Modern AI Systems: Agents and System Optimizations

LLM Serving Basics III: scheduling, paging and parallelism

Juncheng Yang

Harvard John A. Paulson School of Engineering and Applied Sciences

HARVARD UNIVERSITY MaDSys

Measurement and Design of Systems Lab

Plan for today

- LLM serving performance evaluation
- Scheduling and resource management
- Parallelism

Logistics

- Assignment 4 due next Tuesday midnight
  - Reading and tune engine flags
  - Speculative decoding
  - Agent-engine co-design
  - Measurement / observation + reasoning in write-up and video
    - **what you saw, what you did, what changed and why, what is the trade-off**
    - **more than one counter and metric when you measure**
- Project proposal due last Wednesday

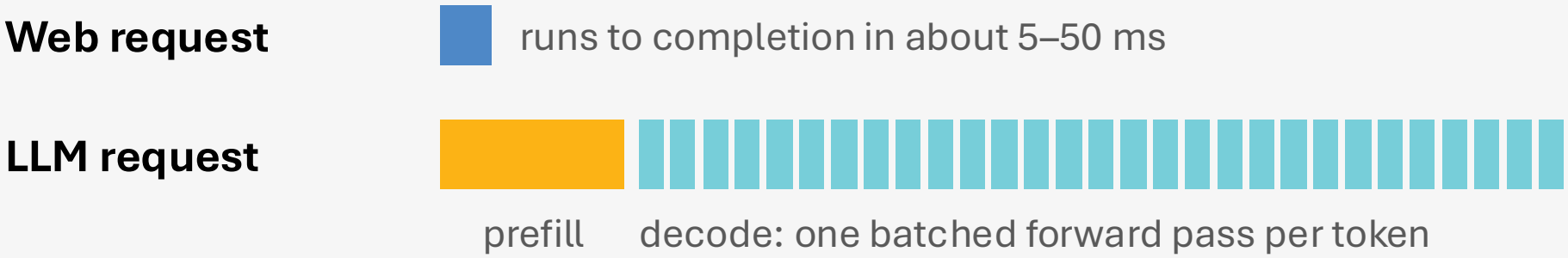
Why spend several classes on LLM serving?

Huge compute
LLMs need GPUs and other accelerators

Paid on every token
each user request reruns the model

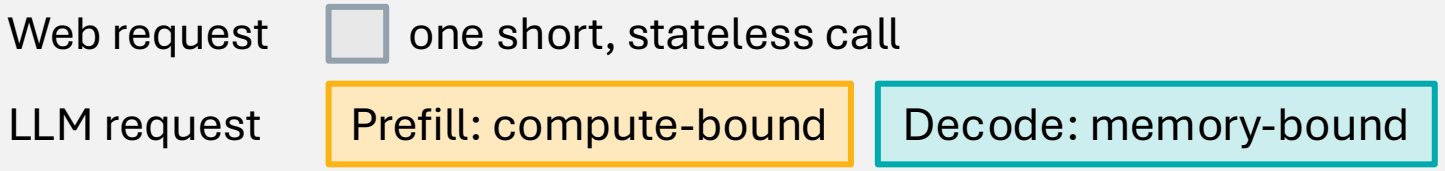
GPUs are costly
serving efficiency sets cost and speed

LLM serving is different from previous computing, e.g., web service, batch jobs

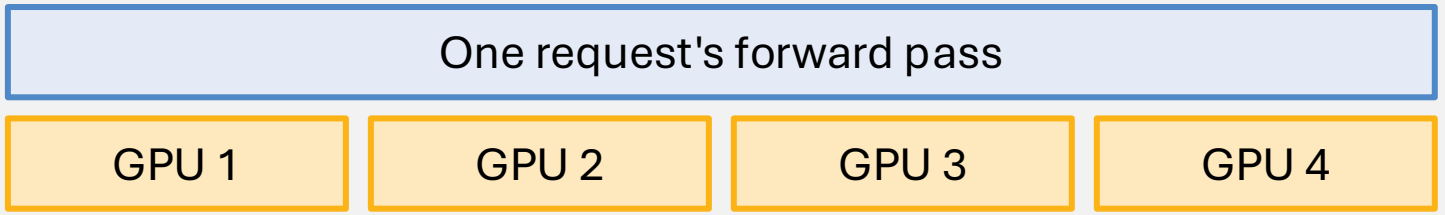


How LLM serving differs: inside a single request

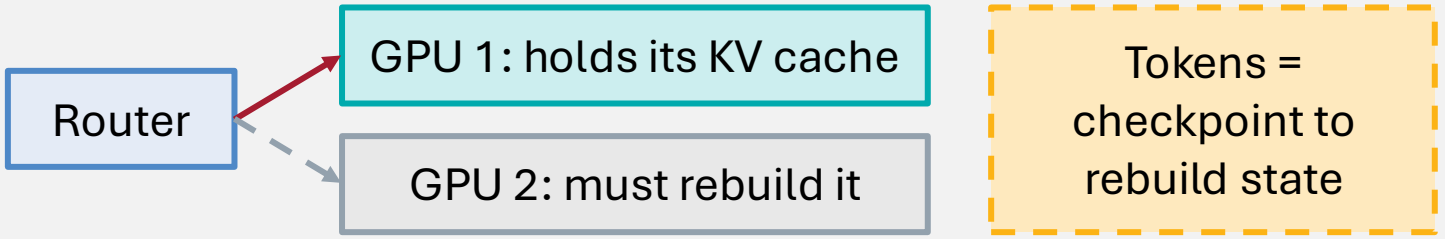
Two phases, two bottlenecks



Multi-GPU, multi-server compute and communicate



Soft, growing state

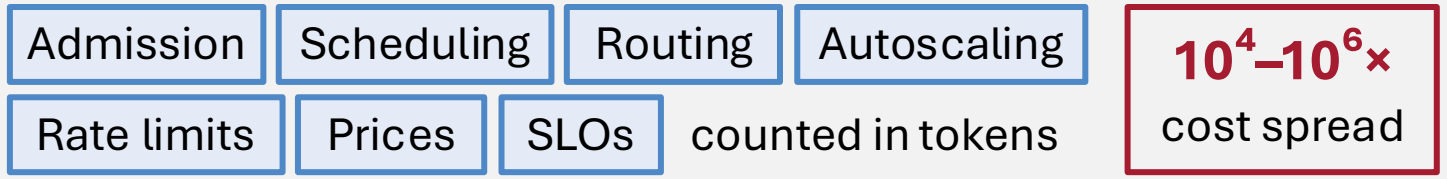


How LLM serving differs: across many requests

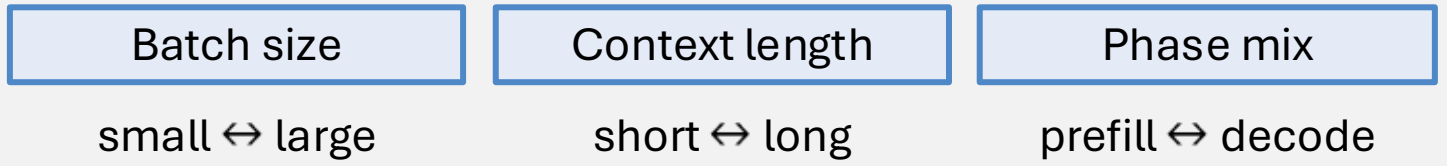
Unknown output length



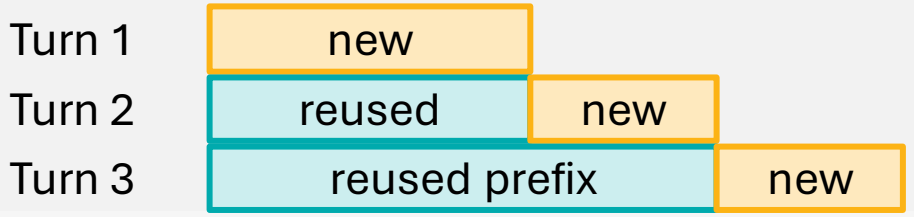
Tokens as the unit of work



Shifting bottlenecks



Agents and reasoning widen every gap



≈90% prefix reuse

KV cache becomes a storage problem

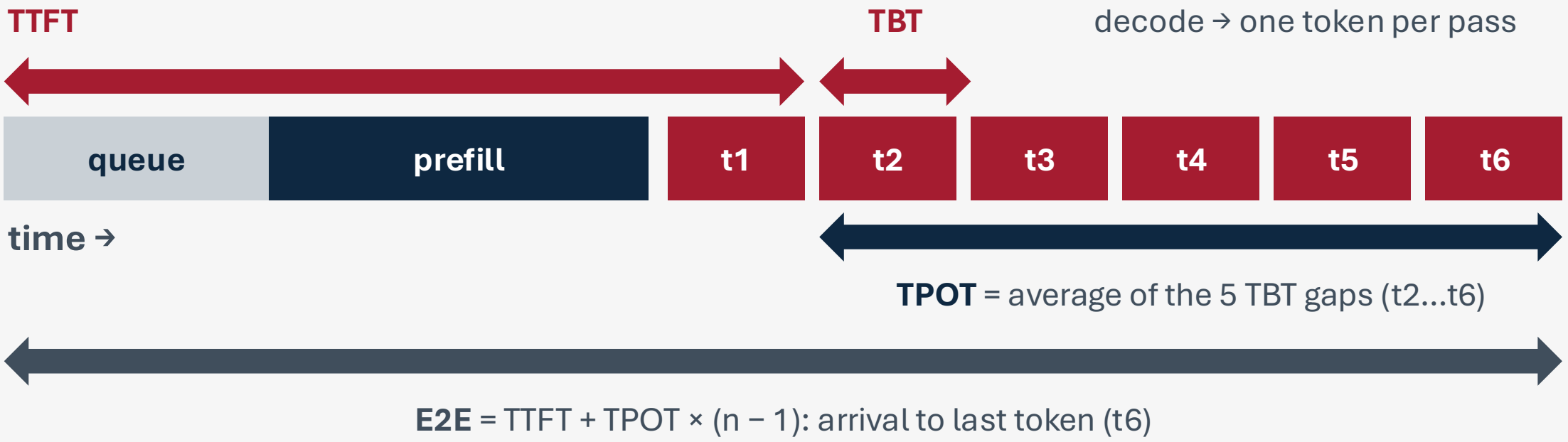
LLM inference metrics and evaluation

The two phases of LLM serving: prefill and decode

	Prefill	Decode
Granularity	All S prompt tokens at once	1 token / step
Dominant op	Matrix–matrix GEMM	GEMV / thin GEMM
Bottleneck	Compute (FLOPs)	Memory bandwidth
Latency driver	TTFT	TPOT × output length

Serving effect: one request mixes a compute-bound phase and a memory-bound phase

Serving metrics on one request's timeline



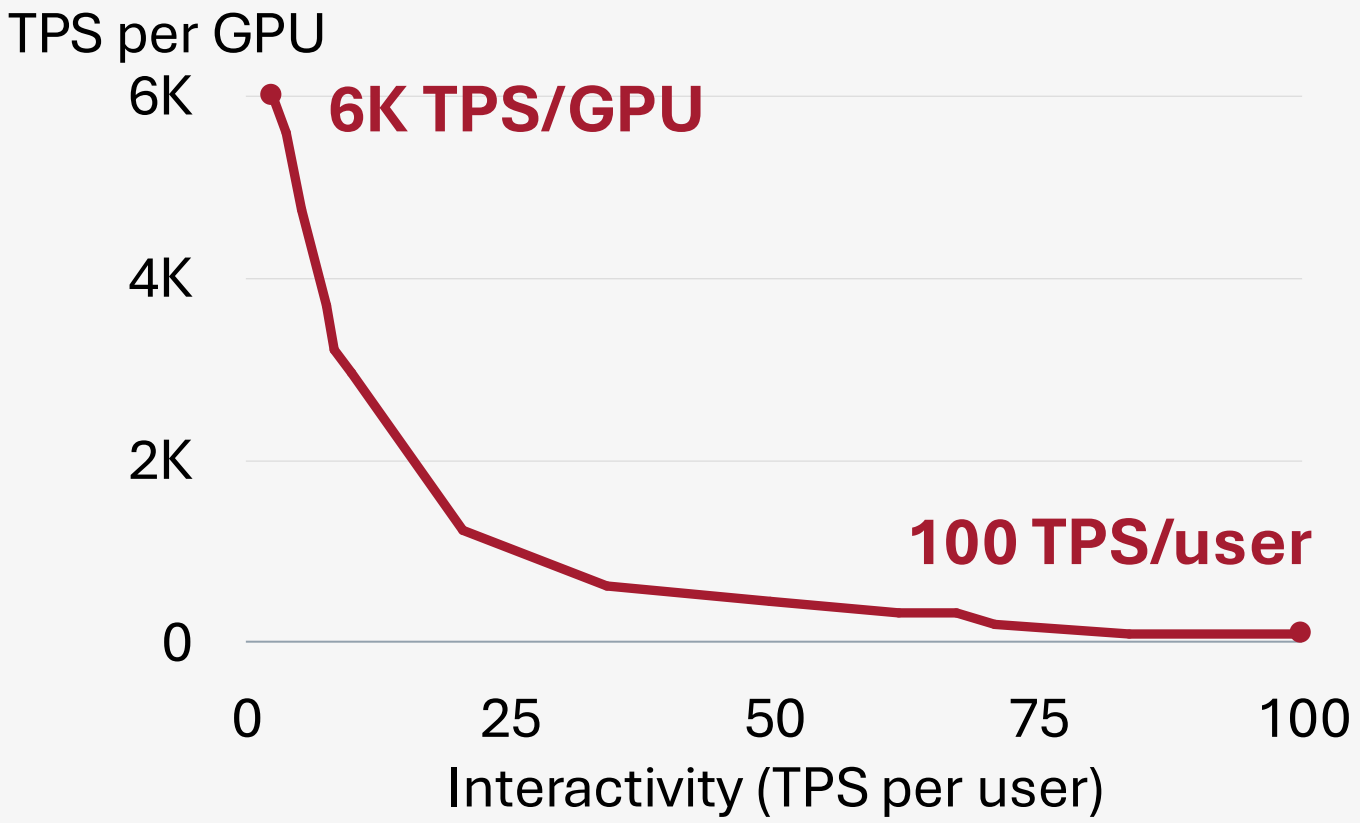
TBT is one gap, measured per token, so its P99 catches stalls.

TPOT averages every gap: (E2E − TTFT) ÷ (n − 1). A few long stalls barely move it.

Throughput is tokens/s per GPU across all requests, maximized by large batches.

Goodput is throughput counting only requests that meet their TTFT and TPOT SLOs.

GPU throughput vs user throughput



GPU throughput
all tokens the server emits per second

User throughput
1 ÷ TPOT — what one reader feels

Batch 1 → 64
20× the GPU, but 3× slower per user

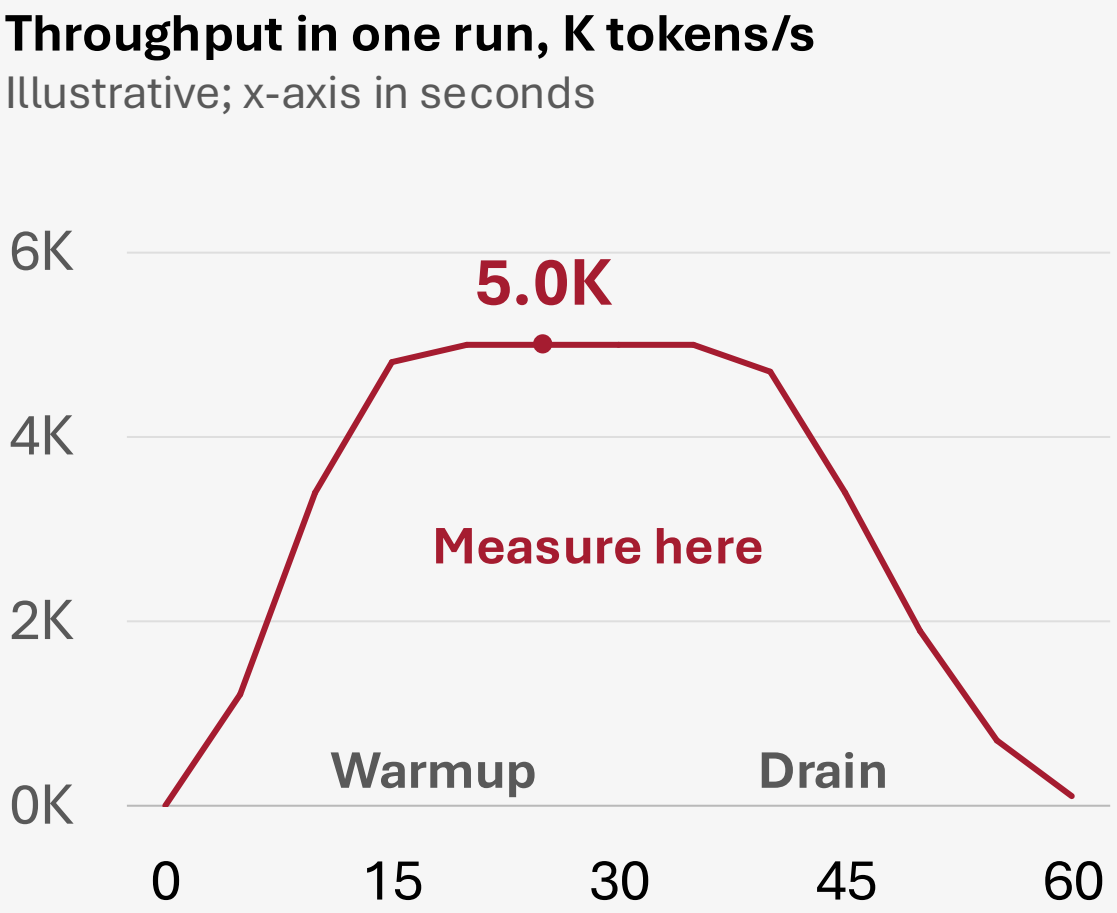
Pitfall 1: Short, transient runs

Example: 500 requests sent at once

- Early ones pay for kernel compilation, lazy loading, GPU clock ramping, and cold prefix caches;
- Late ones run in a shrinking batch.

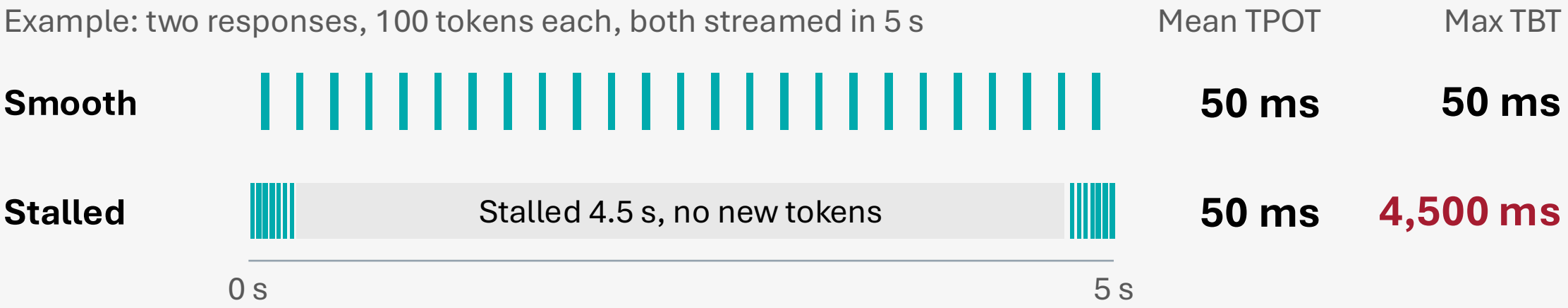
A whole-run average blends both.

What you should do: report steady state performance.



Pitfall 2: Reporting misleading or opaque metrics

Requests differ in prompt length, output length and latency sensitivity, so one average can hide what a user sees.



Same 50 ms TPOT, yet one user watches a frozen stream for 4.5 s.

What you should do: (1) show the full distribution; (2) report TTFT, TPOT, and throughput.

Pitfall 3: Unrealistic workloads

Token content

Benchmark `kq 7z ##vl 3x`

Real `Summarize this email`

Draft tokens rarely get accepted

Length distribution

Benchmark

Real **320K**

One 320K prompt stalls a batch

Request arrival

Benchmark

Real

Bursts push P99 TTFT way up

Prefix cache hits

Benchmark

Real

A shared prompt skips prefill

What you should do: replay real traces, with real lengths and arrival times.

Pitfall 4: Unfair comparison

Unequal resources or tuning

Ours: 8 GPUs



Baseline: 4 GPUs



Twice the GPUs can explain a 2× win

Untuned baselines

Default baseline



Tuned baseline

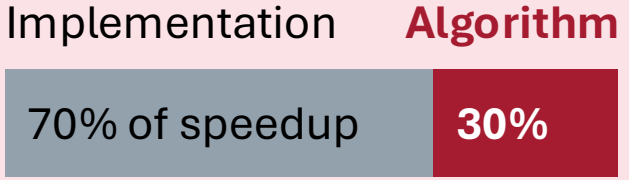


Ours, tuned



Real gain over tuned: 1.8 ÷ 1.6 ≈ 1.1×

Not isolating the algorithm



Only 30% comes from the algorithm

Fix: tune every baseline on the same hardware, then switch each technique off to measure its impact.

Batching, scheduling, paging and caching

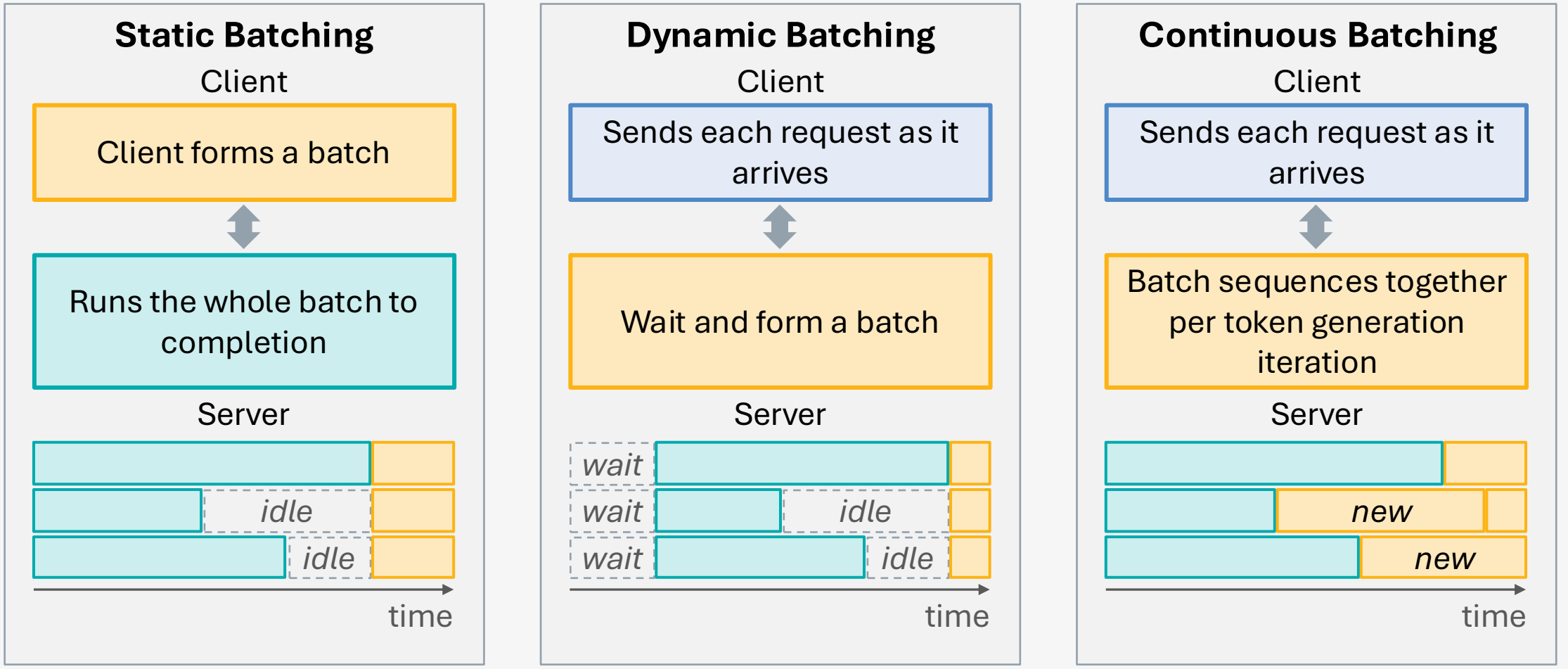
Batching

Request scheduling

Memory management

Prefix caching

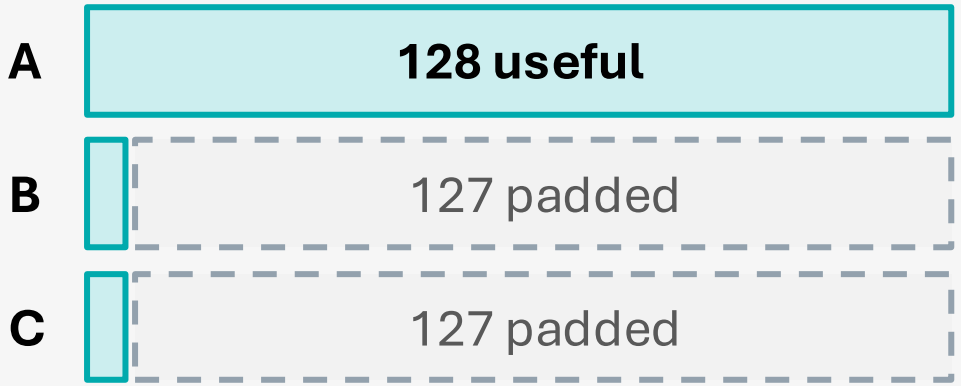
Continuous batching and iteration-level scheduling



Ordinary batching breaks when requests differ in input length and KV history

Request	Phase	Tokens this step	KV-cache length
A	Prefill	128	0
B	Decode	1	500
C	Decode	1	1,000

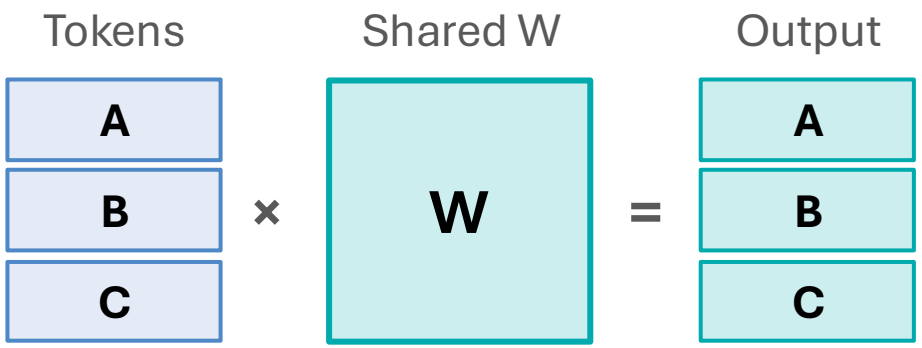
Pad every input to 128 tokens



Yu et al., OSDI 2022 (Orca)

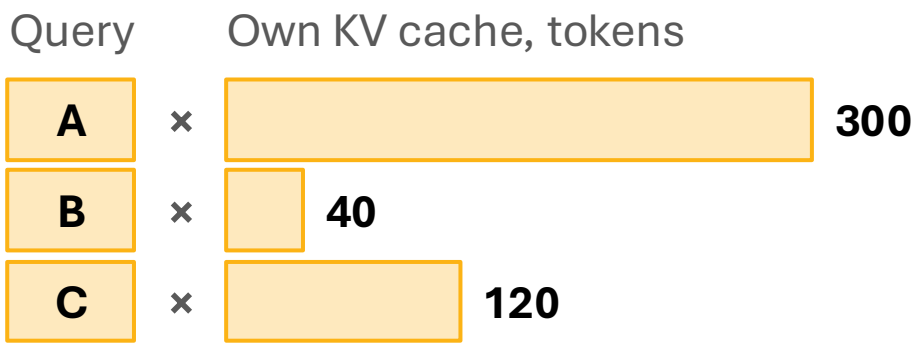
Attention can't share one GEMM: each request reads its own KV cache, and the lengths differ

Linear layer: batched



Every token multiplies the same W, so one GEMM serves all requests

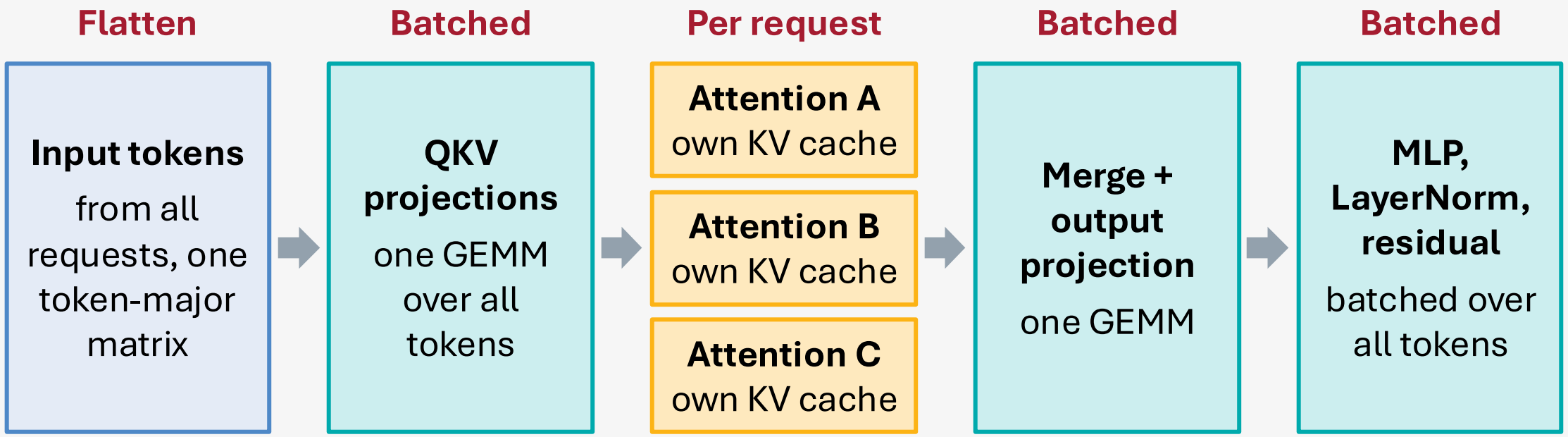
Attention: per request



No shared operand and ragged lengths, so each request runs its own attention

Orca's observation: not every Transformer operation needs the same batching strategy.

Selective batching: batch every linear layer, split only attention by request



Why it works: linear layers treat every token the same, so one GEMM serves all requests; only attention needs each request's own KV cache.

Request scheduling

FCFS/SJF · chunked prefill · disaggregation

Request scheduling: first-come, first-served (FCFS)

vLLM and SGLang default to FCFS

- Admits waiting requests in arrival order as KV-cache space frees up
- Simple and starvation-free, but one long request at the head delays all behind it

But how to mix the prefill of an incoming request vs ongoing decode?

Prefill immediately

Larger batches, but blocks all decoding requests



Decodes stall here

Wait until decode finishes

TPOT less affected, but long queueing for new requests



New request waits

Scheduling

Prefill-first schedulers protect TTFT; decode-first schedulers protect TPOT

Prefill-first SGLang



+ **TTFT:** new prompts start at once
+ **Throughput:** batch refills every step
– **TPOT:** each arrival stalls all decodes

Decode-first FasterTransformer



+ **TPOT:** steady token stream
– **TTFT:** new prompts wait for a drain
– **Throughput:** batch shrinks as requests end

Two ways to break the trade-off

Chunked prefill
Mix small prompt chunks into decode batches

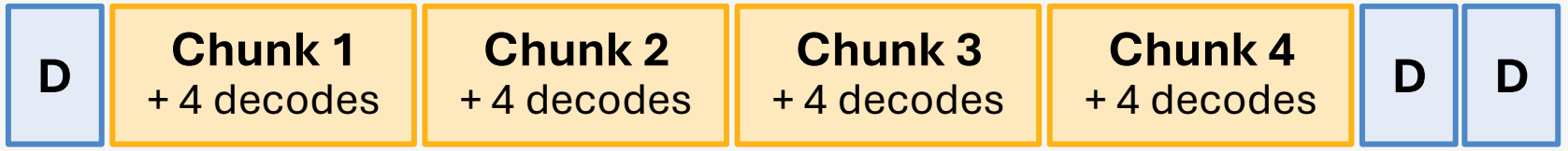
Disaggregation
Run prefill and decode on separate GPUs

Framing from Agrawal et al., Sarathi-Serve, OSDI 2024

Chunked prefill aims to strike a balance

What is the downside of chunked prefill?

512-token chunks



Each step: 512 prompt tokens + 4 decodes, so a token every **~23 ms**

A per-step token budget bounds iteration latency

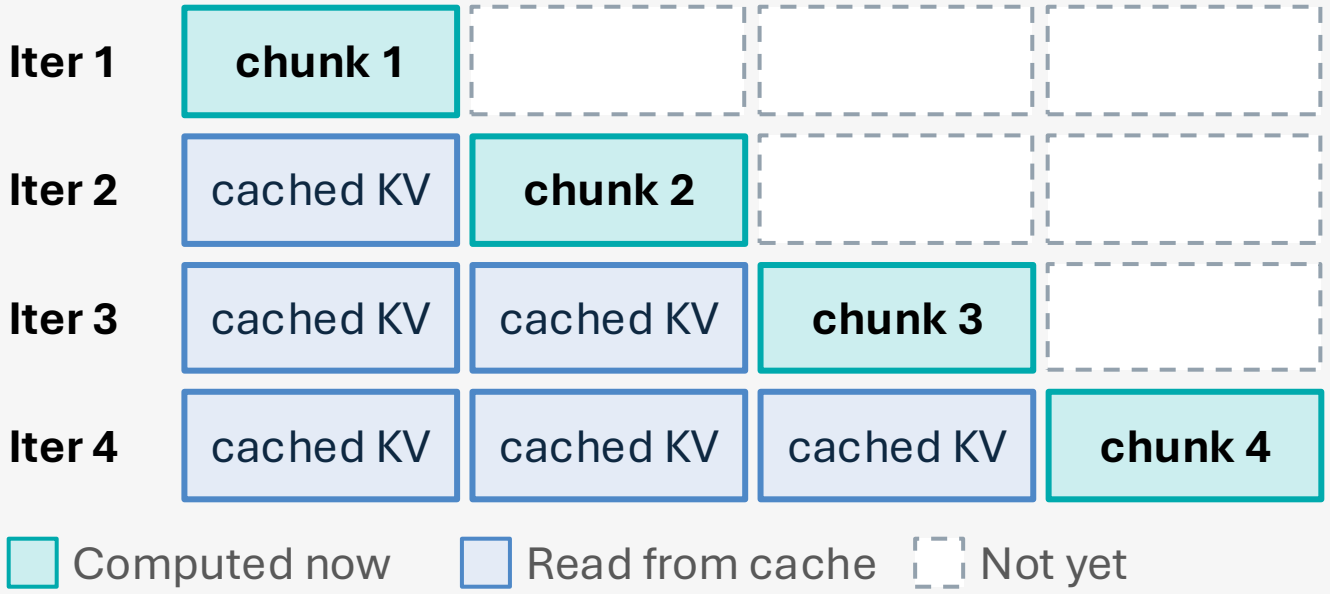
This relies on mixed-batch kernels being as efficient as pure prefill or decode kernels.

> **SGLang**: prefill-prioritized (with decode protection --prefill-decode-interval N)
> **vLLM**: decode-prioritized (with chunked prefill)

Agrawal et al., Sarathi-Serve, OSDI 2024; SGLang and vLLM scheduler docs

Chunked prefill needs more memory reads

2,048-token prompt, chunk size 512



Cost: KV is re-read

Chunk k reads (k − 1) × 512 tokens

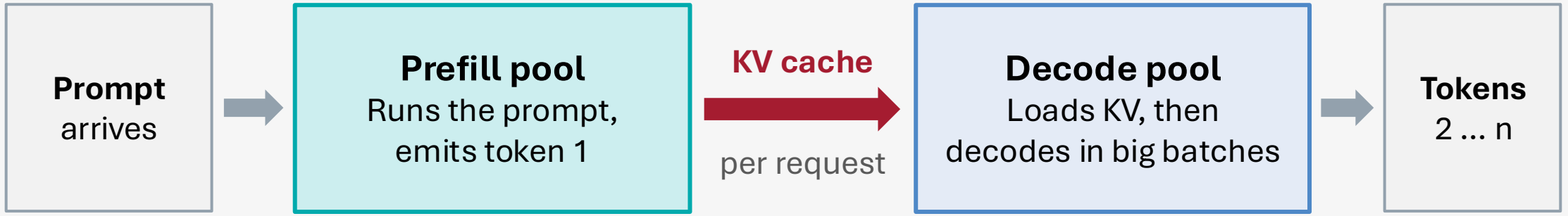
Same total prefill FLOPs, spread over 4 shorter steps.

Chunk size trades decode latency against prefill efficiency

Agrawal et al., Sarathi (2023) and Sarathi-Serve (OSDI 2024); vLLM: enable_chunked_prefill

PD disaggregation separates prefill and decode

One request's path



Prefill pool
Compute-bound; sized and parallelized for TTFT

KV transfer
Prompt KV sent layer by layer over NVLink or RDMA

Decode pool
Memory-bound; large batches, sized for TPOT

DistServe (Zhong et al., OSDI 2024); Splitwise (Patel et al., ISCA 2024); Mooncake (Qin et al., FAST 2025)

Each pool gets its own parallelism and batch size, tuned to its own SLO

Configure each pool separately

	Prefill pool	Decode pool
Target	TTFT SLO	TPOT SLO
Batch	one or a few prompts	100s of sequences
Parallelism	more TP, less latency	more replicas
Best GPU	high FLOPS	high HBM bandwidth

The costs: a KV transfer per request, a P:D ratio to tune, two or more instances.

Zhong et al., DistServe, OSDI 2024; Patel et al., Splitwise, ISCA 2024

PD disaggregation requires moving the KV cache between nodes

KV size: Llama-3.1-70B (GQA), FP16

2 (K, V) × 80 layers × 8 KV heads
× 128 dims × 2 bytes

= 320 KB per token

× 65,536 tokens = 20 GB

Its prefill ≈ 11.5 s

2 × 70B × 65,536 ÷ (4 × 200 TFLOPS)

Time to move 20 GB

Link	Bandwidth	Time
NVLink, H100	450 GB/s	**44 ms**
NVLink, A100	300 GB/s	**67 ms**
InfiniBand 400 Gb/s	50 GB/s	**400 ms**
PCIe 4.0 x16	32 GB/s	**625 ms**
Ethernet 100 Gb/s	12.5 GB/s	**1.6 s**

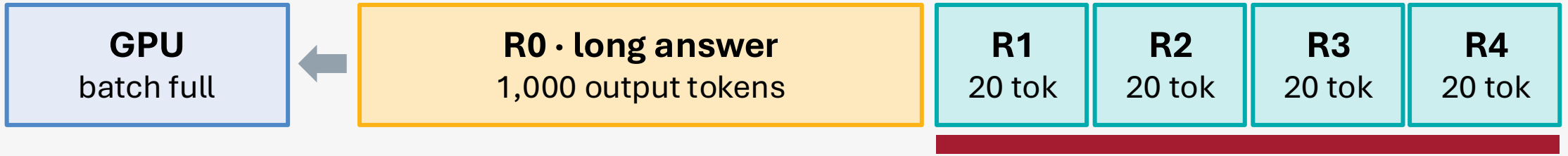
Even 400 ms is under 4% of the prefill.

Streaming per layer hides most of it: layer i's KV ships while layer i+1 computes.

Scheduling

From FCFS to SJF: short requests should not wait behind long ones

Head-of-line blocking: a full batch, served in arrival order



Queue head on the left; arrival order R0 → R4

R1–R4 wait for all of R0's tokens

FCFS

Serve in arrival order. Default in vLLM and SGLang; one long job delays every request behind it.

SJF

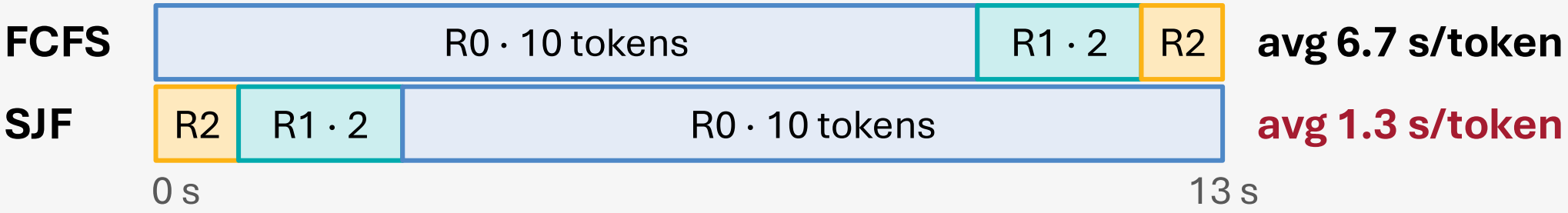
Serve the shortest job first. Minimizes average waiting time, but needs each job's size up front.

For LLMs, job size depends on output length, and it is unknown until decoding ends.

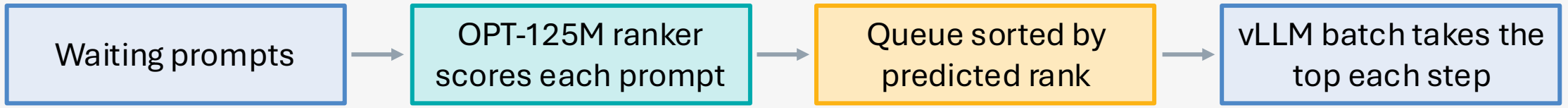
ADVANCED — skim, or read more on your own

Shortest-job-first needs output lengths, so learn to rank them

Three requests arrive together; the GPU emits one token per second



The catch: output length is unknown until decoding ends, so it must be predicted



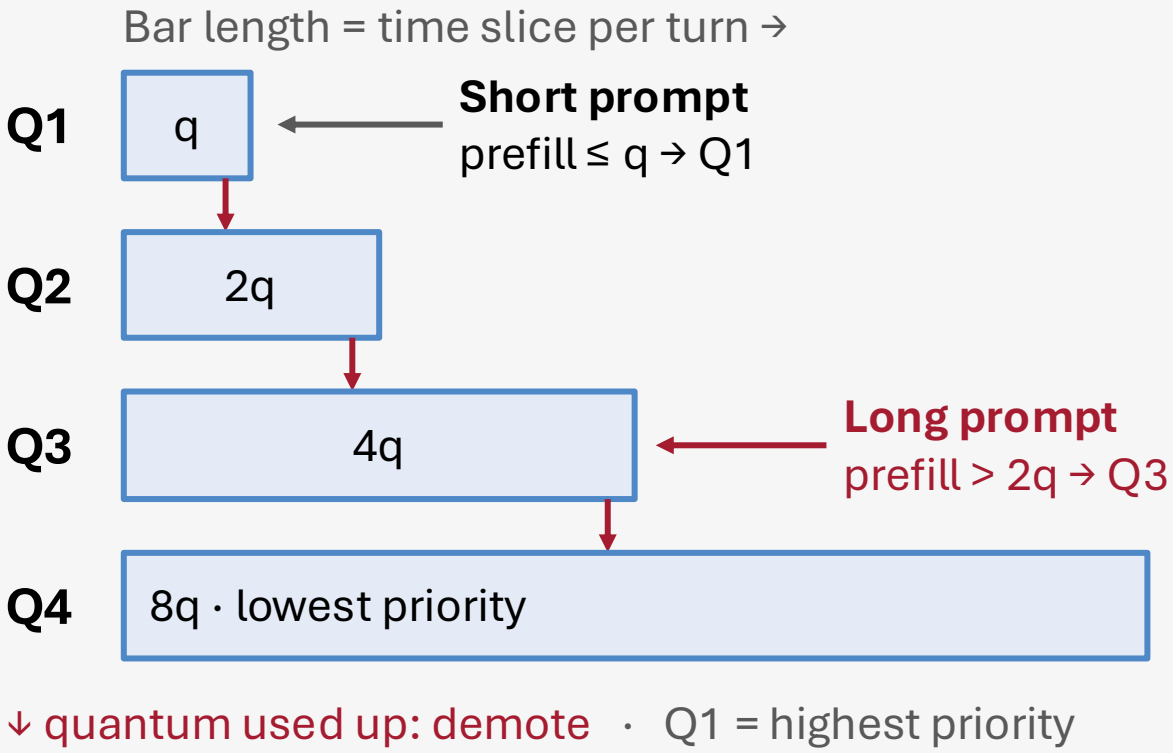
Rank instead of regress
Trained with a ranking loss; only relative order matters

Starvation guard
A request that waits too long is promoted

Fu et al., Efficient LLM Scheduling by Learning to Rank, NeurIPS 2024

ADVANCED — skim, or read more on your own

FastServe: token-level preemption with a skip-join multi-level queue



Skip-join by prompt length

A new job enters the highest queue whose quantum covers its prefill time

Demotion

A job that uses up its quantum drops one queue

Promotion

Jobs that wait too long are promoted back up to prevent starvation

Why it works: long jobs sink and short jobs finish first, so it **approximates SRPT** (shortest remaining processing time first) without knowing job sizes

Wu et al., FastServe, NSDI 2026 (arXiv 2305.05920)

Check your understanding: request scheduling

Q1. P99 TBT spikes whenever a long prompt arrives. Which techniques can fix this?

Answer: Chunked prefill caps prompt tokens per step; PD disaggregation keeps prefill out of decode steps.

Q2. When is PD disaggregation not worth deploying?

Answer: Short prompts or small fleets: KV transfer, P:D tuning and extra instances cost more than the stalls.

Q3. SJF needs job sizes. Are there other methods to avoid it for agentic workloads?

Answer: Use uncached prefill tokens as job size.

Memory management

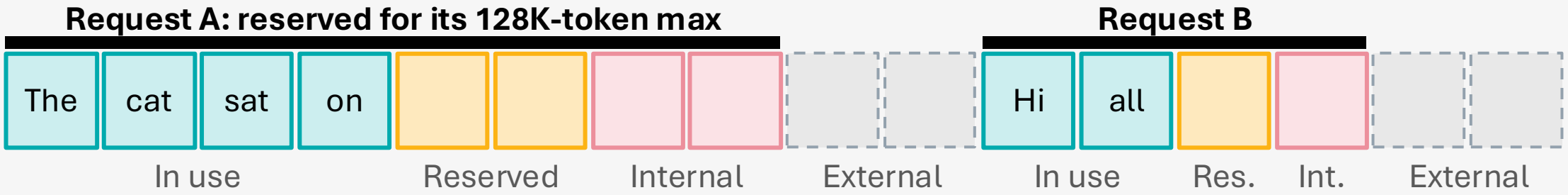
PagedAttention

Unknown output length forces KV reservation that wastes memory three ways

Llama-3.1-70B on two H200s, 282 GB of memory



320 KB per token · **40 GB** per request at the 128K-token limit · **~3** such requests fill the KV budget



1 Reserved

Slots held for tokens not yet generated. Used later, but idle until then.

2 Internal fragmentation

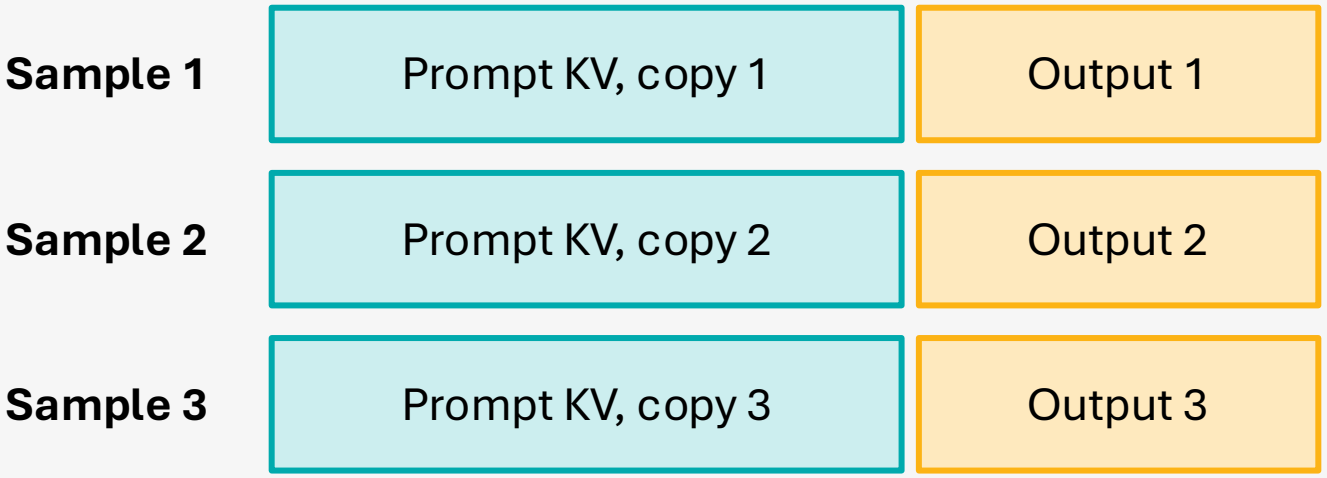
Slots past the real end. The reply hits EOS before its maximum length.

Only 20–38% of KV memory held real tokens in Orca-style contiguous allocators.

Sources: Kwon et al., SOSP 2023, Fig. 2–3; Meta Llama 3.1 70B config, 2024; NVIDIA H200 datasheet

Contiguous caches cannot be shared, so n samples store the same prompt n times

Parallel sampling, n = 3, contiguous slabs

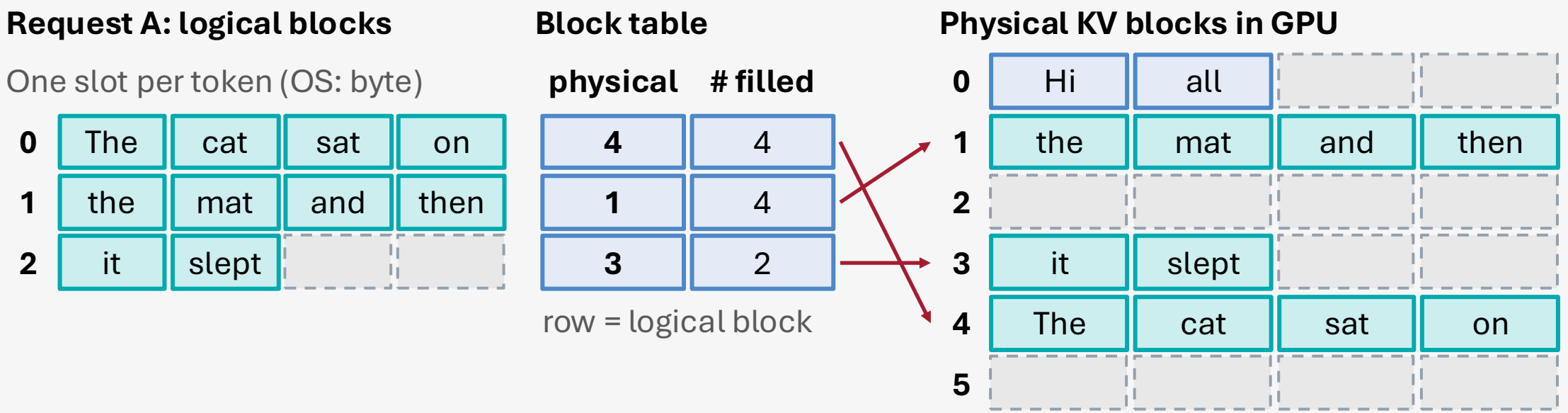


Parallel sampling: n replies to one prompt copy the prompt n times

Beam search: beams share the prompt and a prefix that shifts every step; a fork copies the whole slab

Shared system prompt: every user request repeats the same instructions

PagedAttention manages the KV cache like OS paging



Demand paging
New block only when the last one fills

Shared pages, copy-on-write
Shared prompt blocks with reference counts

Swap to disk
Swap blocks to CPU memory

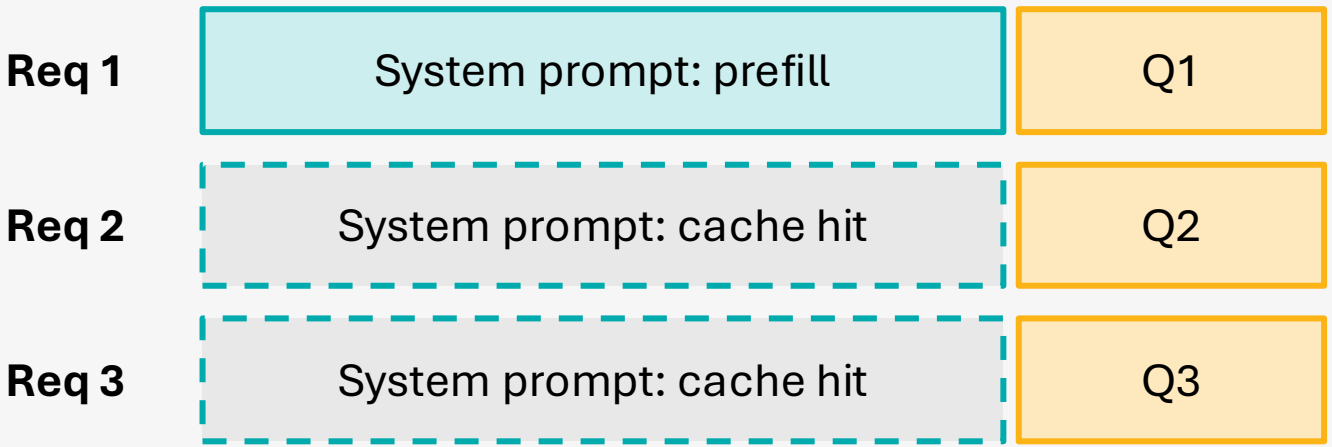
One level of indirection removes the need for contiguous memory.

Source: Kwon et al., PagedAttention and vLLM, SOSP 2023, Fig. 6. Blocks hold 16 tokens; 4 shown.

Prefix cache

Prefix cache: reuse the KV of a prefix the server has already computed

Three chat requests share one system prompt



Prefill work, 1,000-token prompt + 50-token question

No cache: 3 × 1,050 = 3,150 tokens

Prefix cache: 1,050 + 50 + 50 = 1,150 tokens

Where shared prefixes come from

System prompts: every request repeats the same instructions

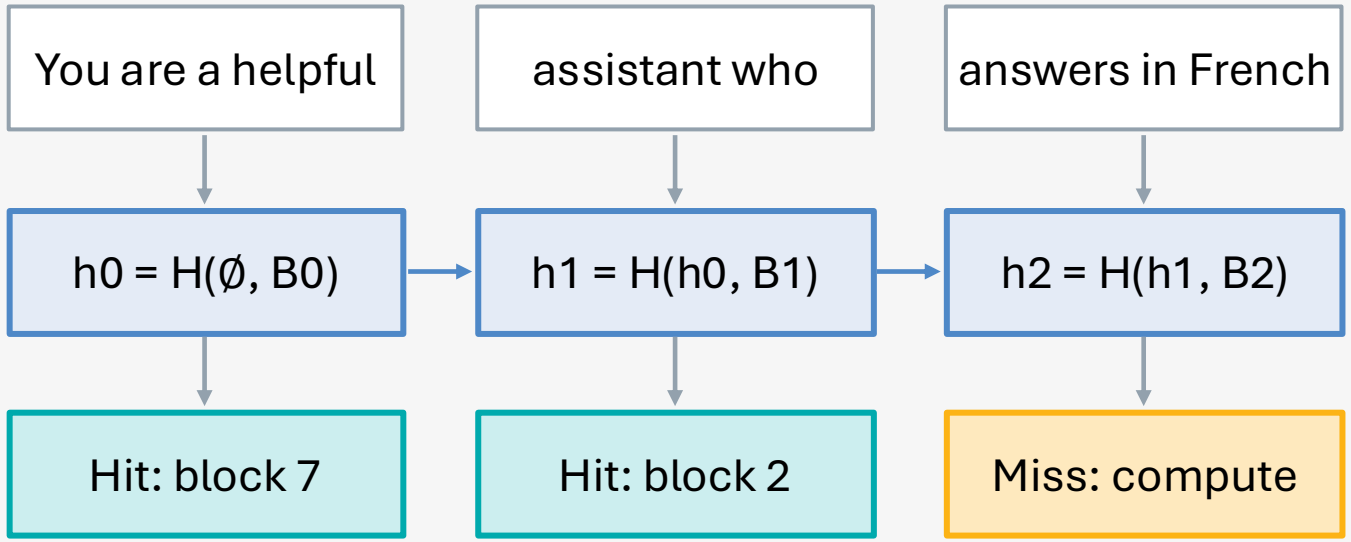
Multi-turn chat: each turn resends the history

Agents: each step resends tools and the trace so far

Exact match only: a token's KV depends on every token before it.

Block hashing: find cached KV blocks by a hash of the whole prefix

One prompt, 4 tokens per block (vLLM uses 16)



Each hash folds in its parent, so **h2** considers all 12 tokens, not just the last 4.

Lookup: hash each full block of a new prompt; stop at the first miss

Sharing: a hit raises the block's reference count; nothing is copied

Eviction: unreferenced blocks stay cached until memory is needed, then go in LRU order

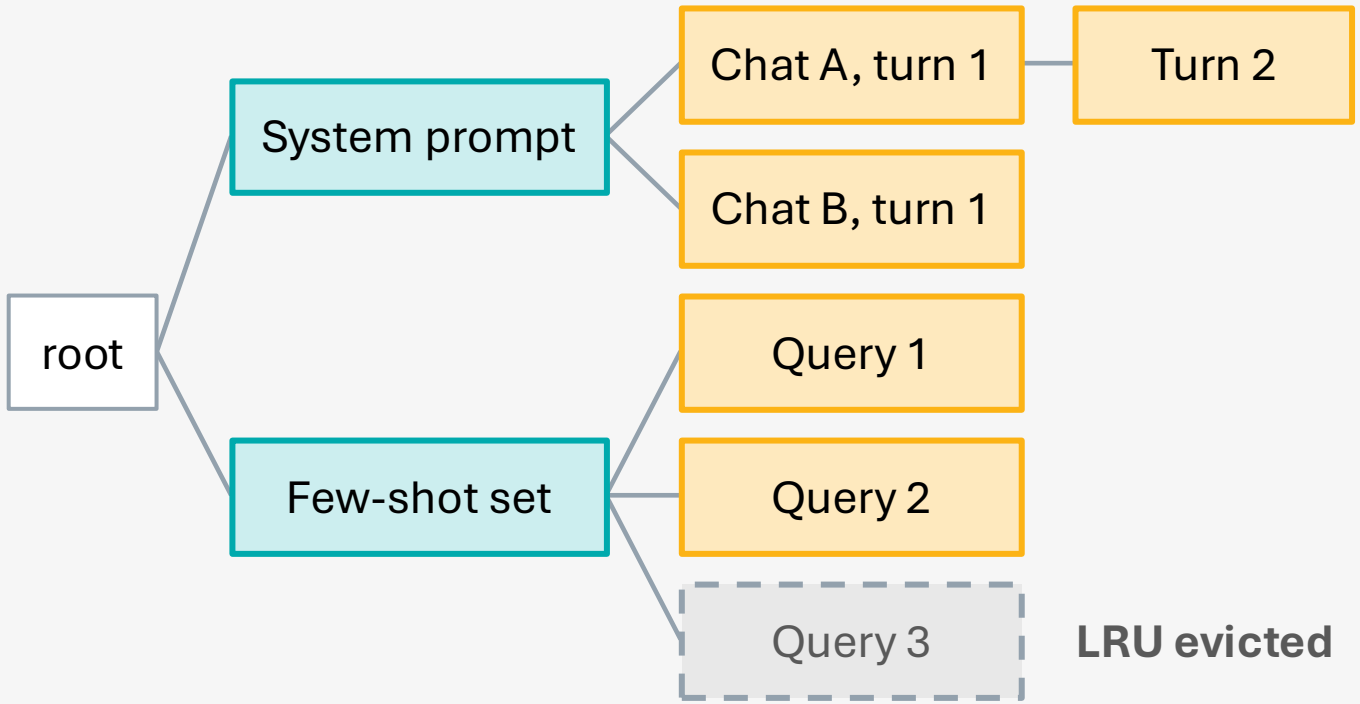
Granularity: a partly filled last block is never shared

A hash table on top of PagedAttention turns shared blocks into a cache.

Source: vLLM documentation, Automatic Prefix Caching design, 2024

SGLang RadixAttention: keep every request's KV in a radix tree and share its prefixes

One tree holds the KV of every cached request



Arrive: match the longest prefix in the tree; prefill only the rest

Finish: insert the new tokens as a branch; the KV stays on the GPU

Evict: drop least-recently-used leaves no running request uses

Branches share their parent's KV, so a common prefix is stored once.

Source: Zheng et al., SGLang, NeurIPS 2024, Sec. 3 and Fig. 3

Check your understanding: prefix caching

Q1. Two prompts differ only at token 3 of 1,000. How much KV can they share?

Answer: Two tokens at most, and none with 16-token block hashing: each token's KV depends on all tokens before it.

Q2. Why does maximizing prefix-cache hits conflict with fair scheduling?

Answer: Serving the client with the hottest cache first can starve others.

Q3. A cached prefix ends mid-block. Which index can still reuse it: block hash or radix tree?

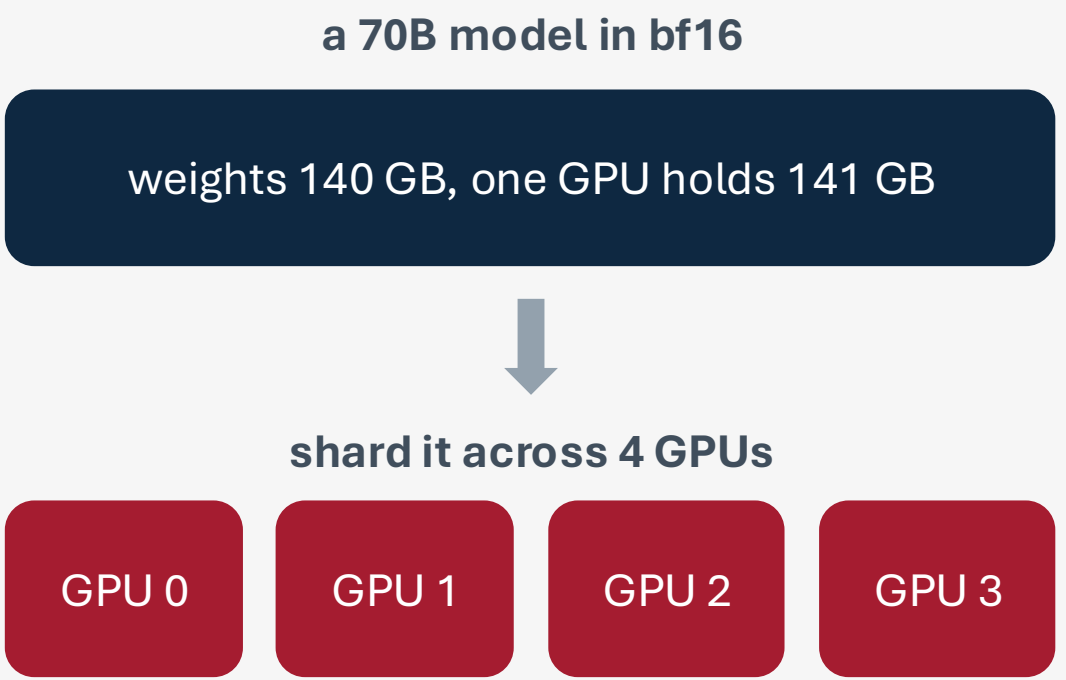
Answer: The radix tree, which matches at token granularity; block hashing reuses only full blocks.

Parallelism: one model, many GPUs

Data parallelism (DP)
Tensor parallelism (TP)
Pipeline parallelism (PP)
Expert parallelism (EP)
Sequence parallelism (SP)

Why one GPU is not enough

- Weights alone outgrow a single GPU; KV cache competes for the same HBM
- One GPU's bandwidth caps decode speed
- Parallelism buys capacity and speed



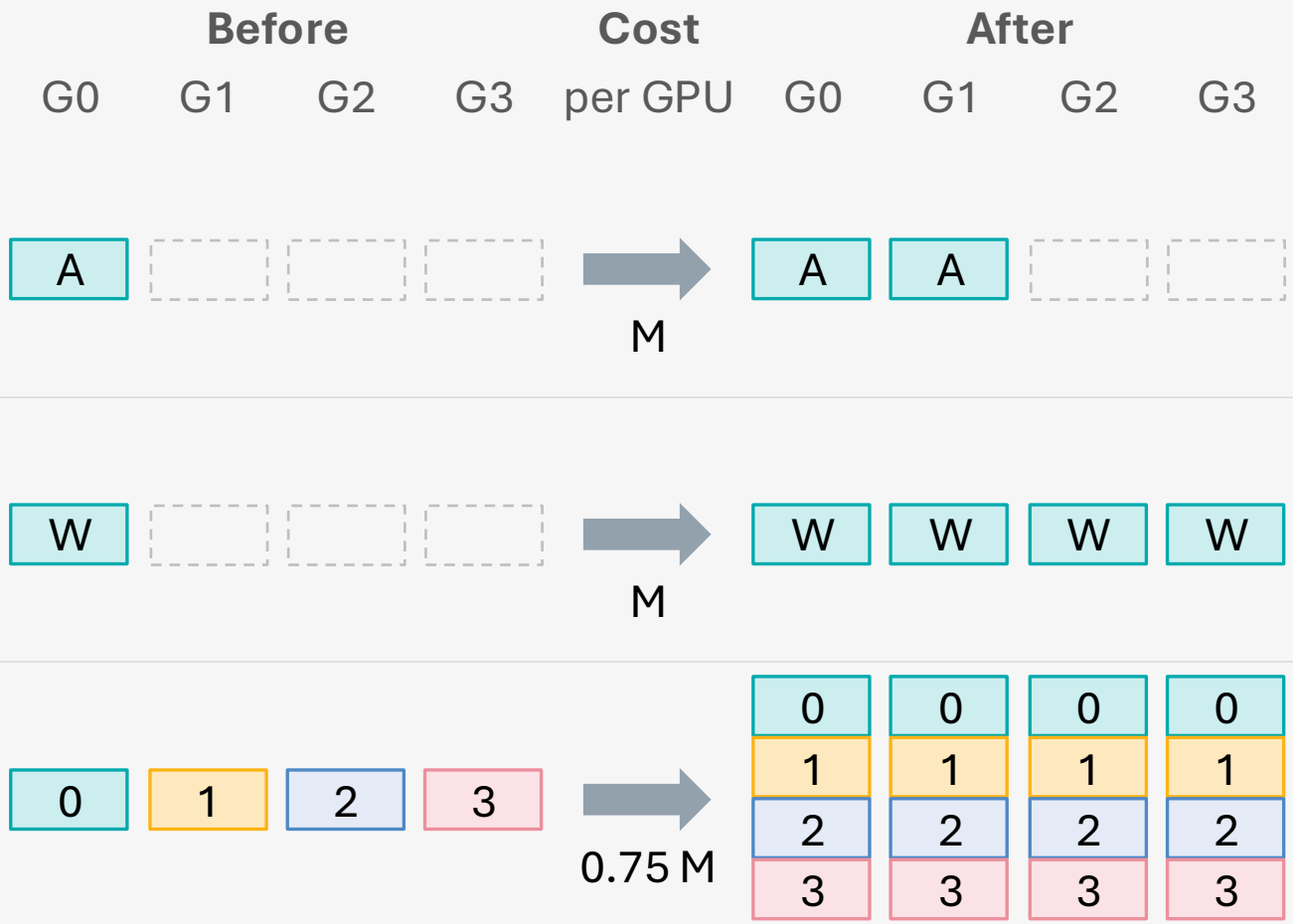
Back-of-envelope

$$t_{\text{token}} \geq 2N \div \text{BW}_{\text{HBM}}$$

70B bf16 on H200: 140 GB ÷ 4.8 TB/s ≈ 29 ms

Collectives: send/recv, broadcast, all-gather

Color = GPU that produced the data;
number = slice; M = size of A, W, or 0–3



Send / recv

One GPU sends a buffer to one peer

Used by: PP stage hops, SP ring

Broadcast

One GPU copies its buffer to all

Used by: batch inputs to TP ranks

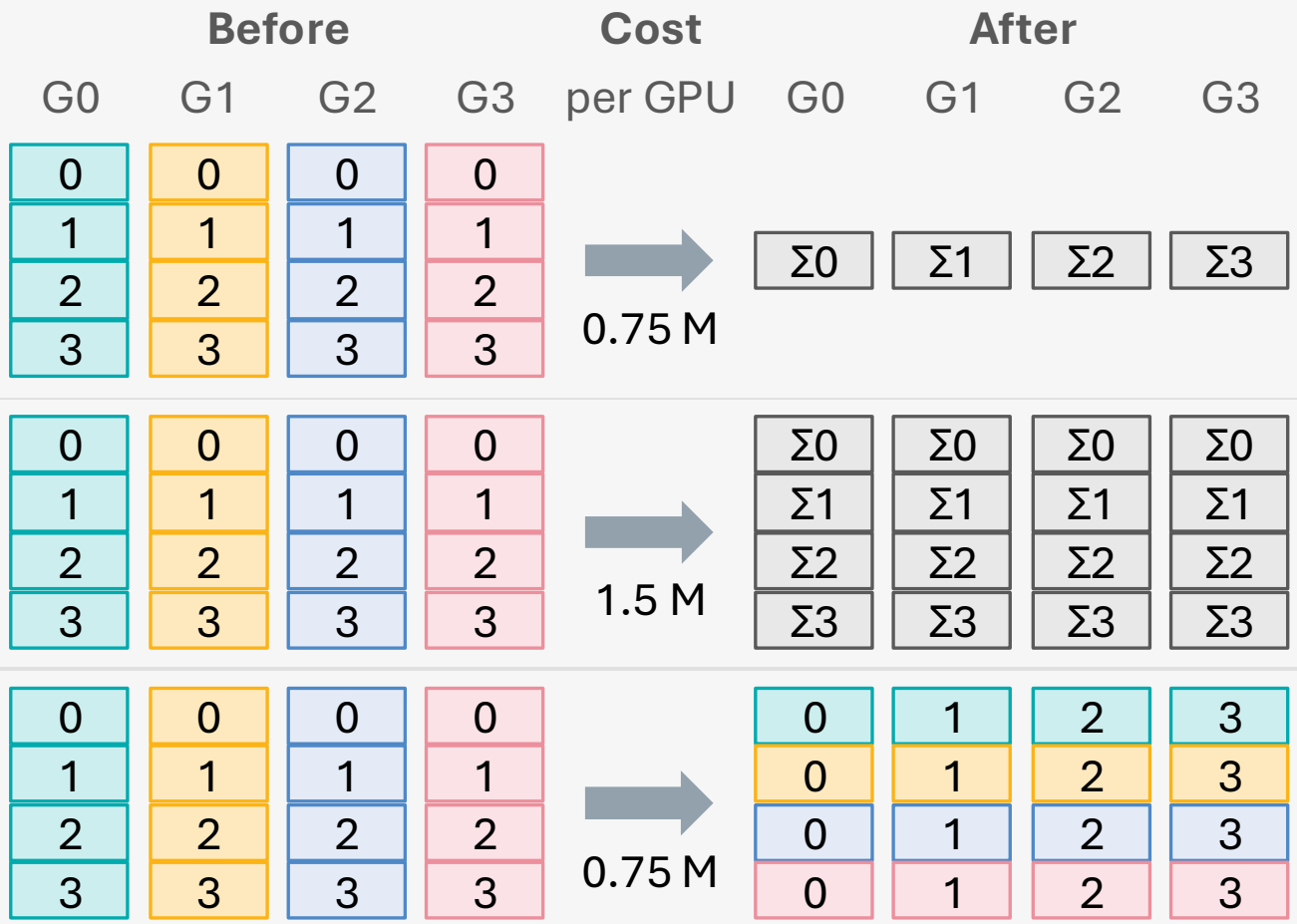
All-gather

Each adds a shard; all get the whole

Used by: TP outputs, SP K/V

Collectives: reduce-scatter, all-reduce, all-to-all

Color = source GPU; number = slice
Gray Σ = sum over GPUs; M = all 4 slices



Reduce-scatter

Sum across GPUs; each keeps one slice

Used by: TP with sequence parallelism

All-reduce

Sum across GPUs; every GPU keeps all
= reduce-scatter + all-gather

Used by: TP after row-parallel layers

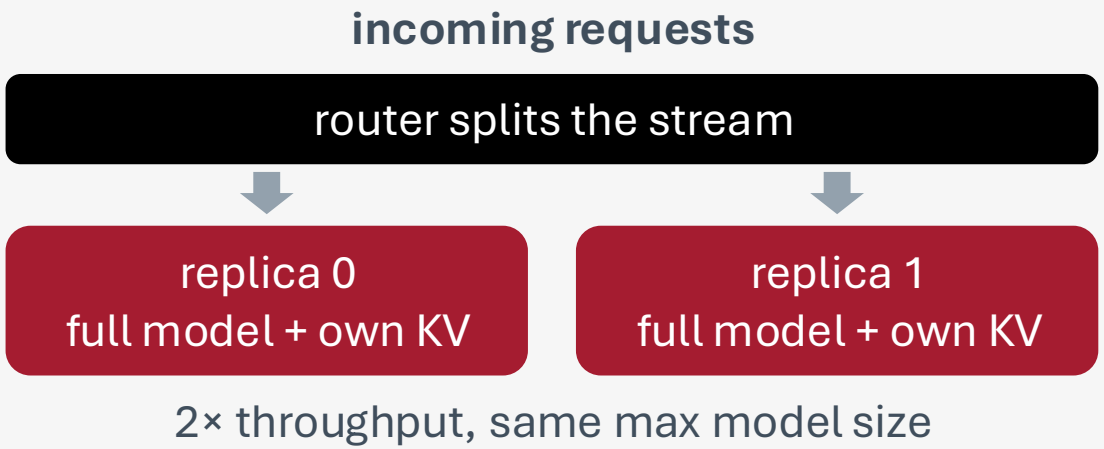
All-to-all

GPU i sends slice j to GPU j

Used by: EP token dispatch and combine

Data parallelism: more replicas

- Every GPU holds the whole model
- Requests are split across replicas
- Scales throughput
- Each replica keeps its own scheduler and KV cache



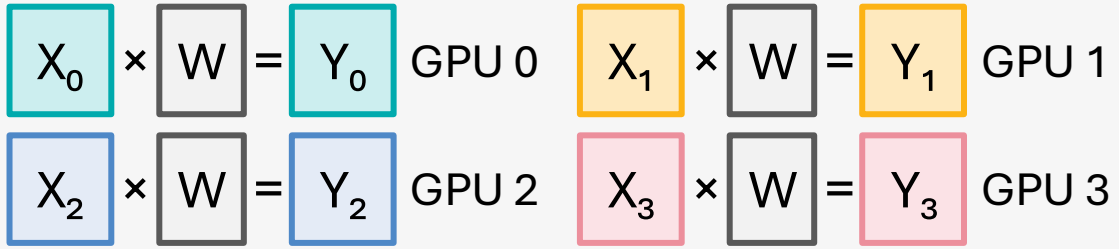
Back-of-envelope

$$\text{Throughput} \approx G \times \Theta_1$$

G replicas; memory per GPU stays 2N + KV

Comm: 0, replicas never talk

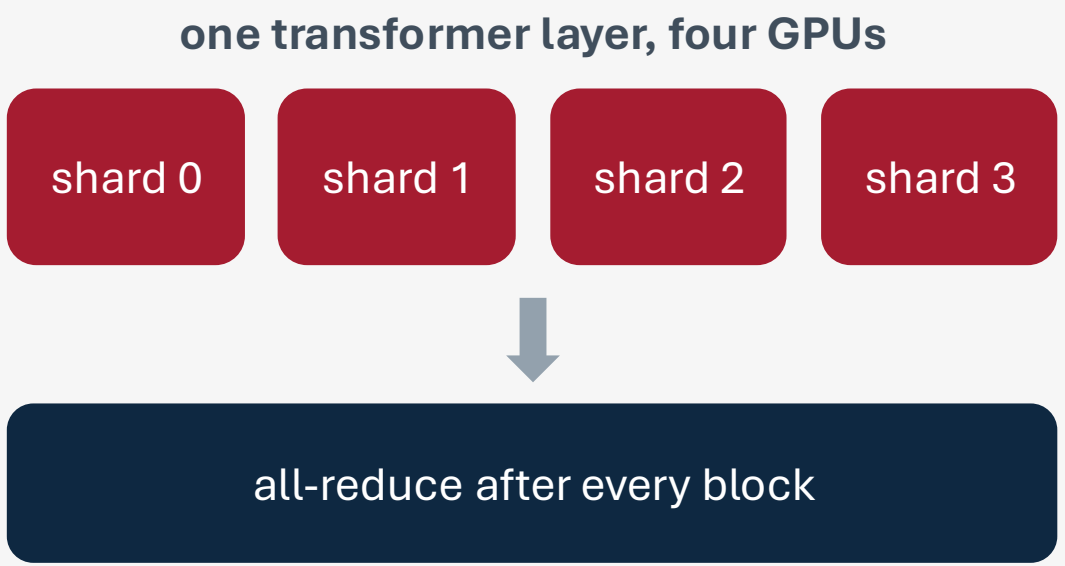
Matrix view



Same W on every GPU; the batch splits by rows

Tensor parallelism: split within a layer

- Each layer's matrices are sliced across GPUs
- Every GPU computes part of every layer
- An all-reduce joins the slices each block
- Needs NVLink-class bandwidth, so keep it inside one node



more GPUs means more communication

Back-of-envelope

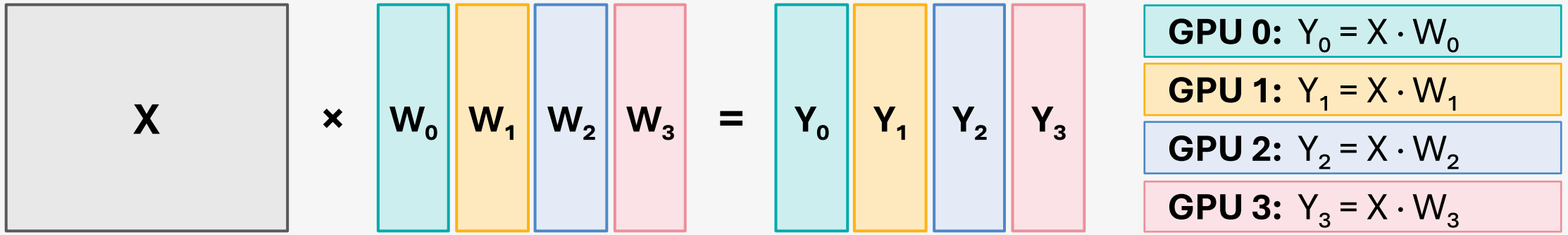
$$\text{Weights per GPU} = 2N \div G$$

70B at G = 4 GPUs → 35 GB each

Comm per GPU: 2 × 1.5 × 2bh bytes per layer

Column-parallel linear: split W by columns, so each GPU owns whole output columns

Each GPU holds a **vertical slice of W** and a full copy of X, so it computes its own output columns with no communication.



b tokens, h hidden, f output width

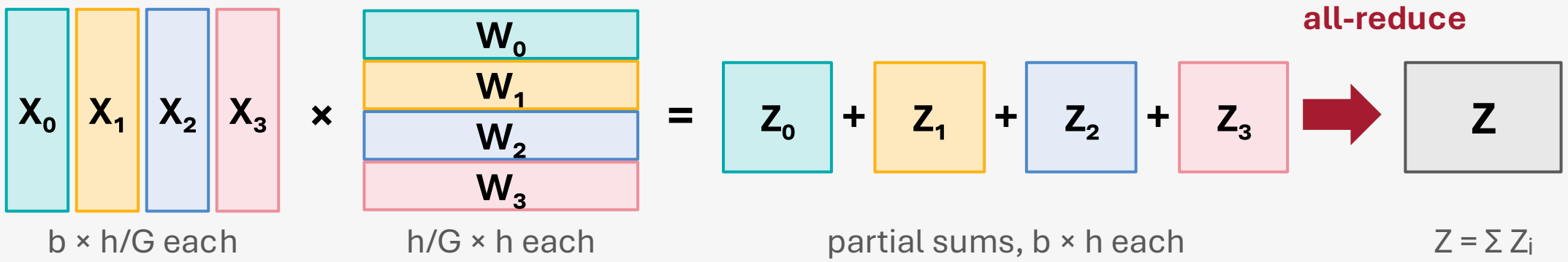
Communication

None inside the layer. Add an all-gather only if the next step needs the full Y.

Shoeybi et al., Megatron-LM, arXiv 1909.08053, 2019

Row-parallel linear: split W by rows, so each GPU holds a partial sum to all-reduce

Each GPU holds a **horizontal slice of W** and the matching columns of X. Its product is full size but only part of the answer.



b tokens, h hidden size; GPU i computes $Z_i = X_i \cdot W_i$

Communication

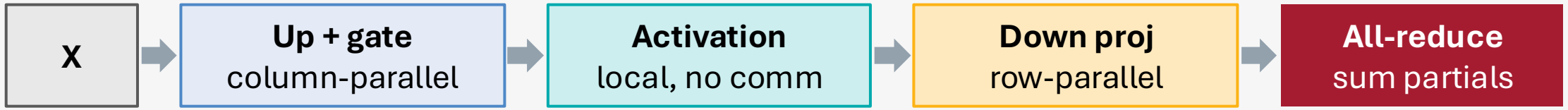
One all-reduce of the b × h output (2bh bytes in bf16); each GPU sends 2(G − 1)/G × 2bh = 3bh at G = 4

Shoeybi et al., Megatron-LM, arXiv 1909.08053, 2019

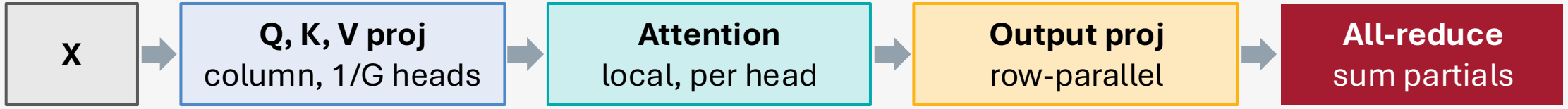
Megatron TP block: column-parallel then row-parallel, one all-reduce per block

The column-parallel layer's output shards are **exactly the input** the row-parallel layer needs, so GPUs only talk once, at the end.

MLP



Attention



Shoeybi et al., Megatron-LM, arXiv 1909.08053, 2019

Pipeline parallelism: split the layers

- Layers are cut into consecutive stages
- Each GPU owns a contiguous block of layers
- Micro-batches keep every stage busy
- Tolerates slower links, so it can span nodes

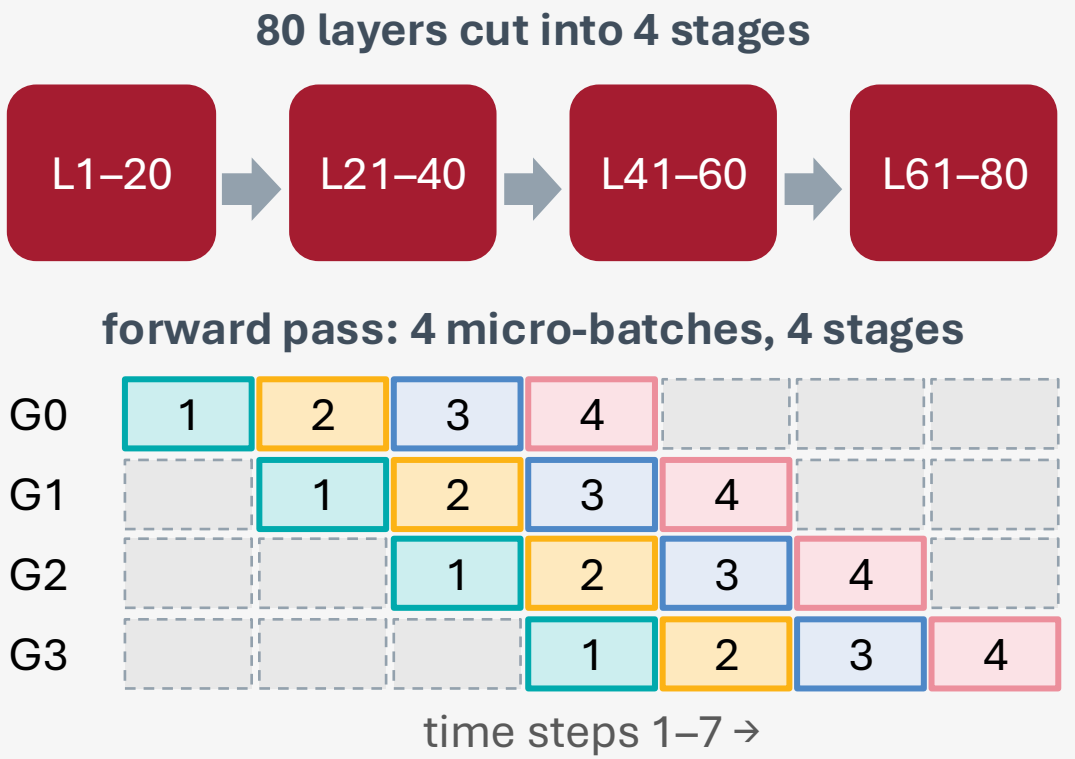
Back-of-envelope

$$\text{Bubble} = (G-1) \div (m+G-1)$$

G = GPUs (stages), m = micro-batches

G = 4: m = 4 → 43%, m = 16 → 16%

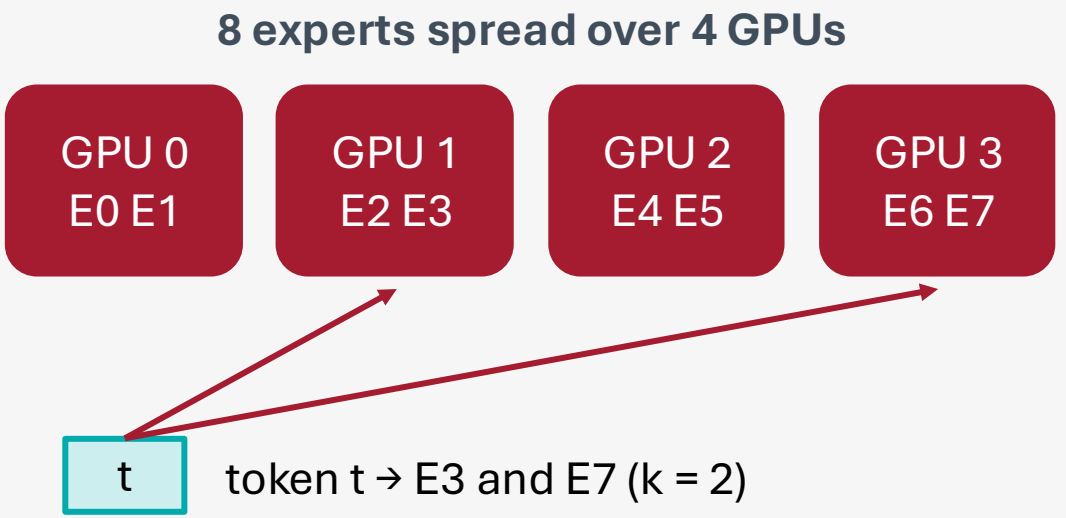
Comm: one send of 2bh bytes per stage hop



Grey = bubble: 12 of 28 slots idle = 3/7 ≈ 43%

Expert parallelism: split the experts

- An MoE layer holds many experts; each token uses a few
- Experts are spread across GPUs
- An all-to-all ships tokens to their experts
- Uneven routing leaves some GPUs idle



Back-of-envelope

$$\text{Load per GPU} \approx k \times b \div G$$

top-2, 1,024 tokens, 4 GPUs → 512 each

Comm per GPU: 2 × 0.75 × 2kbh bytes per layer

1 Dispatch: send t to its k experts

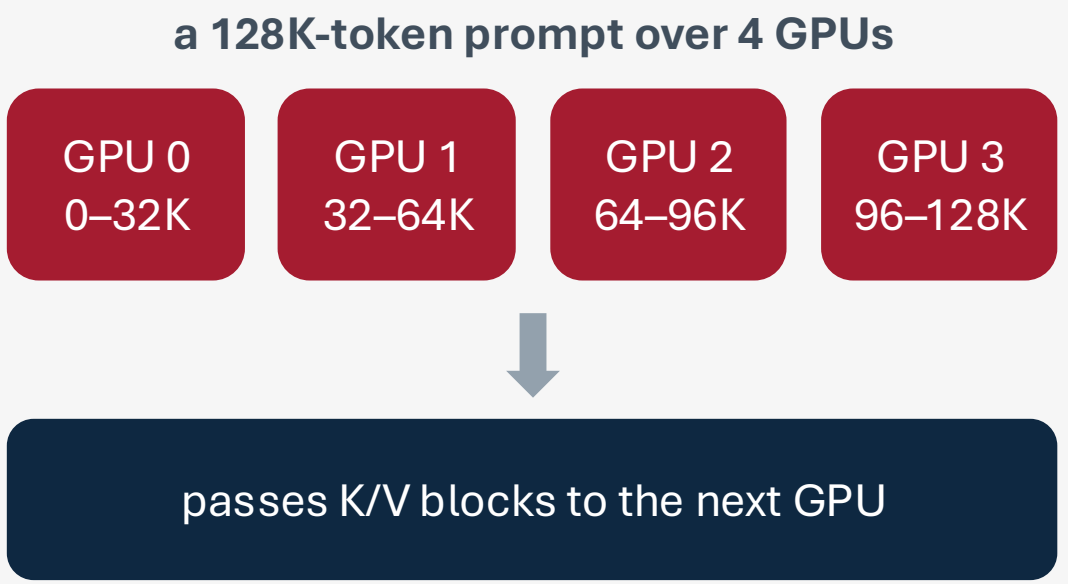
2 Compute: each expert runs locally

3 Combine: send results back, sum

All GPUs do this at once: all-to-all

Context parallelism: split the tokens

- Long prompts overflow one GPU
- Each GPU holds a block of tokens
- Ring attention passes K/V blocks
- Helps prefill, not decode



Back-of-envelope

$$\text{Tokens per GPU} = s \div G$$

128K-token prompt, G = 4 → 32K tokens each

Comm: G − 1 K/V block passes per layer

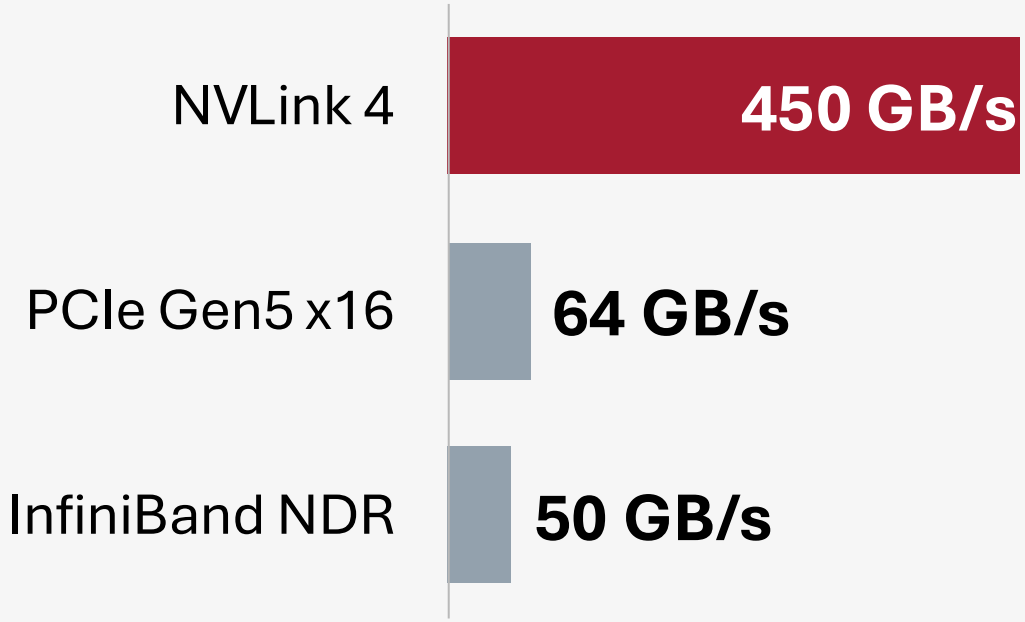
TP and EP move 240× more bytes per token than PP

Strategy	Collective	How often	Bytes / token / GPU
Data	none	never	0
Tensor (G = 4)	all-reduce	2 per layer	3.9 MB
Pipeline (G = 4)	send / recv	1 per stage hop	16 KB
Expert (G = 4, top-2)	all-to-all	2 per MoE layer	≈ 3.9 MB

70B-shape model: h = 8,192, 80 layers, bf16, decoding one token.
Ring all-reduce moves 2(G − 1)/G of the data; MoE in every layer, 3 of 4 experts remote.

TP and EP need NVLink

Link bandwidth per GPU, GB/s per direction



Cost of one transfer

t = bytes ÷ BW + hops × latency

Tensor traffic, 3.9 MB/token/GPU

NVLink ≈ 9 µs; InfiniBand ≈ 78 µs, plus overhead

Rule of thumb

TP inside a node; PP across nodes

H100 SXM: NVLink 4 (900 GB/s bidirectional), PCIe Gen5 x16, InfiniBand NDR 400 Gb/s

Check your understanding: parallelism

Q1. A 70B bf16 model on one node of 4 H200s (140 GB each). Which split first, and why?

Answer: TP = 4 over NVLink or DP2 + TP2 depends on context size.

Q2. You add a second node. What parallelism to use?

Answer: PP across nodes: one 2h-byte hop per token, while TP all-reduces every layer, about 240× more traffic.

Q3. Why can DP attention + EP outperform TP on long context but underperform on short ones?

Answer: Long: TP replicates KV when KV heads < ranks, while DP supports larger batches. Short: all-to-all overhead dominates.

Parallelism: what to take away

Data parallelism scales throughput

G replicas serve G× the requests, but every replica must hold the whole model

Tensor parallelism cuts per-token latency

Each GPU reads 1/G of the weights, at the cost of all-reduces every layer; keep it inside an NVLink node

Pipeline and sequence parallelism add capacity

Pipelines span nodes with light traffic but add bubbles; sequence parallelism splits very long prompts

Expert parallelism serves MoE models

Experts spread across GPUs; all-to-all traffic and expert load imbalance are the costs

Next class: speculative decoding

Today in brief

Measurement: TTFT and TBT percentiles, goodput

Batching: continuous, selective

Scheduling: request scheduling, chunked prefill, disaggregation

Memory management: paged, shared KV cache

Parallelism: DP, TP, PP, EP, SP

Optional reading

Optional reading: the papers behind each section

Evaluation

Agrawal et al., On Evaluating Performance of LLM Inference Systems, arXiv 2025

Batching and scheduling

Yu et al., Orca, OSDI 2022

Agrawal et al., Sarathi-Serve, OSDI 2024

Sheng et al., Fairness in Serving LLMs, OSDI 2024

PD disaggregation

Zhong et al., DistServe, OSDI 2024

KV-cache memory

Kwon et al., PagedAttention and vLLM, SOSP 2023

Zheng et al., SGLang, NeurIPS 2024

Parallelism

Shoeybi et al., Megatron-LM, arXiv 2019

Korthikanti et al., Reducing Activation Recomputation, MLSys 2023

Thakur, Rabenseifner and Gropp, Collective Operations in MPICH, IJHPCA 2005