CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026

15: LLM Serving and Inference: Batching, Speculation, and KV Cache Management

Lecturer: *Hao Zhang*
Scribes: *Suraj Ranganath, Vaishak Menon, Arunima Anand, Jackson Wilke, Atharv Nair, Rishika Mundada, Anish Patnaik, Kris Dcosta, Ishan Jain*

1 Execution Model: Prefill, Decode, and the KV Cache

The lecture started from the basic execution pattern of an autoregressive language model. Given a prompt, the model first runs a **prefill** pass over all input tokens. After that, it enters a **decode** loop. Each decode iteration consumes the token generated in the previous iteration and produces exactly one new token. The loop stops when the sequence reaches a maximum length or when the model emits a termination token such as `<EOS>`.

This separation matters because prefill and decode stress the GPU in different ways. In prefill, the sequence length can be large, so the model sees many tokens at once. In decode, the effective sequence length for the current forward pass is one token. The model is still large, but the batch of useful computation is small unless many requests are served together.

The key dimensions are batch size b, sequence length s, hidden size h, number of heads n, and per-head dimension d. Many tensors contain an s dimension. Large s is expensive for attention because attention contains an $s \times s$ interaction inside each sequence. But $s = 1$ is also a problem when b is small: the GPU is asked to process one vector at a time. The lecture emphasized this tension because it explains why the same model needs different optimization strategies under serving and single-request inference.

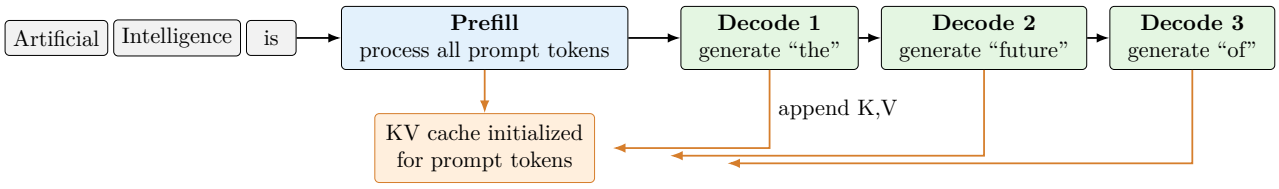


Figure 1: Autoregressive execution has one parallel prefill pass followed by a serial decode loop. The KV cache stores per-layer keys and values so later decode steps do not recompute attention states for earlier tokens.

The **key-value cache** stores the attention keys and values produced by each transformer layer for every token that is still active. During decode, a new query attends to all previous keys and values,

then the new token's key and value are appended. From a systems perspective this object is not a cache in the usual sense. A cache normally trades on an uncertain hit rate. The KV cache has a hit rate of essentially 100 percent: every future decode step needs the previous tokens. It is better to think of it as the **working set** of active generation.

Its lifetime is also different from both parameters and ordinary activations. Model weights are persistent for the whole server process. Layer activations inside one forward pass are mostly ephemeral: a layer consumes them, produces the next activations, and the old values can be discarded. KV entries sit between these extremes. They are allocated at prefill, persist across all decode iterations for that request, grow by one token per decode step, and are freed only when the request finishes.

2 Serving and Inference Are Different Regimes

The lecture used *serving* and *inference* to distinguish two operating regimes. The model computation is the same, but the performance objective is not.

Regime	Typical batch size	Main objective
Online serving	Large effective batch size b, because many users send requests to the same endpoint	High throughput and low cost per query, while meeting reasonable latency limits
Single-request inference	$b = 1$, or a very small batch	Low latency for a small number of requests

In serving, the system is handling many independent requests. Prompts arrive over time, have different lengths, and generate different numbers of output tokens. The GPU should be kept busy across all of them. In single-request inference, there may be no other user to batch with, so the main goal is to finish that one request as quickly as possible.

This is why large-b serving and $b = 1$ inference need different techniques. Serving needs scheduling, dynamic batching, and memory management. Single-request inference needs to reduce the number or cost of serial decode steps.

A practical way to remember the distinction is to ask what the expensive GPU is doing. In hosted serving, the operator is paying for a fleet of GPUs and wants those GPUs to serve as many users as possible subject to latency constraints. In local or edge inference, such as a laptop or phone running one assistant request, there may be no useful batch to form. The bottleneck is not aggregate throughput; it is the latency of that one generation.

3 Bottlenecks in Large-b Serving

For online serving, the first bottleneck is batching prompts of different lengths. If request R_1 has a 20-token prompt and request R_2 has a 500-token prompt, the naive tensor shape is obtained by padding R_1 to length 500. That is simple, but it wastes compute on padding tokens. In a high-throughput service, padding overhead directly lowers the number of useful tokens processed per second.

Decode has a different problem. Each active request contributes only one new token per iteration, so the per-request sequence length for the current computation is $s = 1$. However, the number of active

requests b is large. This large b prevents the main projections and MLPs from degenerating completely: the work is still closer to a matrix-matrix multiply (GEMM) than a matrix-vector multiply (GEMV), because the request dimension supplies useful parallelism. The hard part is that the number of decode iterations is not known ahead of time. One request may emit `<EOS>` after a few tokens while another continues for hundreds of tokens. A static batch has to wait for the longest request or leave GPU slots idle.

The memory bottleneck is the KV cache. For each active request, the cache grows with the number of prompt and generated tokens. With many concurrent users, total KV memory is approximately linear in the total number of active tokens. If the cache is allocated or managed poorly, memory capacity becomes the limit before compute does.

The serving challenge can be summarized as:

$$\text{throughput} \quad \text{requires} \quad \begin{cases} \text{little or no padding in prefill,} \\ \text{dynamic replacement of finished requests,} \\ \text{KV cache storage proportional to useful active tokens.} \end{cases}$$

4 The $b = 1$ Case and Speculative Decoding

When $b = 1$, the prefill batching problem mostly disappears, but decode becomes a latency problem. Each decode step processes one new token. Large matrix multiplications become closer to matrix-vector operations, so the arithmetic intensity drops. Informally,

$$\text{arithmetic intensity} = \frac{\text{operations}}{\text{bytes moved}}.$$

With one token, the GPU cannot reuse model weights across many tokens or many requests. The computation often becomes memory-bandwidth bound: each step repeatedly streams the model's weights from high-bandwidth memory into faster on-chip storage, performs a small amount of useful computation for one token, and then repeats. This is why the lecture described the $b = s = 1$ case as wasteful even on very expensive accelerators such as B200-class GPUs. The hardware has many SMs, but the current operation exposes too little parallel work to keep them busy.

The serial nature of autoregressive decoding gives a simple latency model:

$$\text{latency} = \text{step latency} \times \text{number of decode steps}.$$

If one output token is generated per expensive target-model step, long generations are slow. **Speculative decoding** attacks the second factor: it tries to generate and verify multiple tokens per target-model step.

The important constraint is algorithmic, not merely implementation-related. Autoregressive generation cannot know token $t + 1$ until token t has been chosen. A large model may therefore move hundreds of gigabytes of weights, or in trillion-parameter MoE settings a much larger active weight footprint, just to advance one token. Speculative decoding does not remove autoregression; it changes how often the large target model has to perform this expensive step.

4.1 How Speculative Decoding Helps

The idea is to use a cheaper draft model to propose several future tokens, then use the large target model to verify them in one parallel pass. For greedy decoding, the target model accepts the proposed

tokens until the first position where the draft token differs from the target model's choice. For sampling, the accept/reject rule is adjusted so the output distribution matches the target distribution.

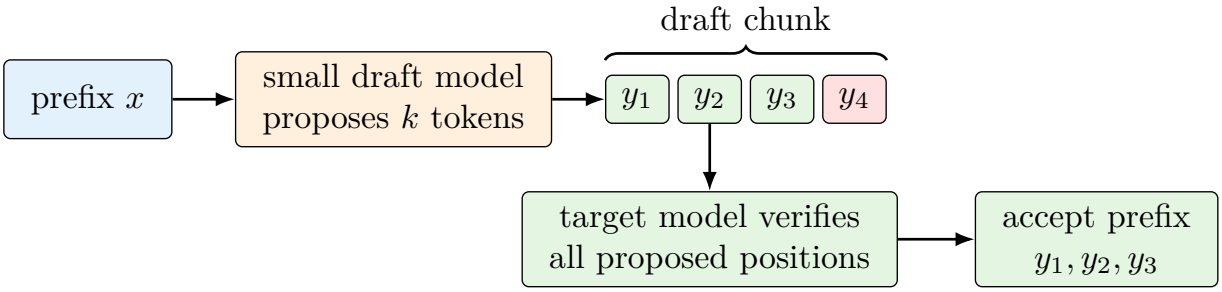


Figure 2: Speculative decoding trades extra draft-model work for fewer expensive target-model decode steps. It is useful when target-model decode is memory-bandwidth bound and the draft acceptance rate is high.

The target model still decides what is accepted. The speedup comes from amortizing the target model's memory movement over several accepted tokens. This is why speculative decoding is especially natural in the $b = 1$, $s = 1$ regime: the expensive model is underutilized per step, and reducing target steps can reduce end-to-end latency.

Speculation is not free. It adds draft-model computation and can waste work if many proposed tokens are rejected. Its benefit depends on three quantities: draft model speed, target model verification cost, and the average number of accepted tokens per verification pass. The right mental model is that speculative decoding may increase total arithmetic, but it can reduce wall-clock latency because it amortizes the target model's memory movement over several accepted tokens.

5 Static Batching Leaves Work on the Table

The lecture then moved back to the large-b serving case. Production systems improve GPU utilization by batching requests together, but a static batch is a poor fit for text generation.

The problem is easiest to see with the lecture's running example. Suppose the serving engine can run at most three requests at once. Requests R_1 and R_2 enter first. Later, R_3, R_4, and R_5 arrive. If the system uses a static batch, it may keep running R_1 and R_2 until both finish. If R_2 emits `<EOS>` early, its slot becomes idle. If R_3 arrives while the original batch is running, it may have to wait even though the GPU could have done useful work for it.

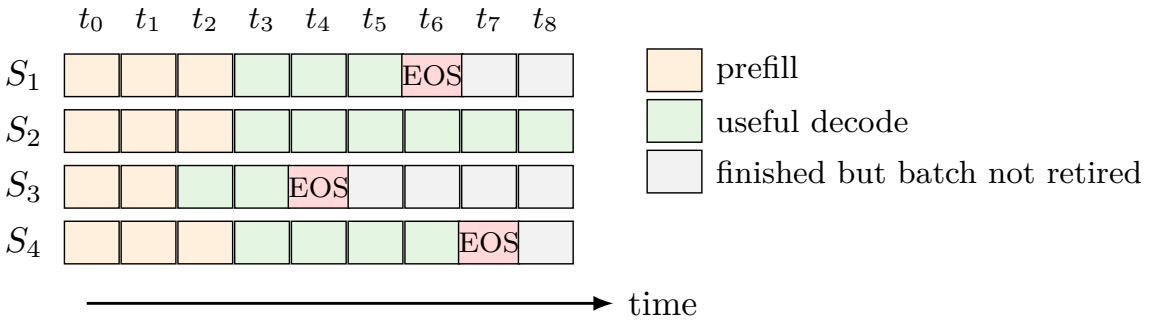


Figure 3: Static batching wastes iteration slots after shorter requests finish. From the user's view, a request may have generated its answer but still wait for the longest request in the static batch.

Static batching therefore has three related issues:

- requests complete at different decode iterations;
- finished requests leave idle GPU slots until the static batch drains;
- newly arrived requests cannot start immediately, increasing queueing latency.

6 Continuous Batching

Continuous batching fixes this by making the batch dynamic at each iteration. A request pool on the CPU holds waiting work. An execution engine on the GPU runs a batch of active work up to a configured capacity. At every iteration, the scheduler can remove finished requests, admit newly arrived requests, and mix prefill and decode work.

The useful abstraction is **iteration-level scheduling**. The batch is not a contract that lasts until every member finishes. It is rebuilt at each iteration from the active requests and the waiting request pool. This gives the system three abilities that static batching lacks: a finished request can leave immediately, a late request can join as soon as there is capacity, and a new prefill can run in the same iteration as ongoing decode work.

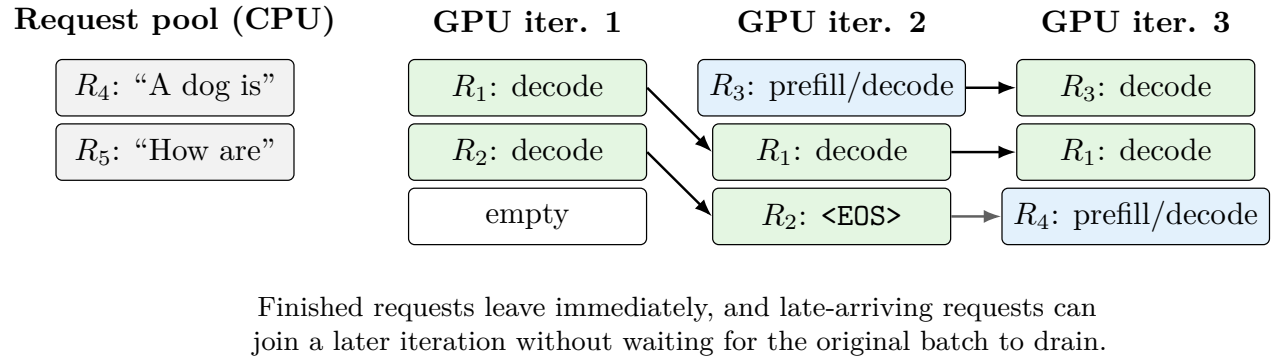


Figure 4: Continuous batching follows the lecture's example: early exits and late arrivals are handled at iteration granularity. The exact scheduler can be FIFO or more sophisticated, but the batch is not fixed for the lifetime of a request.

This design improves both utilization and user-facing latency. The GPU sees a fuller batch over time, and each user can still receive an independent stream of tokens as soon as their request produces them.

The policy for choosing which requests enter the GPU is a scheduling problem. A simple implementation may use FIFO, as in the lecture example where R_3 enters before R_4 and R_5. More advanced systems consider prompt length, remaining budget, priority, fairness, or latency service-level objectives. The lecture's main point was not a particular policy, but the mechanism: the serving engine must be able to change the batch continuously.

One visible consequence is streaming. Modern hosted LLM APIs often return tokens incrementally rather than waiting for the whole completion. Continuous batching is one reason this is practical: each request can produce and return its own token stream while sharing GPU iterations with unrelated requests.

7 How Ragged Requests Can Be Batched

Continuous batching is only useful if the model computation can run efficiently on ragged requests. The key observation from the lecture is that not every transformer operation depends on the sequence dimension in the same way.

Attention is sequence-dependent. A token's attention output depends on which previous tokens it can attend to, and different requests may have different KV-cache lengths. The attention kernel therefore needs per-request sequence metadata.

Many other transformer operations are token-dependent rather than sequence-dependent. RMSNorm normalizes one token representation; the $Q/K/V$ projections map one token hidden vector into attention vectors; the output projection, MLP, and SwiGLU/MoE feed-forward computation also apply per token. For these operations, the system can collapse the ragged batch into a single list of tokens:

$$(b, s, h) \quad \longrightarrow \quad \left(\sum_i s_i, h\right).$$

Then one large kernel processes all useful tokens, without padding shorter prompts to the longest prompt.

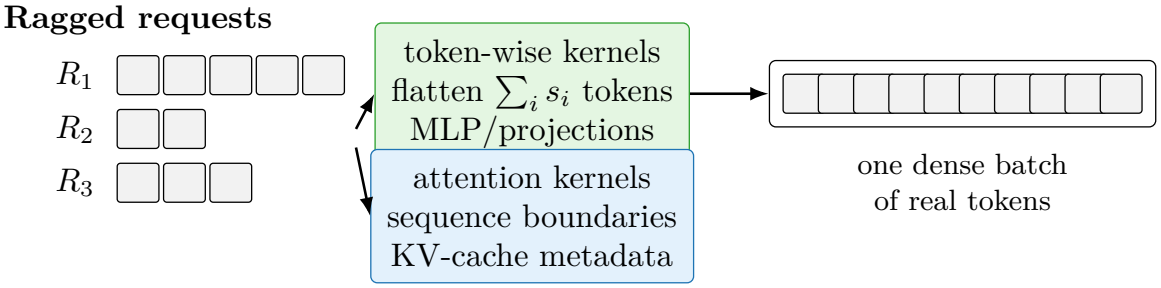


Figure 5: The system avoids padding by separating token-wise work from sequence-dependent attention. This is why continuous batching can be effective when MLP/projection work dominates runtime.

This decomposition also explains how different phases can be mixed:

- **Prefill with prefill:** prompts have different lengths, so token-wise work can be flattened while attention uses ragged sequence boundaries.
- **Decode with decode:** each request contributes one query token, but those query tokens attend to KV caches of different lengths.
- **Prefill with decode:** new prompts and ongoing decode steps can share token-wise kernels, while attention handles full prompt attention for some requests and single-query attention for others.

The difficult implementation work is in the attention kernels and metadata management. Modern serving engines such as vLLM and SGLang include kernels that support variable-length prefill, decode, and mixed prefill-decode batches. The lecture emphasized the principle: do not force every operation into a padded rectangular tensor when the operation itself only needs a list of tokens.

This decomposition matters because, for large models at moderate context length, MLPs and projections often dominate the compute budget. The lecture gave the rule of thumb that attention can be roughly 20–30% of the work while MLP-like operations can be 70–80%. In that regime, batching token-wise work captures most of the benefit. At very long context length the balance can flip, and attention can dominate. That is why recent serving systems invest heavily in variable-length attention kernels rather than treating attention as an unbatched special case.

7.1 Kernel Trend: Variable-Length Attention

The current industry trend is to fuse more of this ragged attention logic into specialized GPU kernels. Variable-length attention batching, online softmax as used in FlashAttention-style algorithms, and prefill/decode attention are increasingly handled together inside one kernel family. This is less elegant than flattening token-wise MLP work because the kernel must track sequence boundaries, KV-cache locations, causal masks, and different context lengths. But it is the direction production systems have moved because long contexts and mixed prefill-decode batches are now common.

The lecture highlighted FlashInfer as an example of this kernel-focused infrastructure layer. Serving frameworks and training frameworks increasingly rely on libraries that provide optimized kernels for LLM inference, and sometimes diffusion workloads too. The important takeaway is not a specific API, but the division of labor: high-level serving systems decide scheduling and memory layout, while specialized kernel libraries make ragged attention and token-level computation fast enough to keep expensive accelerators busy.

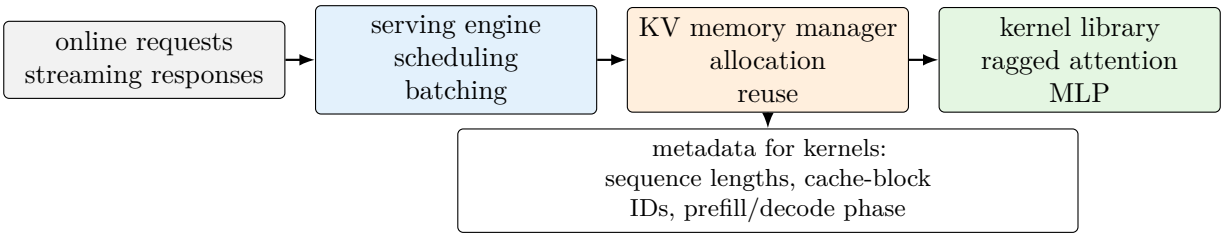


Figure 6: Modern serving stacks split responsibilities. The scheduler decides which requests run; the memory manager decides where KV entries live; optimized kernel libraries execute token-wise and ragged attention work using the metadata supplied by the engine.

8 KV Cache Management

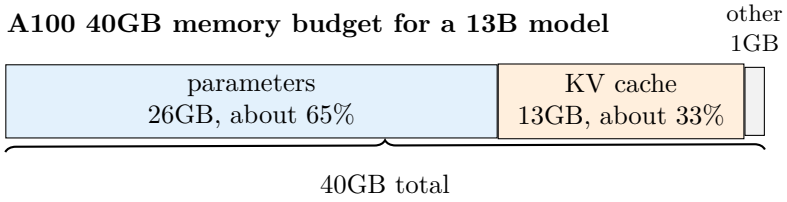
For each transformer layer ℓ and active request r, the KV cache stores

$$K_\ell^{(r)}, V_\ell^{(r)} \in \mathbb{R}^{T_r \times n_{\mathrm{KV}} \times d_{\mathrm{head}}},$$

where T_r is the number of cached tokens for that request, n_{KV} is the number of key-value heads, and d_{head} is the head dimension. The factor of two is for keys and values. For L layers and a bytes per element, the approximate memory use is

$$M_{\mathrm{KV}} \approx 2 \cdot L \cdot n_{\mathrm{KV}} \cdot d_{\mathrm{head}} \cdot a \cdot \sum_{r \in \mathrm{active}} T_r.$$

Thus KV memory scales with the total number of active tokens, not only the number of active requests. During decode, T_r grows by one per generated token. When a request finishes, its KV memory should be released quickly.



The lecture's slide used this example to show why KV cache management is central. After storing the weights, only about 14GB remain for cache and runtime overhead. If each request reserves cache pessimistically, concurrency is low. If cache allocation follows actual tokens and avoids fragmentation, many more requests can fit.

Figure 7: In the 13B-on-A100 example, parameters dominate memory, but the remaining capacity is quickly consumed by KV cache.

The A100 example is historically important because this was when KV-cache management became an obvious systems bottleneck. Modern accelerators such as B200 provide much larger memory capacity, but the same structure remains: model weights claim a fixed base allocation, cache grows with active tokens, and throughput improves only if enough concurrent work fits in memory.

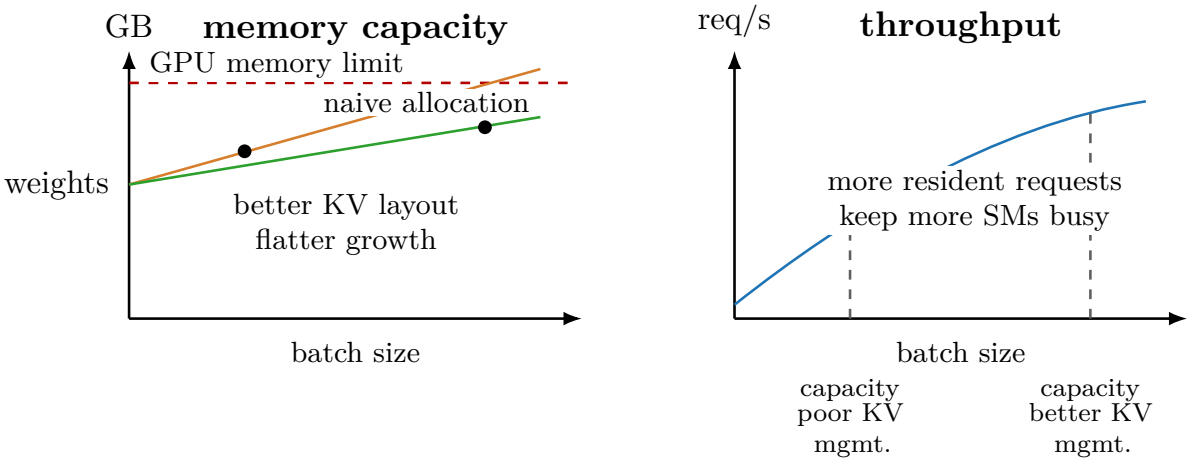


Figure 8: KV-cache management links memory capacity to throughput. If cache allocation grows too fast with batch size, the server hits the memory wall before the GPU reaches its useful throughput region.

A concrete calculation helps explain the slide's intuition. For a 13B-class transformer with roughly $L = 40$, $n_{\text{KV}} = 40$, $d_{\text{head}} = 128$, fp16/bf16 cache elements, and a 2048-token sequence,

$$M_{\text{KV}} \approx 2 \cdot 40 \cdot 40 \cdot 128 \cdot 2 \cdot 2048 \approx 1.6 \text{ GB per request.}$$

Eight such full-length requests require about 13GB of KV cache, matching the memory scale shown in the lecture. Better cache management can raise the number of concurrent requests by avoiding over-reservation and fragmentation. The slides used the vLLM/PagedAttention story to preview the next lecture: existing systems in that example hit the wall around 8 concurrent requests, while vLLM-style cache management aims to support many more, on the order of 40 in the slide. The systems lesson is not the exact number, which depends on model and workload, but the mechanism: flattening KV memory growth increases resident batch size, which raises throughput until another bottleneck appears.

A useful way to understand the PagedAttention preview is to separate the logical sequence from the physical KV-memory layout:

Layout idea	**What it means**	**Serving intuition**
Contiguous reservation	Reserve KV space per request, often for an upper-bound output length.	Simple indexing, but unused tail space and fragmentation reduce the number of resident requests.
Block-based allocation	Store KV entries in fixed-size token blocks and use a block table to map logical token order to physical blocks.	The request behaves as if its cache is contiguous, while memory can grow only as real tokens arrive.
Throughput effect	Lower memory waste before the capacity wall.	Fits more resident requests, raising the effective batch size.

Table 1: PagedAttention borrows the virtual-memory idea of separating logical address order from physical storage. For KV cache, this reduces fragmentation and over-reservation.

The lecture also mentioned prefix caching. If many requests share a prefix, such as the same system prompt, their prefix KV states can be reused instead of recomputed and stored separately. Serving engines often organize reusable prefixes with a trie or radix tree over token IDs. This is separate from

continuous batching, but it attacks the same serving objective: spend memory and compute only on useful, non-duplicated tokens.

9 Takeaways

The main conceptual split is between **latency optimization for small batches** and **throughput optimization for serving**. In the $b = 1$ case, decode is serial and memory-bandwidth bound, so speculative decoding reduces the number of expensive target-model steps. In large-b serving, the hard problems are dynamic batching and KV memory management.

Continuous batching improves serving by letting requests join and leave at iteration granularity. Its implementation depends on separating token-wise transformer work from sequence-dependent attention. The current kernel trend is to make even the sequence-dependent part less special, using variable-length attention kernels and FlashAttention-style online softmax inside production kernel libraries. KV cache management is equally important because the cache grows with active tokens and can become the capacity bottleneck even when the GPU has enough compute.

The examples in lecture were meant to build this systems intuition. Padding looks convenient, but wastes compute. Static batches look simple, but waste slots and increase queueing. KV cache management matters because it directly limits how many active users can fit on the GPU. If many requests share the same prompt prefix, prefix caching and RadixAttention let the server reuse those KV entries instead of recomputing them or storing extra copies.