Memory Management with PagedAttention



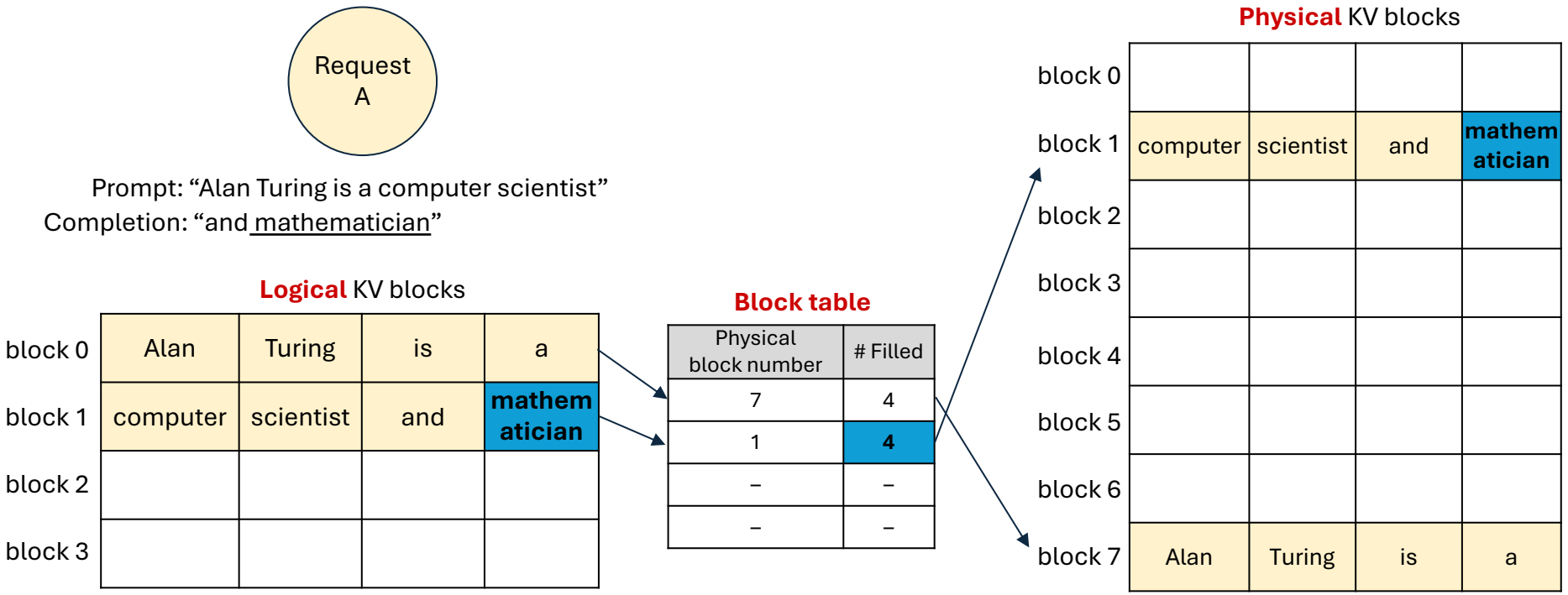
slides from vllm: Efficient Memory Management for Large Language Model Serving with PagedAttention

Memory Management with PagedAttention



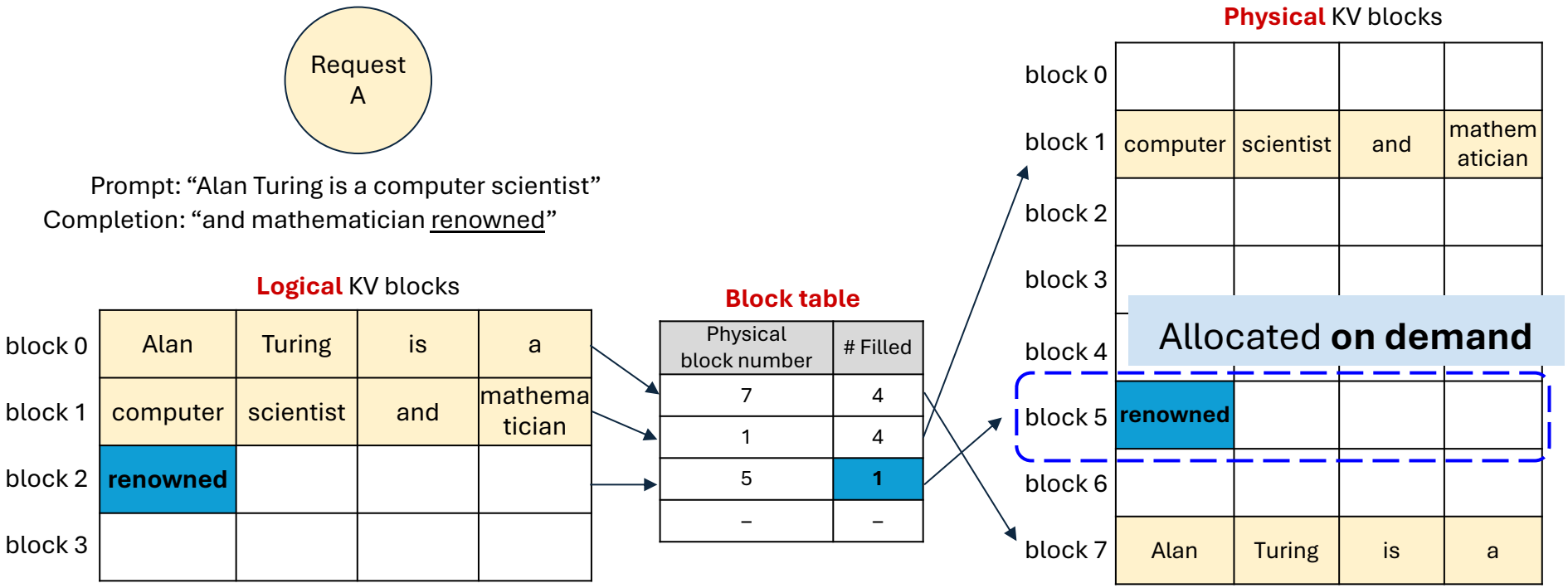
slides from vllm: Efficient Memory Management for Large Language Model Serving with PagedAttention

Memory Management with PagedAttention



slides from vllm: Efficient Memory Management for Large Language Model Serving with PagedAttention

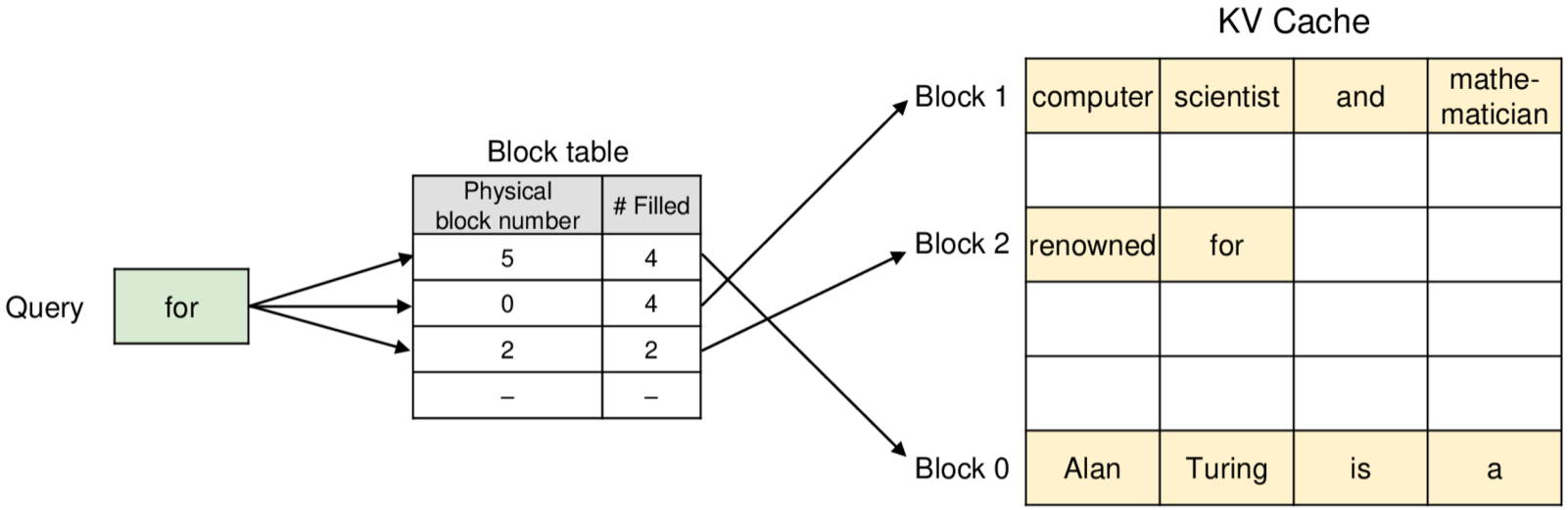
Memory Management with PagedAttention



slides from vllm: Efficient Memory Management for Large Language Model Serving with PagedAttention

Attention with Virtualized KV Cache

1. Fetch non-contiguous KV blocks using the block table
2. Apply attention on the fly



Takeaway: 20–26% higher attention-kernel latency than a contiguous kernel, but 2–4× higher end-to-end throughput because batches get much bigger

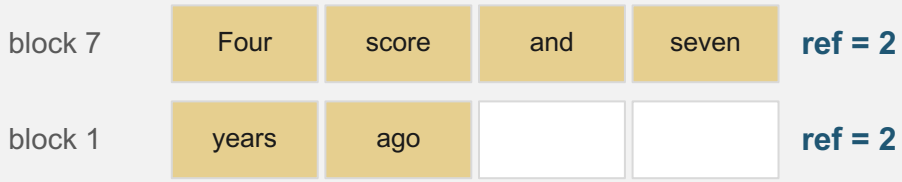
What’s the best block size?

Sharing blocks: parallel sampling and copy-on-write

Before: two samples share the prompt

`A1: [7, 1]`

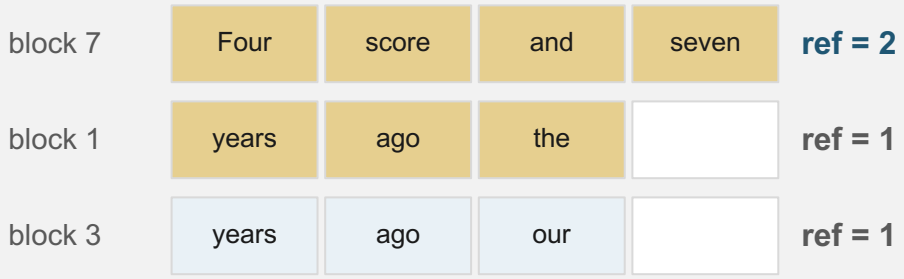
`A2: [7, 1]`



After: A1 writes “our”, A2 writes “the”

`A1: [7, 3]` ← `copy-on-write`

`A2: [7, 1]`



- Logical -> Physical mapping allows multiple logical pages to point to the same physical page (like in OSes)
- Writing into a shared, partly filled block triggers a copy of that one block
- Example: Beam search which generates multiple candidate text sequences

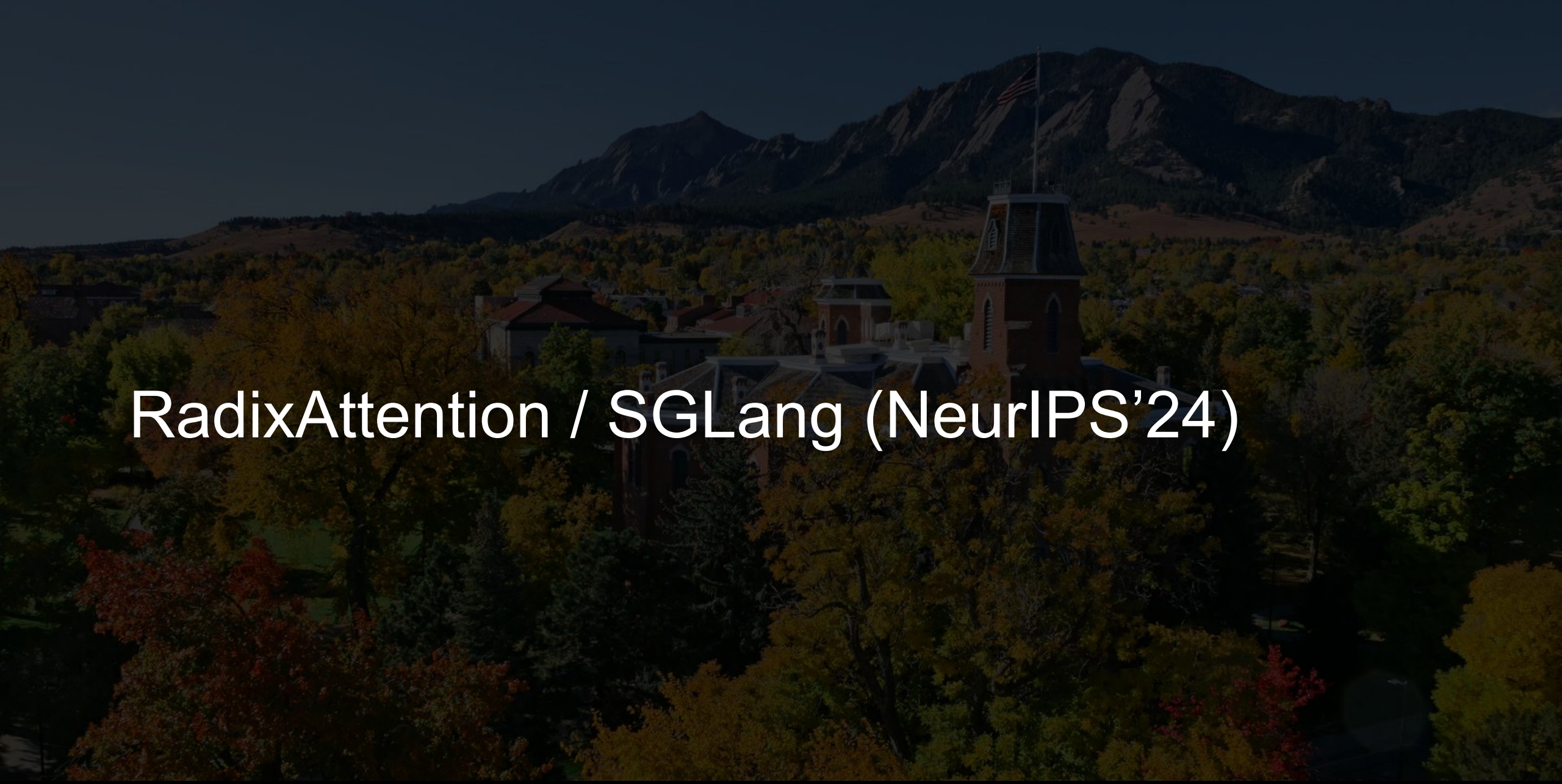
What happens when memory runs out?

- We often run much higher batch sizes than could fit in GPU memory (assuming all sequences generate max_len).
  - Assumption: most sequences won’t get close to max_len
  - … But what happens if they do and we OOM?

PagedAttention -> vLLM

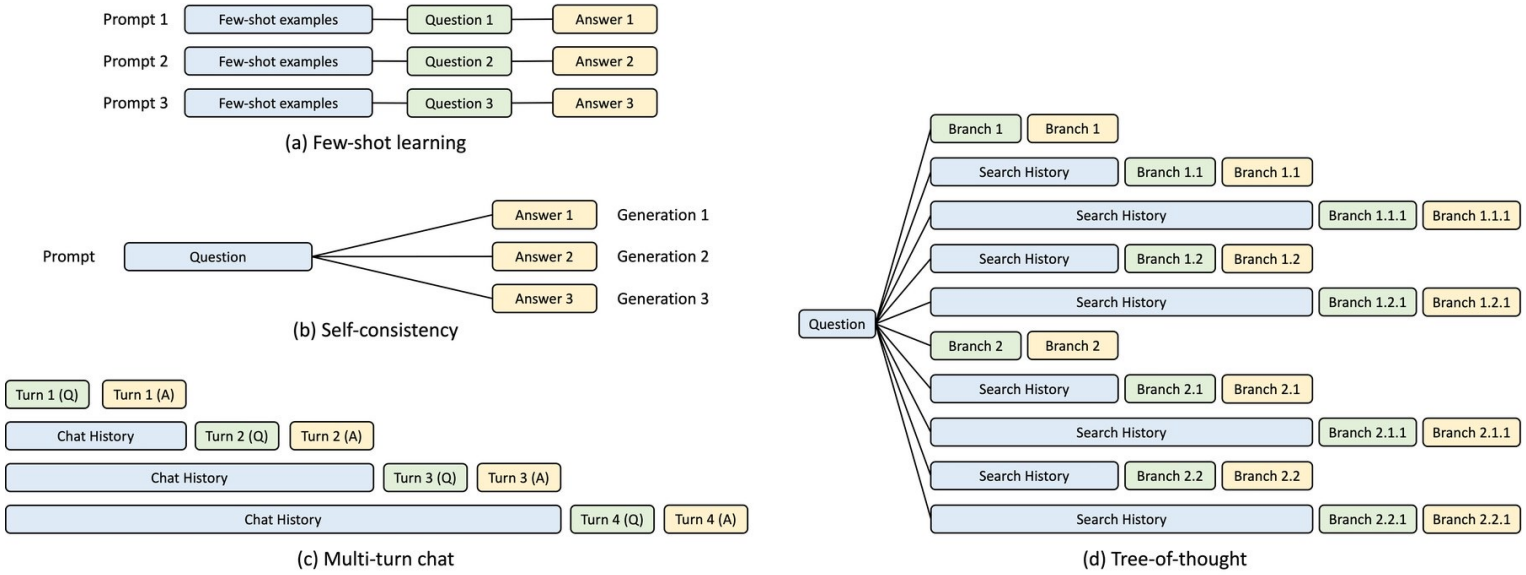






RadixAttention / SGLang (NeurIPS'24)

Requests share prefixes



- Examples:
  - System prompts
  - Multi-turn chat re-sends whole history
  - Few-shot examples consistent across queries
  - Tree-of-thought branches
  - Agents with growing context
  - …

Insight: Keep track of prompt prefix -> KV cache blocks, if there are already KV cache blocks generated for your prompt, reuse it instead of recomputing it

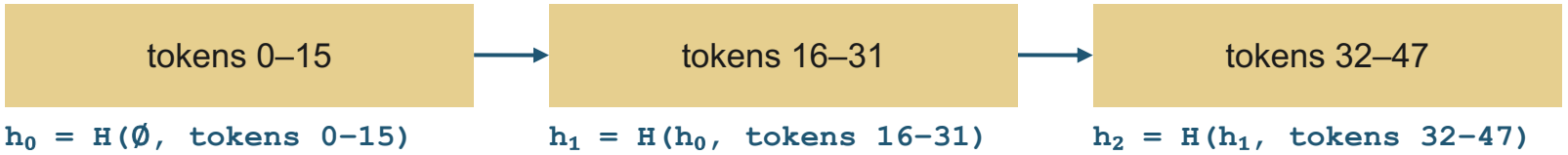
Only exact prefixes are reusable

Request 1 (cached)	system	tools	doc A	doc B	question 1
Request 2	**system**	**tools**	**doc A**	**doc B**	**question 2**
Request 3	**system**	**tools**	**doc B**	**doc A**	**question 3**
Request 4	**timestamp**	**system**	**tools**	**doc A**	**question 4**

cache hit miss: recompute

- Token i's K and V depend on every token ≤ i and on its position, valid only if the entire prefix matches exactly.
- How can we reuse more prefix cache?

vLLM automatic prefix caching: content-addressed blocks



hash table

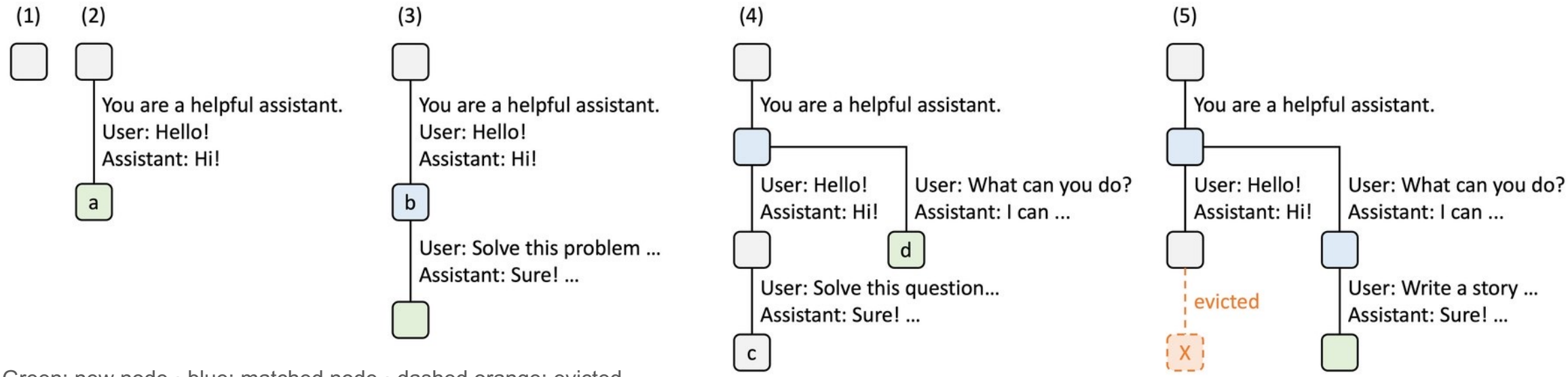
h_0 → physical block 12
h_1 → physical block 3
h_2 → physical block 40
…

New request: hash its blocks in order and look each one up. Reuse until the first miss, then prefill from there.

- A block's identity includes its parent's hash, so a hit implies the entire prefix matches

- **Prefix caching:** No need to "zero-out" physical blocks once a sequence has finished
  - Can reuse KV cache blocks if a prompt comes in that has a shared prefix
  - Keep finished requests' block on GPU with ref count 0, evict blocks when needed via LRU

RadixAttention: a radix tree over token sequences



Green: new node · blue: matched node · dashed orange: evicted

Insight: Structure memory as a prefix tree, reuse KV cache prefixes by performing a longest-prefix match

- On arrival: match the longest cached prefix, prefill only the rest
- On completion: insert the new path (new branch in tree)
- Under memory pressure: evict least-recently-used leaves whose reference count is 0
  - Keeps “root” of tree alive, which is more valuable for future prefix reuse

Cache-aware scheduling

Example: six requests, two shared prefixes (A, B)

FCFS order	**A miss**	**B miss**	**A miss**	**B miss**	**A miss**	**B miss**	**6 misses**
Longest-prefix-match	**A miss**	**A hit**	**A hit**	**B miss**	**B hit**	**B hit**	**2 misses**

Insight: Sort awaiting requests by matched-prefix length (walk radix tree, DFS order), schedule based on priority

Cons?

How much does prefix caching matter?

Model pricing

The following table shows pricing for all Claude models:

Model	Base tokens		Prompt caching		
Name	Input	Output	5m writes	1h writes	Hits and refreshes
Claude Fable 5.1 For demanding reasoning and long-horizon agentic work	$10 / MTok	$50 / MTok	$12.50 / MTok	$20 / MTok	$0.25 / MTok
Claude Opus 5.5 For long-running agentic coding and knowledge work	$4 / MTok	$20 / MTok	$5 / MTok	$8 / MTok	$0.20 / MTok
Claude Sonnet 5.5 The best combination of speed and intelligence	$2 / MTok	$10 / MTok	$2.50 / MTok	$4 / MTok	$0.10 / MTok
Claude Haiku 5.5 For high-volume, latency-sensitive tasks such as classification, extraction, and routing	$0.10 / MTok for prompts up to 100,000 tokens	$0.50 / MTok	$0.125 / MTok	$0.20 / MTok	$0.01 / MTok
	$0.50 / MTok for prompts over 100,000 tokens	$2.50 / MTok	$0.625 / MTok	$1 / MTok	$0.05 / MTok

Flagship models

Standard | Batch | Flex | Fast | Ultrafast

Our latest models

Prices per 1M tokens.

	Short context				Long context			
Model	Input	Cached input	Cache writes	Output	Input	Cached input	Cache writes	Output
gpt-6-astra	$10.00	$1.00	$12.50	$50.00	$20.00	$2.00	$25.00	$75.00
gpt-6.1-sol	$2.00	$0.10	$2.50	$10.00	$4.00	$0.20	$5.00	$15.00
gpt-6-luna	$0.10	$0.01	$0.125	$0.50	$0.20	$0.02	$0.25	$0.75

RadixAttention -> SGLang





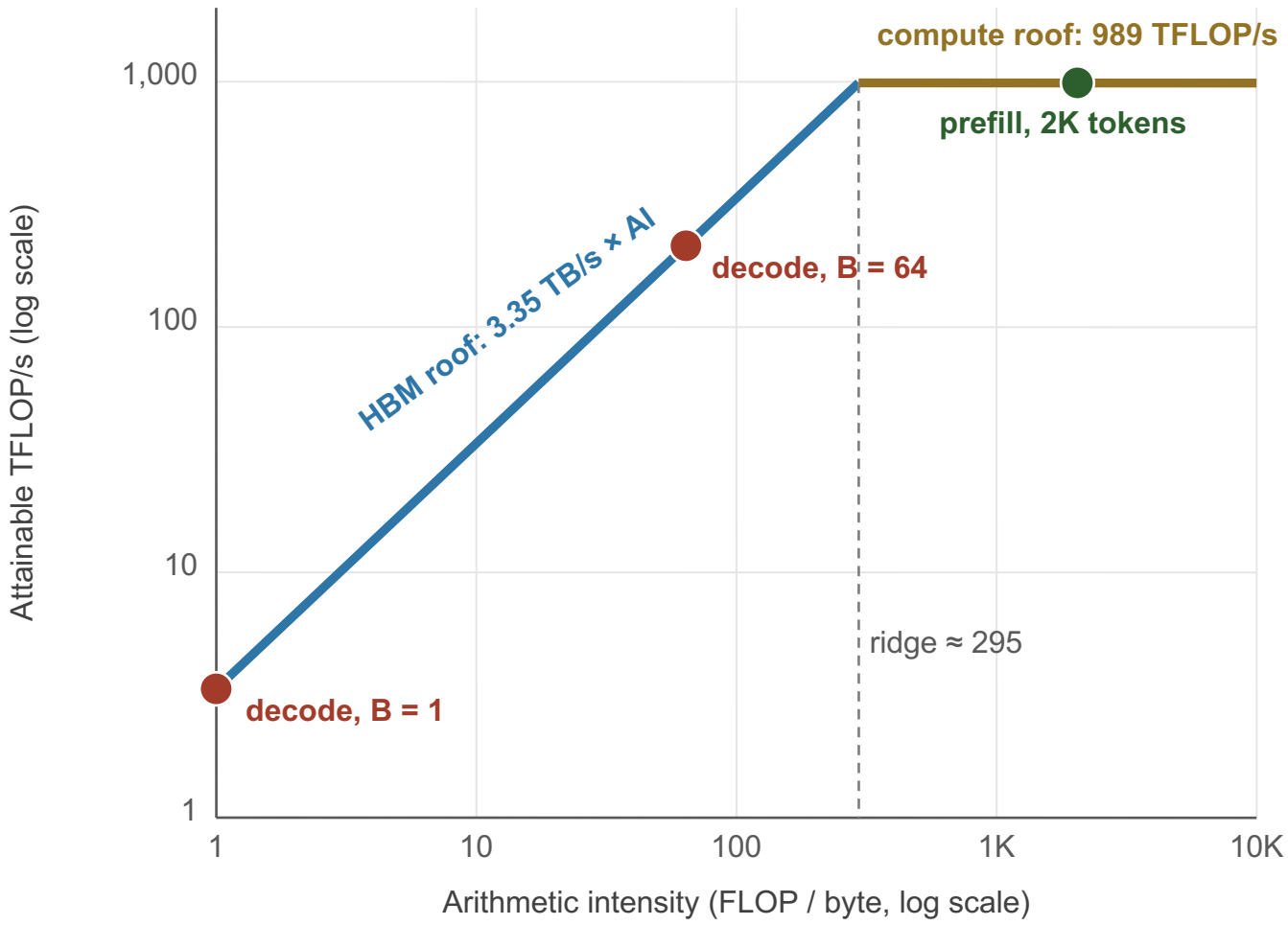


Prefill / Decode Interference

Where we are

- Challenge: High batch sizes to increase arithmetic intensity of decode
  - Solution: Continuous batching
- Challenge: Need KV cache, but comes at the (massive) cost of linearly-increasing memory w.r.t. context length
  - Solution: PagedAttention / RadixAttention / Prefix Caching
  - Challenge part 2: KV working set outgrows HBM…
    - ???
- Prefill and Decode are distinct stages of inference, exhibit completely different hardware requirements (memory vs compute)
  - ???

Recap: The roofline

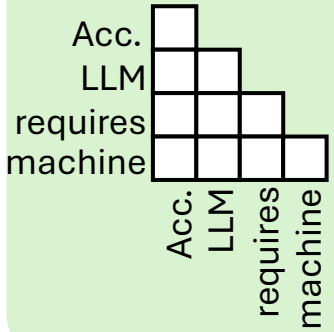


H100 Roofline

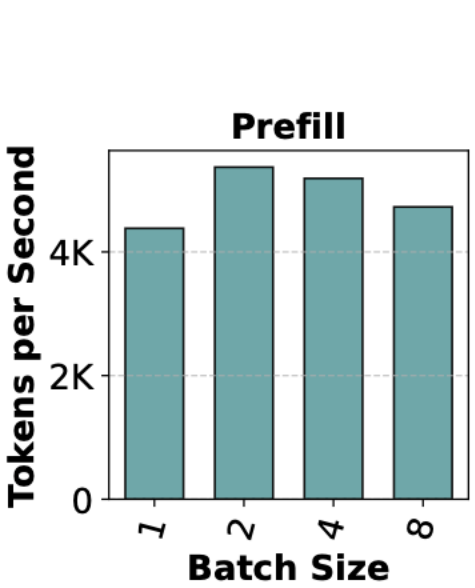
Where are the bottlenecks?

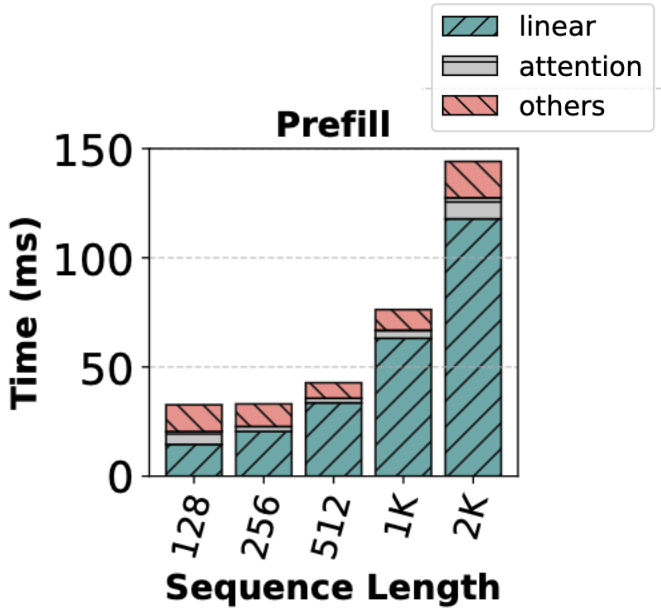
- Prefill phase:
  - For each request, compute the KV$ / Attention Matrix for *all* input tokens

Attention Matrix



Prompt: [Accelerating LLM requires machine … {1000s of more tokens}]





KV Cache

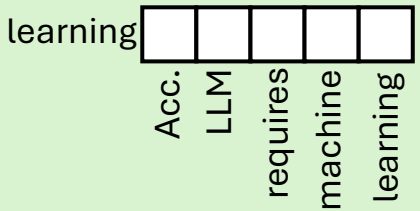


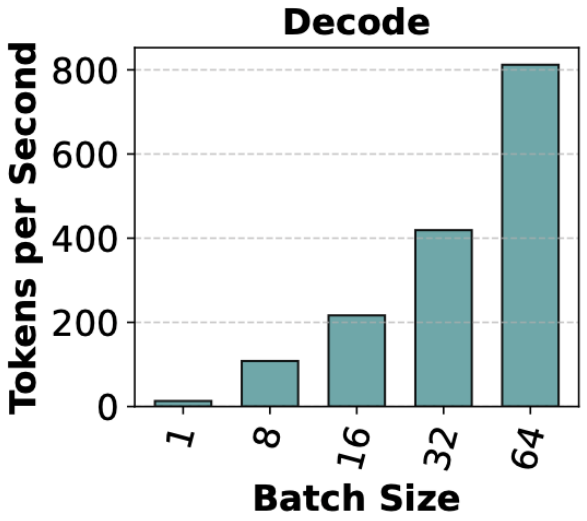
Where are the bottlenecks?

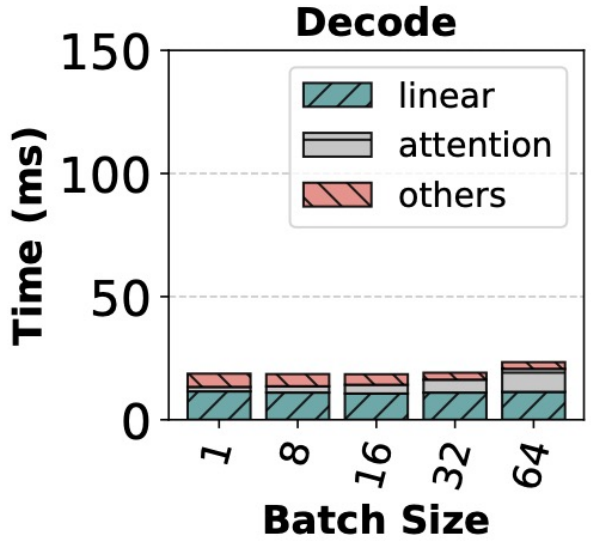
- Decode:
  - Produce a *single* output token

Prompt: [Accelerating LLM requires machine … {1000s of more tokens}]
Output: Learning

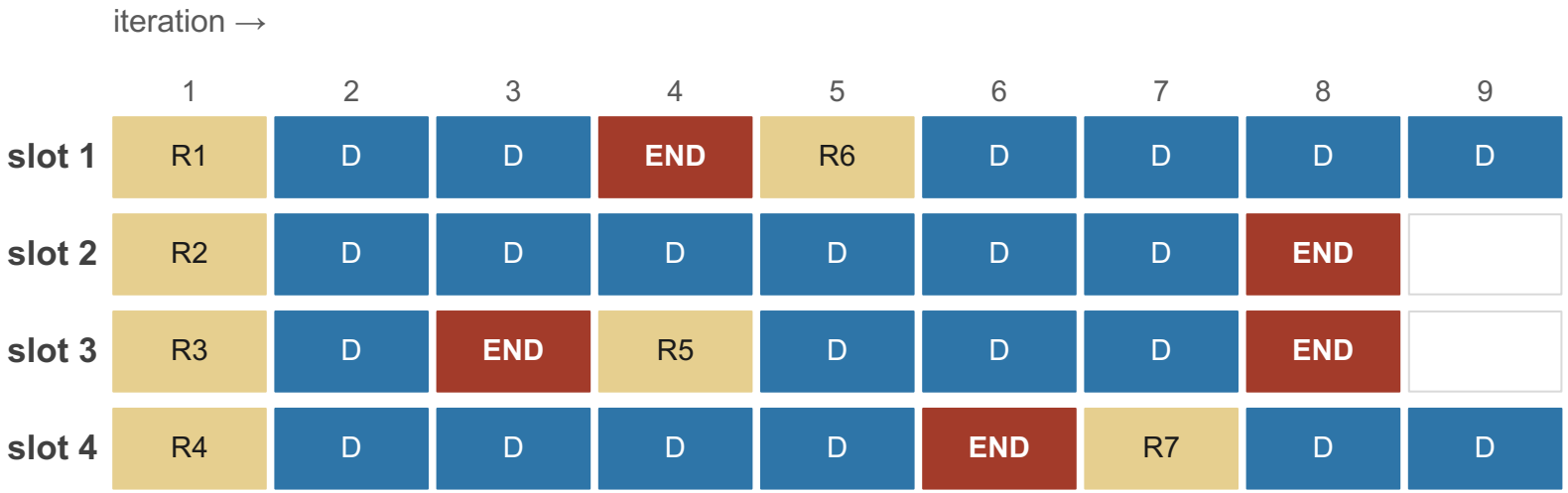
Attention Matrix







Recap: Continuous Batching



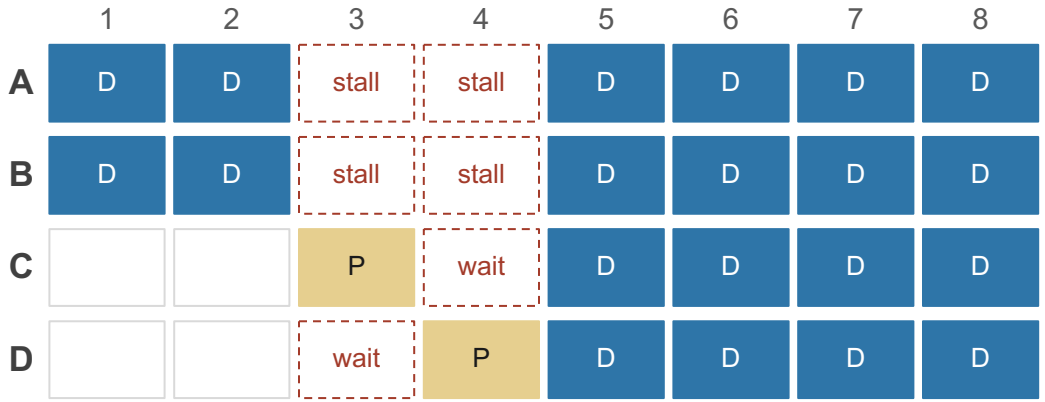
- At every iteration: retire finished requests, admit waiting ones, run one step
- Short requests leave as soon as they finish
- Potential issues with this approach?

Challenge: O(100ms) prefill runs in the same iteration as O(1ms) decode step -> every decode sees massive TPOT spikes!

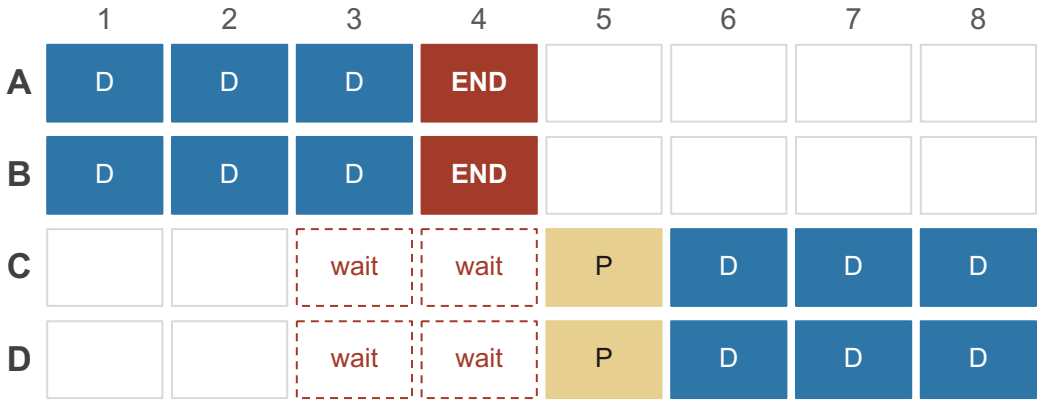
How can we schedule prefill and decode stages to avoid this interference, while maximizing high batches during decode?

Example: A and B decoding, C and D arrive

Option 1 · Prefill-prioritizing (original vLLM)



Option 2 · Decode-prioritizing (FasterTransformer)



Prefill first

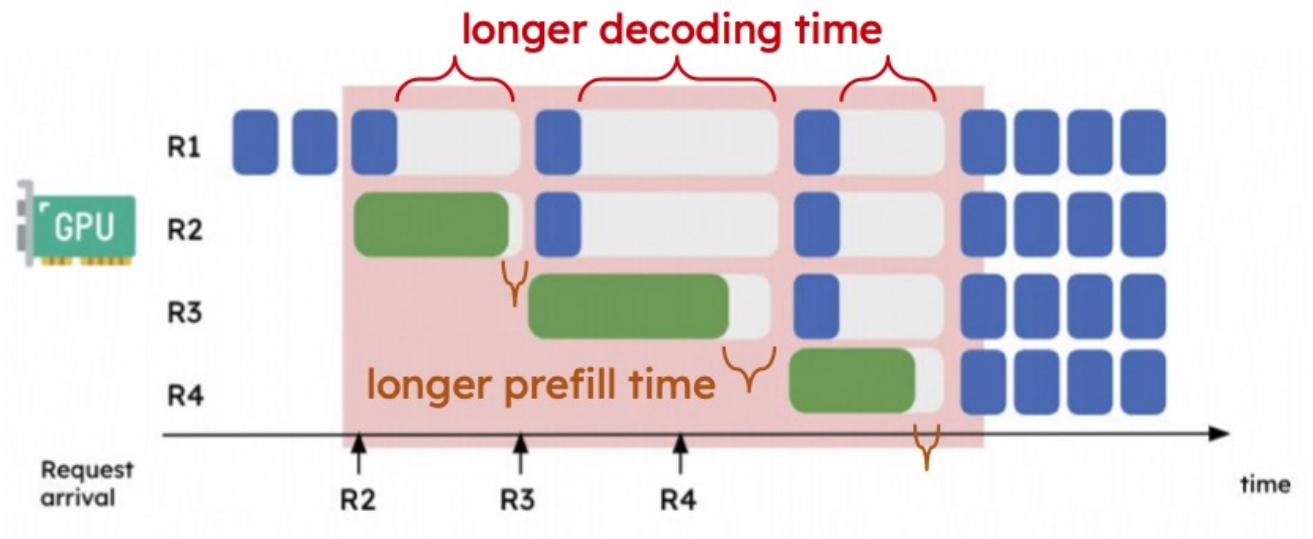
- + Bigger decode batches → higher throughput
- − A and B stall for two whole prefills: TBT spikes

Decode first

- + Smooth TBT for A and B
- − C and D wait (TTFT ↑); smaller batches → lower throughput

Existing policies trade latency for throughput. Can we get both?

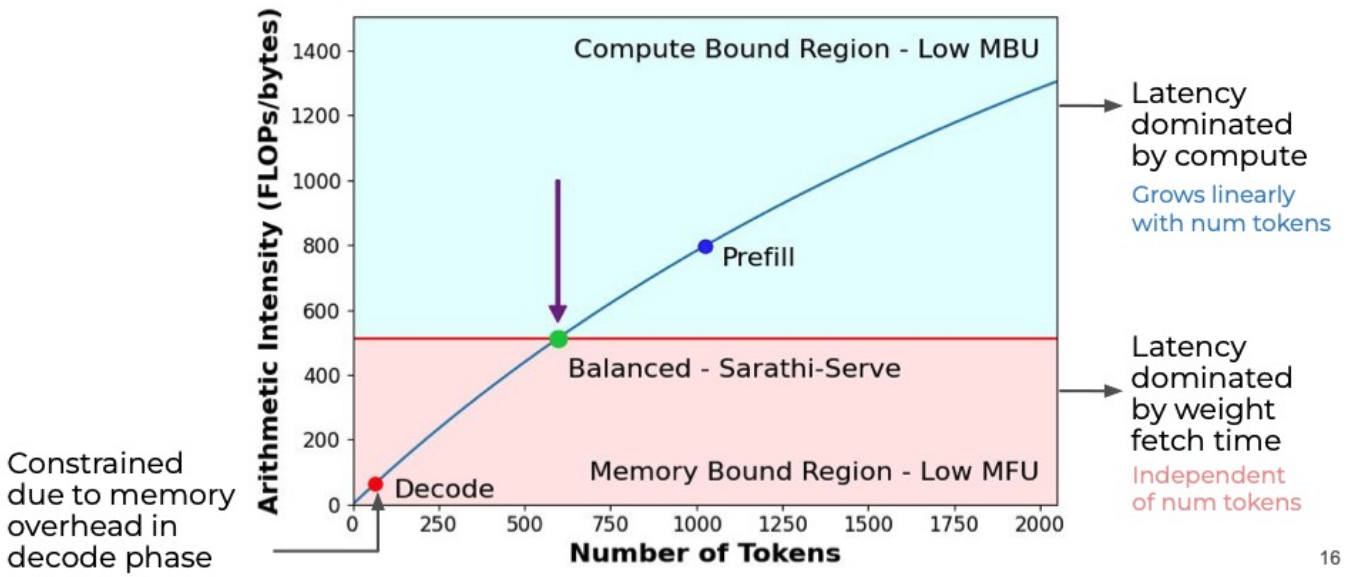
Hybrid batches



- Go back to Orca-style mixed iteration – mix prefill and decode tokens in one step
- A and B keep making progress, but TBT lasts as long as C's prefill
- C's prefill is slowed by the co-scheduled decodes too

Better than prefill-first, worse than decode-first for TBT. Can we do better?

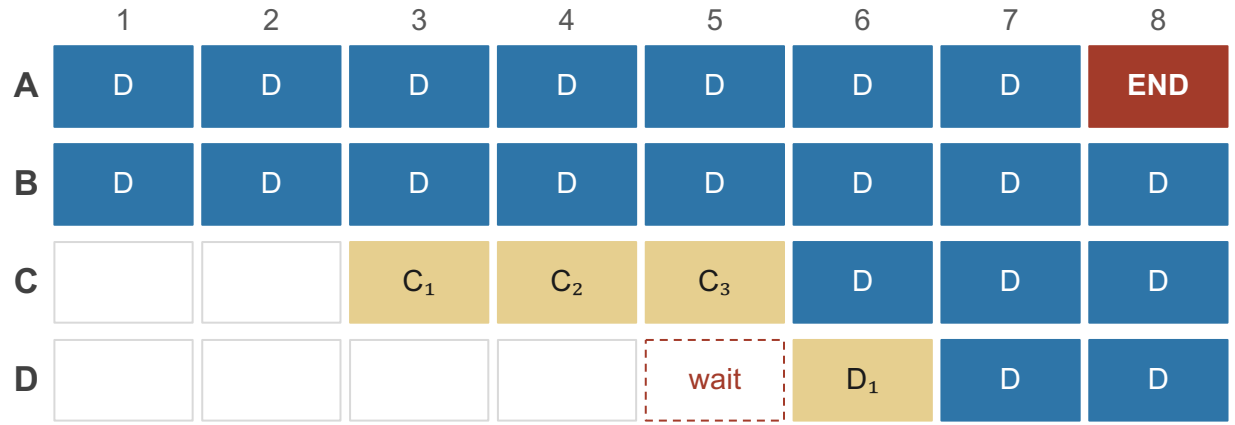
Go back to the roofline



- A decode batch sits deep in the memory-bound region: the GPU's FLOPs are mostly idle
- Adding prefill tokens to that step is almost free, until the step crosses into compute-bound
- Mistral-7B on an A100: one prefill saturates compute at ≈ 512 tokens

Key Insight: Run hybrid batches, but slice up one prefill across multiple iterations – choose just enough prefill tokens to use the compute slack left over by the decode tokens

Chunked prefill + stall-free batching



Keep every column at ≤ τ tokens. Prioritize decodes first, then prefill chunks

Each iteration, with token budget τ:

1. Add every running decode (1 token each)
2. Continue in-progress prefills, one chunk each
3. Admit new prefills while budget remains

Running decodes are never paused → no generation stalls

- Every iteration has about the same cost (bounded by τ) so TBT is predictable
- Prefill still makes progress every step, but A and B aren't stalled
- Uniform iterations also shrink pipeline-parallel bubbles (distributed inference)

Choosing the token budget τ

small τ ⟷ **large τ**

Small τ: low TBT, but expensive prefill

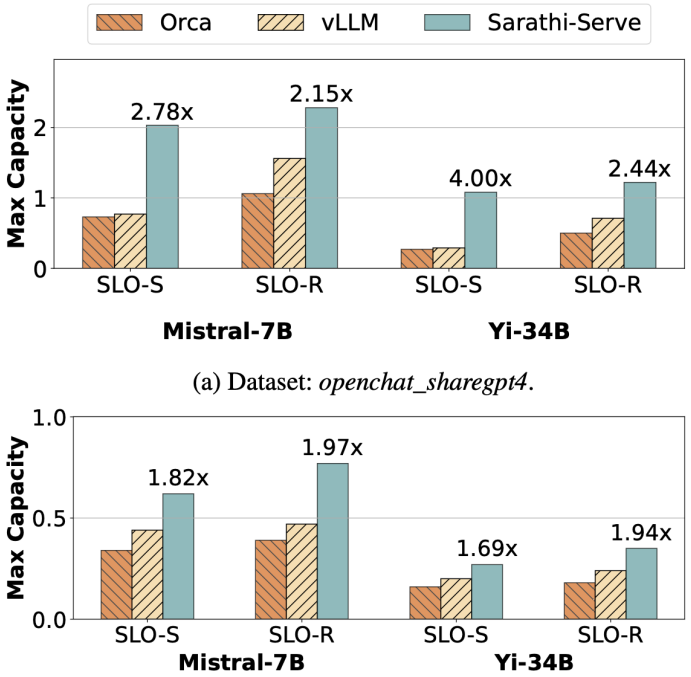
- Shorter iterations → low, stable TPOT
- More iterations per prompt → higher TTFT
- Each chunk re-reads the KV of all earlier chunks: KV reads grow ~ quadratically in the number of chunks
- Fixed per-iteration overheads repeat

Large τ: efficient prefill, but spiky TBT

- Fewer, more compute-efficient prefill steps
- Longer iterations → TPOT spikes for every running decode
- Too high just results in the Orca hybrid batch again

Rule of thumb: Profile step time vs. tokens per step on your model and GPU, then pick the largest τ whose step time fits the TBT SLO.

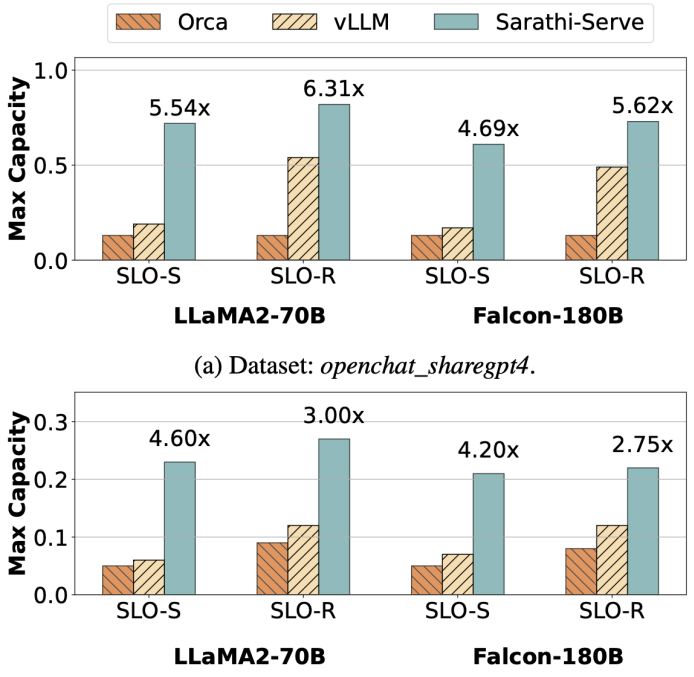
Chunked pre



(a) Dataset: *openchat_sharegpt4*.

(b) Dataset: *arxiv_summarization*.

Figure 10: Capacity (in queries per second) of Mistral-7B and Yi-34B with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.



(a) Dataset: *openchat_sharegpt4*.

(b) Dataset: *arxiv_summarization*.

Figure 11: Capacity of LLaMA2-70B and Falcon-180B (models with pipeline parallelism) with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.

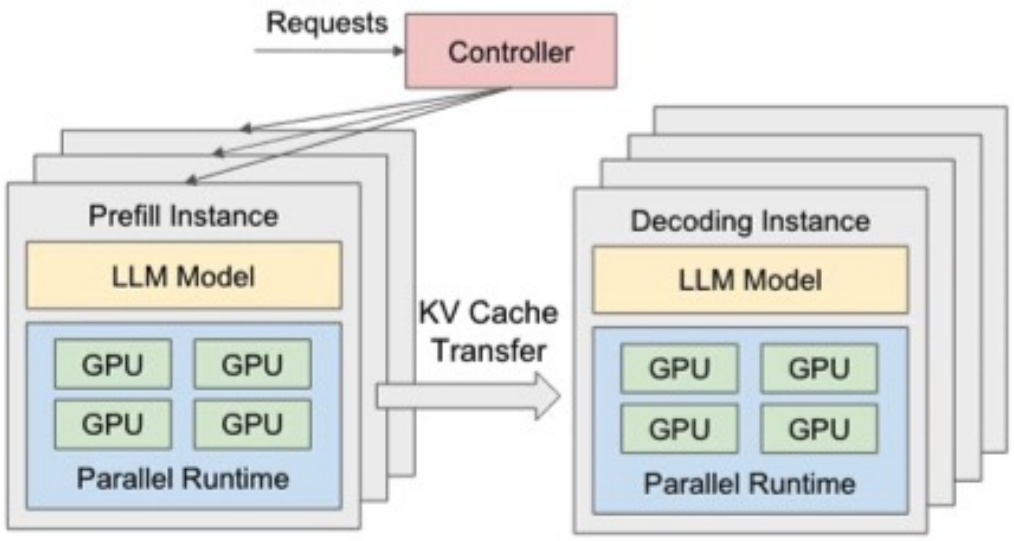
Widely-used in production: on by default in vLLM V1 and supported in SGLang and TensorRT-LLM



Prefill / Decode Disaggregation

What are some downsides with chunked prefill?

Disaggregation: separate GPU pools for prefill and decode



Key Idea: Eliminate P/D interference by *disaggregating the two stages*

- **No interference:** a decode step never shares an iteration with a prefill
- **Customize each pool:** parallelism, batching policy, hardware, …
- **Scale the pools independently:** x prefill instances per y decode instances (xPyD)
- Cons?

DistServe (OSDI '24); Splitwise (ISCA '24)

Goodput example

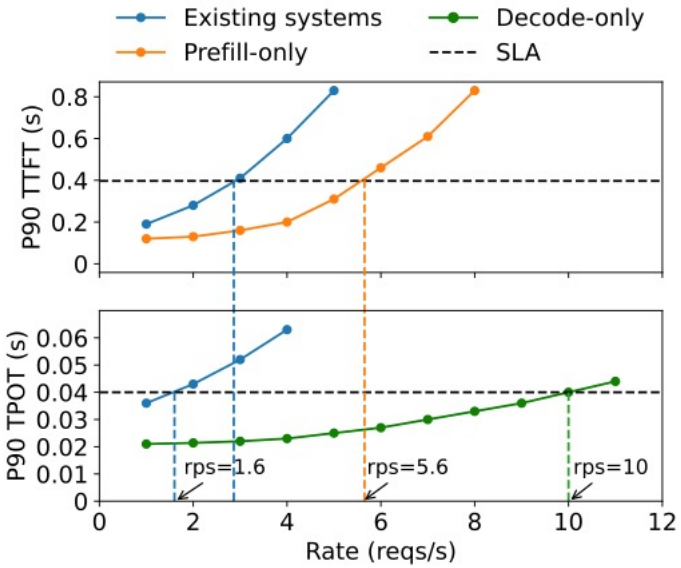


Figure 1: Performance when serving an LLM with 13B parameters under a synthetic workload with input length = 512 and output length = 64 on one NVIDIA 80GB A100. *Upper*: The P90 time-to-first-token (TTFT) latency comparing existing systems vs. a system serving only the prefill phase. *Down*: The P90 time-per-output-token (TPOT) latency comparing existing systems vs. a system serving only the decoding phase.

Goodput = the highest request rate per GPU at which the SLO attainment target (e.g., 90% of requests within both TTFT and TPOT) still holds.

Existing Systems (co-locate prefill and decode):

- 1.6 requests per second on an A100 under TPOT SLO

Disaggregation

- 5.6 requests per second for prefill under TTFT SLO
- 10 requests per second for decode under TPOT SLO

2 prefill GPUs + 1 decode GPU → 10 rps on 3 GPUs = 3.3 rps per GPU = 2.1× colocation

Disaggregation benefits

- **Phase-specific parallelism**
  - Prefill favors tensor parallelism to cut TTFT
  - Decode favors bigger batches and pipeline parallelism / replication for throughput
  - Ex: DistServe chooses TP4 prefill, TP2xPP2 decode for OPT-66B
- **Independent scaling**
  - Customize P:D ratio for the workload
  - Long prompts (RAG, summarization) → more prefill
  - Long outputs (reasoning, code) → more decode
- **Heterogeneous hardware**
  - Splitwise: compute-heavy GPUs for prefill, cheaper or power-capped GPUs for decode
  - Optimizes throughput per dollar and per watt

Costs of moving KV cache

Llama-3-70B, 4,096-token prompt: 4,096 × 320 KiB = 1.34 GB of KV cache

Link	Bandwidth (one direction)	Transfer time
NVLink, H100 (same node)	450 GB/s	3.0 ms
8 × InfiniBand NDR (one 400 Gb/s NIC per GPU)	400 GB/s	3.4 ms
1 × InfiniBand NDR	50 GB/s	27 ms
25 Gb Ethernet	3.1 GB/s	**430 ms**

For comparison, prefill is ~ 0.3 s on 4 × H100

- For RDMA fabrics, typically transfer << prefill
  - Can also be streamed layer by layer while prefill continues (pipeline transfer with compute)
- GQA/MLA further reduces KV transfer costs
- Longer contexts favor disaggregation, **why**?

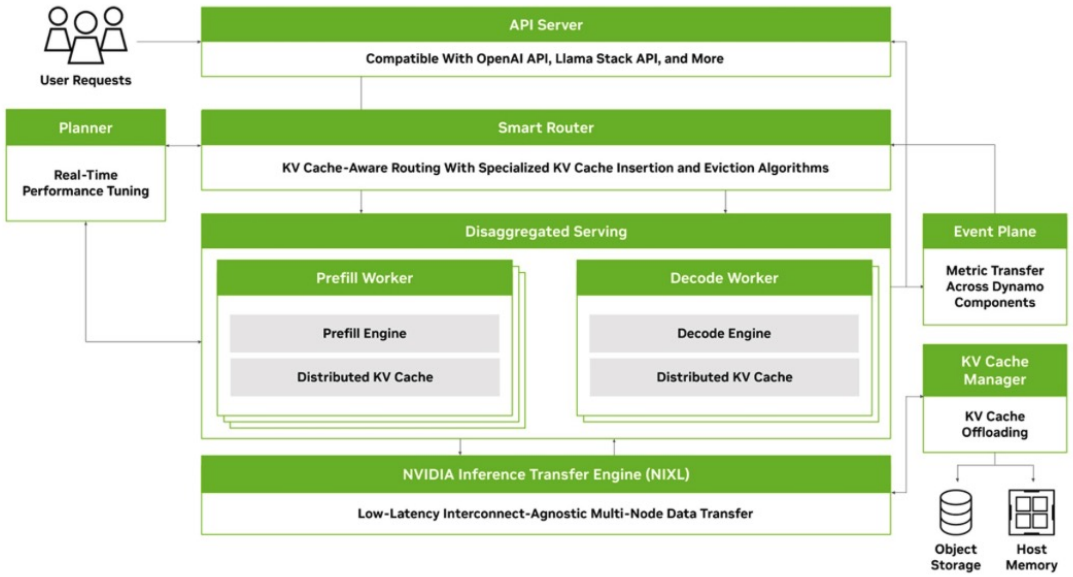
Chunked prefill or disaggregation?

	Colocated + chunked prefill	Disaggregated prefill / decode
Interference	Bounded by the token budget	None
Utilization	High if you can fill every step to the budget	Risk of an idle pool if the P:D ratio is off
Parallelism	One plan for both phases	Customize for each phase
Network	None needed	RDMA-class fabric for KV transfer
Minimum scale	One GPU	At least two instances, better at DC-scale
Best for	Throughput-oriented jobs, small clusters, short prompts	Strict TTFT + TPOT SLOs, long prompts, large fleets

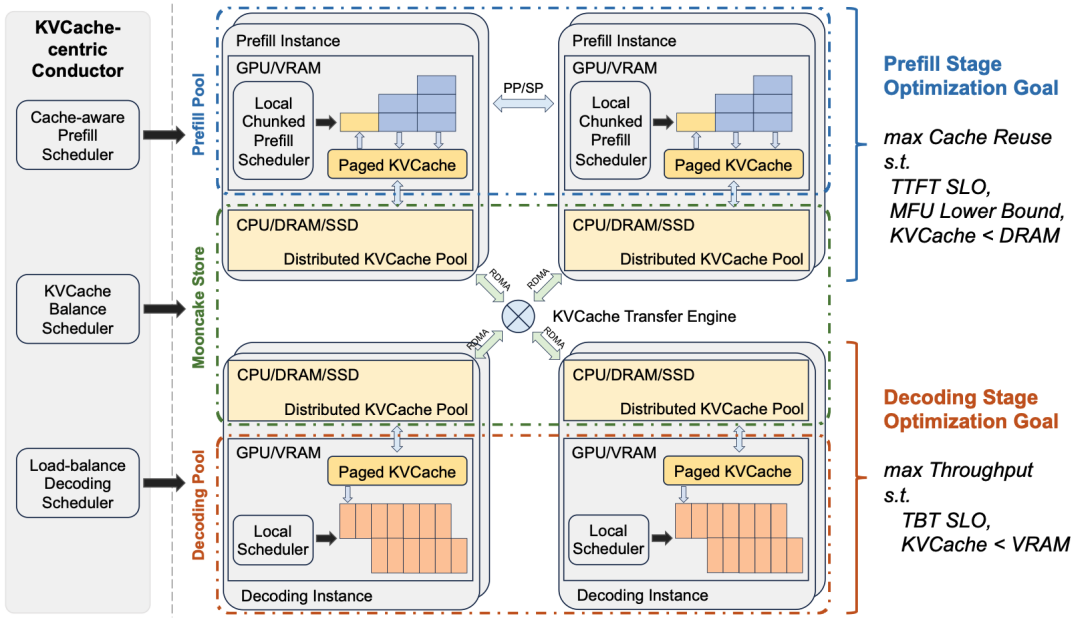
Not either/or: many production systems can choose to do one or the other dynamically, for example disaggregate only when a prompt is long enough and the remote prefill queue is short enough, otherwise prefill on decode instances

https://docs.nvidia.com/dynamo/dev/advanced-customizations/conditional-disaggregation

Disaggregation in production



NVIDIA Dynamo



Mooncake (Moonshot AI / Kimi)

- **DeepSeek-V3:** prefill deployed in 32-GPU units, decode in 320-GPU units, with different expert-parallel degrees per phase
- **Frameworks:** Dynamo and llm-d orchestrate vLLM / SGLang / TensorRT-LLM engines, transfer KV via NIXL or Mooncake's transfer engine
- **Going further:** attention–FFN disaggregation for MoE models (e.g., MegaScale-Infer)



Hierarchical and distributed KV caching

Prefix caching is useful, but how much KV cache can we fit?

Tier (8-GPU H100 node)	Capacity	Bandwidth to one GPU	Llama-3-70B tokens
GPU HBM	80 GB per GPU	3.35 TB/s	≈ 448K per 4-GPU replica
Host DRAM	≈ 2 TB per node	≈ 64 GB/s (PCIe Gen5 x16)	≈ 6M
Local NVMe SSD	≈ 30 TB per node	≈ 7–14 GB/s per drive	≈ 94M
Remote pool (RDMA)	Cluster-wide	≈ 50 GB/s per 400 Gb/s NIC	Effectively unbounded

- Prefix-cache hit rate is capped by how long KV survives before eviction
  - Useful for workloads with large contexts: multi-turn chat, agents re-reading growing trajectories, long documents and codebases
- Similar memory hierarchy as training: ~10–100× more capacity for ~10–50× less bandwidth
- **Question:** Is moving KV across the hierarchy cheaper than recomputing it?

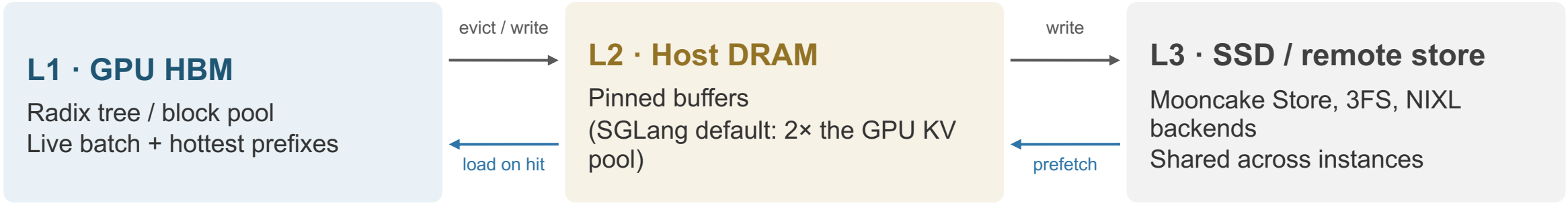
Load or recompute?

Back of the envelope calculation:

- Load latency: roughly KV_bytes_per_token / load_bandwidth
- Compute latency: roughly 2 * active_params / effective_FLOPs (e.g., 50% MFU)

Model	KV per token	Prefill FLOPs per token	FLOPs per KV byte	Break-even BW for H100 GPU
Llama-3.1-8B	128 KiB	16 GFLOP	≈ 123K	**≈ 4.0 GB/s**
Llama-3-70B	320 KiB	141 GFLOP	≈ 431K	**≈ 1.15 GB/s**
DeepSeek-V3 (37B active)	69 KiB	74 GFLOP	≈ 1.05M	**≈ 0.47 GB/s**

Anatomy of a hierarchical KV cache

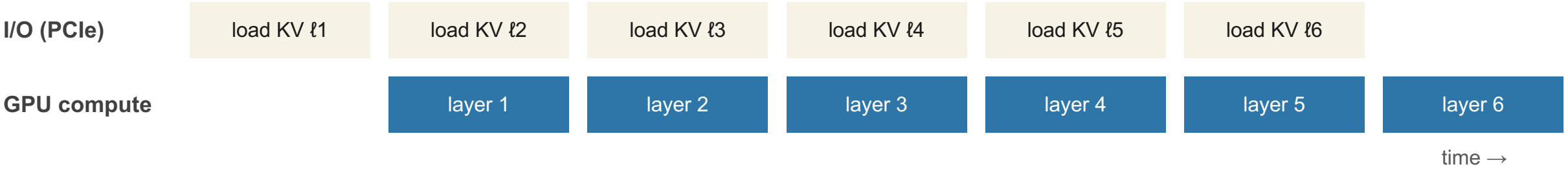


Cache placement and eviction drastically impacts benefit/performance

- **Eviction policy**: What block to evict when your tier is full? Want to keep "hottest" blocks in / close to L1
- **Write policy:** When do you write to lower tiers? Write-through vs. write on eviction
- **Prefetch policy**: When do you fetch from L3 to L2/L1? Best effort? Wait for completion? Wait with timeout? Predict when it's needed?

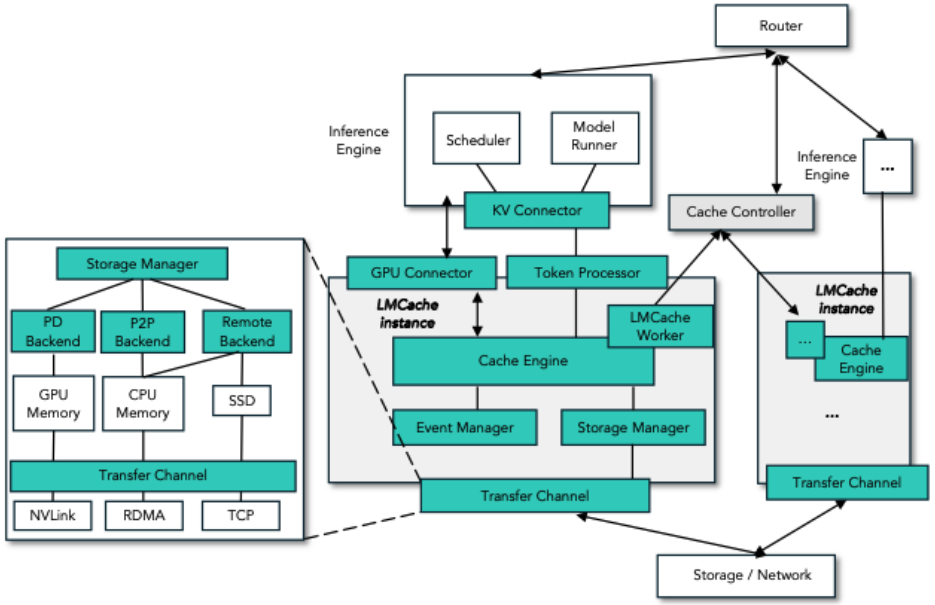
Xie, Xu, Zhao, An, Mailthody, Mahlke, Garland & Kozyrakis, "Strata: Hierarchical Context Caching for Long Context LLM Serving," OSDI '26.
SGLang HiCache https://www.lmsys.org/blog/2025-09-10-sglang-hicache/

Layer-wise offloading



- Layer ℓ's cached K, V are loaded while layer ℓ − 1 computes on the new suffix tokens
- Hidden if per-layer load time ≤ per-layer compute time
- The same pipelining argument applies for writing KV cache (asynchronous write-back) and PD transfer (send layer ℓ while computing ℓ + 1)
- If compute is short / little compute to hide behind → the load is exposed in TTFT

Beyond one node: distributed KV stores

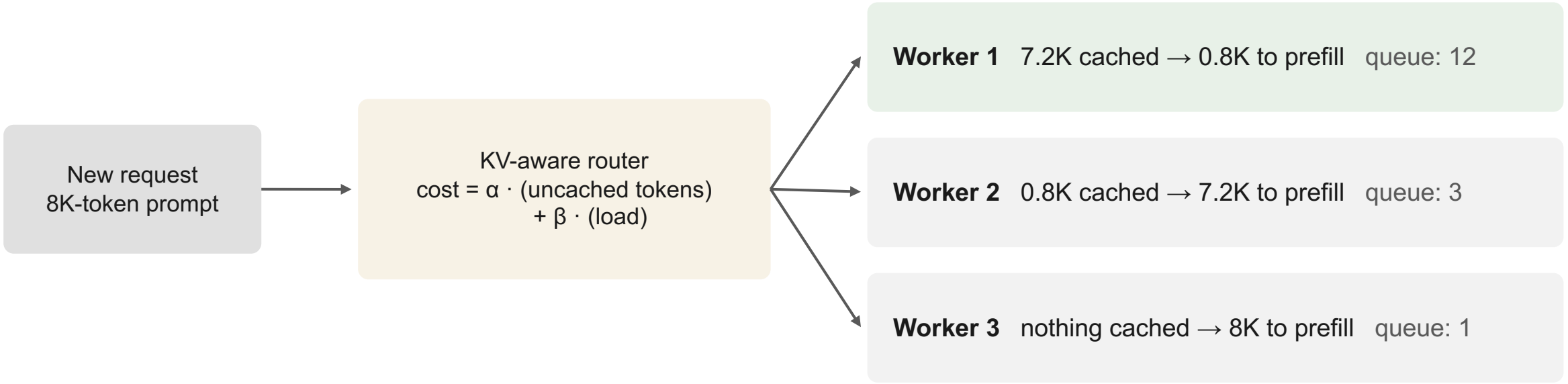


LMCache architecture

- **Mooncake (FAST ’25):** pools the cluster’s CPU DRAM and SSD into one KV store + RDMA transfer engine + scheduler places requests by cache locality and load
- **LMCache:** KV layer under vLLM / SGLang — offload to CPU, disk or remote, share across instances
- **CachedAttention (ATC ’24):** multi-turn KV kept in a DRAM/SSD hierarchy across turns
- **Dynamo KV block manager:** tiered offload GPU → CPU → SSD → remote over NIXL

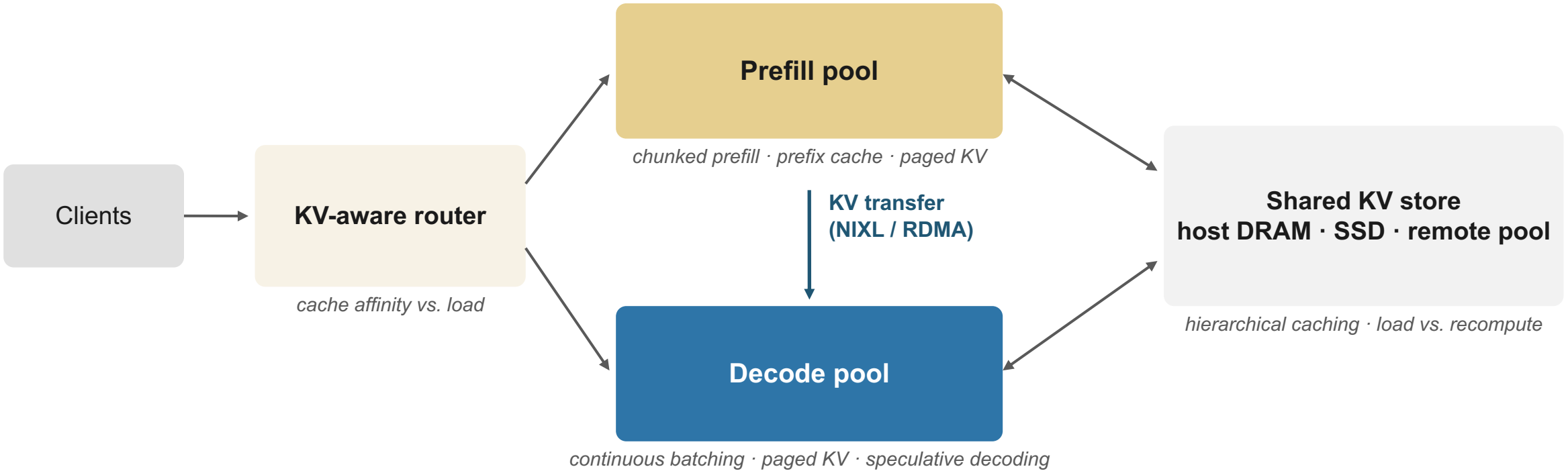
https://lmcache.ai/en/

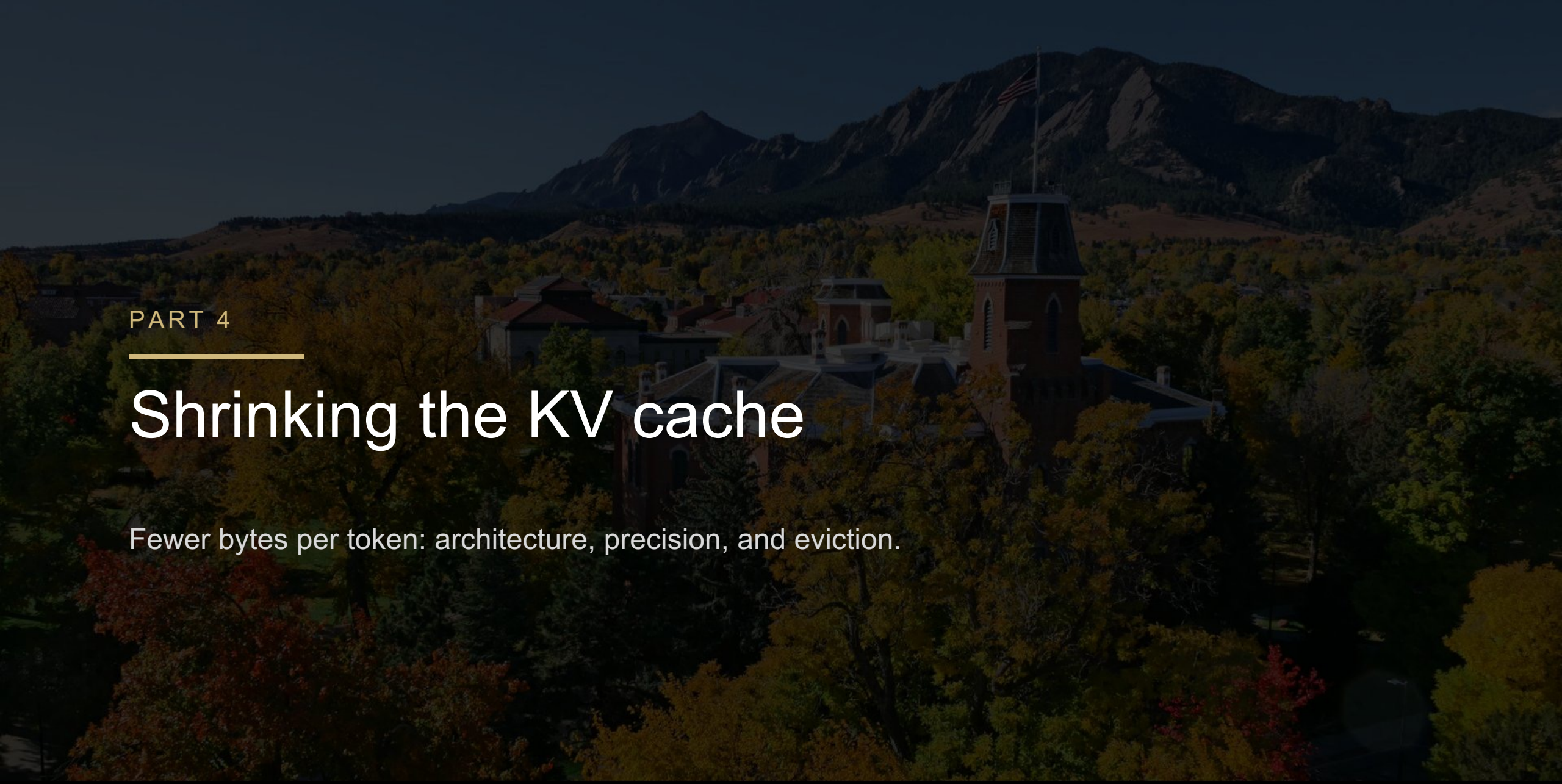
Routing with cache affinity



- **We need to design request routing alongside KV cache placement**
- Prefix caches are per-replica state, so routing to the right replica will determine hit rate
- But there's a trade-off…
  - If you maximize for cache hit → the worker who holds popular prefixes will get overloaded ("hotspot")
  - If you maximize for load balancing → every replica caches everything poorly
- Production routers combine both using global view of cached blocks + cost model to balance between hits/load balancing

Putting it together: a modern serving stack



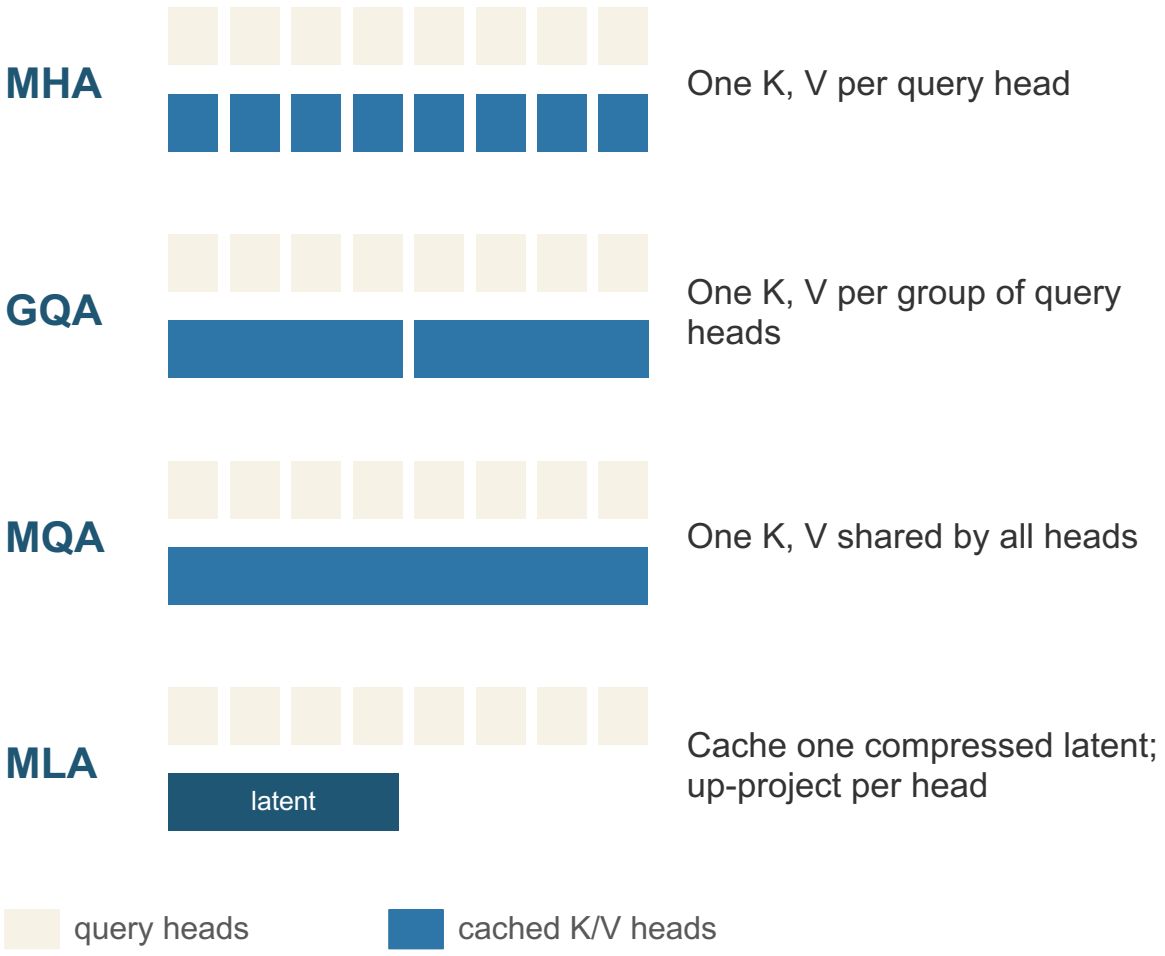


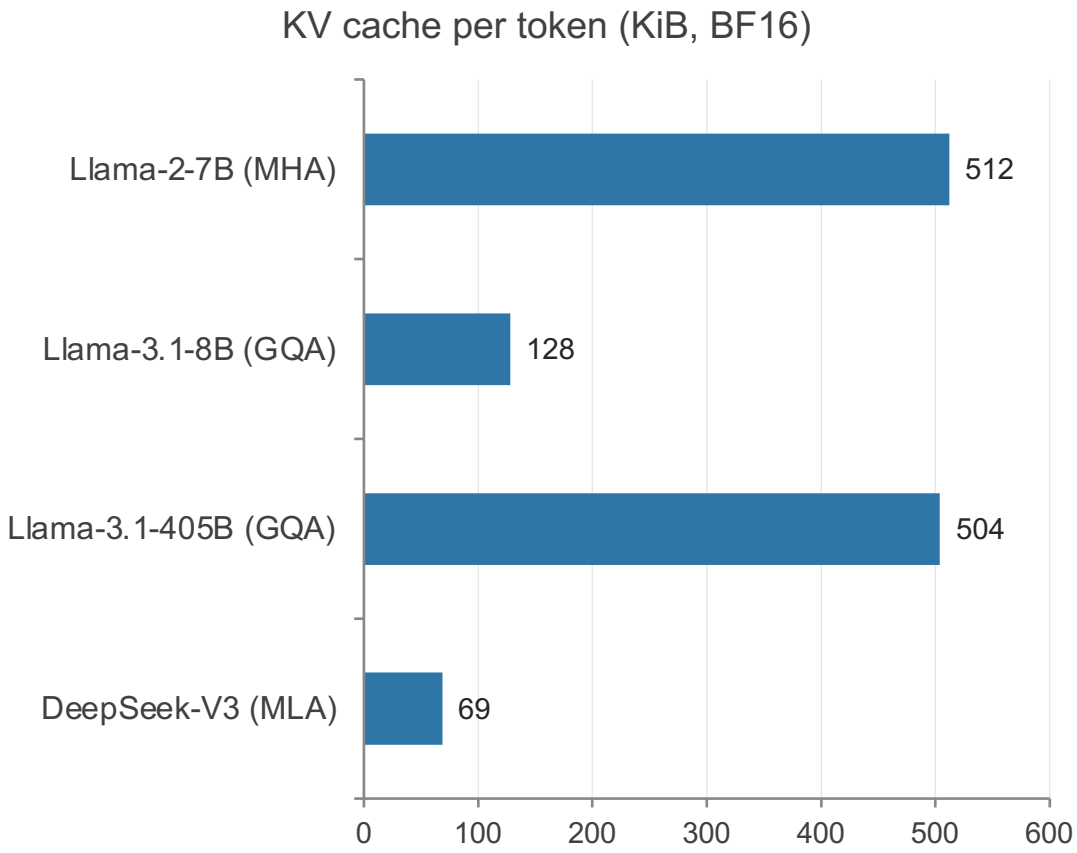
PART 4

Shrinking the KV cache

Fewer bytes per token: architecture, precision, and eviction.

Architecture: fewer KV bytes per token





Other ways of shrinking

Quantize the KV cache

- FP8 halves capacity and halves KV bandwidth in decode
- Per-tensor or per-head scales; small accuracy cost
- INT4/INT2 is active research (e.g., KIVI)
- **Lossy, usually small**

Evict or sparsify

- StreamingLLM: keep “attention sinks” + a recent window
- H2O: keep heavy-hitter tokens
- SnapKV: compress the prompt’s KV using observed attention
- **Lossy: changes outputs**

Summary

Technique	Problem it addresses	Benefit / Gains	Cons / Cost
KV cache	$O(n^2)$ recomputation	FLOPs	HBM capacity
Continuous batching	Idle slots, waiting arrivals	GPU time, TTFT	Dynamic memory, scheduler work
PagedAttention	Fragmentation	HBM capacity → batch size	Kernel indirection, small blocks
Prefix caching	Repeated prefill	FLOPs, TTFT	Capacity, exact-prefix rule
Chunked prefill	Generation stalls	TBT	Prefill efficiency
PD disaggregation	Interference, coupled plans	Goodput	KV transfer, P:D balancing
Hierarchical / distributed KV	HBM-limited hit rate	FLOPs, TTFT	PCIe / network / storage bandwidth
GQA / MLA / FP8 KV	Bytes per token	Capacity and bandwidth	Training-time choice; accuracy