# Model Inference Systems

University of Edinburgh

Speaker: Luo Mai

# Overview: Follow the Bottleneck

Each technique in this lecture fixes one bottleneck — and exposes the next one.

### 1 · Scheduling

How do we keep the GPU busy? Batching and continuous batching.

### 2 · Memory

How do we fit more requests? Paged KV cache and prefix reuse.

### 3 · Prefill vs decode

How do we serve both phases well? Chunked prefill and disaggregation.

### 4 · Speculation

How do we get more than one token per forward pass?

### 5 · Multi-GPU

What if one GPU is not enough? TP, PP, DP and EP.

### 6 · Kernels

CUDA graphs, fusion, FlashAttention and quantization.

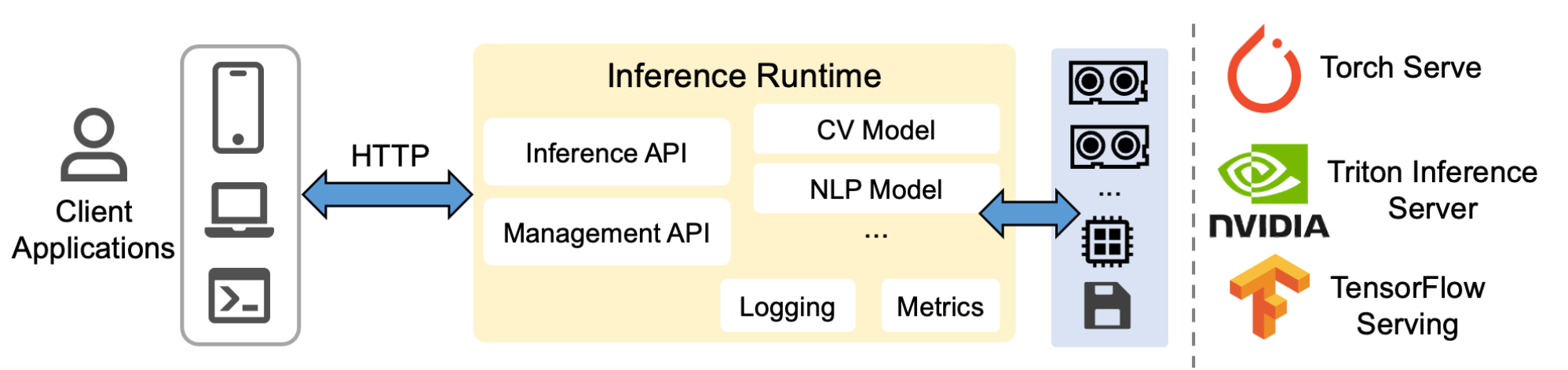
### 7 · Cluster scale

New workloads, routing, autoscaling and many models.

### Recap

Which technique helps which metric — and what it costs.

# Background: Model Inference Pre-ChatGPT




- Saved models need a dedicated runtime environment — the system must host the model, listen for requests, perform inference, and return responses
- Classic inference systems serve multiple heterogeneous models at once (CV classifier, BERT-based NLP, speech, etc.)

# Background: LLM Inference Becomes Dominant

- In the past: **small-to-medium models** with throughput-oriented batching and ML compilers
- Today: **LLM inference accounts for a large share** of production workloads
- A wave of **LLM-specialized inference engines** has emerged — and, above them, **cluster-level orchestration**

**ML compilers (past focus)**









**LLM serving engines (today)**




NVIDIA TensorRT LLM







**Cluster layer (2025 onwards)**

- NVIDIA Dynamo
- llm-d
- Mooncake (KV store)

# Background: Autoregressive Generation

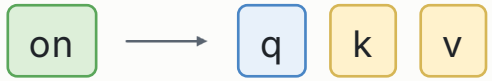



- LLMs generate text one token at a time — each new token depends on **all** previous tokens
- This is fundamentally sequential: the output tokens of one request cannot be computed in parallel

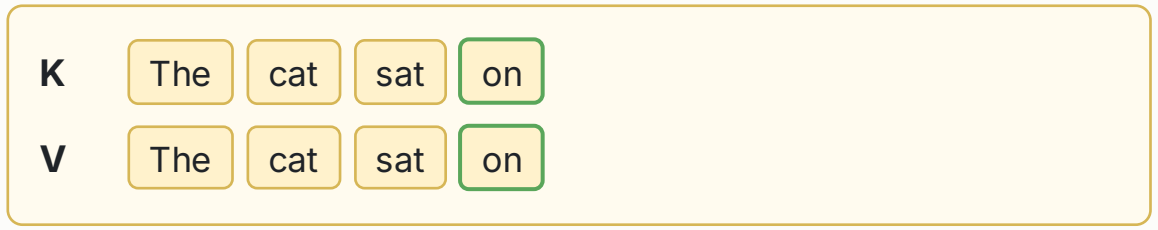
# Background: Attention and KV Cache

The cache already holds “The cat sat”. The new input token is “on”.

**1. Compute q, k, v for the new token only**




**2. Append k, v to the KV cache**




**3. q attends over all cached K, V → next token “the”**

### Without KV cache

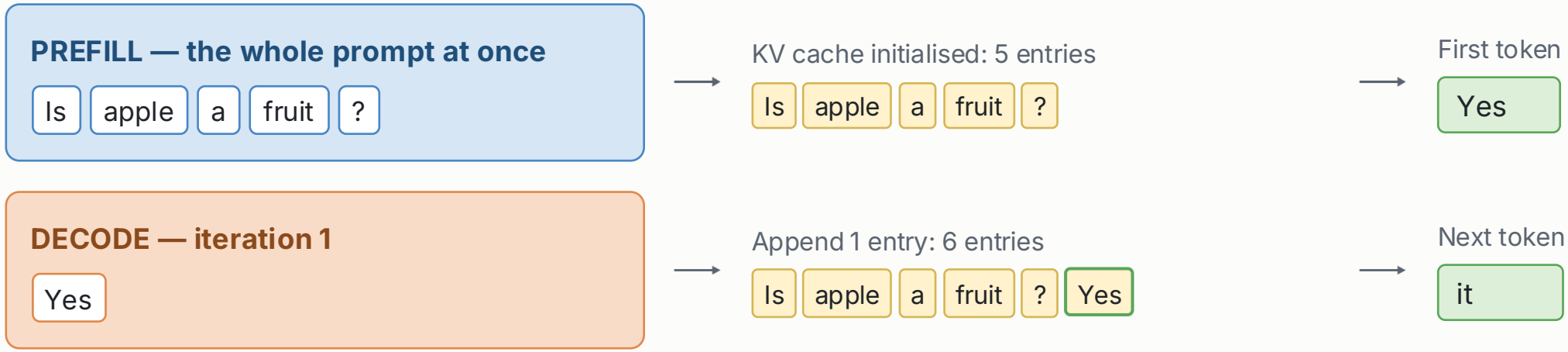
Every step recomputes K and V for all t tokens so far. Generating n tokens costs $O(n^2)$ K/V projections in total.

### With KV cache

Each step computes K and V only for the new token and appends them: O(n) in total.

But attention still **reads all t cached entries** every step, so memory traffic grows with the context length.

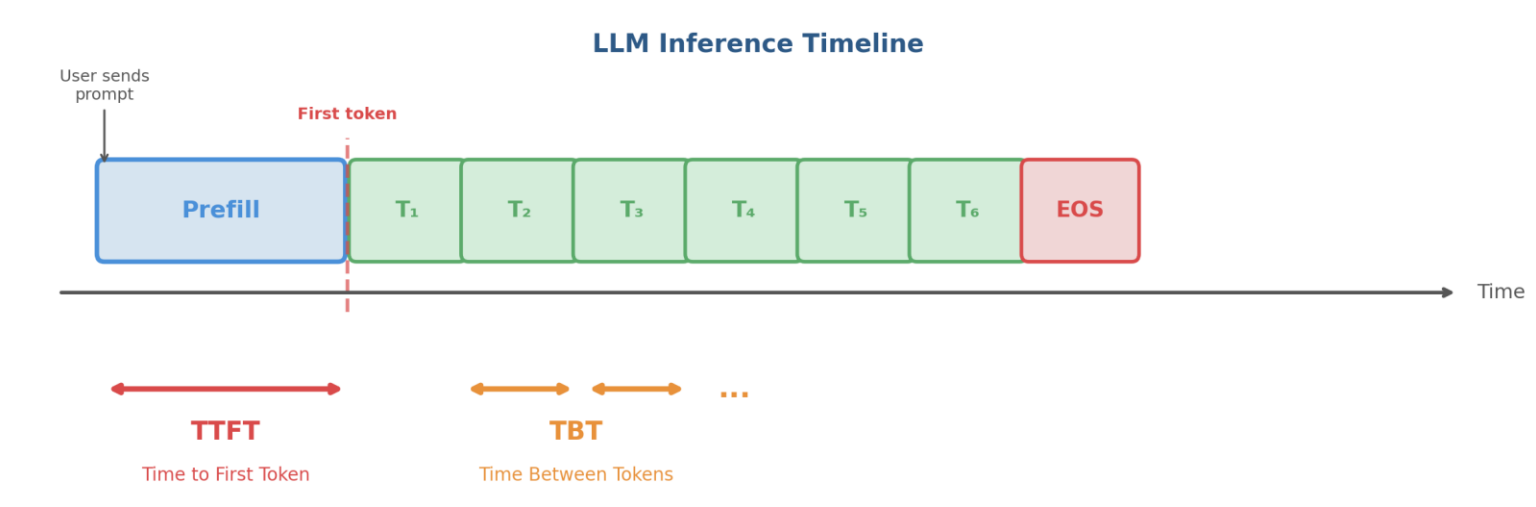
# Background: Prefill and Decode Phases




... and so on: each decode iteration feeds back the last token and grows the cache by one entry.

- **Prefill**: process the entire prompt in one forward pass (parallel), fill the KV cache, and produce the first output token
- **Decode**: generate the remaining tokens one by one, appending one K/V entry per step (sequential)

# Background: Key Latency Metrics (TTFT & TBT)




- **TTFT (Time to First Token)**: time from the request's arrival to the first output token — **queueing delay + prefill**
- **TBT (Time Between Tokens)**: latency between consecutive output tokens, i.e. one decode iteration
- **Key takeaway**: optimising inference = reducing TTFT (less queueing, faster prefill) and/or TBT (faster decoding)

# Background: SLOs and Goodput

| Metric | What it measures | Why it matters |
|---|---|---|
| TTFT | Arrival → first token (queueing + prefill) | How responsive it feels |
| TBT / TPOT | Gap between output tokens; TPOT = average gap over one request | Smooth streaming, faster than people read |
| End-to-end latency | Arrival → last token ≈ TTFT + TPOT × (output tokens − 1) | Agents and batch jobs wait for the full answer |
| Throughput | Tokens (or requests) per second, over all users | Cost per token |
| Goodput | Throughput that counts only requests meeting their SLOs | What a service really optimises |

- An SLO (service-level objective) is a latency target, e.g. “P99 TTFT under X ms”. Report **tail latency** (P90 / P99), not only averages — one slow call can stall a whole agent pipeline.

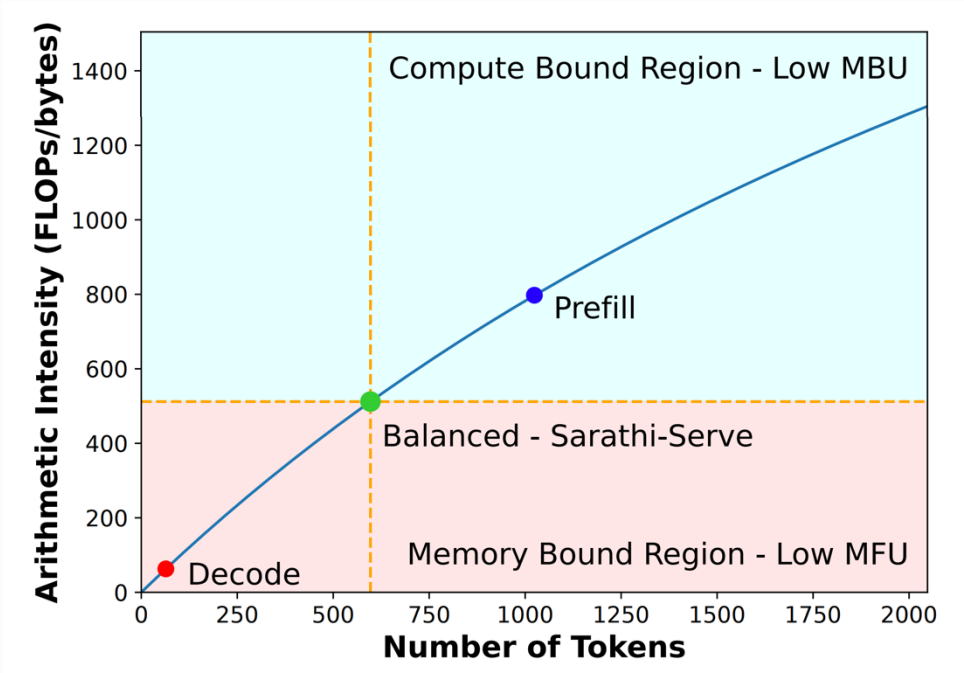
Goodput: Zhong et al., DistServe, OSDI’24

# Background: Training vs. Inference

| Aspect | Training | Inference |
|---|---|---|
| Workload | Fixed dataset, controlled | Variable requests, unpredictable |
| Execution | Forward + backward passes | Forward only, autoregressive |
| State | Gradients, optimizer state | KV cache (grows with sequence) |
| Objective | Throughput (samples/sec) | Latency (TTFT, TBT) + throughput, i.e. goodput under SLOs |
| Cost metric | GPU-hours per training run | Cost per million tokens served |

Training runs for weeks on a fixed job; inference runs forever on traffic you do not control. Most serving techniques exist to handle that unpredictability.

# Background: Prefill vs Decode Characteristics




| Phase | Bottleneck | Characteristics |
|---|---|---|
| Prefill | Compute-bound | Many tokens in parallel; latency grows with input length |
| Decode | Memory-bound | One token per request per step; each step reads all weights (shared by the batch) plus each request's own KV cache |

This difference drives most optimization techniques we'll discuss.

Balanced point: Agrawal et al., Sarathi-Serve, OSDI'24

# Background: How Memory-Bound Is Decode?

**≈ 295**

**FLOP/byte: H100 ridge point**

989 TFLOPS (BF16, dense) ÷ 3.35 TB/s. Below this a kernel is memory-bound.

**≈ 1**

**FLOP/byte: decode, batch 1**

Each 2-byte BF16 weight is read once and used for one multiply-add (2 FLOPs).

**≈ 0.3%**

**of peak compute used**

At batch 1 (≈ 1 / 295). Batch 8 → ≈ 2.7%. Batch ≈ 300 → compute-bound.

- For the weight matrix multiplies, arithmetic intensity grows roughly **linearly with batch size** → batching is the main lever (next section)
- Attention over the KV cache does **not** improve with batching: every request reads its own K and V
- So with long contexts, decode stays memory-bound even at large batch sizes

H100 SXM datasheet figures; estimates ignore activations and KV reads

# Background: How Big Is the KV Cache?

```
KV bytes per token = 2 (K and V) × layers × KV heads × head dim × bytes per value
Llama-3-8B, BF16: 2 × 32 × 8 × 128 × 2 B = 131,072 B = 128 KiB per token
```

**128 KiB**

per token

**1 GiB**

per request with an 8K-token context

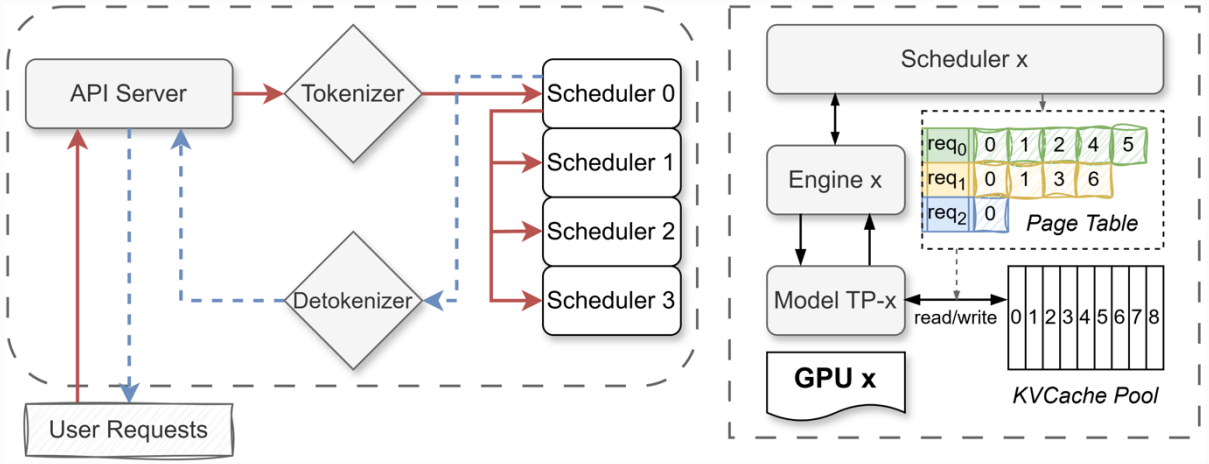
**≤ 60**

such requests on an 80 GB GPU, after ~16 GB of weights

- KV memory, not compute, usually limits how many requests run at once
- **Grouped-query attention (GQA)** shares each K/V head across several query heads: Llama-3-8B has 32 query heads but only 8 KV heads → a 4× smaller cache. MLA (later) compresses it further.

Model configuration: Meta Llama 3 (2024)

# Background: System Architecture Overview




| Component | Function |
|---|---|
| API Server | Receives user requests, handles networking, returns (streams) responses |
| Tokenizer / Detokenizer | Converts text ↔ token IDs |
| Scheduler | Queues and organizes requests, decides which sequences run in the next iteration |
| KV Cache Manager | Allocates and manages KV cache memory (page table → KV cache pool) |
| Model Engine | Executes forward passes on the GPU(s) |

# Scheduling: Why Batching Matters

### Single request (batch = 1)

Read weights: **14 GB** (7B model, FP16)

Compute: **14 GFLOP** (1 token)

Step time ≈ 14 GB ÷ 3.35 TB/s ≈ **4.2 ms**

Compute utilisation ≈ **0.3%**

### Batched requests (batch = 8)

Read weights: **14 GB** (same!)

Compute: **112 GFLOP** (8 tokens)

Step time ≈ **4.2 ms** (still memory-bound)

Compute utilisation ≈ **2.7%**

- Decode reads the **full model weights once per step**, whatever the batch size
- Batching amortises that read: **8× the tokens for about the same step time** — until the batch becomes compute-bound or KV cache reads dominate
- Goal: increase arithmetic intensity → better GPU utilisation

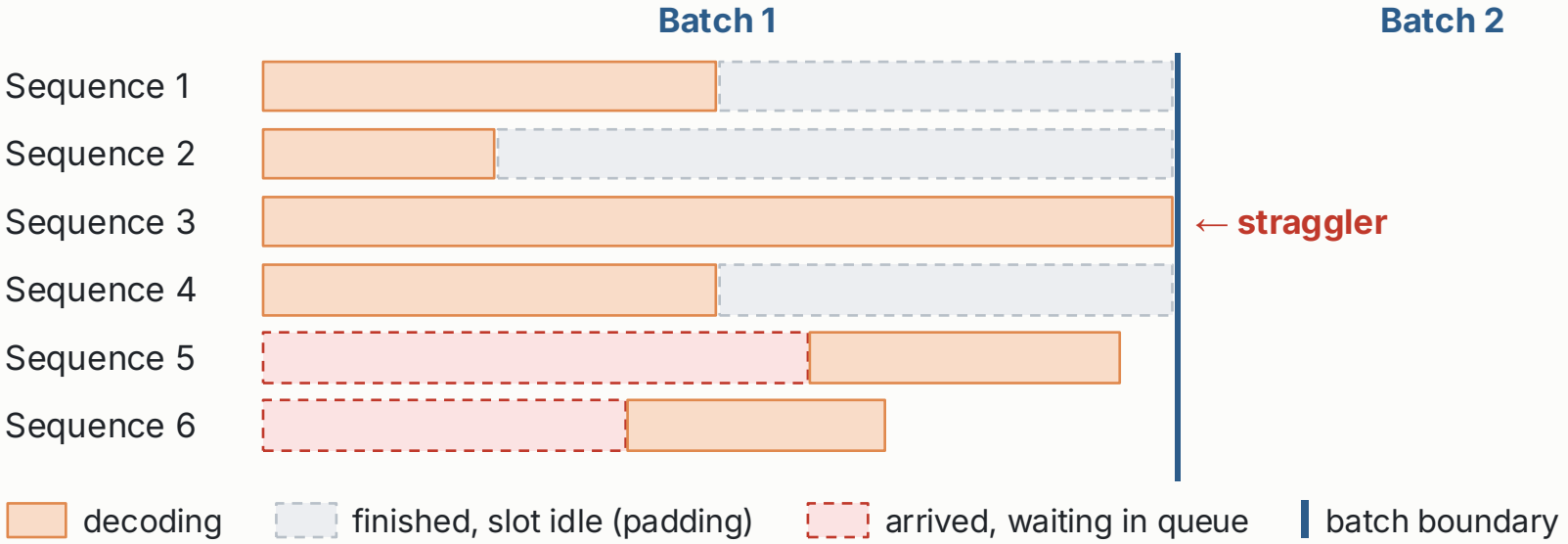
Illustrative numbers for an H100 SXM; ignores KV cache and activation traffic

# Scheduling: Batching Strategies Overview

| | Static batching | Dynamic batching | Continuous batching |
|---|---|---|---|
| Batch size | Fixed | Flexible | Changes every iteration |
| Batch starts | Wait until full | When full or on timeout | Any iteration |
| Padding | Pad to max length | Pad to max length | None (variable-length kernels) |
| Requests leave | All together | All together | As soon as each finishes |
| Scheduling unit | Whole batch | Whole batch | One iteration (one token) |

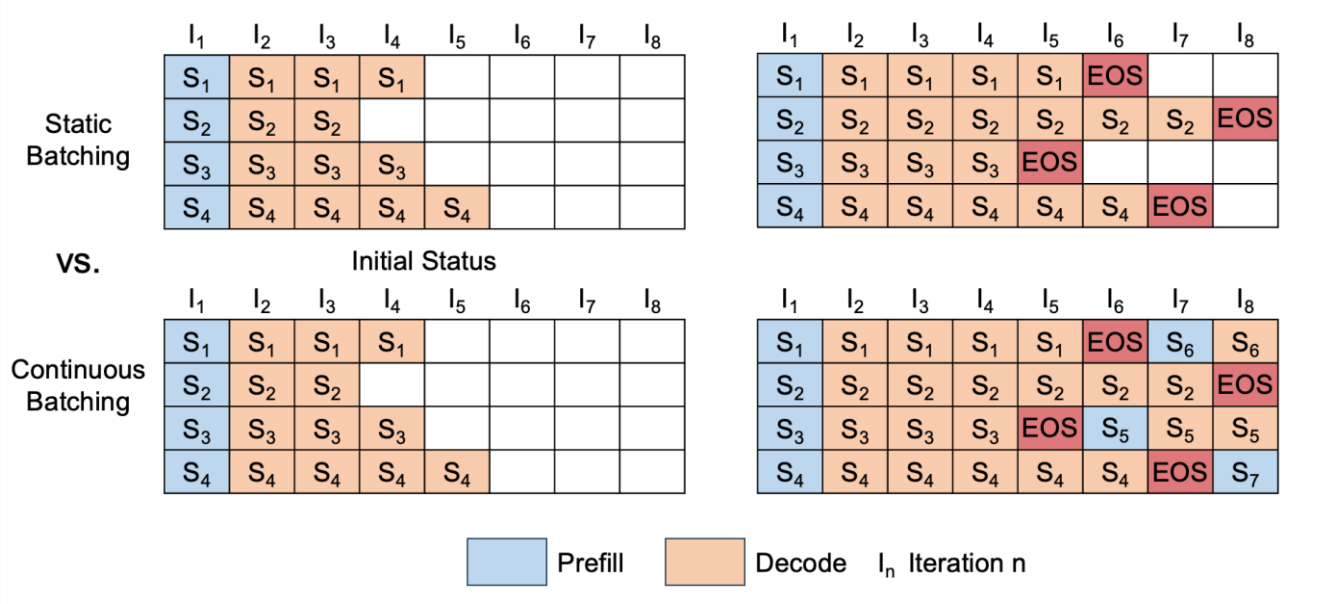
- **Static / dynamic**: the entire batch waits for its slowest sequence (the **straggler** problem)
- **Continuous**: sequences finish independently, and new ones join immediately

# Scheduling: Problems with Static/Dynamic Batching




- **Batch boundary**: new requests must wait until the current batch completes
- **Stragglers**: the entire batch waits for the slowest request (Sequence 3)
- **No mid-decode admission**: new requests cannot join during the long decoding phase
- **Bad user experience**: results are returned only when all requests in the batch finish

# Scheduling: Continuous Batching




- The scheduler runs **one forward pass (iteration)** at a time
- After each iteration: finished requests leave, new requests join
- **All modern systems (vLLM, SGLang, TensorRT-LLM) use continuous batching**

Iteration-level scheduling: Yu et al., Orca, OSDI'22

# The Next Bottleneck: Memory

**Continuous batching solved**

- ✓ No waiting for stragglers
- ✓ High GPU utilisation
- ✓ Low queueing latency

→

**… but created a new problem**

Many concurrent requests

↓

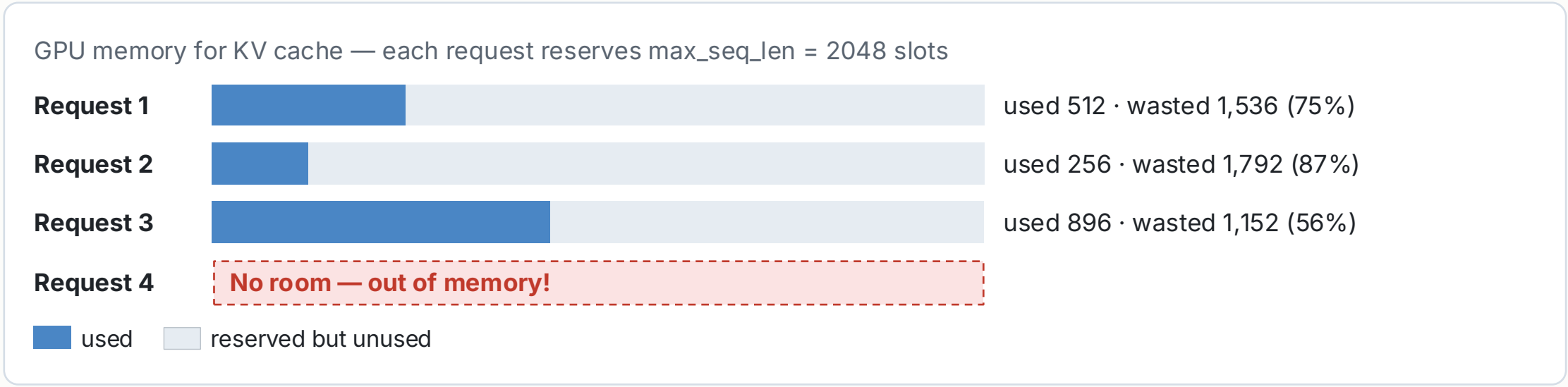
Each needs its own KV cache

↓

**GPU memory exhausted!**

**Question: how do we manage KV cache memory efficiently to maximise concurrency?**

# Memory Management: Naïve KV Cache Allocation




- **Naïve approach**: reserve KV slots for the maximum sequence length of every request
- **Internal fragmentation**: most requests finish early, so most reserved space is never used
- **External fragmentation**: reservations of different sizes leave unusable gaps between them
- **Result**: memory runs out after 3 requests, although actual usage is low. The vLLM authors found earlier systems used only **20–38%** of KV cache memory for real token states.

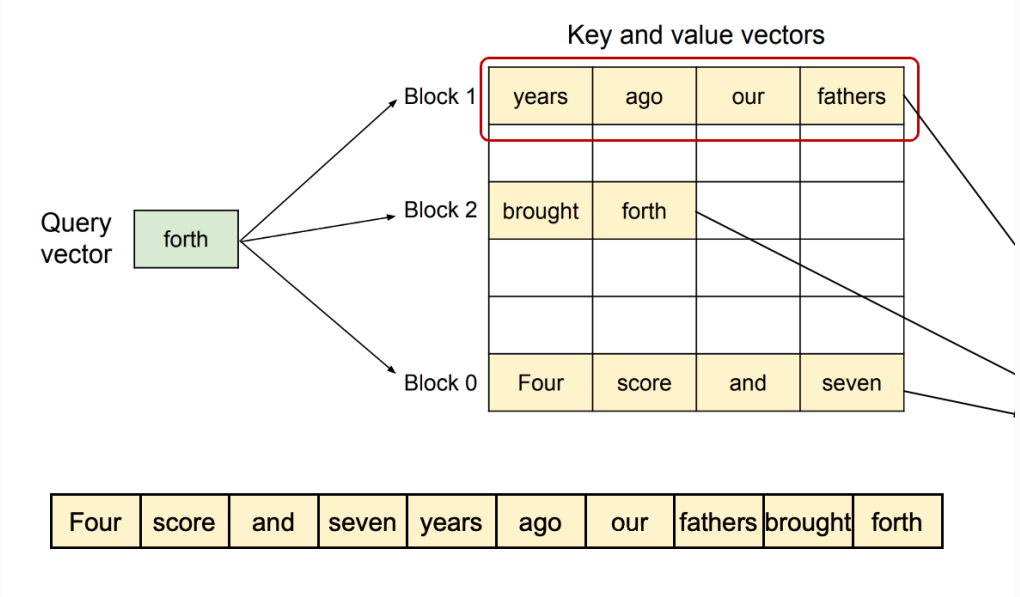
# Memory Management: PagedAttention

Inspired by **paging and virtual memory** in operating systems:

- KV vectors are stored in **fixed-size, non-contiguous blocks** (e.g. 16 tokens per block)
- The attention kernel walks the request's blocks: for each block, compute the scores $\mathbf{q}\cdot\mathbf{K}^T$, then add the score-weighted sum of **V** (softmax is combined across blocks)

**Benefits:**

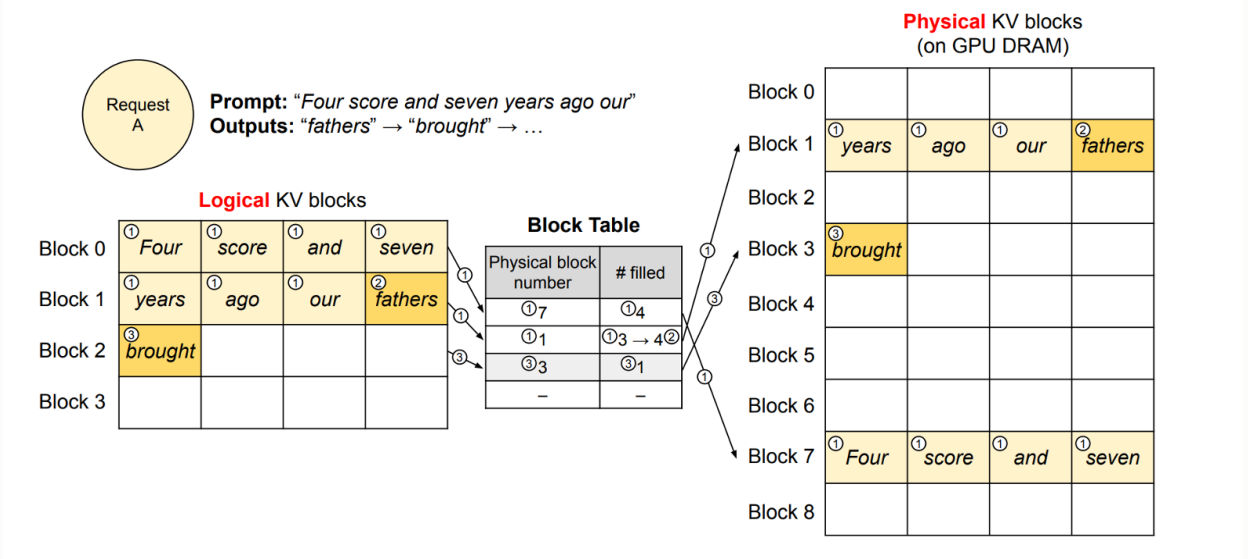
- **No external fragmentation**; internal waste only in each request's last, partly filled block (less than one block)
- Blocks allocated on demand → near-full memory utilisation
- Higher request concurrency




The query for "forth" attends over three blocks (block size 4) that are not contiguous in memory.

Figure: Kwon et al., vLLM, SOSP'23

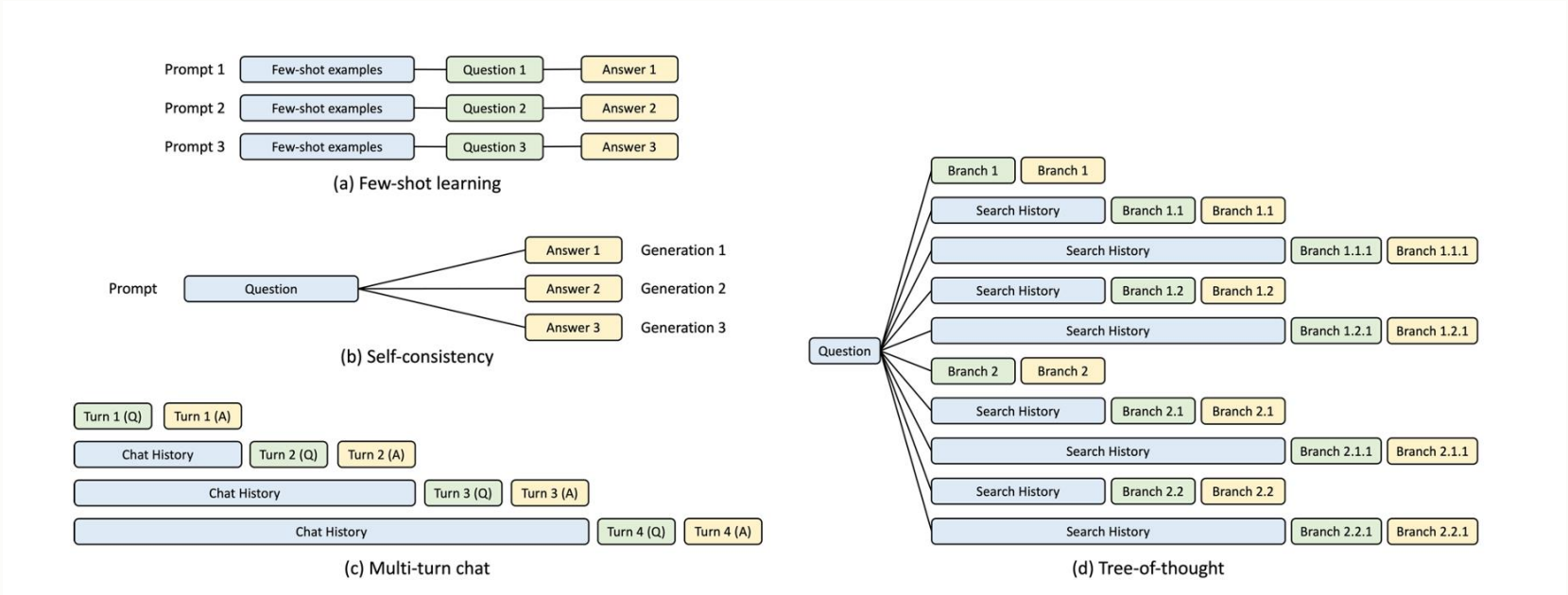
# Memory Management: Block Table




- A request's KV cache = a series of **logical KV blocks**
- The **block table** maps logical → physical blocks for each request (like a page table)
- Neighbouring logical blocks need not be contiguous in physical memory
- Physical blocks can be **shared** by several requests (reference counts + copy-on-write) — the basis for prefix sharing and parallel sampling

Figure: Kwon et al., vLLM, SOSP'23

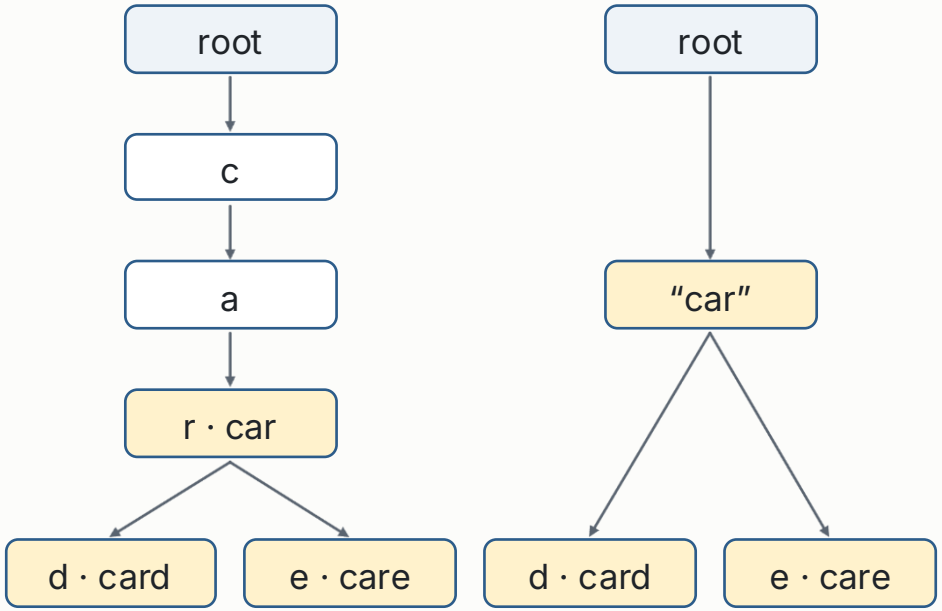
# Memory Management: KV Cache Reuse




- Different prompts with the **same prefix** can share KV cache (system prompts, few-shot examples, chat history)
- **Dual benefit**: saves memory AND skips the redundant prefill computation for the shared prefix
- Multi-turn chat and agents re-send the whole history on every turn, so reuse can skip most of the prefill

Figure: Zheng et al., SGLang, NeurIPS'24

# Memory Management: RadixAttention




Trie: one element per edge

Radix tree: compressed

- A **radix tree** is a compressed trie: one edge holds a **sequence** of elements, not just one
- In SGLang the elements are **token IDs**, and each node points to the KV cache pages of its tokens
- A new request walks the tree to find its **longest cached prefix**, then prefills only the rest
- Eviction: **LRU over leaf nodes**; nodes used by running requests are protected by reference counts
- **Cache-aware scheduling**: run requests with longer cached prefixes first to raise the hit rate
- vLLM’s automatic prefix caching gets a similar effect by **hashing full KV blocks**

# The Next Bottleneck: Prefill-Decode Interference

**Memory management solved**

- ✓ Near-zero fragmentation
- ✓ High memory utilisation
- ✓ KV cache reuse across requests

→

**… but a new problem appears**

Prefill: compute-bound

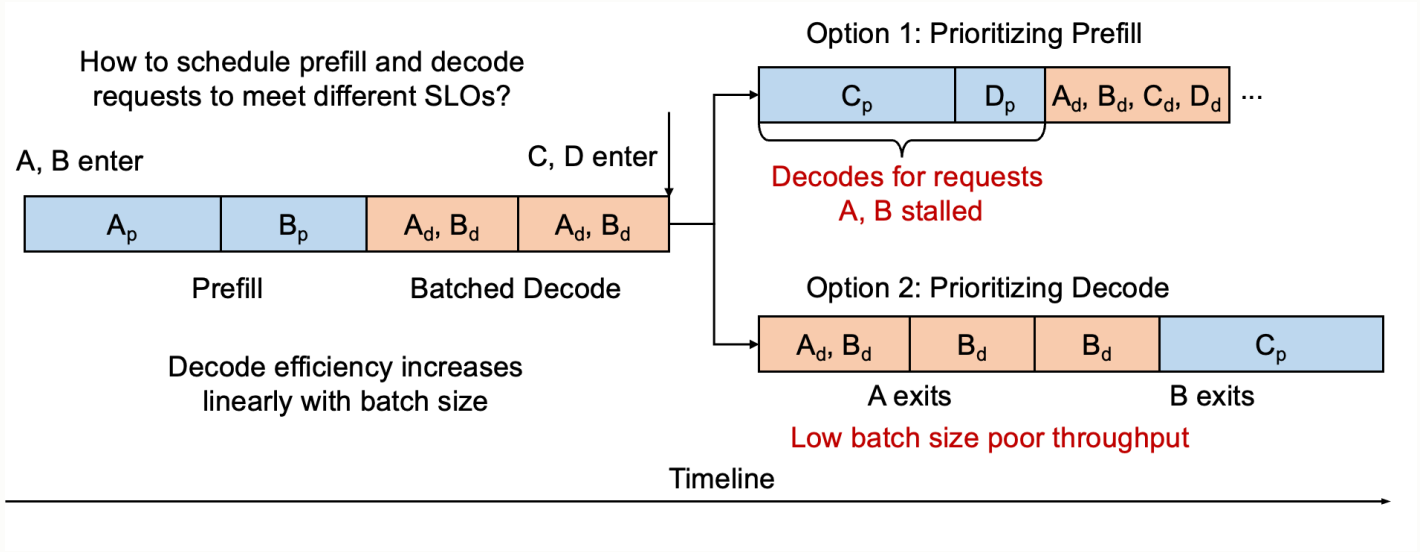
Decode: memory-bound

↓

**They compete for the same GPU — how do we schedule both?**

**Question: how do we balance prefill and decode to optimise both TTFT and TBT?**

# Prefill-Decode Optimization: Throughput-Latency Tradeoff




- **Prioritise prefill**: new requests start sooner (good TTFT), but decodes of running requests stall (bad TBT)
- **Prioritise decode**: smooth TBT, but small batches → poor throughput, and new requests wait (bad TTFT)

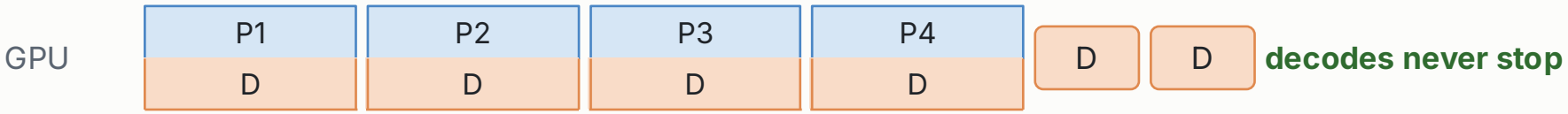
Further reading: Agrawal et al., Sarathi-Serve, OSDI'24

# Prefill-Decode Optimization: Chunked Prefill

**Without chunked prefill — a long prefill blocks all decodes**

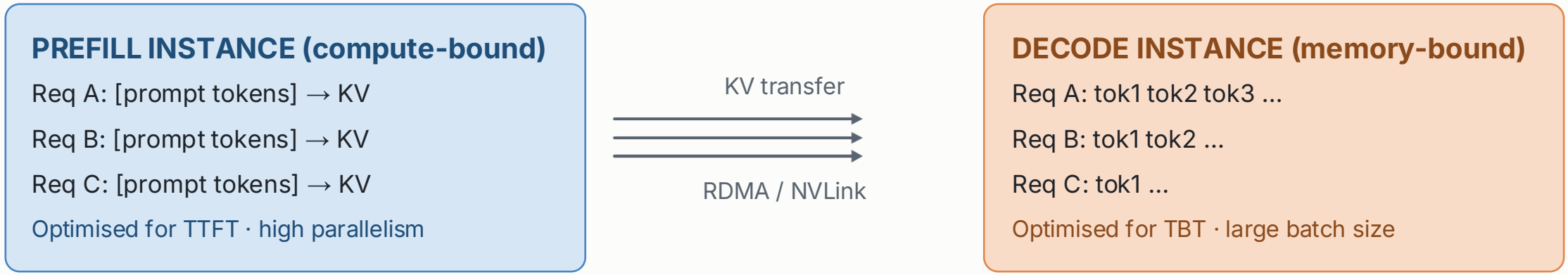



**With chunked prefill — every iteration has a token budget: decodes first, a prefill chunk fills the rest**




- Split a long prompt into chunks, and piggyback each chunk onto an ongoing decode batch
- Decode is memory-bound → the spare compute in each decode iteration runs the prefill chunk almost for free
- **Main win: bounded TBT** (no generation stalls) while keeping throughput high
- **Cost**: a long prompt's TTFT grows a little — its prefill is spread over several iterations, and each chunk re-reads the KV of earlier chunks. The chunk size sets this trade-off.

# Prefill-Decode Optimization: Disaggregation




- Separate instances let each phase use its own hardware, parallelism and batch size — **no interference**
- **Cost**: the KV cache must move between instances; this needs high-bandwidth links, and the transfer can be overlapped layer by layer with compute
- **Not always worth it**: for short prompts or small deployments, the transfer and extra instances may cost more than they save
- Research: DistServe (OSDI'24), Splitwise (ISCA'24), Mooncake (FAST'25). Now supported in vLLM, SGLang, NVIDIA Dynamo and llm-d.

# Prefill-Decode Optimization: Which One When?

| | **Chunked prefill** | **Disaggregation** |
|---|---|---|
| Idea | Mix prefill chunks into decode batches on the same GPUs | Run prefill and decode on separate GPU pools |
| Main benefit | Bounded TBT and high utilisation, with no extra data movement | Each phase tuned on its own; no interference at all |
| Main cost | Interference is reduced, not removed; slightly higher TTFT for long prompts | KV transfer bandwidth; two pools to size and keep balanced |
| Good fit | Single node, moderate load, mixed prompt lengths | Large scale, long prompts, strict TTFT and TBT targets |

The two compose: large deployments often disaggregate, and still use chunked prefill inside the prefill pool to bound per-iteration work.

# Speculative Decoding: Overview

**Standard decoding — 1 token per target forward pass**

pass 1 → tok1 | pass 2 → tok2 | pass 3 → tok3 | pass 4 → tok4 — 4 passes, 4 tokens

**Speculative decoding — draft proposes, target verifies in one pass**

| | | | | | | |
|---|---|---|---|---|---|---|
| Draft (fast), $\gamma = 4$ | tok1 | tok2 | tok3 | tok4 | | |
| Target: ONE pass | tok1 accept | tok2 accept | tok3 accept | ~~tok4~~ | tok4′ from target | |

**→ 3 accepted + 1 corrected = 4 tokens from 1 target pass ($\gamma + 1 = 5$ if all are accepted)**

- With acceptance rate $\alpha$, a pass yields on average **$(1 - \alpha^{(\gamma+1)}) / (1 - \alpha)$** tokens
- Pays off only if the draft is cheap and $\alpha$ is high — the draft passes are not free
- Gains are largest at **small batch sizes** (memory-bound decode); at large batches the extra verification compute eats the benefit

Leviathan et al., ICML'23; Chen et al., 2023

# Speculative Decoding: Draft-Target Method

The classic implementation uses two models:

| Model | Role |
|---|---|
| Draft model | Small and fast; proposes $\gamma$ candidate tokens |
| Target model | Large and accurate; checks all proposals in one forward pass |

**Process:**

1. The draft model generates $\gamma$ tokens, one after another (cheap)
2. The target model scores all $\gamma$ positions in a single forward pass
3. Accept draft tokens left to right with rejection sampling; stop at the first rejection
4. The target adds one token: a corrected token at the rejected position, or a bonus token if all $\gamma$ were accepted
5. Discard everything after the first rejection and continue from there

# Speculative Decoding: Rejection Sampling

| Token | P(draft) | P(target) | Accept prob. |
|---|---|---|---|
| “years” | 0.92 | 0.93 | 1.00 |
| “ago” | 0.87 | 0.80 | 0.92 |
| “your” | 0.52 | 0.41 | 0.79 |
| “mother” | 0.58 | 0.79 | 1.00 |

Each token is only tested if every earlier token was accepted. E.g. 0.92 = 0.80 ÷ 0.87.

If P(target) ≥ P(draft): **accept**

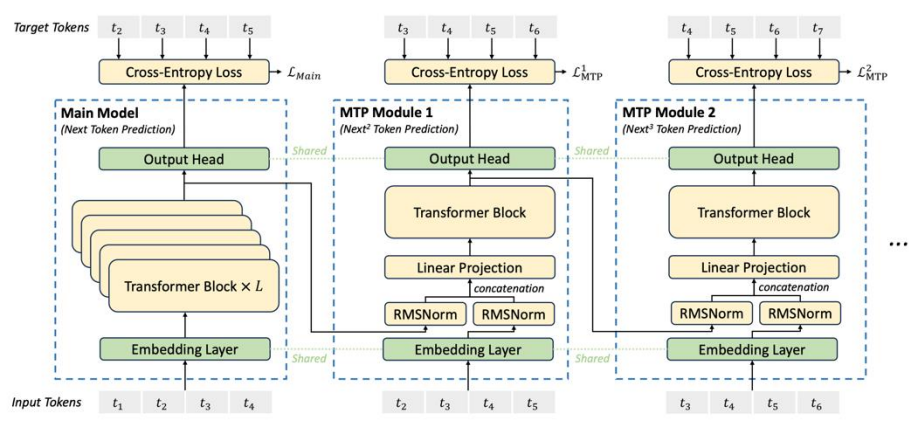
Otherwise: accept with probability **P(target) / P(draft)**

On rejection: sample a replacement from the **residual** distribution norm(max(0, p_target − p_draft)), and drop all later draft tokens

**Together, these rules make the output distribution exactly equal to the target model’s.**

Leviathan et al., ICML’23; Chen et al., 2023

# Speculative Decoding: Multi-Token Prediction




- DeepSeek-V3 adds **sequential MTP modules** after the main model. Module k combines the previous depth's hidden state with the embedding of the next input token, runs one Transformer block, and predicts one token further ahead
- This keeps the **causal chain** between predicted tokens — unlike Medusa, whose extra heads predict each position independently
- The embedding layer and output head are **shared** with the main model, so each module is small
- Trained as an extra objective; at inference the modules can be dropped, or reused as a **built-in drafter** — no separate draft model

Figure: DeepSeek-AI, DeepSeek-V3 Technical Report, 2024

# Speculative Decoding: Where Do Drafts Come From?

| Method | Drafter | How drafts are made | Notes |
|---|---|---|---|
| Draft model | Separate small LM, same tokenizer | Runs $\gamma$ autoregressive steps | Simple; one more model to host |
| Medusa | Extra heads on the target | One head per future position, in parallel; tree verification | Heads are independent → acceptance drops for later positions |
| EAGLE | Light autoregressive head on target features | Predicts next features and tokens step by step | High acceptance; widely supported in vLLM and SGLang |
| MTP | Modules trained with the model | Each module predicts one token further | Free if the model was trained with MTP |
| N-gram / prompt lookup | No model | Copy spans from the prompt or history | Strong for code edits, RAG and agent loops |

Cai et al., Medusa, ICML'24; Li et al., EAGLE, ICML'24; DeepSeek-V3 Technical Report, 2024

# Scaling Beyond a Single GPU

So far we assumed one GPU per model. But what if:

### Problem 1: the model does not fit

**DeepSeek-V3**: 671B parameters ≈ 700 GB in FP8 — far beyond any single GPU

Even a 70B model in BF16 (≈ 140 GB) fits on one B200, but leaves little room for KV cache

### Problem 2: one GPU is too slow

**Per-request speed** (tokens/s for one user) → split each step across GPUs

**Aggregate throughput** (tokens/s for all users) → add more replicas

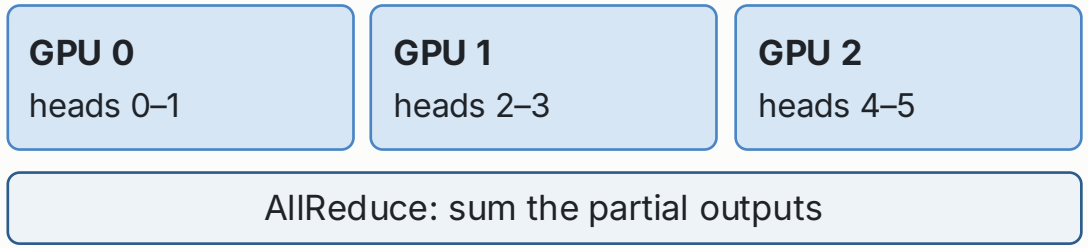
**Solution**: distribute the model and/or the workload across multiple GPUs. **Question**: how do we partition it?

HBM per GPU: H100 80 GB · H200 141 GB · B200 up to 192 GB · MI325X 256 GB · B300 288 GB

# Multi-GPU Serving: Tensor & Pipeline Parallelism (TP, PP)

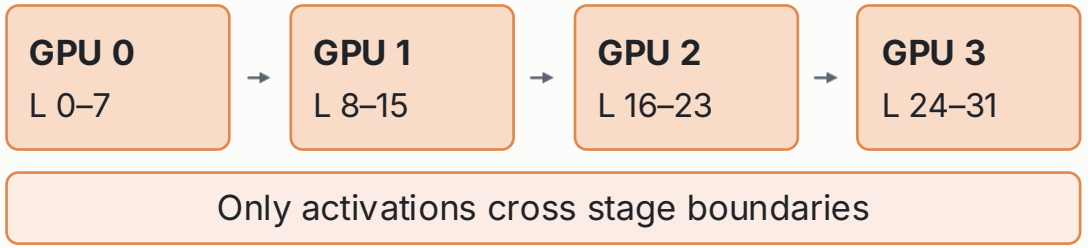
**TP: split each layer across GPUs**

Layer 1 (attention heads)




**PP: assign layer groups to GPUs**

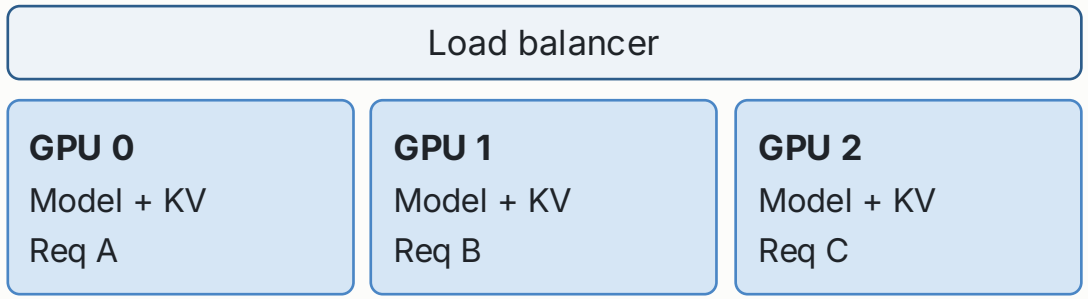
Request → stage 0 → stage 1 → … → output



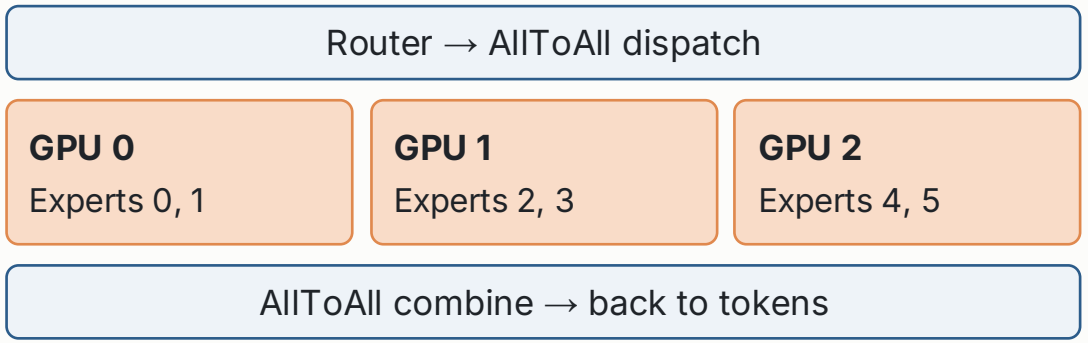

- **TP**: each GPU holds a slice of every layer (attention heads, MLP columns/rows). Two AllReduces per layer → lower per-token latency, but it needs a fast interconnect (NVLink), so usually stays within a node
- **PP**: little communication and works across nodes. But a request passes the stages one after another, so PP **adds** latency rather than reducing it. Bubbles stay small when many requests are in flight

# Multi-GPU Serving: Data & Expert Parallelism (DP, EP)

**DP: replicate the model, distribute requests**




**EP: distribute MoE experts across GPUs**




- **DP**: simple scaling, but every replica keeps a **full copy of the weights**. (The KV cache is per request, so it is not duplicated.)
- **EP**: efficient for MoE. Tokens go to the GPUs that host their experts (AllToAll dispatch) and results come back (AllToAll combine). Load imbalance across experts is the main challenge; libraries such as DeepEP optimise these AllToAlls

# Multi-GPU Serving: Composing Strategies

Example: DeepSeek-V3 hybrid parallelism, as reported by DeepSeek

**Attention layers** — Data parallel over requests (each group of GPUs owns its requests and their KV cache), with TP4 + sequence parallelism inside a group

↓ AllToAll dispatch · ↑ AllToAll combine

**MoE layers** — Expert parallelism: routed experts spread across all GPUs of the deployment unit

| Stage | Min. deployment unit | Attention | MoE |
|---|---|---|---|
| Prefill | 4 nodes, 32 GPUs | TP4 + SP, with DP8 | EP32 |
| Decode | 40 nodes, 320 GPUs | TP4 + SP, with DP80 | EP320 |

Prefill and decode are deployed separately, with different parallelism. The choice depends on: model architecture, hardware topology, latency vs throughput goals, and KV cache memory.

Source: DeepSeek-AI, DeepSeek-V3 Technical Report (2024), Inference and Deployment section

# Multi-GPU Serving: Case Study — DeepSeek MLA

**Multi-head attention: cache per token, per layer**

K, V head 1 | K, V head 2 | K, V head 3 | … | K, V head 128

Tens of thousands of values per token per layer

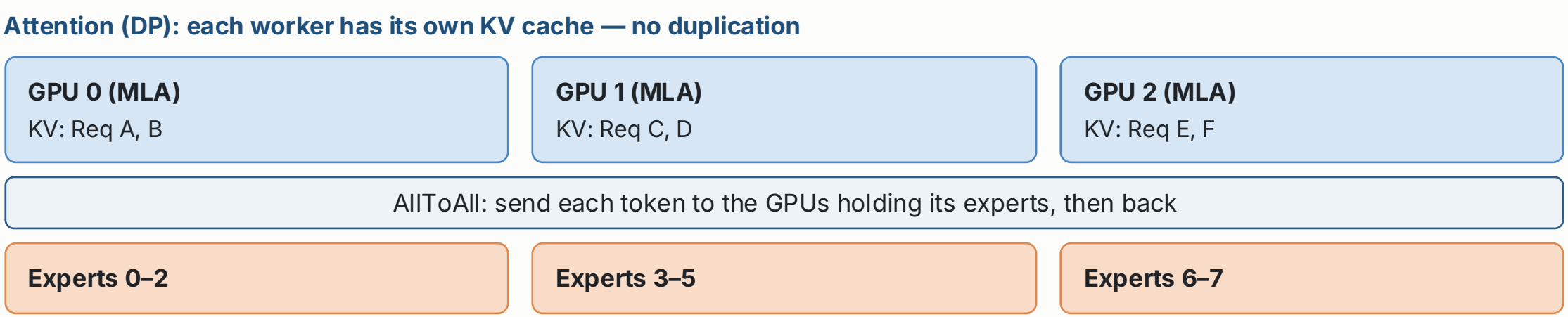
**MLA: one shared latent vector**

latent c (512) | RoPE key (64)

576 values per token per layer (DeepSeek-V3), shared by all heads

- Instead of caching full per-head K/V tensors, MLA caches a **low-dimensional latent vector** per token
- K/V for all heads are reconstructed from it at attention time (or the up-projections are absorbed into the matmuls)
- **Problem with TP + MLA**: TP splits attention by heads, but the latent vector is **shared by all heads**, so it cannot be split. Every TP rank must store the whole latent cache → the KV cache is duplicated TP times → smaller maximum decode batch

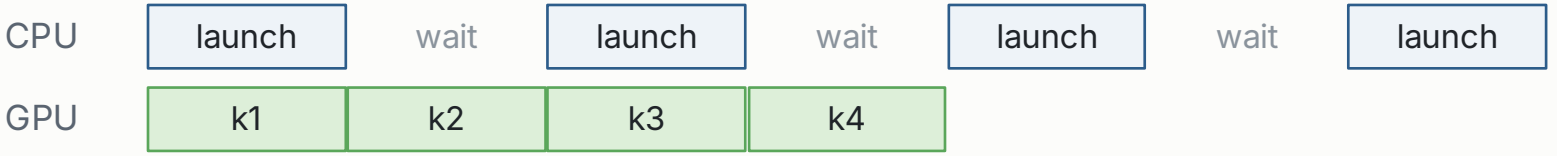
# Multi-GPU Serving: Case Study — DP-Attention




- **DP for attention**: each GPU owns its requests and their (MLA) KV cache → no duplication, much larger decode batches
- **EP for MoE**: GPUs exchange tokens only at the MoE layer boundaries
- The name “DP-Attention” comes from SGLang’s implementation; the same pattern is used in vLLM and in DeepSeek’s own deployment

# System Optimization: CUDA Graphs

**Without CUDA Graphs — one CPU launch per kernel**




Launch overhead (a few μs each) adds up across hundreds of small kernels; the GPU idles in between

**With CUDA Graphs — one launch replays the whole sequence**




kernels run back-to-back, no CPU in the loop

- **Capture once**: record the kernel sequence during warm-up
- **Replay many times**: a single cudaGraphLaunch triggers the entire sequence
- Biggest win in **decode**: each step runs many tiny kernels, so launch overhead can rival the GPU work itself

# System Optimization: CUDA Graphs Constraints

**Constraint**: a graph replays fixed kernels with fixed arguments, so it needs **static shapes** (fixed tensor sizes / batch size) and **fixed memory addresses**.

### Engineering solution

- Pre-capture graphs at startup for a set of batch sizes
- At runtime, pad the batch up to the nearest captured size
- Replay the matching pre-captured graph

```
captured sizes: [1, 2, 4, 8, 16, 32, 64, …]
batch = 5 → pad to 8 → replay graph_8
```

### In modern engines

**Piecewise CUDA graphs** (e.g. vLLM V1 with torch.compile): capture everything except attention, which runs outside the graph and handles variable sequence lengths

Engines capture many batch sizes, often up to a few hundred

Large prefill steps gain little: their kernels are long, so launch overhead is small

# System Optimization: Kernel Fusion

**Without fusion: 3 kernels**

Kernel 1: GEMM (Linear)

↓ write output to HBM, read it back

Kernel 2: add bias

↓ write to HBM, read it back

Kernel 3: GELU → write to HBM

**3× the activation traffic, 3 launches**

**With fusion: 1 kernel**

**GEMM with a fused epilogue**

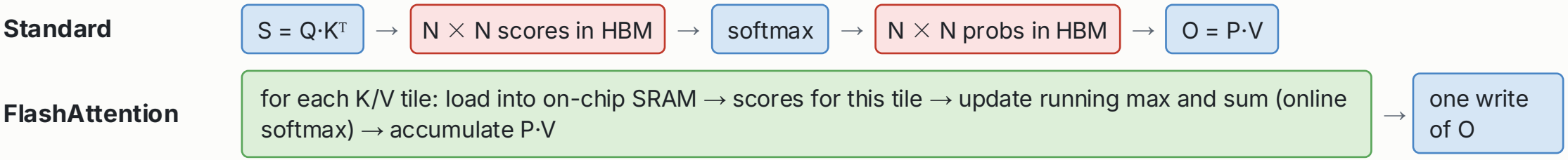
bias + GELU applied in registers, before the single write

**1 read + 1 write, 1 launch**

Other common fusions: RMSNorm + residual add, RoPE + KV-cache write, and attention itself (FlashAttention, next slide)

- **Kernel fusion**: combine several operations into one GPU kernel
- Removes intermediate HBM reads/writes → higher arithmetic intensity. A memory-bound chain may stay memory-bound, but it moves far fewer bytes
- **CUDA Graphs vs fusion**: graphs cut launch overhead (CPU side), but kernels stay separate; fusion cuts memory traffic (GPU side) by merging operations

# System Optimization: FlashAttention




- Standard attention writes the N × N matrix to HBM and reads it back: memory traffic grows with $N^2$
- FlashAttention **tiles** Q, K, V into SRAM and uses an **online softmax** → the exact same result, with no N × N matrix in HBM
- **Decode is different**: one query row per request, so a kernel that parallelises over queries leaves most of the GPU idle. **Flash-Decoding** splits the KV sequence across thread blocks (split-K) and merges the partial results
- Serving kernels today: FlashInfer (paged KV, many variants), FlashMLA (DeepSeek's MLA decode kernel), FlashAttention-3 (Hopper)

Dao et al., NeurIPS'22; Dao, ICLR'24; Shah et al., NeurIPS'24; Ye et al., FlashInfer, MLSys'25

# System Optimization: Quantization

Decode time ≈ bytes moved ÷ memory bandwidth. Fewer bytes per weight and per KV entry → faster decode and more room for KV cache.

| What | Format | Effect | Notes |
|---|---|---|---|
| Weights + acts | FP8 (W8A8) | Half the bytes of BF16; FP8 tensor cores | Hopper and newer; usually near-lossless |
| Weights | INT4 (AWQ, GPTQ) | 4× fewer weight bytes → faster decode | Dequantised inside the GEMM kernel |
| Weights + acts | FP4 (NVFP4, MXFP4) | 4× fewer bytes; FP4 tensor cores | Blackwell; needs block scaling |
| KV cache | FP8 / INT8 | 2× more tokens per GB; less traffic | Helps long contexts most |

- Check accuracy for each model and task — reasoning and long-context tasks are often the most sensitive

Lin et al., AWQ, MLSys'24; Frantar et al., GPTQ, ICLR'23

# Beyond One Engine: Workloads Have Changed

### Reasoning models

Very long outputs (thousands of tokens of “thinking”) → **decode dominates**

TBT, KV cache size and speculative decoding matter most

### Agents and multi-turn chat

Every call re-sends a long, growing history → **prefix reuse** is the key

Prefix caching, KV offloading to CPU memory and SSD, KV-aware routing

### RL rollouts and batch jobs

Huge numbers of sampled trajectories → **throughput over latency**

Offline batching, very large batches, efficient MoE batch inference

Tiered KV storage (e.g. LMCache, Mooncake) keeps reusable KV cache in CPU memory or SSD, so a later request can load it instead of recomputing the prefill.

# Beyond One Engine: Serving at Cluster Scale

### Request routing

Send each request to the replica that already holds its prefix (**KV-aware routing**), while balancing load

e.g. NVIDIA Dynamo, llm-d

### Autoscaling and cold start

A new replica must load tens to hundreds of GB of weights. **Fast checkpoint loading** and locality-aware placement cut start-up time

e.g. ServerlessLLM (OSDI'24)

### Many models, many adapters

Serve many fine-tuned **LoRA adapters** on one base model, batched together with special kernels

e.g. S-LoRA, Punica (MLSys'24)

The pre-ChatGPT question — how to host many models on shared hardware — returns, now at cluster scale.

# Recap: Which Technique Helps Which Metric?

| Technique | Main metric improved | Cost / trade-off |
|---|---|---|
| Continuous batching | Throughput, queueing delay | More complex scheduler |
| PagedAttention | Concurrency (memory) | Indirection inside attention kernels |
| Prefix caching | TTFT, prefill compute | Cache management and eviction |
| Chunked prefill | TBT (no stalls) | Slightly higher TTFT for long prompts |
| Disaggregation | TTFT and TBT, tuned separately | KV transfer; two pools to size |
| Speculative decoding | TBT at small batch sizes | Draft cost; less gain at large batch |
| TP / PP / DP / EP | Model fits; latency (TP) or throughput (DP, EP) | Communication; weight copies |
| CUDA graphs, fusion | Step latency | Static shapes; kernel engineering |
| Quantization | TBT, KV capacity | Accuracy risk |

# References


### Scheduling and memory

Yu et al. Orca: iteration-level scheduling. OSDI'22

Kwon et al. vLLM / PagedAttention. SOSP'23

Zheng et al. SGLang / RadixAttention. NeurIPS'24

### Prefill and decode

Agrawal et al. Sarathi-Serve. OSDI'24

Zhong et al. DistServe. OSDI'24

Patel et al. Splitwise. ISCA'24

Qin et al. Mooncake. FAST'25

### Speculative decoding

Leviathan et al. Fast inference via speculative decoding. ICML'23

Chen et al. Accelerating LLM decoding with speculative sampling. 2023

Cai et al. Medusa. ICML'24 · Li et al. EAGLE. ICML'24

### Models and parallelism

DeepSeek-AI. DeepSeek-V2 and DeepSeek-V3 Technical Reports. 2024

### Kernels and quantization

Dao et al. FlashAttention. NeurIPS'22 · Dao. FlashAttention-2. ICLR'24

Shah et al. FlashAttention-3. NeurIPS'24

Ye et al. FlashInfer. MLSys'25

Lin et al. AWQ. MLSys'24 · Frantar et al. GPTQ. ICLR'23

### Cluster serving

Fu et al. ServerlessLLM. OSDI'24

Sheng et al. S-LoRA. MLSys'24 · Chen et al. Punica. MLSys'24


### Suggested deep reading for this week

vLLM (SOSP'23); for the trade-offs, Sarathi-Serve (OSDI'24)