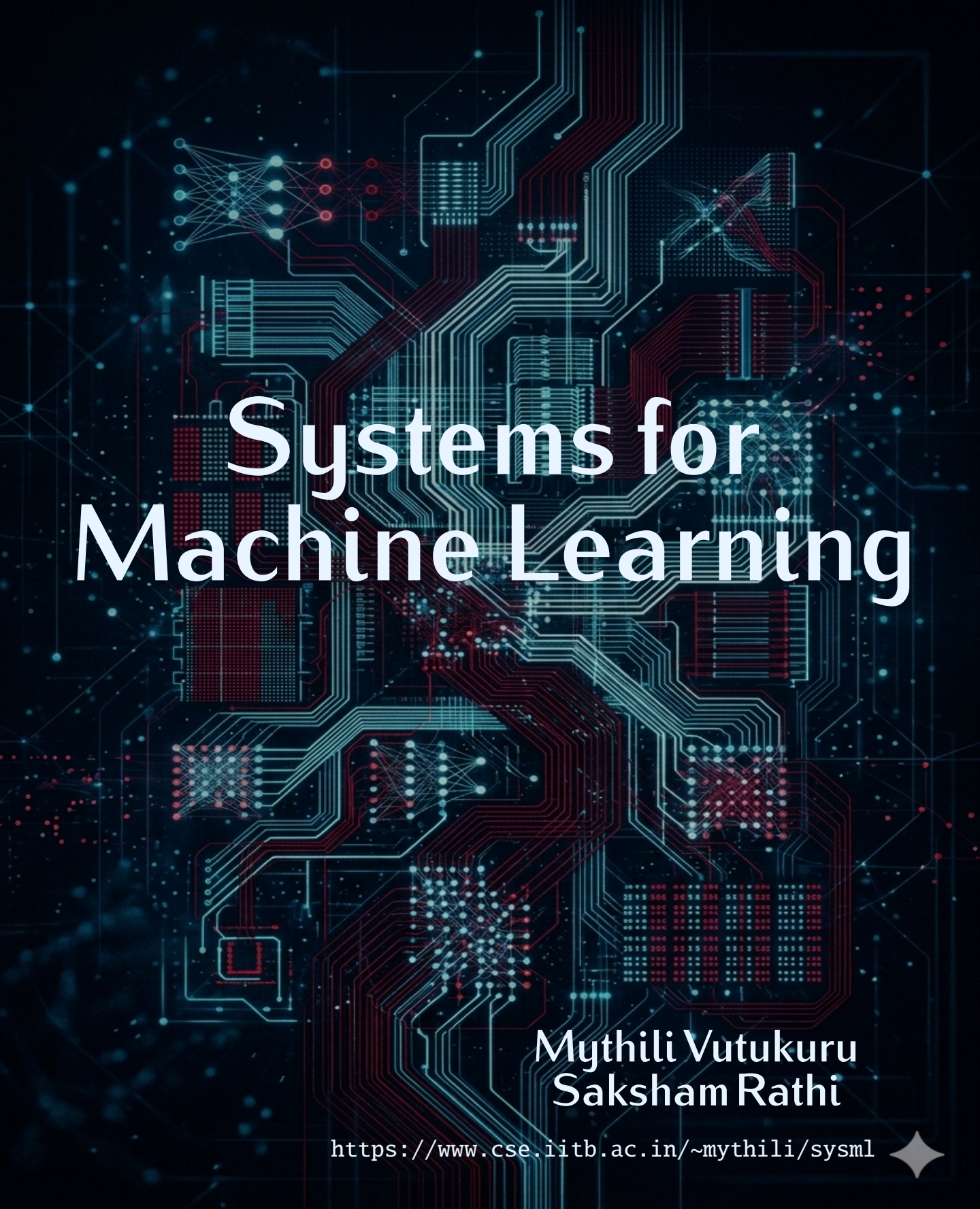

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

October 2026

8

LLM Inference Optimizations

8.1 What is Inference? **3**
8.1.1 Output Generation 3
8.1.2 Speculative Decoding 4
8.1.3 Prefill, Decode, and KV Caching 5
8.1.4 Batching and KV Caching 6
8.1.5 Performance Metrics 8
8.2 Memory Management **12**
8.2.1 Traditional Memory Management for KV Caches 12
8.2.2 PagedAttention 13
8.2.3 Sharing Memory Across Sequences 17
8.3 Batching and Scheduling **24**
8.3.1 Request-Level Scheduling and Its Limits 24
8.3.2 Iteration-Level Scheduling in Orca and vLLM 24
8.3.3 Prefill-Prioritizing and Decode-Prioritizing Schedules 27
8.3.4 Chunked Prefills in Sarathi-Serve 28
8.3.5 Fusing Prefill and Decode Attention in POD-Attention 34
8.3.6 Disaggregating Prefill and Decode 38
8.4 Prefix Sharing **45**
8.4.1 The Radix Tree 45
8.4.2 Tree-Structured Attention Kernels 49
8.4.3 Prefix Homogeneity: A Scheduling Signal Beyond Batch Size 52
8.4.4 Retrieval-Augmented Generation 53
8.5 CPU Offloading **66**
8.5.1 Why Offload, and What Can We Offload? 66
8.5.2 Offloading the KV Cache 67

8.5.3 Offloading Compute 70
8.5.4 Offloading for Long-Context Serving 72
8.6 KV Cache Compression and Sparsity 78
8.6.1 Why the KV Cache is Worth Compressing? 78
8.6.2 Model-Level Compression: Grouped-Query Attention 79
8.6.3 Runtime Compression: Differentiated KV Cache Management 80
8.6.4 Sparse Attention and the KV Cache 81
8.6.5 ECHO: Offloading the KV Cache of Sparse-Attention Models 82
8.6.6 Communicating KV Cache in Disaggregated Serving 84
8.7 An Inference Engine in Practice: Nano-vLLM 87
8.7.1 Core Components of an Inference Engine 87
8.7.2 Starting the Engine 89
8.7.3 Scheduler 90
8.7.4 Model Runner, Attention, and CUDA Graphs 94
8.7.5 One Full Step of Generation 95
8.8 Summary 97

The previous two chapters looked at how we train large models. We looked at distributed training techniques, and how large clusters of GPUs are networked to support such training efficiently. In this chapter we turn to a different but equally important problem: how to serve a trained large language model efficiently. Unlike training, inference is more real-time in nature, and encounters very different resource bottlenecks. For example, with LLM inference, a serving engine should be able to provide quick responses to user requests, while also ensuring that the batch of requests being processed utilizes the GPU hardware resources efficiently. Meeting both these goals simultaneously is the central challenge of LLM inference. We focus mostly on LLM inference (and not very much on inference in other deep learning models) because the Transformer architecture is the most-resource intensive, and hence the hardest to optimize. That said, some of these ideas apply to inference on other deep learning models as well.

We will begin this chapter by revisiting what inference actually computes and how text is generated one token at a time. We understand the main performance bottlenecks during the prefill stage, where the user prompt is processed, and the decode phase, where output tokens are generated autoregressively. We will then study the main ideas that make this process efficient in practice, including key-value caching, batching and scheduling, efficient memory management, offloading compute and memory to the host, and parallelizing inference across multiple devices.

8.1 What is Inference?

Inference is the process of using a trained model to produce output for a new input, by executing the forward pass of the model for the input. For a large language model, inference involves processing the text in the user prompt, and generating a continuation of that text, one token at a time. Understanding how this generation process works, and why it behaves so differently from training, is the starting point for understanding every optimization discussed in this chapter.

8.1.1 Output Generation

As we saw in Chapter 2, a transformer-based language model does not directly output words. The input text is first broken into tokens by a tokenizer, and each token is mapped to an embedding vector from the model's vocabulary. These embedding vectors are passed through the transformer, and after the final layer, an unembedding step projects the model's internal representation back into a vector whose size equals the size of the vocabulary. This vector of logits, after a softmax operation, gives the probability distribution over every possible token in the vocabulary for the next token. Figure 8.1 illustrates this process for the input "Birds of a feather", showing how the tokenizer converts text into embedding vectors (equivalently, to the indices of the tokens in the model's embedding vector table), and how the model converts these embeddings into output logits over the vocabulary.

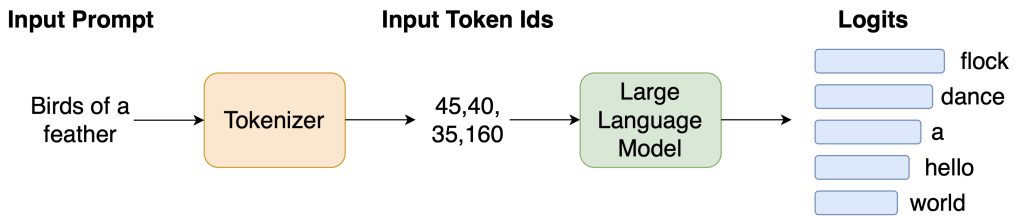


Figure 8.1: LLM inference

Once a next token has been chosen from this distribution, it is appended to the input, and the whole process repeats: the new, longer sequence is fed back into the model to produce a fresh distribution over the token that should come after it. This is what makes language model generation autoregressive — each new token depends on every token that came before it, including the ones the model itself just generated. Naively, this means that every single step reprocesses the entire prefix from scratch, which results in a compute cost that grows quadratically with the length of the generated sequence, as shown in Figure 8.2. We will see shortly why this naive approach is far from what real systems do.

Given a probability distribution over the vocabulary at each step, the model still needs a rule for actually picking the next token. We covered the main decoding strategies in detail in Chapter 2, so we only recap them briefly here. The simplest approach is greedy decoding, which always picks the

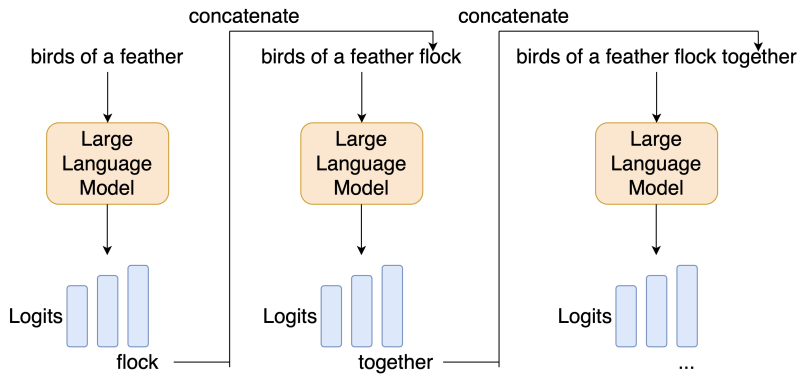


Figure 8.2: Output generation

token with the highest probability at every step. Beam search improves on greedy decoding by keeping track of several of the most likely partial sequences at once, rather than committing to a single path, which reduces the chance of missing a high probability sequence that starts with a locally weaker token. Temperature sampling samples from the probability distribution, rather than always taking the maximum, with a temperature parameter controlling how sharply peaked or flat that distribution is. Top-k sampling restricts sampling to only the k most probable tokens at each step, and top-p, or nucleus, sampling restricts sampling to the smallest set of tokens whose cumulative probability exceeds a threshold p. Each of these strategies trades off determinism, diversity, and quality of the generated text in different ways, and the right choice depends on the application.

8.1.2 Speculative Decoding

The autoregressive nature of decoding means that generating each token requires one full forward pass through the model, and these passes cannot be parallelized across tokens, because each one depends on the token before it. As we will see later, this makes the decode phase memory-bandwidth bound rather than compute bound, and much of the GPU's compute capacity sits idle while it waits on memory accesses. Speculative decoding is a technique that exploits this idle compute to reduce the effective latency of generation without changing the final output distribution.

The idea is to use a small, fast draft model alongside the original, much larger target model. The draft model first proposes several consecutive tokens very quickly. The target model then checks all of these proposed tokens in a single forward pass. Note that this verification by the target model can be done in parallel for all proposed tokens at once, and this compute-bound operation can be almost done for free by an underutilized GPU during the decode phase. This is illustrated in Figure 8.3: the draft model proposes "a", "feather", "dance", "birds" following the input "birds", "of", and the target model verifies all four candidates in parallel.

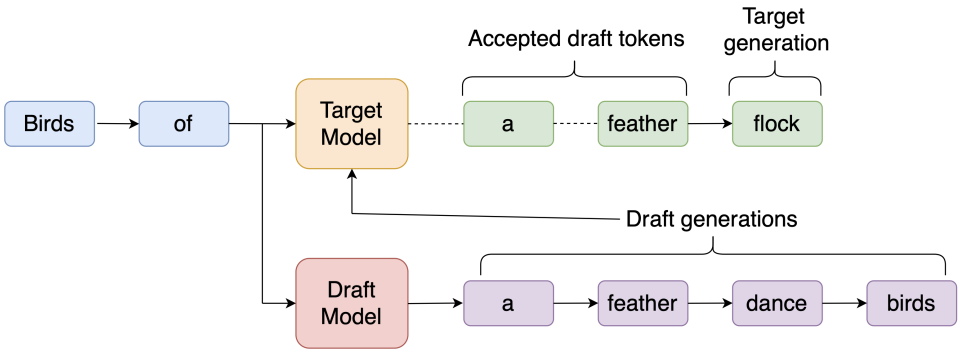


Figure 8.3: Speculative decoding

The target model does not simply check whether it would have generated the exact same tokens as the draft model. Instead, it compares the probability it assigns to each proposed token, $P(\text{Target})$, against the probability the draft model assigned to that same token, $P(\text{Draft})$. If the target model's probability is at least as high as the draft model's, the token is accepted outright. If the target model's probability is lower, the token is accepted only with probability equal to the ratio $P(\text{Target})/P(\text{Draft})$; otherwise it is rejected, and a new token is instead resampled from an adjusted distribution constructed from the target model's own output. This rejection sampling scheme guarantees that the tokens ultimately produced are distributed exactly as if the target model had generated them on its own, so speculative decoding is a lossless optimization — it changes only the speed of generation, not its quality. Crucially, as soon as one proposed token is rejected, every token proposed after it must also be discarded, since the draft model generated them conditioned on a continuation that the target model did not actually accept.

The overall speedup from speculative decoding depends on how often the draft model's proposals are accepted. A draft model that closely mimics the target model's behaviour, perhaps because it is a smaller version of the same model family or was distilled from it, will have a high acceptance rate and yield a large speedup, while a poorly matched draft model may propose tokens that are rejected so often that the extra verification work outweighs any benefit.

8.1.3 Prefill, Decode, and KV Caching

We noted earlier that naive autoregressive sampling reprocesses the entire input sequence at every single generation step, giving quadratic overall cost. This inefficiency comes from recomputing the same intermediate values, specifically the key and value tensors used by the attention mechanism, over and over again for tokens that have not changed since the previous step. Chapter 2 introduced key-value caching as the fix for this problem; here we look at it more closely, since it is central to understanding how inference systems are built.

Real inference systems split generation into two distinct phases. The first is prefill, in which the entire prompt is processed in a single forward pass. Because all the input tokens are available at once, this phase consists of large matrix multiplications and is compute bound. During prefill, the key and value tensors for every token in the prompt are computed and stored in memory, initializing what is called the KV cache. The second phase that follows is decode, in which output tokens are generated one at a time. Because each new token depends on the one before it, this phase is inherently sequential, and cannot be parallelized across steps in the way prefill can be. At each decode step, the model only needs to compute the query, key, and value for the single new token. Then, it computes attention scores for that query against the keys and values of every token seen so far, by reading them from the KV cache. Finally, it appends the new token's own key and value to the cache for future steps. Because so little new computation happens per step relative to the amount of memory that must be read, decode is memory-bandwidth bound rather than compute bound (Figure 8.4).

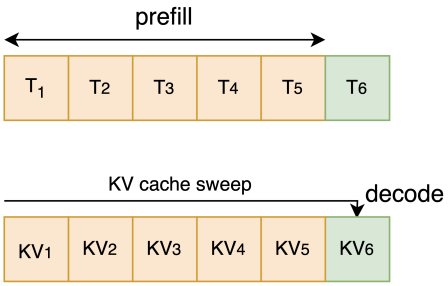


Figure 8.4: Prefill and decode

This split explains why prefill and decode place very different demands on hardware. Prefill has high arithmetic intensity, meaning it performs a large amount of computation for every byte of data it moves from memory, and so it can make good use of the GPU's compute units. Decode has very low arithmetic intensity, since a single query attends to a large KV cache using comparatively little arithmetic, and so its throughput is limited by how quickly memory can be accessed, rather than by how fast the GPU can compute.

8.1.4 Batching and KV Caching

Because a single decode step generates only one token at a time and performs so little computation, processing one user request at a time until completion leaves most of the GPU's compute capacity unused during decode, even though the prefill phase for that same request may have utilized the GPU satisfactorily (Figure 8.5). Processing requests one after another, alternating between a large prefill block and several small decode blocks, is there quite sub-optimal with respect to GPU utilization.

The fix is to batch requests together (Figure 8.6), so that many decode steps run in parallel on

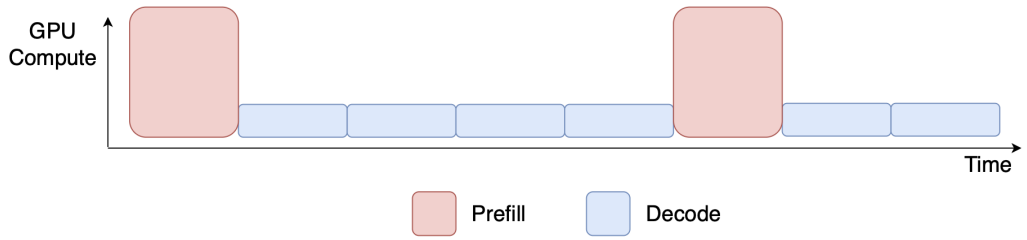


Figure 8.5: Prefill and decode without batching

the GPU rather than one at a time. Instead of processing a request to completion before starting the next, a server can run the prefill phase of several requests first, and then batch together the decode step of every request for each subsequent token generation. Because the sequential dependency in decode is within a request rather than across requests, batching the decode steps of multiple requests together allows easy parallelization and improves GPU utilization, without changing the output of any individual request.

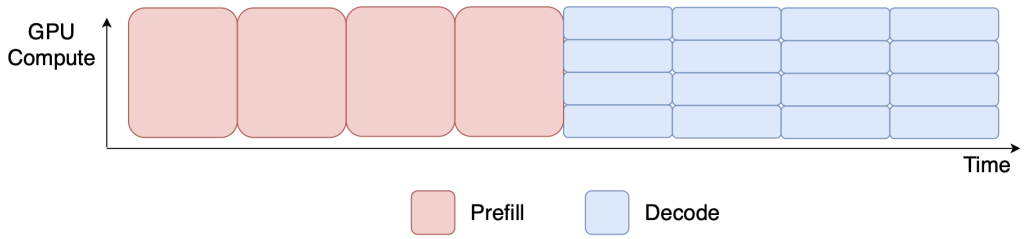


Figure 8.6: Prefill and decode with batching

Batching, however, interacts directly with KV caching, and this interaction is what ultimately limits how large a batch can be. Every request in a batch has its own KV cache, and because every decode step for a request must attend to the request's entire cache, all of these caches need to reside in GPU memory simultaneously for as long as the corresponding requests are still generating new output tokens (Figure 8.7). As more requests are added to a batch, more KV caches must be kept resident in GPU memory alongside the batched decode computation. Because GPU memory is finite, the total size of the KV caches in flight places a hard limit on the maximum batch size a system can support during decode, independent of how much spare compute might otherwise be available. This tension between wanting large batches for compute efficiency and being constrained by KV cache memory is one of the central problems in LLM serving systems, and it motivates much of the memory management work we will study later in this chapter.

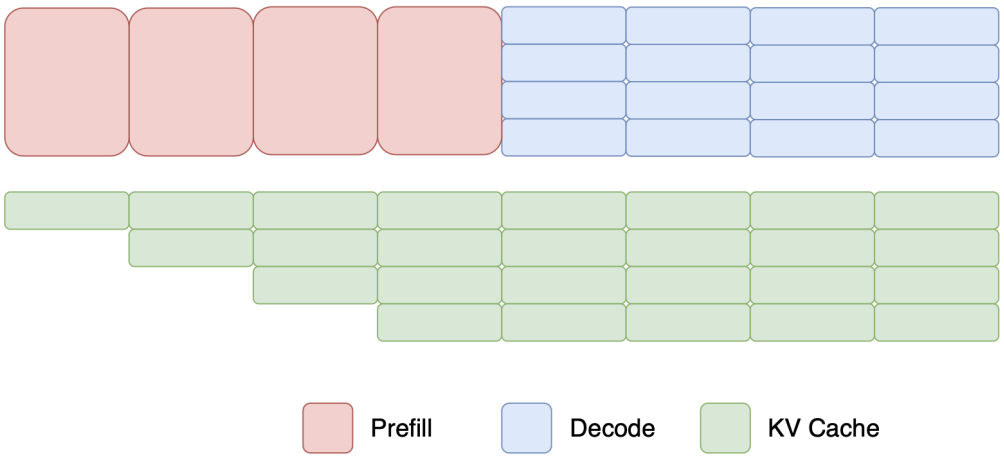


Figure 8.7: KV cache with batching

8.1.5 Performance Metrics

We now define a few metrics that help us reason about and compare the performance of inference systems. Time To First Token (TTFT) measures how long it takes to complete the prefill phase and produce the very first output token, and reflects the initial delay a user experiences before a response begins to appear. Time Per Output Token (TPOT), sometimes called Time Between Tokens (TBT), measures how long each subsequent decode step takes, and corresponds to how quickly text streams in once generation has started.

Since decode is typically memory-bandwidth bound, its minimum possible step time can be estimated directly from how much data must be read from memory relative to how fast that memory can be accessed. Every decode iteration of a batch must read the model's parameters as well as the KV caches of every request in the batch, so the theoretical minimum step time taken for one decode iteration of an entire batch is approximately:

$$\text{Theoretical Min Decode Step Time} = \frac{\text{Batch Size} \times \text{KV Cache Size} + \text{Parameter Size}}{\text{Total Memory Bandwidth}}.$$

This relationship makes explicit why larger batches, while good for utilization, also increase the amount of memory traffic every decode step must pay for.

A third metric, throughput, measured in tokens processed or generated per second, can be computed separately for the prefill phase, the decode phase, or the system as a whole. Because prefill and decode are bound by different resources, prefill throughput is generally limited by the GPU's peak compute capability, while decode throughput is generally limited by memory bandwidth. Figure 8.8 confirms this empirically for Mistral-7B running on a single A100 GPU — prefill throughput barely

changes as batch size grows, since it is already compute bound and additional batching adds little further parallelism to exploit, whereas decode throughput scales almost linearly with batch size, since decode starts out with so much spare compute capacity that batching lets the system approach the maximum throughput dictated by the memory bandwidth limit. Therefore, batching is much more important during the decode phase than during prefill, and is one of the most important optimizations to imporve token generation throughput of an LLM serving system.

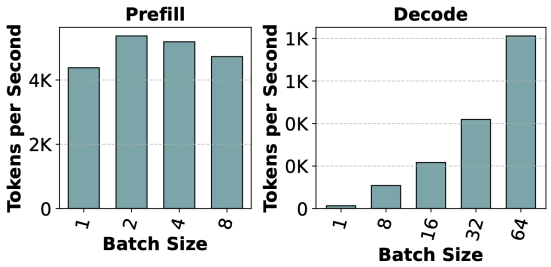


Figure 8.8: Throughput vs. batch size for prefill and decode; Image credit: [Agrawal, Kedia, Panwar, Mohan, Kwatra, B. S. Gulavani, et al., 2024]

Together, TTFT, TPOT, and throughput capture the things users and system operators care about most — how responsive a system feels, and how efficiently it utilizes expensive hardware. As we will see in the rest of this chapter, most optimizations used in modern LLM serving systems, including batching/scheduling policies, memory management techniques, and strategies for offload and parallelism across devices, strive to improve these metrics, sometimes by trading one off against another.

Practice Problem 8.1

Answer the following questions.

(i) Which of the following is/are valid reasons for why speculative decoding is beneficial?

(A) The main target model can verify outputs of the draft model in parallel

(B) The draft model can generate outputs quicker than the main target model

(C) The main model can accept some outputs of the draft model without verification

(D) The main model only processes some, but not all of the tokens

Solution: (A), (B)

(A) True. The target model checks all the draft tokens in a single forward pass, so several tokens are verified at the cost of roughly one decode step.

(B) True. The draft model is much smaller, so it can cheaply propose several candidate tokens faster than the target model could generate them.

(C) False. Every draft token must be verified by the target model; tokens are accepted only because verification succeeded.

(D) False. The target model processes all the draft tokens in the verification pass (in parallel); the benefit comes from parallelism, not from skipping tokens.

(ii) Which of the following statements is/are true about prefill and decode phases?

(A) Prefill is more compute bound, while decode is more memory bound

(B) Batching is more beneficial in decode than in prefill

(C) The memory bandwidth of the GPU is fully utilized during decodes

(D) The SMs of the GPU are fully occupied during the prefill phase

Solution: (A), (B)

(A) True. Prefill processes many tokens at once (high arithmetic intensity), whereas decode handles one token per request and is limited by loading weights and KV cache from memory.

(B) True. Batching in decode amortizes the cost of loading model weights across many requests, while prefill already has many tokens per request to keep the GPU busy.

(C) False. Decode is memory bound, but achieved bandwidth is not guaranteed to reach the peak (e.g., small batches, kernel overheads).

(D) False. SM occupancy is not guaranteed in prefill; for example, short prompts may not provide enough work to occupy all the SMs.

(iii) Which of the following statements is/are true about batching decodes?

(A) All KV cache tensors of all decode requests of the currently active batch must be in GPU memory

(B) Only the KV tensors of the last token of every request of a batch need to be in GPU memory

(C) Techniques that reduce the KV cache size will help increase the decode batch size

(D) Smaller models have smaller KV caches and hence can run with larger decode batch sizes

Solution: (A), (C), (D)

(A) True. Attention for each new token needs the keys and values of all previous tokens, so the entire KV cache of every active request must be resident in GPU memory.

(B) False. Keeping only the last token's KV is insufficient, since attention attends over the whole preceding context.

(C) True. A smaller KV cache per request leaves room for more requests in GPU memory, enabling larger decode batches.

(D) True. Smaller models (fewer layers/heads/hidden size) have smaller per-token KV caches and weights, leaving more memory for additional requests.

(iv) Which of the following statements is/are true about LLM inference performance metrics like time to first token (TTFT) and Time between tokens (TBT)?

(A) TTFT primarily depends on prefill latency, while TBT on decode latency

(B) Improving (reducing) the TBT of an inference system will always lead to higher (worse) TTFT

(C) Lower (better) TBT usually correlates to higher decode throughput in tokens/sec

(D) The throughput in tokens/sec of an inference system is directly proportional to the FLOP/s of its GPU

Solution: (A), (C)

(A) True. The first token is produced after the prompt is prefilled, and subsequent inter-token gaps are determined by decode iterations.

(B) False. There can be a trade-off, but not always; better kernels or hardware can improve TBT without hurting TTFT.

(C) True. If each token for a request arrives faster, the system generates more tokens per second, so decode throughput generally rises.

(D) False. Decode is largely memory-bandwidth bound, so throughput also depends on memory bandwidth, batch size, and KV cache capacity, not just FLOP/s.

8.2 Memory Management

We have seen that decode throughput of an LLM inference engine is limited by how efficiently the GPU memory can be accessed at every step, rather than by how much arithmetic the GPU can perform. This section examines GPU memory management closely, covering several techniques used by modern inference engines, particularly the widely used idea of PagedAttention.

8.2.1 Traditional Memory Management for KV Caches

When an LLM is serving user requests on a GPU, the GPU memory is primarily consumed by three things. The first and largest static component is the model's parameters, i.e., the weights of every layer which must remain resident in GPU memory for the entire time the model is being served. The second is the KV cache introduced in the previous section, which stores the key and value tensors for every token of every request currently being processed, and which, unlike the parameters, grows and shrinks continuously as requests arrive, generate tokens, and complete. The third is a smaller and more transient category of activation memory (the intermediate tensors produced and consumed while a forward pass is executing), along with framework and CUDA runtime overhead.

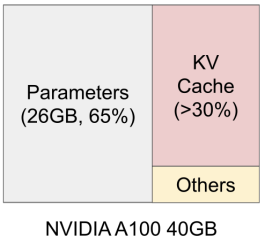


Figure 8.9: Representative GPU memory layout when serving a 13B parameter model on an NVIDIA A100 40GB GPU; Image credit: [Kwon et al., 2023]

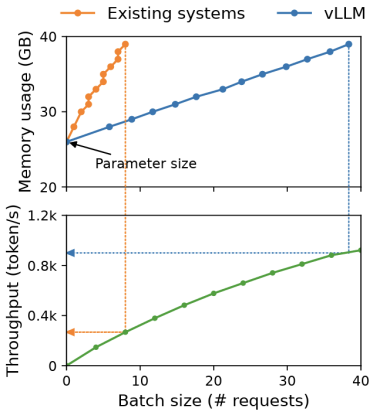


Figure 8.10: Memory usage and throughput vs. batch size; Image credit: [Kwon et al., 2023]

Figure 8.9 shows a representative memory layout for a 13 billion parameter model served on a single 40GB A100 GPU — parameters occupy about 65 percent of memory and persist unchanged throughout serving, while the KV cache can account for over 30 percent and, unlike the parameters, is allocated and freed dynamically for every request. Every design decision discussed in the rest of this section is ultimately about how to pack as many requests' KV caches as possible into whatever memory the parameters leave behind, since that packing directly determines the maximum batch size,

which in turn determines decode throughput.

Early inference engines treated the KV cache the way a program treats an array. To store the key and value tensors for a request, the engine would allocate one contiguous block of memory, large enough to hold the KV tensors for the maximum possible sequence length any request could ever reach. This allocation would be made up front statically, before the engine knew how many tokens the request would actually generate. This design is simple to implement, but it wastes memory in two distinct ways. First, each request is assigned a contiguous memory region that is typically sized conservatively to accommodate future token generation, to reduce the overhead of repeated memory reallocations. Since the inference engine cannot predict in advance how long a sequence will ultimately become, the allocated region often reserves space for tokens that are never generated. This can leave portions of the allocated region unused, resulting in internal fragmentation. Second, because each request occupies a single contiguous block, allocating and freeing requests of different sizes over time leaves many small non-contiguous gaps between allocated regions. Although the total memory may be sufficient, these scattered gaps cannot be combined to satisfy larger allocations, leading to external fragmentation, a well-known problem in memory-allocators (Figure 8.11).

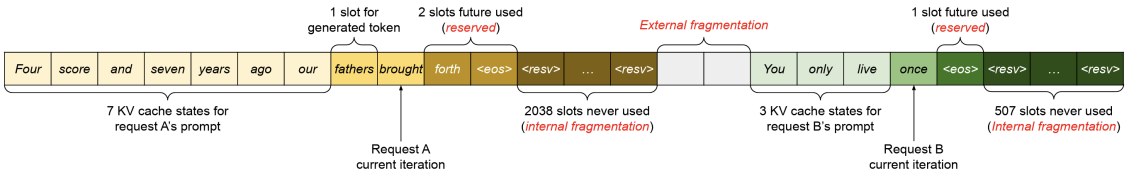


Figure 8.11: KV cache memory management in existing systems; Image credit: [Kwon et al., 2023]

The cumulative effect of these sources of waste is severe. Measurements from the vLLM paper [Kwon et al., 2023] found that in serving systems using this style of allocation, only 20 to 40 percent of KV cache memory was actually holding useful token states, with the rest lost to reservation, internal fragmentation, and external fragmentation. Since the theoretical minimum decode step time derived earlier scales with batch size, and batch size is capped by how much KV cache memory is available, this waste translates directly into lower achievable throughput: a system that can only make effective use of a third of its KV cache memory can support only a third of the batch size it otherwise could, and therefore only a fraction of the decode throughput the hardware is capable of.

8.2.2 PagedAttention

The fix to this problem, introduced by the vLLM system, borrows a well established idea from operating system — virtual memory paging. Rather than allocating one large contiguous span of memory per request, PagedAttention divides the KV cache into fixed-size blocks, each holding the key and value vectors for a small, fixed number of tokens, for example, 16 tokens per block. These blocks are the unit of memory allocation. A request's KV cache is built up out of as many blocks

as it currently needs, and a new block is only allocated once the previous block has filled up, so a request never reserves more space than the tokens it has actually generated so far (up to the granularity of a single block). Because blocks are allocated on demand rather than up front, both internal fragmentation and reservation waste shrink to at most one partially filled block per request, and because a request's blocks do not need to sit next to each other in physical memory, external fragmentation disappears entirely. This leads to memory saving in the GPU, eventually resulting in larger batch sizes and higher throughput, as shown by the vLLM authors (Figure 8.10).

The non-contiguous layout of the KV cache in memory now requires a change to how attention itself is computed. In ordinary attention, the query for a token can simply read a contiguous span of keys and values in memory. With paged attention, a request's key and value vectors are logically one contiguous sequence but physically scattered across non-contiguous blocks (Figure 8.12), so the attention kernel must instead iterate over the sequence one block at a time, gathering each block's tensors from wherever it happens to sit in physical memory, computing a partial attention score against that block, and accumulating the result. Denote the key block $K_j = (k_{(j-1)B+1}, \ldots, k_{jB})$ and value block $V_j = (v_{(j-1)B+1}, \ldots, v_{jB})$. The attention computation is transformed into the following block-wise computation:

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t \mathbf{1} / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top,$$

where $A_{ij} = (a_{i,(j-1)B+1}, \ldots, a_{i,jB})$ is the row vector of attention scores on the j-th KV block. Intuitively, instead of computing one giant softmax over the entire sequence at once, the query score against each block is computed separately (the numerator), but the denominator still normalizes across all blocks seen so far — so mathematically it's identical to standard attention, just computed piece by piece. In practice, a single GPU warp is assigned to process one block at a time, which keeps memory accesses within a block coalesced even though the blocks themselves are scattered.

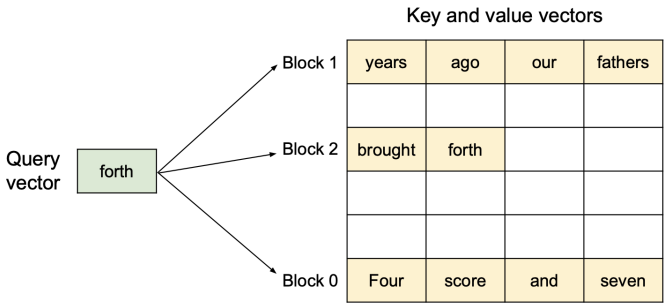


Figure 8.12: PagedAttention algorithm; Image credit: [Kwon et al., 2023]

The indirection between a request's logical view of its KV cache and the physical blocks that

actually hold it is maintained through a block table, directly analogous to a page table in an operating system. Each request has its own sequence of logical blocks, numbered starting from zero, and the block table maps each of these logical block numbers to the physical block in GPU memory that currently stores it, along with how many of the slots in that physical block are filled. Figure 8.13 shows this mapping for a single request — its logical blocks are numbered in order, but the physical blocks they map to can be anywhere in GPU DRAM, since a centralized allocator hands out physical blocks from a shared free pool as they are requested. When multiple requests are batched together, as in Figure 8.14, each request's logical blocks remain contiguous from its own point of view, but the underlying physical blocks for different requests are interleaved arbitrarily throughout the same pool of GPU memory.

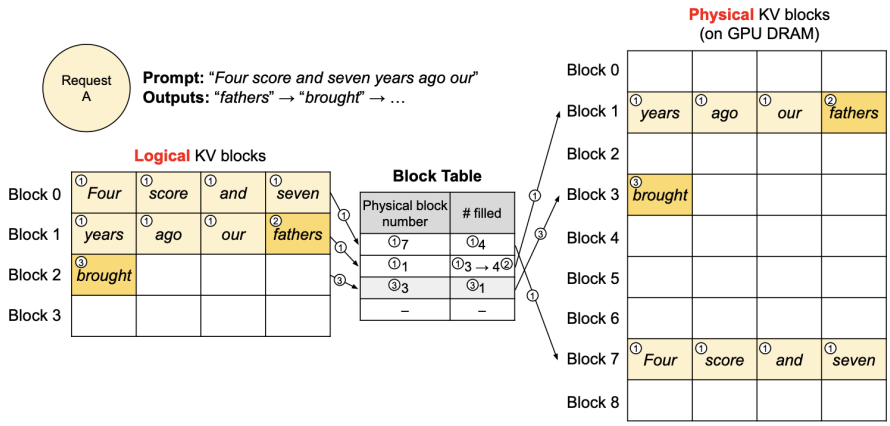


Figure 8.13: Block table translation; Image credit: [Kwon et al., 2023]

Making this scheme work in a real serving system requires a centralized component responsible for handing out and reclaiming physical blocks, since many requests are being scheduled and evicted concurrently and must never be given overlapping physical memory, unless they are deliberately sharing it. Figure 8.15 shows this at a system level. A scheduler decides, at every iteration, which requests to run and how many new tokens each one needs space for, and consults a KV cache manager to allocate space for the required tokens. Recall that every request running on the GPU needs all of its KV cache blocks in GPU memory. So every GPU worker in the system, whether it holds the whole model or only a shard of it under tensor or pipeline parallelism, runs a local cache engine that executes the physical reads, writes, and copies of KV blocks that the scheduler's decisions require.

The KV cache manager in turn delegates the actual bookkeeping of free and used physical blocks to separate allocators for GPU memory and for CPU memory. The separation between GPU and CPU allocators is not incidental. When the GPU runs out of free blocks, one option is to simply

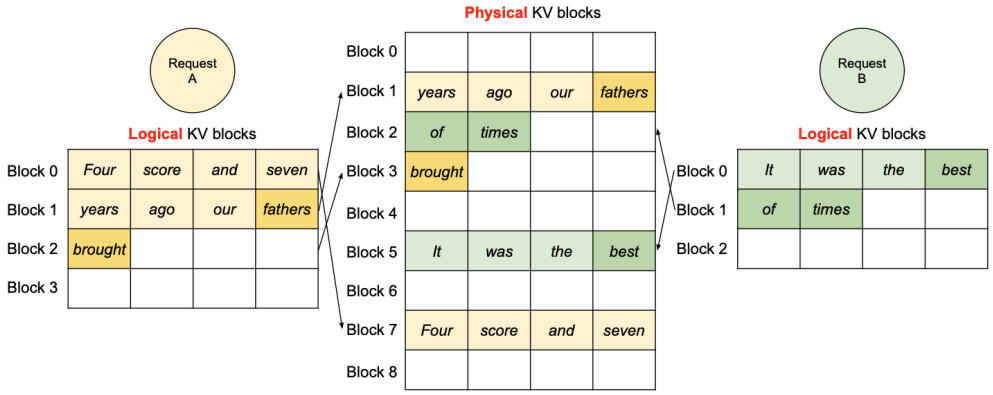


Figure 8.14: Storing the KV cache of two requests; Image credit: [Kwon et al., 2023]

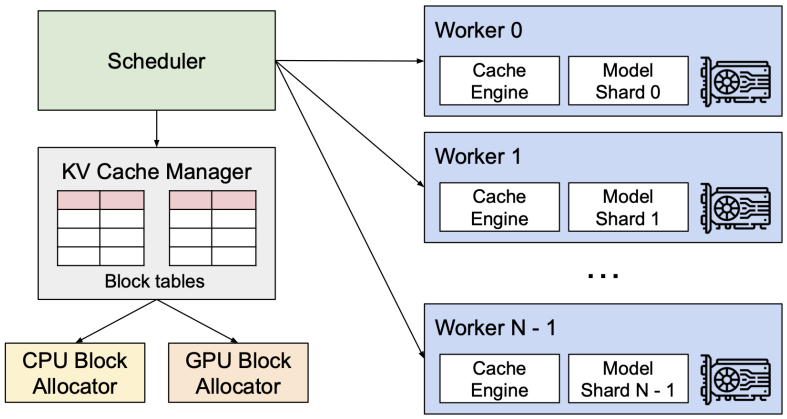


Figure 8.15: vLLM system overview; Image credit: [Kwon et al., 2023]

pause the request that needs a new block until other requests finish and free their blocks. A second option, useful when pausing would waste too much progress, is to swap that request's existing blocks out to CPU memory and swap them back in once GPU space becomes available again, exactly as an operating system would swap pages out to disk under memory pressure. Because blocks are a fixed, uniform size, this copying between GPU and CPU memory is a simple, predictable operation rather than the ad hoc, request-specific copies that contiguous allocation would have required.

Choosing the block size involves a trade-off. Larger blocks amortize the bookkeeping cost of the block table and can improve the efficiency of the block-wise attention kernel, but they also increase the expected internal fragmentation in the last, partially filled block of every request. Smaller blocks reduce that fragmentation, but increase the overhead of maintaining and looking up block tables, and can reduce the effectiveness of memory coalescing inside the attention kernel. Serving systems typically settle on a block size in the range of a few tokens to a few dozen tokens as a reasonable middle ground.

8.2.3 Sharing Memory Across Sequences

Beyond eliminating fragmentation, organizing the KV cache into blocks with a level of indirection between logical and physical addresses opens up a second, largely independent benefit. When multiple sequences being processed share common tokens, the logical KV cache blocks corresponding to the common tokens can point at the very same physical block, and the serving system only needs to allocate physical memory once, while tracking the logical blocks pointing to it. This is exactly the technique operating systems use to let independent processes share pages of physical memory, and it turns out to be extremely useful whenever an LLM serving workload produces multiple sequences that share tokens, some examples of which we will see below.

It is important to note that sharing tokens across sequences necessarily means sharing prefixes and not arbitrary chunks in the middle. The reason follows directly from how self-attention works. Because attention at every layer lets a token look backward at all preceding tokens, the key and value vectors computed for a token depend not just on that token and its position, but on its entire preceding context. Two sequences can therefore only have identical KV entries at a given position if their token histories are identical up to and including that position, and the only way to guarantee this for a contiguous run of positions is for the sequences to share a common prefix. Once the sequences diverge at some point, every token after that point sees a different history through attention, so the corresponding KV entries differ at every layer and can no longer be shared.

The simplest case of sharing is parallel sampling, where a single prompt is used to generate several independent completions, for example by sampling several times at a non-zero temperature to give the user a choice of outputs. Every one of these completions begins with an identical prompt, so their KV caches are identical for as long as the prompt lasts. Rather than duplicating that memory once per sample, the KV cache manager can let every sample's logical blocks for the prompt map to the same physical blocks, and track how many samples are currently sharing each block with a reference count. Figure 8.16 illustrates this with an example — two samples drawn from the same

prompt share every physical block until the token at which their generated text diverges. At that point, the block that used to be identical between them can no longer be shared, since writing the two samples' different next tokens into it would corrupt each other's data. So the manager allocates a fresh copy of that one block for one of the samples and lets the two diverge from there. This copy-on-write strategy means memory is duplicated only for the specific blocks where sequences actually differ, not for the shared history that came before.

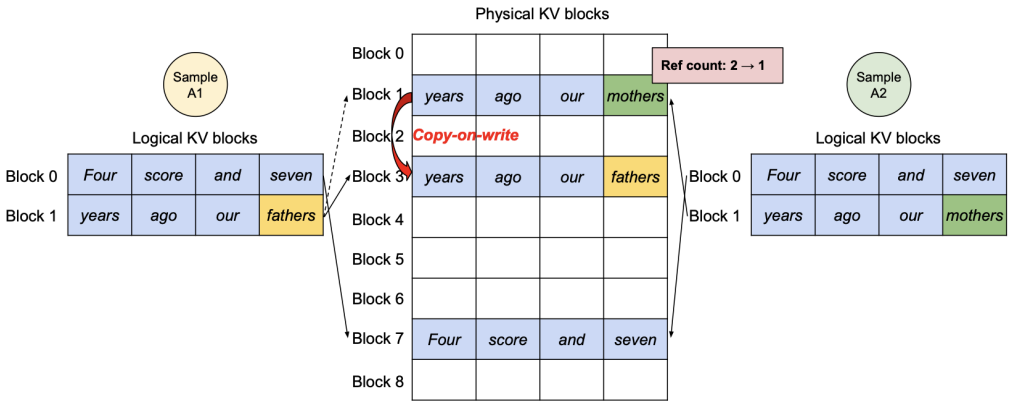


Figure 8.16: vLLM parallel sampling; Image credit: [Kwon et al., 2023]

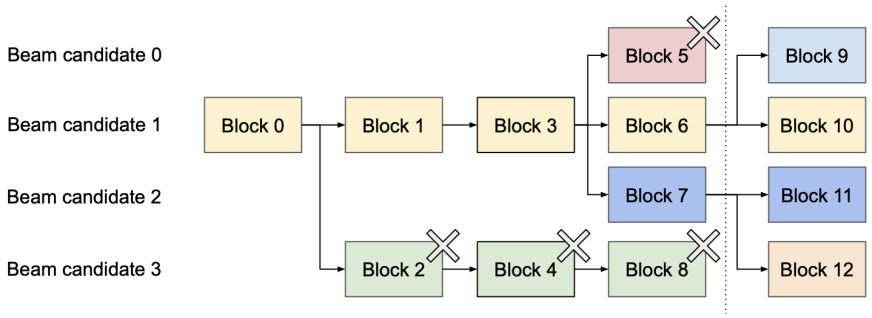


Figure 8.17: vLLM beam search; Image credit: [Kwon et al., 2023]

Beam search benefits from the same mechanism in a more elaborate way. At every step of beam search, several candidate continuations are extended from a shared history, and crucially, most candidates are eventually discarded as the search proceeds, and only the most promising ones survive to be extended further. Earlier serving systems, lacking any way to share memory between beams, had to physically copy a beam's entire KV cache history every time it was extended into

multiple new candidates, an expense that grows with how far the beam search has progressed. With block level sharing, the blocks belonging to a common ancestor of several beam candidates are shared exactly as in parallel sampling, and reference counts tell the system precisely when it is safe to free a block, namely once every beam candidate that could still reach it has been pruned. Figure 8.17 shows several beam candidates branching out of, and eventually pruning back to, a shared tree of blocks.

A third and, for many production workloads, more consequential form of sharing is prefix sharing across entirely different requests that happen to share a common prefix in their input, even though the requests were issued independently and have no logical relationship to each other from the application's point of view. This is common whenever many requests are built from the same template, for example a fixed set of few shot examples followed by a different task input on every request, as in Figure 8.18.

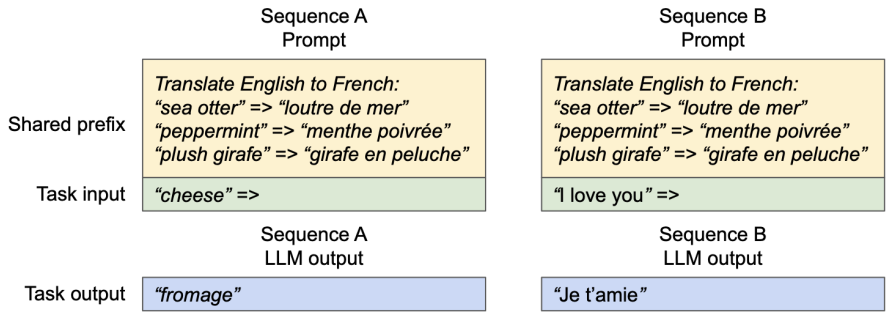


Figure 8.18: Shared prompt example; Image credit: [Kwon et al., 2023]

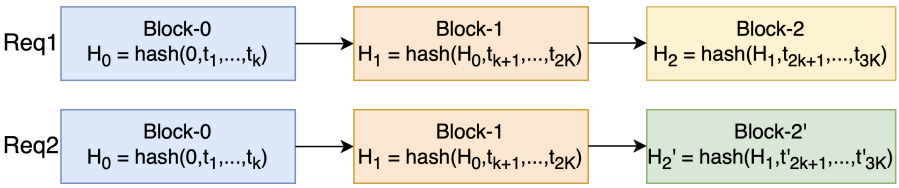


Figure 8.19: Content hashes of each block, chained together across the sequence, let the KV cache manager detect and reuse identical blocks across otherwise unrelated requests

To detect this kind of sharing automatically, the KV cache manager can compute a hash of each block's contents, incorporating the hash of the preceding block so that the hash of a given block also depends on everything that came before it in the sequence (Figure 8.19). When a new request arrives, the manager checks whether a block with a matching hash already exists in the cache; if

it does, the new request's logical KV cache block is simply pointed at the existing physical block instead of triggering fresh computation for it, which means the corresponding tokens can skip prefill entirely. This idea of sharing KV caches corresponding to shared prefixes has since grown from an opportunistic optimization inside a single engine into an important design consideration in its own right for workloads with heavily reused prompts, such as retrieval augmented generation, few shot prompting, and multi-turn conversation.

Practice Problem 8.2

Answer the following questions related to KV cache memory management in LLM inference engines.

(i) Consider the two possible KV-cache layouts in an inference engine like nanonvLLM:

```
kv_cache_A = torch.empty(
    2,
    num_layers,
    num_blocks,
    block_size,
    num_kv_heads,
    head_dim
)
```

```
kv_cache_B = torch.empty(
    2,
    num_layers,
    num_blocks,
    num_kv_heads,
    block_size,
    head_dim
)
```

Here, `num_blocks` is the number of KV cache blocks in the KV cache, each of which has `block_size` number of tokens. The other variables are self-explanatory. Which layout (A or B) is more efficient for attention computation, and why? Assume that different GPU threads parallelize across different attention heads.

Solution: Layout B is more efficient for attention computation. When GPU threads parallelize across attention heads, each thread handles one head's slice of the KV cache. Attention computation requires iterating over all tokens in a block for a fixed head index, i.e., accessing elements across `block_size` for a given `num_kv_heads` index. In layout B, a thread assigned to one attention head reads `block_size` $\times$ `head_dim` elements sequentially, which leads to coalesced accesses.

(ii) Unlike nanovLLM, vLLM maintains a queue of freed KV cache blocks. When serving a new request, blocks are allocated from the front of this queue. Once a request completes, its blocks are returned to the free queue. We can choose to append freed blocks either to the front or to the back of the queue. Why might appending blocks to the front be more beneficial than appending them to the back?

Solution: When a completed request's blocks are pushed to the front, the next incoming request immediately receives those recently-freed blocks. Because these blocks were just actively used, their physical memory pages are likely still warm in the GPU cache, reducing cache misses during the first writes of the new request.

(iii) Continuing from the previous part, can you think of a workload where appending freed-up blocks to the back of the queue (without clearing the contents of the KV cache) would be more advantageous? If so, describe such a scenario.

Solution: Consider a system serving many requests that share a long common prefix (e.g., a system prompt in a chatbot, or a shared document). With prefix caching, KV cache blocks for the common prefix are

shared across requests and should not be overwritten prematurely. If freed blocks are pushed to the front and reallocated greedily, a shared-prefix block that was freed could be quickly reclaimed and overwritten, destroying the cached prefix before the next request can reuse it, which forces recomputation.

By appending to the back, freed blocks are the last to be reallocated. This gives recently-freed, prefix-shared blocks a longer survival window, increasing the probability that an arriving request finds the prefix still intact.

Practice Problem 8.3

Answer the following questions based on vLLM.

(i) Which of the following statements is/are true?

(A) The prefill phase takes the whole user prompt as input, computes the probability of the first new token, and is the memory-bound phase of attention.

(B) The autoregressive generation (decode) phase can only stop when an end-of-sequence token is emitted, or the maximum number of tokens are reached.

(C) For batching, there is no need to pad the inputs and outputs of the requests, even if they vary in their lengths.

Solution: (B)

(A) False. Prefill processes all prompt tokens together and is the compute-bound phase; it is decode that is memory bound.

(B) True. Generation terminates only on an end-of-sequence token or on reaching the maximum token limit.

(C) False. Requests of different lengths generally need padding (or special handling) so they can be batched into regular tensors.

(ii) Which of the following statements is/are true?

(A) In vLLM, memory sharing between different sequences associated with the same request (e.g., during parallel sampling) is possible, but sharing memory across entirely different requests is fundamentally unsupported.

(B) Previous to vLLM, LLM serving systems statically allocated a chunk of memory for a request based on the request's maximum possible sequence length.

(C) The autoregressive generation (decode) phase uses matrix-vector multiplication. As a result, this phase severely underutilizes GPU computation and becomes memory bound.

(D) Under the PagedAttention paradigm, one can conceptualize the blocks as pages, the tokens as bytes, and the requests as processes.

Solution: (B), (C), (D)

(A) False. vLLM can also share blocks across different requests, for example a common prefix such as a shared system prompt.

(B) True. Earlier systems reserved contiguous memory up front for the maximum possible length, wasting

space on tokens that may never be generated.

(C) True. Decode processes one token per request, giving low arithmetic intensity, so the GPU's compute is underused and memory bandwidth becomes the bottleneck.

(D) True. This is the OS analogy used by vLLM — KV blocks are like pages, tokens like bytes, and requests like processes.

(iii) Which of the following statements is/are true?

(A) LLM serving systems prior to vLLM suffer from both internal and external fragmentation.

(B) The PagedAttention algorithm allows the KV blocks to be stored in non-contiguous physical memory.

(C) In vLLM, on GPU workers, a block engine allocates a contiguous chunk of GPU DRAM and divides it into physical KV blocks.

(D) In parallel sampling, users get an option to choose a favourite output from various candidates. In such cases, vLLM enables the sharing of most of the space across multiple output samples, with the exception of the final logical block, which is managed by a copy-on-write mechanism.

Solution: (A), (B), (C), (D)

(A) True. Over-reserving for the maximum length causes internal fragmentation, and differently sized allocations leave unusable gaps between requests (external fragmentation).

(B) True. A block table maps logical blocks to physical blocks that can be anywhere in GPU memory.

(C) True. The block engine reserves one large contiguous region of GPU DRAM and carves it into fixed-size physical KV blocks.

(D) True. Samples share the prompt's blocks; only the last, partially filled block is copied when a sample writes to it (copy-on-write).

(iv) Which of the features are present in the memory management of both modern operating systems and vLLM (assume that we are not considering UVM)?

(A) Logical to physical address translation

(B) Demand paging triggered by access

(C) Copy-on-write

(D) Shared Pages

Solution: (A), (C), (D)

(A) True. vLLM's block tables translate logical KV blocks to physical blocks, just like page tables in an OS.

(B) False. vLLM allocates blocks explicitly as tokens are generated and has no access-triggered page-fault mechanism.

(C) True. vLLM copies a shared block only when one sequence needs to modify it, as an OS does for forked processes.

(D) True. Multiple sequences or requests can map to the same physical blocks, like shared pages between processes.

(v) Which of the following statements is/are true?

(A) Except recomputation, there is no other way to retrieve an evicted KV cache block.

(B) If the block size is too small, vLLM may not fully utilize the GPU's parallelism for reading and processing KV cache. If the block size is too large, external fragmentation increases and the probability of sharing decreases.

(C) The overhead of recomputation remains constant across different block sizes, as recomputation does not utilize the KV blocks. Whereas, swapping incurs excessive overhead with small block sizes.

Solution: (C)

(A) False. Evicted blocks can also be swapped out to CPU memory and brought back later, so recomputation is not the only option.

(B) False. A block size that is too large increases internal (not external) fragmentation, so the statement is incorrect even though the small-block part and the lower sharing probability are reasonable.

(C) True. Recomputation is independent of block size, while swapping many tiny blocks over PCIe incurs heavy overhead.

(vi) Which of the following statements is/are true about paged attention?

(A) The KV cache tensors are stored in discontiguous physical memory on the GPU

(B) The attention computation does not access KV cache tensors in a logically sequential order (from token 1 to N) anymore

(C) The attention code needs to change to work with KV cache tensors in discontiguous blocks rather than in contiguous memory

(D) Paged attention leads to larger batch sizes in general because it reduces the space used by the KV cache

Solution: (A), (C), (D)

(A) True. The KV cache is divided into fixed-size blocks that can be placed anywhere in GPU memory, much like pages in virtual memory.

(B) False. Attention still processes tokens logically from 1 to N; a block table maps logical positions to physical blocks, so only the physical layout is non-sequential.

(C) True. The attention kernel must look up the block table and gather KV data from scattered blocks instead of assuming one contiguous tensor.

(D) True. By avoiding fragmentation and over-allocation, paged attention uses KV memory more efficiently, allowing more requests to fit in a batch.

8.3 Batching and Scheduling

The previous section examined how the KV cache is stored in memory. This section looks at a closely related question: given a stream of incoming requests, how should the serving system decide which requests to run together, and in what order, at each moment in time. As with memory management, the naive answer turns out to waste a great deal of the GPU's capacity.

8.3.1 Request-Level Scheduling and Its Limits

Early transformer serving systems treated a batch of requests the way a traditional deep learning system treats a batch of training examples — a fixed group of requests is assembled, the model is run on all of them together, and the batch is considered complete only once every request in it has finished generating. This is request-level scheduling, and it inherits its simplicity directly from training, where all examples in a batch really do take the same number of steps.

However, generation does not behave this way. Different requests in a batch will finish at different times, since some prompts call for short answers and others for long ones, and the length of a generated sequence is not known in advance. Under request-level scheduling, the whole batch is held hostage by whichever request happens to run the longest — once a shorter request finishes generating its final token, the GPU cannot simply move on to a new request in its place, because the batch as a unit is not yet complete. The slot vacated by the finished request is either left idle or padded, so the GPU spends cycles doing no useful work at all, or work whose output is thrown away. A new request that arrives while a batch is in flight fares no better — it must wait for the entire batch to drain, even if most of the requests already running are close to finishing, which introduces a queuing delay unrelated to the actual amount of computation required. The overall effect is that GPU utilization is determined by the longest request in the batch, and every other request either idles or is padded until that one straggler completes.

8.3.2 Iteration-Level Scheduling in Orca and vLLM

Orca [G.-I. Yu et al., 2022], introduced in 2022, identified request-level scheduling as the source of this inefficiency and proposed replacing it with iteration-level scheduling, sometimes also called continuous batching. The core idea is to move the unit of scheduling from an entire request down to a single iteration of the model, meaning a single token generation step. Instead of committing to a fixed batch of requests until all of them finish, Orca's scheduler makes a fresh decision before every iteration — it looks at the current pool of requests, some of which may have just arrived and some of which may have just produced their final token, and selects whichever set of requests should run for the next single step.

Figure 8.20 illustrates this with Orca's own system diagram. Incoming requests are placed into a request pool rather than directly into a running batch. On every iteration, the scheduler selects a set of requests from this pool, hands their input tokens to the execution engine, and receives back the newly generated tokens once the iteration completes. Because this selection happens fresh on

every single iteration, a request can join the running batch the moment it arrives, without waiting for an existing batch to drain, and a request can leave the running batch the instant it produces an end-of-sequence token, without waiting for its batch-mates to catch up. As shown in the figure, request x_1 has already run two iterations and generated two tokens beyond its original prompt, while request x_3 has only just arrived and has not been scheduled for a single iteration yet, and both can coexist in the pool because the scheduler is free to include or exclude either one independently on the next step.

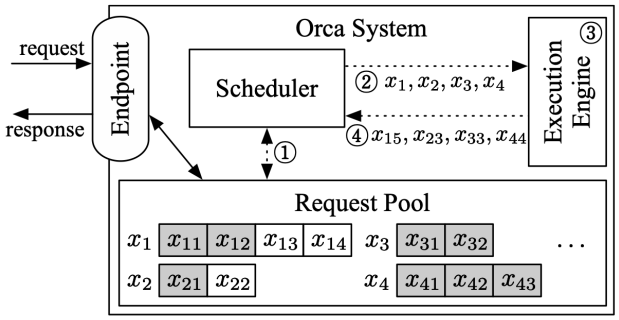


Figure 8.20: System overview of Orca, showing requests scheduled iteration by iteration. Shaded tokens represent input tokens received from the clients, while unshaded tokens are generated by Orca; Image credit: [G.-I. Yu et al., 2022]

This single change removes both of the problems identified above. Since a finished request is retired from the running batch as soon as it completes, its slot is immediately available to a waiting request rather than sitting idle or padded, so the GPU is doing useful work in that slot on the very next iteration. Since a new request can be admitted into the batch on any iteration, it no longer has to wait for an unrelated batch of possibly much longer requests to finish before it is even started. Because requests generally have very different output lengths, and arrive at unpredictable times relative to each other, allowing membership in the batch to change every iteration keeps the batch close to full far more than a scheme that only changes membership once an entire batch has drained.

Making request-level scheduling work efficiently requires one further change beyond scheduling itself. If different requests in a batch are at different points in their sequence, in particular if some are still processing a multi-token prompt during prefill while others are only generating one token at a time during decode, then naively stacking their tensors into a single batched tensor for every operation in the model would require padding shorter sequences up to the length of the longest one, which reintroduces the same wasted computation that iteration-level scheduling was meant to eliminate. Orca addresses this with what it calls selective batching — operations that do not depend on how tokens relate to each other, such as the linear projections and activation functions, are still

applied to all requests as one batched operation for efficiency, since these operations do not care whether the underlying sequences are the same length. Only the attention operation, which does depend on a token's position relative to the rest of its own sequence and must not mix information across different requests, is computed separately per request. This lets Orca batch together requests that are in different phases of generation, and even requests with prompts and outputs of very different lengths, without paying a padding tax on the operations that make up most of the model's computation.

vLLM adopts iteration-level scheduling as well, and builds its scheduling policy directly on top of the PagedAttention block manager described earlier, so that the decision of which requests to run and the decision of where their KV cache lives are made together rather than separately. On every decode iteration, the scheduler first selects the maximum number of requests that can be packed into a batch given the GPU memory available, and then uses first-come-first-served (FCFS) scheduling to pick the requests to be admitted in that batch. Once a set of requests has been chosen for the iteration, the scheduler makes sure every one of them has the physical KV cache blocks it needs in GPU memory. For tokens that are part of the input and have not yet been allocated physical blocks, blocks are allocated on the GPU, and as the iteration produces new tokens for each request, additional physical KV blocks are allocated on demand to hold them as well.

While admitting requests into a batch, the scheduler will eventually run into a situation where GPU memory has no free blocks left to give a request that needs one. At this point, the scheduler can stop the batch formation and dispatch the batch to execute. Further, as requests in a batch generate output tokens and consume GPU memory, the scheduler may not be able to pack all existing requests in the batch for future iterations as well. When this happens, vLLM preempts one or more running requests to free up space, and it has two mechanisms for doing so. One way is to evict the request's KV cache to CPU memory rather than discarding it, so that it can later be swapped back in once GPU space is available again, and generation can resume where it left off. While a request is swapped out, its processing simply halts. Because a partial KV cache is of no use if the request cannot continue generating from it, eviction always takes a request's entire KV cache at once rather than only some of its blocks, and if the evicted request shares any of its blocks with other requests through the reference counted mechanism described earlier, all requests sharing that block are evicted together, since the block cannot be safely freed on the GPU while some requests using it remain resident.

Swapping assumes there is enough CPU memory to receive the evicted KV cache, which is not always true. vLLM has another way of simply discarding the evicted request's KV cache outright and recomputing it later from scratch once the request is rescheduled, trading the cost of a swap for the cost of redoing the prefill computation for that request. To keep memory pressure from getting worse while requests are being evicted, the scheduler also stops admitting new requests into the batch until the preempted requests have completed. Once a request finishes generating entirely, whether because it produced an end-of-sequence token or reached the maximum possible length, all of the physical blocks that were exclusively allocated to it are freed and returned to the shared pool, ready to be handed out to the next request the scheduler admits.

8.3.3 Prefill-Prioritizing and Decode-Prioritizing Schedules

Once a serving system adopts iteration-level scheduling, a new question arises that request-level scheduling never had to answer — on any given iteration, if both new prefills and ongoing decodes are competing for space in the batch, which one should the scheduler favor? A decode-prioritizing schedule (Figure 8.21) keeps the two phases of a request cleanly separated at the level of the whole batch. All the prefills belonging to a batch of requests are computed first, and only once every request in that batch has produced its first token does the scheduler move on to decoding, generating one token per request per iteration, until every request in the batch is finished. A request that arrives while a batch is already decoding is not admitted immediately; it waits in a queue until the current batch has fully drained, at which point it can be included in the next round of prefills. This policy has an appealing property from a point of view of token generation latency — once a request enters its decode phase, nothing else is allowed to interrupt it, so the time between one generated token and the next stays predictable and low. The cost of this predictability is throughput. Requests in the same batch rarely need the same number of decode steps, so as some requests finish generating early, the batch does not make room for new requests; it simply continues running with fewer and fewer active requests until the last one completes. Therefore, GPU cycles that could have gone toward a waiting request are spent on a batch that is only partially full.

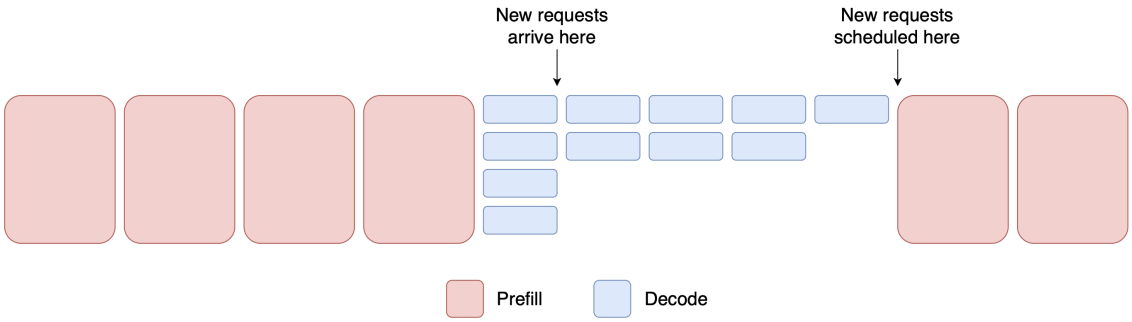


Figure 8.21: Decode-prioritizing schedule: prefills of a batch are all computed up front, decodes then run together until every request finishes, and any request that arrives during decoding must wait for the whole batch to drain

A prefill-prioritizing schedule (Figure 8.22), which is the policy adopted by systems such as vLLM, makes the opposite choice. As soon as a new request arrives and the necessary memory can be allocated for it, its prefill is scheduled at the very next opportunity, ahead of any pending decodes. Only when there is no prefill left to run does the scheduler fall back to batching together whatever decodes are currently in flight. This keeps the pipeline of incoming requests moving quickly and tends to produce larger decode batches once prefills are out of the way. This design choice is good for throughput, but negatively impacts token generation latency. A prefill iteration processes an

entire prompt, which can run into the thousands of tokens for some workloads, and the GPU time required for that iteration scales with the prompt length. Any decode requests that happen to be running when a large prefill is scheduled must wait for that entire prefill to finish before they can produce their next token. This produces a generation stall — a single request with a long prompt can freeze the token generation of every other request in the system for a noticeable, and highly variable, stretch of time. Because prompt lengths are not known in advance and can vary significantly across requests, these stalls are also unpredictable, which makes prefill-prioritizing schedules a poor fit for applications that care about smooth, consistent token delivery.

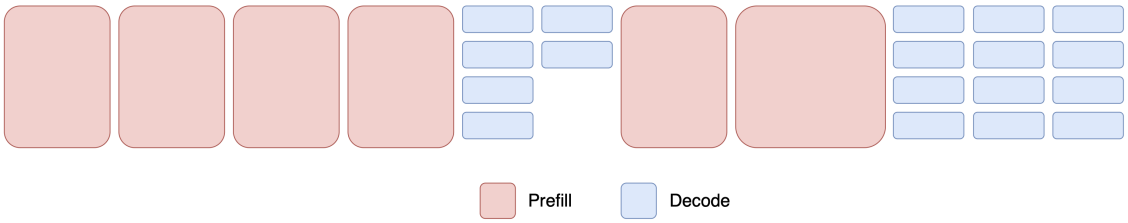


Figure 8.22: Prefill-prioritizing schedule: prefills are scheduled the moment they can be, decodes are batched only in the gaps between them, and prefills can be of variable lengths

The two policies therefore trade-off one performance metric for the other. Decode-prioritizing scheduling optimizes for the time between tokens (TBT) at the cost of throughput and time to first token (TTFT), since new requests cannot join a batch until all previous requests complete. Prefill-prioritizing scheduling optimizes for throughput and TTFT at the cost of TBT, since a long prefill can stall every decode sharing its batch. Neither policy resolves the underlying tension, since both still treat prefill and decode as phases that must be scheduled as whole, indivisible units of work within an iteration.

8.3.4 Chunked Prefills in Sarathi-Serve

Sarathi-Serve [Agrawal, Kedia, Panwar, Mohan, Kwatra, B. Gulavani, et al., 2024] takes a different approach to this tradeoff inherent to prefill or decode prioritizing schedules, by changing the unit of scheduling itself. Rather than treating a prefill as a single, indivisible unit of work that must run in one iteration, Sarathi-Serve splits a prefill into a sequence of roughly equal-sized chunks, each covering only a fraction of the prompt's tokens. This gives the scheduler a much finer-grained unit of prefill work to place alongside decodes, and enables stall-free batching. Instead of choosing between an all-decode batch and an all-prefill batch, the scheduler builds a hybrid batch on every iteration — it first fills the batch with every decode step that is currently owed to an already-running request, and fills up the remaining space in the batch with chunked prefills from requests still in their prefill phase. A chunk should be sized to be small enough to run for a bounded, predictable amount of time, comparable to the time a decode iteration would take, yet should be large enough to keep the

GPU's compute units well utilized. Figure 8.23 shows that how chunks of a new request's prefill are interleaved directly into the batch alongside ongoing decodes over several iterations, so decoding is never paused, leading to stall-free scheduling.

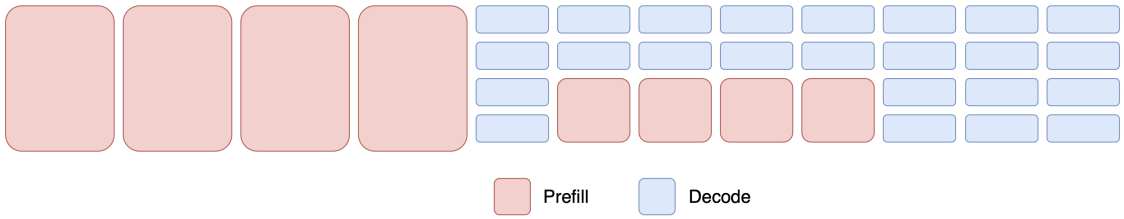


Figure 8.23: Chunked-prefill scheduling, as used in Sarathi-Serve. A prefill is split into equal, compute-sized chunks, and these chunks join the batch alongside ongoing decodes without pausing them.

The batch formation algorithm in Sarathi-Serve relies on a token budget — a cap on the total number of tokens, counting both decode tokens and prefill-chunk tokens, that may be processed in a single iteration. This budget is set once ahead of time based on factors such as the token generation latency target the deployment needs to meet. The scheduling policy of Sarathi-Serve then uses this token budget to add requests to a batch in three phases. First, every request that is already in its decode phase is added to the batch, consuming one token each from the token budget. Second, we add requests whose prefills are partially complete, and the largest chunk of its remaining prompt that can be accommodated is added to the batch. Finally, any leftover space in the token budget is used to admit chunks from new requests that have not yet started their prefill, again sized to fit within whatever budget remains. The following pseudocode summarizes this three-stage logic. Note that the batch size is always limited by the amount of GPU memory available to accommodate the KV caches of all the requests in the batch.

The counter n_t, declared at line 1, tracks how much of the token budget (τ) the current iteration has already committed, and it is reset to zero at the start of every new iteration (line 24), since the budget applies per iteration rather than cumulatively. The first loop (lines 4-7), which walks over every request currently in its decode phase, is where the stall-free property comes from. Every one of these requests contributes exactly one token to the batch and to n_t, with no exceptions and no competition from prefill work at this stage. The second loop (lines 8-12) handles requests whose prefill spans more than one iteration. If a request's prompt was too large to finish prefilling in a single chunk, whatever remains of it is given the next opportunity to advance, ahead of any request that has not started prefilling at all. The chunk size c computed here is capped by however much of the budget is left after decodes have already been accounted for. The third block (lines 13-21), the while loop, is where new work enters the system. It only executes if there is still room left in the budget after decodes and in-progress prefills have taken their share. For each candidate new request,

the scheduler computes the largest chunk of its prompt that fits in the remaining space, and adds it to the batch, and continues this until the token budget completes.

Algorithm 8.1: Stall-free batching (Sarathi-Serve)

Require: Token budget τ, computed from the target TBT SLO

1: $n_t \leftarrow 0$ ▷ Tokens used so far in the current batch
2: $B \leftarrow \emptyset$ ▷ The batch being built for this iteration
3: **while true do**
4: **for** each request R currently in its decode phase **do**
5: add one decode step of R to B
6: $n_t \leftarrow n_t + 1$
7: **end for**
8: **for** each request R with an unfinished prefill **do**
9: $c \leftarrow$ size of the next chunk of R that fits within $\tau - n_t$
10: add a prefill chunk of size c from R to B
11: $n_t \leftarrow n_t + c$
12: **end for**
13: **while** a new, not-yet-started request R_{new} is waiting **and** $n_t < \tau$ **do**
14: $c \leftarrow$ size of the first chunk of R_{new} that fits within $\tau - n_t$
15: **if** $c > 0$ **then**
16: add a prefill chunk of size c from R_{new} to B
17: $n_t \leftarrow n_t + c$
18: **else**
19: **break**
20: **end if**
21: **end while**
22: run one model iteration on the hybrid batch B
23: remove any requests from B that have now finished entirely
24: $n_t \leftarrow 0$, $B \leftarrow \emptyset$
25: **end while**

Once the batch has been assembled through these three stages, it is executed as a single hybrid iteration containing a mixture of decode tokens and prefill-chunk tokens (line 22). After the iteration completes, any request that has now generated its final token is removed from the system (line 23), and the counters are reset before the loop begins again for the next iteration (line 24). Because every request's contribution to a batch, whether a single decode token or a prefill chunk, is bounded by the same token budget, the execution time of every iteration stays close to a fixed SLO target, regardless of how many requests are running or how long their prompts happen to be.

Pipeline Parallelism and the Cost of Uneven Iterations

As we saw in Chapter 6, large models are often also split using pipeline parallelism, where different groups of layers live on different GPUs and a microbatch of requests flows through these groups one after another, like a factory assembly line. Moreover, pipeline parallelism works best when every stage takes roughly the same amount of time to process a microbatch. With iteration-level scheduling, the size of a microbatch's computation is no longer fixed, because a batch can contain any mix of prefill and decode tokens, and this mix can change completely from one iteration to the next. When a slow, prefill-heavy microbatch is followed by a fast, decode-heavy one, the pipeline stages processing the fast microbatch finish early and stall waiting for the slow one to clear the earlier stages, opening up a bubble. Figure 8.24 shows this effect — under Orca's iteration-level scheduling, batches made of varying combinations of prefill and decode work take noticeably different amounts of time to execute, and the resulting gaps between GPU0 and GPU1 are the pipeline bubbles caused by this non-uniformity.

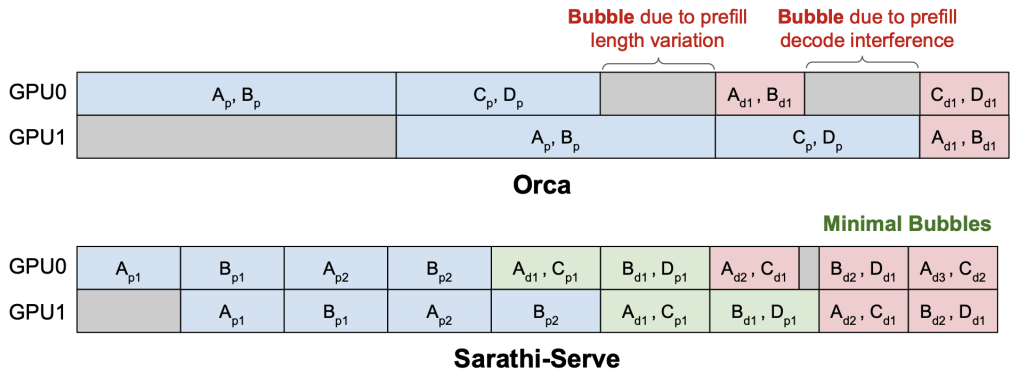


Figure 8.24: A 2-way pipeline-parallel iteration-level schedule across four requests. Orca's batches vary in composition from iteration to iteration, producing visible pipeline bubbles from non-uniform execution times. Sarathi-Serve's uniform-compute batches, built from chunked prefills, keep these bubbles to a minimum; Image credit: [Agrawal, Kedia, Panwar, Mohan, Kwatra, B. Gulavani, et al., 2024]

Sarathi-Serve's token budget, introduced earlier purely as a way to bound decode latency, turns out to solve this problem as well, as a side effect. Because every iteration is built to consume close to the same number of tokens, whether those tokens come from decodes, an in-progress prefill chunk, or a mix of both, the compute cost of consecutive iterations stays close to uniform. A stage that finishes a token-budget-sized iteration can expect its neighboring stages to finish their own iterations in about the same amount of time as well, so the pipeline drains smoothly with little bubble time.

Hybrid Batches and the Compute-Memory Balance

Chunked prefills give the scheduler a second lever beyond just avoiding pipeline bubbles — the chunk size can be tuned to balance how compute-bound or memory-bound a batch is. Recall that decode iterations are memory-bound, since each request contributes only a single query token and the dominant cost is streaming the KV cache and model weights from HBM, while prefill iterations are compute-bound, since a long sequence of tokens gives the GPU plenty of parallel work per byte of memory traffic that is fetched. This balance can be described using arithmetic intensity, which as we saw in Chapter 2, is defined as the number of floating point operations performed per byte read from memory. Figure 8.25 plots arithmetic intensity for the linear operations of a large model as a function of the number of tokens processed together. Decode-only batches sit at very low arithmetic intensity. However, prefill batches with many tokens sit at the opposite end, with high arithmetic intensity.

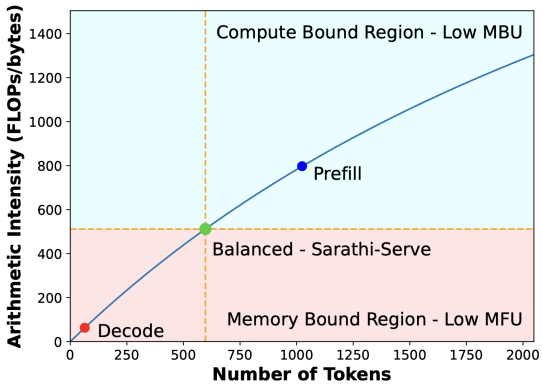


Figure 8.25: Arithmetic intensity of the linear operations of a large model as the number of tokens in a batch grow; Image credit: [Agrawal, Kedia, Panwar, Mohan, Kwatra, B. Gulavani, et al., 2024]

A hybrid batch of decodes and chunked prefill gives the scheduler a way to land somewhere between these two extremes. Because decodes and the prefill chunk are processed together through the same linear operations, the two workloads effectively borrow the resource that the other has spare capacity in. This is why the chunk size is not chosen arbitrarily. A chunk that is too small leaves the batch memory-bound and wastes the GPU's compute units, much like a pure decode batch, while a chunk that is too large pushes the batch toward being compute-bound, reintroducing the very generation stalls that chunking was meant to avoid, and also increases time-between-tokens beyond what the latency target allows. In practice, the chunk size, and by extension the token budget, is chosen to sit near the balanced point in Figure 8.25, close to where the curve crosses from memory-bound into compute-bound territory, so that a hybrid batch makes reasonably good use of both compute and memory bandwidth at once while still respecting the decode latency target.

Issues with Hybrid Batching at the Attention Layer

Everything described so far about hybrid batches, chunk sizes, and arithmetic intensity applies well to the linear operations of the model — the projections, the feed-forward network, and the various normalization layers. These operations are linear in the sense that applying them to a combined batch of prefill and decode tokens gives the same result as applying them separately and concatenating the outputs, so a serving system can stack prefill and decode tokens into one tensor, run them through the same weights in a single kernel call, and split the results apart afterward. This is exactly what makes hybrid batching effective for linear operations — the model weights are streamed from HBM once per iteration and reused across every token in the batch, prefill and decode alike.

Attention does not share this property. Unlike a linear layer, attention is computed separately for every sequence — each query token needs to look back at the key and value vectors of its own sequence, and mixing tokens from different requests or different sequences together inside the same attention computation would be incorrect. As a result, even inside a hybrid batch, current serving systems fall back to computing prefill attention and decode attention as two separate kernel calls, using two differently specialized kernels. This does work correctly, but it throws away the resource-balancing benefit that hybrid batching achieves everywhere else in the model. Figure 8.26 shows why — the prefill attention kernel, much like the linear operations of a prefill-heavy batch, is compute-bound, reaching high compute utilization at long context lengths while barely touching memory bandwidth. The decode attention kernel is the mirror image, reaching high memory bandwidth utilization as batch size grows while its compute utilization stays low. Running these two operations as separate kernels does not always lead to a balanced utilization of GPU compute and memory bandwidth. This inefficiency becomes harder to ignore as context lengths continue to grow, because attention consumes a significant part of the execution time of the model at larger context lengths.

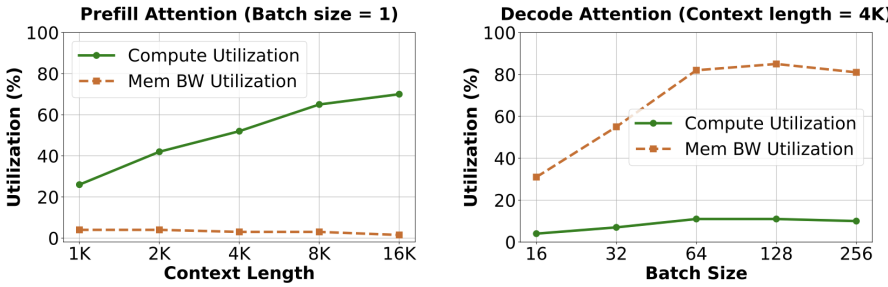


Figure 8.26: Compute and memory bandwidth utilization of prefill attention (left, batch size 1, varying context length) and decode attention (right, context length 4K, varying batch size); Image credit: [Kamath et al., 2025]

Issues with Chunking: Repeated Memory Accesses

Splitting a prefill into chunks solves the scheduling problem described above, but it does not come for free — it introduces a memory-traffic cost that an unchunked prefill never pays. The source of this cost is causal attention itself. When a prompt is prefilled in a single iteration, the attention computation for every query token reads each key and value vector from HBM exactly once, since all tokens are processed together and the keys and values are loaded into the kernel a single time. Chunking breaks this property. For the i-th chunk of a prompt, the query tokens of that chunk must attend not only to the keys and values of the current chunk, but also to $K_{<i}$ and $V_{<i}$, the keys and values of every chunk that has already been processed. Because each chunk runs in its own iteration, and nothing about the GPU's execution model guarantees that a previous iteration's tensors are resident in on-chip memory for the next one, computing chunk i's attention means reading $K_{<i}$ and $V_{<i}$ back from HBM again, even though those exact same tensors were already streamed in once for chunk $i-1$'s attention, and once again before that, all the way back to the prompt's very first chunk.

The result is that the total number of bytes read for a single prompt's own KV cache over the course of its prefill grows roughly quadratically with the number of chunks it is split into. This is a direct, unavoidable consequence of the design choice that makes chunked prefill effective in the first place — bounding the amount of work in an iteration by a token budget necessarily means a long prompt's context gets re-read many times over, once per chunk that follows it. It also means the choice of chunk size is not just a latency knob. Shrinking the token budget to tighten control over TBT increases the number of chunks a long prompt is split into, and therefore increases how many times its growing KV cache prefix gets reloaded. So a serving system that chases a very small TBT target can end up spending a disproportionate share of its memory bandwidth simply re-reading context it has already processed for the same request, rather than making forward progress.

8.3.5 Fusing Prefill and Decode Attention in POD-Attention

POD-Attention [Kamath et al., 2025] starts from a simple observation — if a hybrid batch already contains both compute-heavy prefill work and memory-heavy decode work, then the attention computation for that batch should be able to use both GPU resources simultaneously too, the same way the linear operations already do. The obstacle is that prefill attention and decode attention are two different specialized kernels, and simply launching both of them does not guarantee that they will actually execute at the same time on the same parts of the GPU, thereby balancing the consumption of compute and memory resources.

Why Simply Launching Two Kernels Is Not Enough?

To see why, it helps to recall how a GPU scheduler works. A GPU kernel is launched as a large number of cooperative thread arrays (CTAs), or blocks, where each CTA is a group of threads guaranteed to run together on a single SM, sharing that SM's L1 cache and shared memory. A

modern GPU has around a hundred SMs, and a hardware CTA scheduler assigns queued CTAs to SMs as space becomes available. Kernels are submitted through streams, and while operations within one stream execute in order, operations in different streams can, in principle, run concurrently on the GPU. So the most direct way to overlap prefill and decode attention would be to submit each to its own stream and let the hardware interleave them. In practice this provides only a modest benefit, because streams offer no guarantee that CTAs from the two kernels will actually be co-located on the same SM at the same time; the hardware scheduler is free to run one kernel's CTAs on one set of SMs and the other kernel's CTAs on a different set, or to run them one after another if resources are tight. Without genuine co-location on the same SM, a prefill CTA and a decode CTA cannot actually share that SM's compute units and memory pipelines concurrently, so the desired overlap of compute-heavy and memory-heavy work does not materialize.

Complexities of Fused Kernels

On the other hand, splitting the work by warp, so that some warps in a CTA run prefill code and others run decode code, does guarantee co-location, since all warps of a CTA are on the same SM by construction, but it suffers from a straggler effect — a CTA cannot be replaced by fresh work until every warp inside it finishes, so if one warp's operation runs long, the warps assigned to the other operation sit idle waiting for it. Splitting the work at the level of individual threads, so each thread alternates between prefill and decode instructions, avoids the straggler problem but runs into synchronization barriers — attention kernels rely on barriers to coordinate loading tiles of data into shared memory, and instructions before such a barrier cannot be overlapped with instructions after it, which limits how much overlap this style of fusion can actually achieve in practice. The above two ways of overlap at the warp and thread level also require developing specialized fused kernels, and cannot reuse already optimized kernels for prefill and decode attention.

CTA-Parallel Fusion and SM-Aware Scheduling

POD-Attention solves this problem by fusing prefill and decode attention along the CTA dimension, while adding a mechanism to guarantee that prefill and decode CTAs are actually co-located on the same SM. The kernel is launched with a single combined set of CTAs, some of which are destined to perform prefill work and some of which are destined to perform decode work. However, rather than deciding this split in advance at kernel launch time, each CTA decides for itself, at runtime, whether it should run prefill attention or decode attention, once the hardware has already placed it on a specific SM. This is what the authors call SM-aware CTA scheduling, and it works because a CTA can query which physical SM it has been assigned to and then coordinate with the other CTAs already running on that same SM to keep the mix of prefill and decode work balanced.

Every SM maintains a counter that tracks how many CTAs have already landed on it, along with two further counters tracking how many of those CTAs have been assigned to prefill and how many to decode. When a new CTA starts executing, its first thread reads a special hardware register that reveals which SM it is running on, then atomically increments that SM's counter to obtain a

ticket. The value of the ticket, taken together with the desired ratio of prefill to decode work for the current batch, tells the CTA which operation it should perform. This is what makes the scheme SM-aware: rather than fixing the assignment of CTAs to operations ahead of time, the decision is made adaptively, per SM, based on what the hardware scheduler has actually done. This allows POD-Attention to work in the desired manner no matter how the hardware chooses to distribute CTAs across SMs. Figure 8.27 illustrates this with three SMs: two of them already have a mix of prefill and decode CTAs, while a CTA newly arriving on SM 1 consults the shared counters to decide which operation it should carry out.

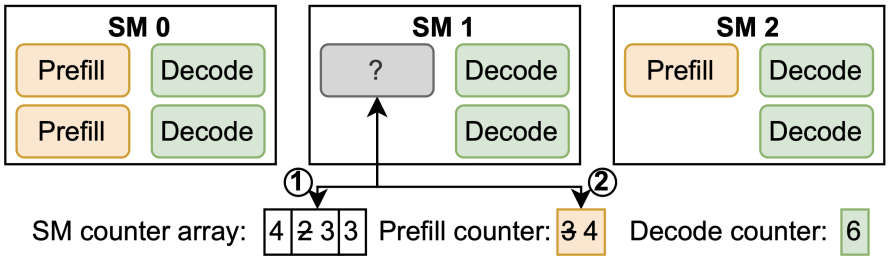


Figure 8.27: SM-aware CTA scheduling; Image credit: [Kamath et al., 2025]

The logic described above is implemented directly in the CTA's own code, shown below, and runs once at the very start of every CTA's execution, before any actual attention computation begins.

Only a single thread in the CTA, the leader thread, carries out the scheduling decision; this is checked at line 1, since there is no need for every thread to repeat the same bookkeeping. The leader begins by reading the SM identifier hardware register at line 3, a low-level instruction that reveals the physical SM this CTA has landed on, something the CUDA programming model does not normally expose directly. Lines 5 through 8 turn this into an operation assignment. The ratio on line 5 adds together the desired number of prefill and decode CTAs for the current batch, and performs an atomic increment on that SM's own counter, `sm_ctr`, to obtain a ticket, taken modulo the ratio, so that tickets cycle repeatedly through the same range as more CTAs arrive on this SM over time. If the ticket falls below the prefill ratio, the CTA is provisionally assigned to prefill, and otherwise to decode, on lines 7 and 9. Because the atomic increment hands out a fresh, unique ticket to every CTA that lands on this SM, this scheme naturally spreads prefill and decode assignments across the SM's CTAs in proportion to the requested ratio, without the CTAs needing to communicate directly.

A provisional assignment is not the final word, however, because the batch has only a limited, precomputed number of CTAs worth of prefill and decode work available overall. Line 10 obtains a global CTA id for the assigned operation by atomically incrementing a global counter, `cta_assign`, shared across all SMs. Lines 12 through 18 handle the case where this global counter has run out of work for the assigned operation. Here, `prefill_ctas` or `decode_ctas` is the maximum number of CTAs that operation actually needs. If a CTA is told to do prefill but every prefill CTA already has

an assignment, it switches to decode instead and obtains a fresh id from the decode counter, and symmetrically if it was told to do decode but decode work has already been exhausted. This ensures that no CTA sits idle simply because its first-choice operation happened to run out of work while the other operation still has some left, so the kernel always keeps making progress.

Algorithm 8.2: SM-Aware CTA Scheduling

```
if threadIdx.x == 0 then                                   ▷ Leader thread finds assignment
    int sm_id;                                             ▷ Find which SM this CTA is on
    asm volatile("mov.u32 %0, %smid;" :  "=r"(sm_id));     ▷ For this SM, what do we want to run?
    const int ratio = (prefill_ratio + decode_ratio);
    int op, ticket = (atomicAdd(&sm_ctr[sm_id], 1) % ratio);
    if ticket < prefill_ratio then
        op = PREFILL;
    else
        op = DECODE;
    end if
    int cta_id = atomicAdd(&cta_assign[op], 1);            ▷ Get the next CTA for operation
    if op == PREFILL and cta_id >= prefill_ctas then       ▷ If the CTA exceeds the max CTA for that op, switch ops
        op = DECODE;
        cta_id = atomicAdd(&cta_assign[op], 1);
    else if op == DECODE and cta_id >= decode_ctas then
        op = PREFILL;
        cta_id = atomicAdd(&cta_assign[op], 1);
    end if                                                 ▷ Write the CTA ID and operation to shared memory
    shared_mem[0] = cta_id;
    shared_mem[1] = op;
end if
__syncthreads();                                           ▷ Barrier: waits for scheduling to finish
int cta_id = shared_mem[0];                                ▷ Fetch the assigned CTA and operation
const int op = shared_mem[1];
__syncthreads();                                           ▷ Perform the appropriate operation
if op == PREFILL then
    prefill_op(cta_id);
else
    decode_op(cta_id);
```

30: **end if**

Once the leader thread has settled on a final operation and CTA id, it writes both values into shared memory at lines 19 and 20, since shared memory is the mechanism through which the other threads in the CTA, who did not participate in this scheduling logic, can learn what was decided. The barrier at line 22 is essential here: every other thread in the CTA must wait until the leader thread has actually finished writing to shared memory before reading from it, otherwise a thread might read a stale or partially written value. Lines 23 and 24 are where every thread reads back the finalized `cta_id` and `op` from shared memory, followed by a second barrier which ensures all threads have finished reading before shared memory is reused later in the kernel for other purposes. Finally, lines 27 and 29 dispatch to the actual attention computation: the CTA calls into the prefill computation function or the decode computation function as assigned.

The net effect of this piece of code, running once per CTA before any real work begins, is that a single kernel launch, containing CTAs destined for both prefill and decode, dynamically sorts itself out so that every SM ends up running a healthy mix of both operations, regardless of the order in which the hardware scheduler happened to dispatch CTAs to that SM. Because prefill and decode now genuinely execute side by side within the same SM, the compute-heavy tensor core work of prefill and the memory-heavy data movement of decode can proceed at the same time, letting a single fused kernel make good use of both compute and memory bandwidth. Also note that this fusion of prefill and decode has been achieved by reusing existing implementations, rather than writing a fused kernel from scratch.

8.3.6 Disaggregating Prefill and Decode

Chunked prefill and POD-Attention both start from the premise that prefill and decode share the same GPU, and then work to make that sharing as seamless as possible. Some recent work like DistServe [Zhong et al., 2024] questions the premise itself, and argues that prefill and decode are computationally so different that co-locating them on the same GPU leads to too much interference. The solution then is to run the two phases on physically separate GPUs, called the prefill instance and the decoding instance, and transfer only the resulting KV cache between them.

What are the costs to co-locating prefill and decode? An LLM service is judged by two latency metrics — time to first token (TTFT), and time per output token (TPOT). A serving system's goal is to maximize its per-GPU goodput, defined as the highest request rate it can sustain while still meeting both the TTFT and TPOT SLOs for a target fraction of requests, since higher goodput per GPU means lower cost per query. Co-locating prefill and decode directly threatens this goal because a prefill step is far more expensive per token than a decode step, so batching them together lets prefill starve decode of GPU time. There is also a second, subtler cost to co-location beyond interference — it forces prefill and decode to share the same resource allocation and the same parallelism strategy, even though their preferences differ. A compute-bound prefill benefits from more GPUs and from intra-operator

parallelism, which shortens execution time and therefore TTFT, at the cost of communication between GPUs. A memory-bandwidth-bound decode benefits far more from a large batch running on one GPU, rather than from splitting a single request's computation across GPUs. When both phases are forced onto the same GPU, the system can only pick one compromise configuration, tuned to whichever of TTFT or TPOT is more stringent, and the other metric suffers, or the deployment is over-provisioned with spare GPUs to compensate.

DistServe solves both problems by giving each phase its own instance, where an instance is defined as a set of GPUs holding one full copy of the model weights, possibly parallelized across several GPUs. A prefill instance receives a request, runs only the prefill computation to produce the first token, and then hands off its KV cache to a decoding instance, which carries out every subsequent decoding step. Figure 8.28 shows the resulting runtime. A controller routes each incoming request to a prefill instance, and the prefill instance's result is transferred to a decoding instance. The two instance types can be scaled independently, typically with more prefill instances feeding fewer decoding instances, because of lower GPU resource consumption by decodes. Separating the phases also makes it easier to pick a good parallelism strategy for each one, since the two choices no longer have to be the same.

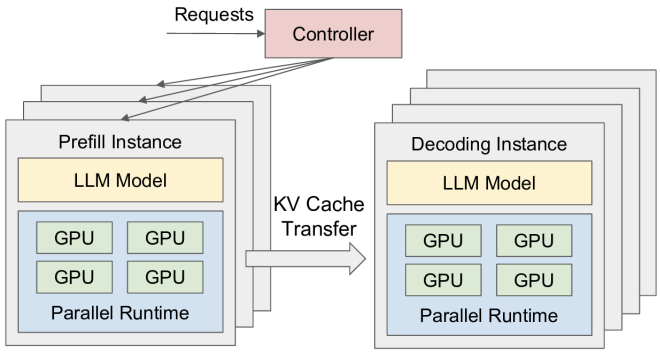


Figure 8.28: DistServe's runtime architecture; Image credit: [Zhong et al., 2024]

Note that disaggregation is only worthwhile if moving the KV cache from the prefill instance to the decoding instance is cheap relative to the savings from eliminating interference, which holds true in practice. For a 512-token request on a 66B-parameter model, the KV cache is only on the order of a gigabyte, and even at ten such requests arriving per second, streaming this data continuously needs well under 100 Gbps of bandwidth, comfortably inside what modern GPU clusters provide over Infiniband, or inside a single node over NVLink. Because arrivals are bursty rather than perfectly smooth, DistServe has the decoding instance pull the KV cache from the prefill instance as decode capacity becomes available, rather than having the prefill instance push it out immediately, using the prefill instance's own memory as a buffer. This keeps a sudden burst of arriving requests from

flooding the decoding instance's memory with KV caches that it is not yet ready to use, letting each instance operate at its own pace.

Practice Problem 8.4

Answer the following questions based on LLM inference engines.

(i) The pseudo-code below shows a scheduling policy using chunked prefill.

Algorithm 8.3: SCHEDULE()

```
τ ← compute_token_budget()
n_t ← 0
B ← ϕ
while true do
    for R in B do
        if is_prefill_complete(R) then
            n_t ← n_t + 1
        end if
    end for
    for R in B do
        if not is_prefill_complete(R) then
            c ← get_next_chunk_size(R, τ, n_t)
            n_t ← n_t + c
        end if
    end for
    R_new ← get_next_request()
    while can_allocate_request(R_new) AND n_t < τ do
        c ← get_next_chunk_size(R_new, τ, n_t)
        n_t ← n_t + c
        B ← B ∪ R_new
    end while
    process_batch(B)
    B ← filter_finished_requests(B)
    n_t ← 0
end while
```

Is the above schedule prefill-prioritizing or decode-prioritizing? Explain your answer briefly.

Solution: The schedule is decode-prioritizing, because in each iteration we first consider all the active decode requests, and then move to prefill requests.

(ii) Which of the following options is/are correct?

(A) Increasing chunk size always eliminates tile-quantization effects.

(B) Increasing the token budget leads to higher compute utilization, but a worse time-between-tokens (TBT).

(C) During the computation of chunked-prefills, the attention operation for every chunk of a prompt needs to access the KV-cache of all prior chunks of the same prompt.

(D) The schedule given in the previous part of the question may allow new requests to join a running batch without pausing ongoing decodes.

Solution: (B), (C), (D)

(A) Tile-quantization arises when workload dimensions are not exact multiples of the hardware tile size. Misalignment can persist for any chunk size that is not a multiple of the tile dimension.

(B) A larger token budget batches more tokens per step, increasing compute utilization.

(C) Attention is inherently a global operation over all previous context. When processing chunk k, the self-attention computation must attend over the KV cache produced by chunks $1, \ldots, k-1$.

(D) Justified, by lines 17-21 of the pseudo-code, which allow new requests, without pausing ongoing decodes.

(iii) Consider a case where the token budget τ is fixed at 120, the batch B has 100 decodes active, and there is a single prefill of length 2100 tokens (not started its execution yet). What is the largest chunk size that can be returned by the function get_next_chunk_size? For this chunk size, how many iterations will be required to compute the prefill? Assume that the decodes remain active for a long enough time.

Solution: After the first 100 decode tokens, we have $n_t = 100$. The remaining budget available for the prefill chunk is 20 tokens. Hence the largest chunk size is 20 tokens. For a prompt of 2100 tokens, the number of iterations required is 105.

(iv) For the above request of 2100 tokens that is processed via chunked prefill, how many times are the KV cache tokens of the first chunk loaded from GPU main memory into the GPU caches, over the duration of the entire prefill? Assume that the GPU caches can hold the KV cache tokens of only one previous chunk at a time while computing attention of subsequent chunks.

Solution: When computing attention for chunk k (for $k = 2, 3, \ldots, 105$), the KV cache of each prior chunk must be loaded from GPU main memory (since only one prior chunk fits in cache at a time). Therefore, the KV cache of the first chunk is loaded 104 times. With correct assumptions, 103 will also be accepted.

(v) Continuing with the previous question, how many KV cache tokens corresponding to all the chunks of the request are loaded from GPU main memory into the GPU caches, over the duration of the entire prefill?

Solution: With 105 chunks in total, chunk i (for $i = 1, \ldots, 104$) must be loaded from main memory once for each of the $(105 - i)$ subsequent chunks. The total number of KV-cache chunk loads is therefore $\sum_{i=1}^{104}(105 - i) = 5460$. Since each chunk contains 20 KV-cache tokens, the total number of KV-cache token loads is $5460 \times 20 = 109{,}200$ tokens.

Practice Problem 8.5

Answer the following questions based on Sarathi-Serve.

(i) Which of the following statements is/are true?

(A) Decodes benefit significantly from batching because larger batches can use GPUs more efficiently whereas prefills do not benefit much from batching.

(B) Decode-prioritizing scheduling faces higher TBT (time between tokens) latency as compared to prefill-prioritizing scheduling.

(C) Prefill-prioritizing schedulers have better throughput because computing prefills first allows subsequent decodes to operate at high batch sizes.

(D) Stall-free scheduling allows new requests to join a running batch without pausing ongoing decodes.

(E) With iteration level batching used by vLLM and related systems, a new request can be added to the batch after an iteration, and there is no need to wait for the entire batch to complete.

(F) FCFS scheduling policy ensures fairness and prevents starvation and is the default scheduling policy in vLLM.

Solution: (A), (C), (D), (E), (F)

(A) True. Decodes are memory bound, so batching amortizes weight loading and raises utilization, whereas a prefill already has enough tokens to saturate the GPU.

(B) False. Decode-prioritizing scheduling keeps decodes running, so it has lower TBT; it is prefill-prioritizing scheduling that causes generation stalls and higher TBT.

(C) True. Running prefills first builds up a large pool of requests, so later decode iterations run at high batch sizes and give better throughput.

(D) True. Stall-free batching adds new (chunked) prefill work into the batches of ongoing decodes, so decodes are never paused.

(E) True. Iteration-level scheduling lets finished requests leave and new ones join after every iteration, without waiting for the whole batch to finish.

(F) True. Serving requests in arrival order is fair and starvation-free, and it is vLLM's default policy.

(ii) Which of the following statements is/are true about multi-GPU LLM inference?

(A) Pipeline Parallelism (PP) shards each layer across the participating GPUs by splitting the model weights and KV-cache equally across GPU workers.

(B) Tensor Parallelism (TP) linearly scales per-GPU batch size but incurs high communication costs due to two expensive AllReduce operations per transformer layer, making it less ideal for cross-node commodity networks.

(C) Pipeline bubbles in LLM inference often occur due to the wildly varying compute times of prefill and decode stages when they are interleaved, creating execution gaps between consecutive micro-batches.

(D) Sarathi-Serve's uniform compute hybrid batches significantly mitigate pipeline bubbles in PP deployments by standardizing the execution time and compute requirements across different micro-batches.

Solution: (B), (C), (D)

(A) False. PP assigns different groups of layers to different GPUs; sharding each layer across GPUs is tensor parallelism, not pipeline parallelism.

(B) True. TP splits weights and KV cache per layer, leaving room for larger batches, but needs two AllReduce per layer, which is costly over slower cross-node links.

(C) True. When micro-batches have very different execution times (long prefills versus short decodes), later pipeline stages sit idle, creating bubbles.

(D) True. Hybrid batches with a fixed token budget have near-uniform compute, so micro-batch execution times match and bubbles shrink.

(iii) Sarathi-Serve requires operators to determine a "token budget" that limits the number of prefill tokens processed in a single iteration. Which of the following statements accurately describe the trade-offs of tuning this token budget?

(A) Reducing token budget always reduces end-to-end latency since it minimizes prefill computation per iteration.

(B) Increasing chunk size always improves GPU utilization and eliminates tile-quantization effects, leading to optimal performance.

(C) During the computation of chunked-prefills, the attention operation for every chunk of a prompt needs to access the KV-cache of all prior chunks of the same prompt.

(D) When latency constraints are relaxed, Sarathi-Serve can utilize a larger token budget to operate more efficiently and achieve higher overall capacity.

Solution: (C), (D)

(A) False. A smaller budget splits a prompt into more chunks, which adds overhead (repeated KV-cache reads, lower efficiency) and can increase end-to-end latency.

(B) False. Larger chunks may improve utilization but increase iteration time (hurting TBT), and tile-quantization effects depend on the chunk size relative to the tile size, so they are not eliminated automatically.

(C) True. Each chunk's tokens must attend to all earlier tokens of the prompt, so the KV cache of the previous chunks must be read.

(D) True. With relaxed TBT limits, a larger budget means fewer chunks and better GPU efficiency, which gives higher capacity.

(iv) Which properties are ensured by stall-free batching in Sarathi-Serve?

(A) Decodes are never delayed due to prefills.

(B) Prefills are executed only after all decodes complete.

(C) Hybrid batches always contain equal numbers of prefill and decode tokens.

(D) It leverages unused compute capacity in decode iterations.

Solution: (A), (D)

(A) True. Ongoing decodes are always included in each batch, and prefill work is limited by the token budget, so decodes do not stall behind prefills.

(B) False. Prefill chunks are packed onto decode batches rather than waiting for all decodes to finish.

(C) False. The prefill portion is whatever fits in the remaining token budget, so the mix of prefill and decode tokens varies.

(D) True. Decode iterations leave compute capacity idle, and adding prefill chunks to them uses that slack.

(v) Which of the following factors directly contribute to Sarathi-Serve's higher serving capacity?

(A) Elimination of generation stalls caused by prefill interference.

(B) Ability to increase batch size beyond what PagedAttention allows.

(C) Reduction in pipeline bubbles via uniform-compute batches.

(D) Better utilization of decode-phase slack via hybrid batching.

Solution: (A), (C), (D)

(A) True. Removing stalls keeps TBT low, so more requests can be served within the latency SLO.

(B) False. Sarathi-Serve is a scheduling technique and does not change KV-cache memory limits; the maximum batch size is still bounded by memory management such as PagedAttention.

(C) True. Uniform-compute batches reduce idle time across pipeline stages, which raises throughput in PP deployments.

(D) True. Filling the spare compute in decode iterations with prefill increases the work done per iteration.

8.4 Prefix Sharing

The scheduling policies discussed so far all assume that every request's KV cache is a private resource, computed from scratch when the request arrives and discarded once it finishes. For many real workloads this assumption throws away a great deal of reusable work. A chatbot typically prepends the same system prompt to every conversation. A few-shot pipeline sends the same set of demonstration examples ahead of a different question each time. A multi-turn conversation resends the entire prior chat history with every new turn, differing from the previous request only in the newest message. In all of these cases, a substantial prefix of the input tokens is identical across many requests, yet a serving system without prefix sharing recomputes the KV cache for that shared prefix again and again, once per request, even though the result is exactly the same every time.

SGLang [Zheng et al., 2024] introduces RadixAttention to address this waste. The idea is to stop treating a request's KV cache as something private that disappears the moment the request finishes, and instead treat it as an entry in a shared cache that persists across requests. When a new request arrives, the runtime checks how much of its prompt already has a matching KV cache computed by some earlier request, and only the portion of the prompt that is new undergoes a prefill. The shared prefix reuses the KV cache tensors that are already in GPU memory, thereby saving the cost of recomputation. Because this matching happens automatically at runtime, it requires no changes to how an application constructs its prompts, and it works whether the sharing comes from a system prompt, a few-shot prefix, a chat history, or a set of branches explored during a reasoning workflow.

8.4.1 The Radix Tree

Making this reuse efficient requires a data structure that can answer one question quickly for every incoming request — what is the longest prefix of this request's tokens that the cache already has stored, and where does that stored data live? SGLang answers this with a radix tree, a compressed prefix tree in which an edge is not restricted to representing a single token but can instead represent a whole sequence of tokens. Where a naive trie would need one node per token, forcing a long shared prefix to be spread across a long chain of single-token nodes, a radix tree collapses any unbranching run of tokens into a single edge, so the number of nodes in the tree reflects only the points where requests actually diverge from one another rather than the raw length of the text.

Each node in the radix tree corresponds to a sequence of tokens reachable by walking from the root, and it stores a mapping from that sequence to the KV cache tensors for those tokens. Because SGLang combines RadixAttention with a paged KV cache layout, the tensors referenced by a node are stored in physical pages rather than as one contiguous block, and a node's edge simply carries pointers to whichever pages hold that segment of tokens.

Figure 8.29 traces how a tree grows and shrinks as requests arrive. At step (2), a first request with a system prompt and one turn of conversation is processed, and the entire sequence becomes a single new edge from the root, shown in green, with node a recording that this sequence's KV cache is available. At step (3), a second request that continues the same conversation with an additional

turn arrives; its prefix through "Assistant: Hi!" already matches what is stored, so that portion, now shown in blue to mark it as existing rather than newly created, is reused, and only the new final turn is appended as a fresh edge at node b. Step (4) shows a different kind of divergence: a new request shares only the system prompt with the previous ones and then branches off with a different first turn, so the tree splits at the point where the two conversations disagree, producing two children of the shared system-prompt node.

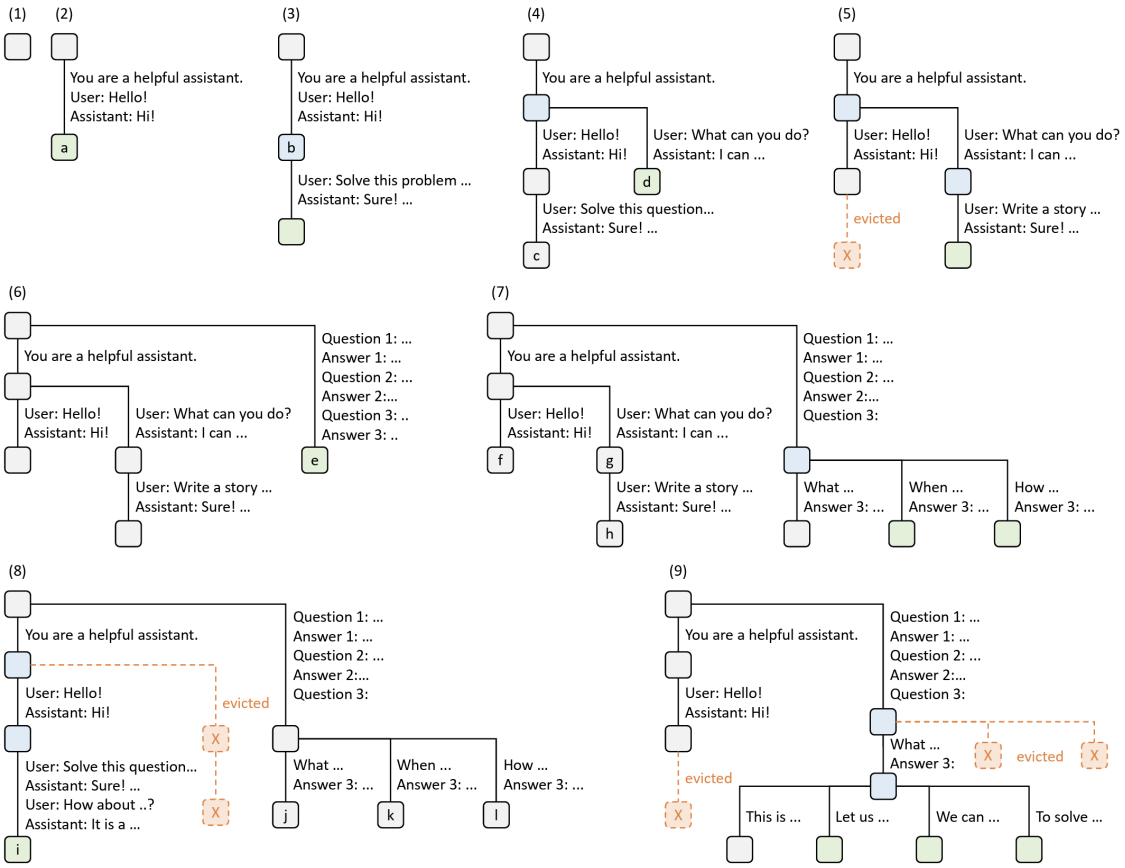


Figure 8.29: Evolution of a radix tree over a sequence of requests. Green nodes are newly created by the current request, blue nodes are existing nodes being reused, and the red node marks a KV cache entry that has been evicted; Image credit: [Zheng et al., 2024]

When a new request arrives, the runtime walks the tree from the root, matching the request's tokens against edges as far as they agree, which is exactly the longest-prefix match the system needs. Whatever portion of the request's prompt matches an existing path can skip prefill entirely and reuse

the stored KV cache directly. Whatever portion falls after the point where the request's tokens first diverge from anything in the tree still has to go through prefill as usual, and once that computation finishes, new nodes are inserted into the tree to record it, extending the tree so that a future request sharing this same prefix can reuse it too. In effect, every request that finishes does not just produce a response, it also grows the cache, making the system progressively more useful to later requests with related prompts.

Reference Counting and Eviction

GPU memory is finite, so the radix tree cannot be allowed to grow without bound. Every node's KV cache occupies physical pages that could otherwise be given to an active request's decoding, so once memory pressure builds up, some of the tree has to be reclaimed. Deciding what to reclaim, however, is complicated by the fact that a single node's KV cache may be in use by more than one active request at the same moment. If two requests currently being served both pass through the same shared system-prompt node, that node and its corresponding KV cache tensors cannot simply be freed, since doing so would disrupt the processing of the requests still reading from it.

SGLang resolves this problem with reference counting. Every node tracks how many currently running requests have a live pointer into its KV cache, and a node is only a candidate for eviction once that count drops to zero, meaning no request in flight still needs to access the corresponding KV cache tensors. Because a radix tree node with children only stores tokens that some longer path also depends on, eviction naturally proceeds from the leaves inward — a leaf whose reference count is zero can be evicted immediately, and once it is removed, its parent may itself become a leaf and become eligible for eviction in turn if its own reference count is also zero. Step (5) of Figure 8.29 shows this happening, where a leaf node from an earlier turn, no longer referenced by any running request, is evicted to free memory for a request extending a different branch of the same conversation. Among the nodes eligible for eviction, the runtime applies a least-recently-used policy, reclaiming whichever unreferenced leaf was accessed furthest in the past first, on the assumption that a prefix nobody has touched in a long time is the least likely to be reused by the next request to arrive.

Patterns of KV Cache Sharing

The benefit of RadixAttention depends on how much sharing actually exists in a given workload, and this varies a great deal depending on how an application is structured. Figure 8.30 summarizes four common patterns, using blue to mark the portion of a prompt that is shareable across requests and green or yellow to mark portions that are specific to a single request.

In few-shot learning, every request sends the same block of demonstration examples ahead of a different question, so the few-shot block is shareable across an entire batch of otherwise unrelated queries. In self-consistency, a single question is sent to the model multiple times so that several candidate answers can be sampled and compared, meaning the entire prompt is shared and only the generated continuations differ. In multi-turn chat, each new request resends the full conversation so far and appends one new turn, so everything up through the previous turn is shareable with the

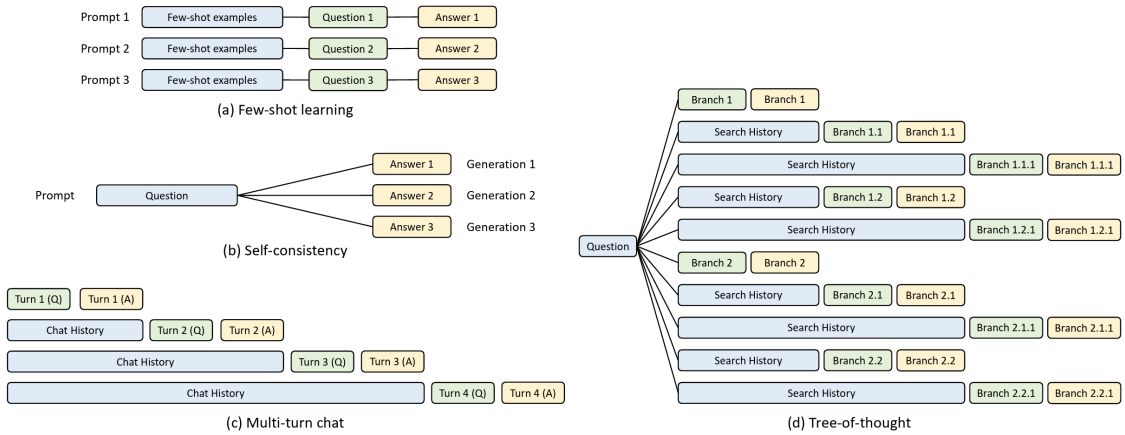


Figure 8.30: Four common patterns of KV cache sharing; Image credit: [Zheng et al., 2024]

request that produced it, and this sharable prefix keeps growing turn by turn as the conversation continues. In tree-of-thought and other branching reasoning workflows, a single question spawns many different lines of exploration that share a common ancestor and then diverge, so the shareable portion is not just a single prefix but an entire subtree of intermediate reasoning steps that multiple branches build on before going their separate ways. Across all four patterns, the same radix tree mechanism applies without modification, since sharing is detected automatically wherever token sequences happen to coincide, regardless of why they coincide.

Cache-Aware Scheduling

Prefix sharing changes what an ideal scheduling decision looks like. The scheduling policies discussed earlier in this chapter, whether first come first served or a decode-and-prefill tradeoff, all rank requests by when they arrived or by what phase of generation they are in, without regard to whether running two particular requests together would let them share KV cache. Once a shared cache is available, however, the order in which requests are scheduled affects the cache hit rate directly. Two requests that share a long common prefix benefit from being scheduled close together in time, since a shared node evicted between one request and the next has to either be reloaded or recomputed from scratch, undoing the savings that prefix sharing was meant to provide. A first come first served policy has no way to account for this, and can easily interleave requests in a way that lets a useful shared prefix fall out of the cache before a related request arrives to reuse it.

SGLang instead proposes to schedule requests by walking the radix tree in depth-first order. Because nodes that share a common ancestor sit next to each other in the tree, visiting requests in the order the tree's structure naturally suggests, rather than in arrival order, tends to group together requests that share the most tokens, which keeps their common prefix's reference count from dropping

to zero and lets it stay resident in the cache across consecutive iterations. Figure 8.31 illustrates the difference with a small example. Four requests, R_1 through R_4, arrive in the order R_1, R_2, R_3, R_4, but R_1, R_3, and R_4 all descend from the same branch of the tree while R_2 descends from a different one. Scheduling them in arrival order produces the sequence R_1, R_2, R_3, R_4, which runs R_2 in between three requests that share a great deal with one another, giving the shared prefix of R_1, R_3, and R_4 a chance to be evicted before R_3 and R_4 get to use it. A depth-first traversal of the tree instead produces the order R_1, R_3, R_4, R_2, which keeps the three closely related requests adjacent and defers the unrelated one until after they have all been served.

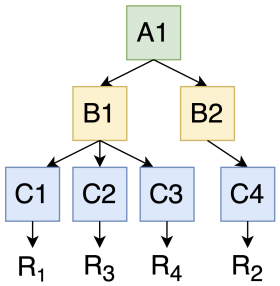


Figure 8.31: Cache-aware scheduling by depth-first traversal of the radix tree. First come first served processes the requests in arrival order R_1, R_2, R_3, R_4; depth-first traversal instead groups requests that share a common ancestor, producing the order R_1, R_3, R_4, R_2

This kind of cache-aware scheduling is a natural complement to iteration-level scheduling rather than a replacement for it. A serving system still has to decide, on every iteration, which requests to admit into the running batch and how to balance prefill against decode, exactly as described earlier in this chapter. What cache awareness adds to this scheduling decision is a preference, among the requests that scheduling would otherwise treat as interchangeable, for grouping together the ones that share prefixes. The gain from such a scheduler depends heavily on how much prefix sharing the workload contains. For workloads such as chat applications, few-shot pipelines, or branching reasoning workflows, where sharing is common by construction, keeping related requests together substantially raises the fraction of tokens that are served from cache rather than recomputed, potentially improving performance.

8.4.2 Tree-Structured Attention Kernels

The radix tree solves the memory side of prefix sharing — it makes sure that a shared prefix's KV cache is stored once and located quickly, no matter how many requests currently depend on it. It says nothing, however, about how the attention computation over that shared cache is actually carried out on the GPU. FastTree [Pan et al., 2025] starts from the observation that existing systems built on top of a radix tree, including SGLang itself, still compute attention exactly as they would if

there were no sharing at all — each query is handed to its own thread block, which independently loads whatever portion of the KV cache that query needs for computing attention. The radix tree enables sharing KV cache memory in HBM, but it does not change how the GPU touches that cache once the model actually runs.

Why a Shared Cache Alone Is Not Enough?

To see why this gap matters, consider three requests that all extend the same system prompt A_1, two of which additionally share a middle segment B_1 before diverging into distinct final segments C_1 and C_2, while the third branches off earlier into a different middle segment B_2 (Figure 8.32). The radix tree stores A_1, B_1, and B_2 each exactly once in memory, so from a memory standpoint the sharing is already realized and useful. But when the three queries reach the attention operation, a conventional kernel still dispatches each of them to a separate thread block, maybe on a separate streaming multiprocessor, and each of those blocks independently loads the tokens of A_1, and for two of them B_1 as well, from the GPU's global memory into its own local memory. Figure 8.33 shows this: SM1 and SM2 both load A_1 and B_1 from HBM separately even though the two copies are identical, simply because the two queries were placed on different blocks with no mechanism for one block's on-chip memory to be visible to the other. The shared memory on an SM could eliminate this duplication, but a query-centric computation pattern never gives it the chance to do so.

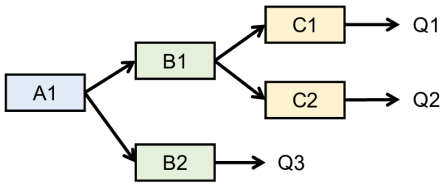


Figure 8.32: KV cache sharing through radix tree; Image credit: [Pan et al., 2025]

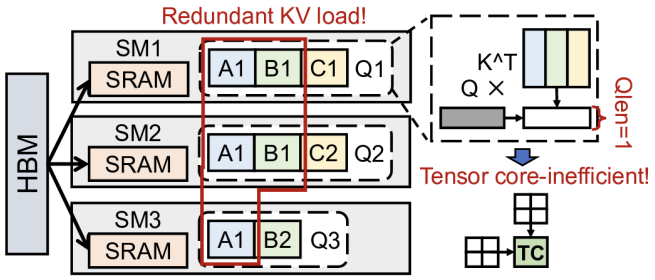


Figure 8.33: Even with a shared radix tree in memory, a conventional attention kernel still loads the shared segment A_1 separately for every query; Image credit: [Pan et al., 2025]

A second problem compounds the first. During decoding, each request contributes only a single new query token at a time, so the query matrix handed to the attention kernel for any one request is really just a vector. Computing attention as $QK^\top$ with a single-row Q turns what should be a matrix-matrix multiplication into a matrix-vector multiplication, an operation that GPU tensor cores, which are built around matrix-shaped inputs of a certain minimum size, cannot execute efficiently. A kernel faced with this shape either falls back to the GPU's slower general-purpose cores or pads

the single query vector up to the tensor cores' minimum tile size, wasting the majority of the padded computation.

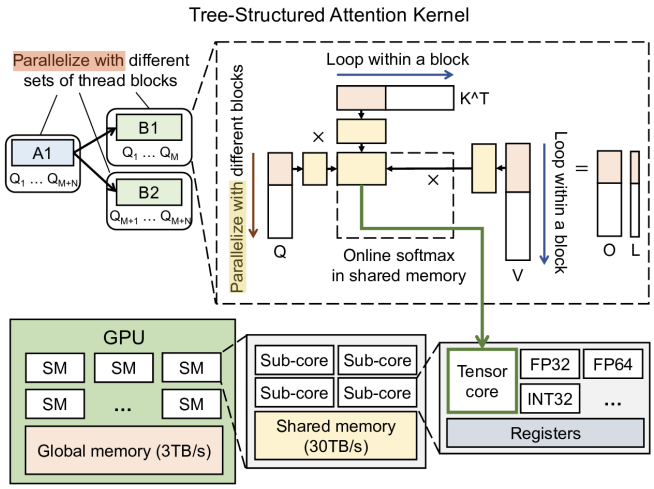


Figure 8.34: FastTree groups queries that share a context node and processes each group with its own set of thread blocks, turning per-query matrix-vector attention into a batched matrix-matrix computation; Image credit: [Pan et al., 2025]

Grouping Queries by Shared Context

FastTree's core idea is to let the tree structure that already governs the KV cache layout also govern how queries are grouped for computation. Rather than dispatching each query to the GPU separately, FastTree identifies groups of queries that share a common context and processes an entire group with a single kernel invocation, loading that shared context into shared memory once and reusing it for every query in the group. Because a group now contributes multiple query rows rather than one, the $QK^\top$ computation becomes a matrix-matrix multiplication again, restoring the tensor cores to an efficient regime. Further, because the shared context is loaded once per group rather than once per query, redundant global memory traffic falls sharply. Figure 8.34 illustrates the resulting picture — queries Q_1 through Q_{M+N} that all descend from context node A_1 can be processed together for the portion of their computation that concerns A_1, before splitting into separate groups for B_1 and B_2, each again processed as a single batched unit rather than as many individual ones. The separate attention computations over shared chunks are eventually combined via the online softmax idea to reconstruct the attentions scores over the entire query.

8.4.3 Prefix Homogeneity: A Scheduling Signal Beyond Batch Size

Every technique discussed so far in this section, like the radix tree and tree-aware attention kernels, all treat a larger batch and more aggressive prefix reuse as unambiguously good, and ask how to pack as many requests together in a batch as memory and compute allow. However, a closer look at decode-time memory access patterns complicates this picture. The decode phase is memory-bound, as every new token generation requires accessing the entire KV cache of previous tokens. In this case, throughput depends not simply on how many requests are batched together but also on how those requests' memory accesses interact with each other. This is the starting point of Feather [Rathi et al., 2026], which shows that a batch's throughput depends heavily on prefix homogeneity — the extent to which every request currently in the batch shares a common prefix — and that this factor can matter more than batch size itself. The reason for this observation is locality — when every request in a batch shares the same prefix, their attention operations sweep the same region of the KV cache at the same time, producing spatial locality that lets the hardware prefetcher pull upcoming cache lines ahead of need, and temporal locality that lets a cache line fetched for one request be reused for a neighbouring request moments later. A heterogeneous batch interleaves accesses across unrelated KV cache regions instead, causing the cache to thrash and fall back to slow global memory accesses. The effect is surprisingly sharp rather than gradual, as demonstrated by a simple experiment. In a controlled experiment on Llama 3 8B served with vLLM, a batch of 500 requests is built from two prefilled 10K-token prefixes A and B, each followed by a short unique suffix. The fraction f of requests drawn from prefix A is varied while all else is held fixed. Moving from a fully homogeneous batch ($f = 1$) to one where a single request is swapped for another prefix (e.g., when $f = 0.998$, 499 requests have prefix A and just 1 has B) is enough to roughly halve decode throughput (Figure 8.35)[1].

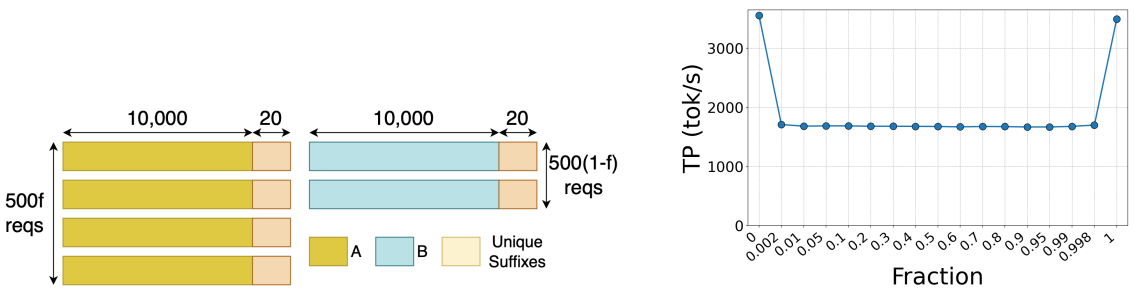


Figure 8.35: Decode throughput as a function of the fraction f of requests sharing prefix A

Simple schedulers like first-come-first-served and cache-aware schedulers such as SGLang's depth-first traversal keep adding requests to a batch as long as doing so raises GPU utilization. But if admitting a request means abandoning prefix homogeneity, the resulting loss of locality can outweigh the utilization gained, so the better decision is sometimes to keep the batch smaller and homogeneous.

[1]No image credit needed, as this figure is from a paper written by the authors of this book itself :)

At the same time, homogeneity cannot be pursued without limit, since a batch too small to keep the GPU's compute units busy loses throughput to underutilization regardless of how well aligned its accesses are. Therefore, there exists a tradeoff between batch size and prefix homogeneity, where a larger batch of several hundred heterogeneous requests can sometimes perform worse than a smaller homogeneous batch where all requests share a long common prefix, and this gap widens as the shared prefix length grows. The scheduler in Feather learns this tradeoff correctly using Reinforcement Learning (RL), batching requests into smaller homogeneous batches where possible, and falling back to larger heterogeneous batches where enough prefix homogeneity does not exist. Furthermore, unlike FastTree's tree-structured kernel or POD-Attention's fused CTA scheduling, a homogeneity-aware scheduler achieves its speedup entirely at the batching layer, without needing custom attention kernels that must be tuned to a specific GPU architecture, making such schedulers easy to deploy using existing optimized attention kernels.

8.4.4 Retrieval-Augmented Generation

Retrieval-augmented generation, or RAG, builds an LLM application in a different way from the techniques discussed so far. Instead of relying only on what the model learned during training, a RAG application keeps a large external knowledge base, usually a collection of document snippets. For every incoming question, it first searches this knowledge base for the snippets most relevant to the question (Figure 8.36). The retrieved snippets are placed before the question, and this longer prompt is what the LLM actually receives. The model can then generate text using up-to-date or private information that was not part of its training data, without any retraining. Retrieval usually runs on CPUs, while the prompt processing and generation run on GPUs. Because the retrieved snippets are much longer than the question itself, the prefill phase becomes the main cost of serving a RAG request. This is why caching and reusing the KV states of snippets is attractive.

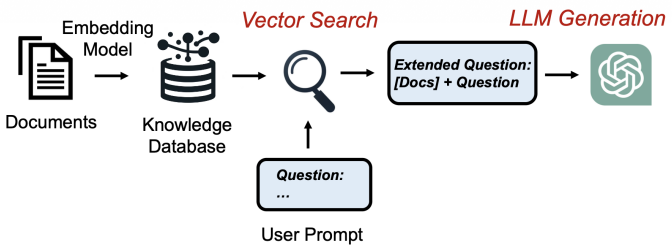


Figure 8.36: RAG workflow; Image credit: [Jin et al., 2025]

Why RAG Breaks Prefix Sharing?

At first glance, RAG looks like an ideal case for the prefix sharing techniques discussed earlier, since the same snippet can be retrieved and placed in front of many different questions. However, the

way RAG builds a prompt is quite different from a chat history or a few-shot block. In a multi-turn conversation, the shared part is always the same fixed sequence at the start of the prompt, and it only grows by appending new turns at the end. In RAG, the knowledge base may hold thousands or millions of snippets, and each new question retrieves a different small subset of them, often in a different order. For example, two questions may share one or two snippets out of five, and even a shared snippet may sit at a different position in each prompt. As we saw before, prefix caching in serving engines such as vLLM and SGLang reuses cached KV blocks only when a new prompt matches an earlier one from the first token onward until the end of a prefix. A radix tree, which can only recognize sharing that starts at the root, finds little to reuse if two prompts contain overlapping text in the middle. Figure 8.37 shows this problem. Request 2 retrieves most of the same documents as the cached Request 1, but the first document differs, so the prefix breaks at the very start and none of the KV cache blocks can be used. In other words, retrieval overlap is a property of sets of documents, while prefix caching needs agreement on an ordered sequence. The rest of this subsection describes two families of solutions to this mismatch of semantics. The first keeps prefix caching but makes prompts match cached prefixes more often, by organizing the cache around document sequences, and by reordering requests and documents. The second gives up the prefix requirement and reuses the KV cache of a snippet wherever it appears in the prompt.

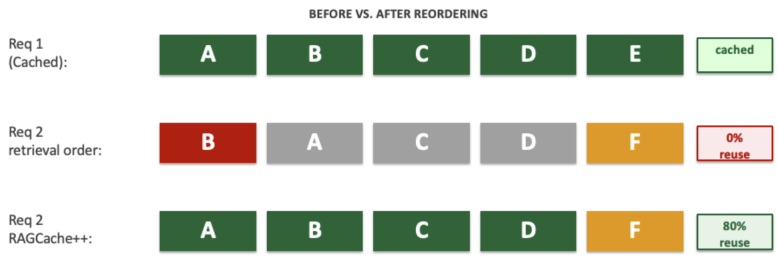


Figure 8.37: Request 1 cached; Request 2 in retrieval order has no reusable prefix; after reordering it has an 80% prefix match; Image credit: [Tan et al., 2026]

Caching Knowledge as a Tree of Documents

A natural first step is to treat each retrieved document as the unit of caching, rather than individual tokens. This is possible because the documents come from a fixed knowledge base, and in practice a small fraction of documents account for most retrievals. The same documents therefore appear again and again over the lifetime of the system. RAGCache [Jin et al., 2025] stores the KV tensors of documents in a knowledge tree, shown in Figure 8.38. The tree works as follows. The root represents the shared system prompt. Every other node represents one document and holds the KV tensors of that document. A path from the root to a node is a sequence of documents, which is exactly the order in which those documents appeared in some earlier prompt. Different requests whose prompts

start with the same documents share the same top part of the path, so their common prefix is stored only once. To serve a new request, the system starts at the root and follows the retrieved document IDs one by one. It stops at the first document that is not among the children of the current node, and the KV tensors along the path that was matched are loaded without recomputation. Note that the tree is simply a radix tree at the granularity of documents instead of tokens. Also, the lookup cost depends only on the height of the tree. Further, every node in the tree is keyed by the whole sequence, not by the document alone, because the KV tensors of a document depend on everything before it. For example, document D_3 placed after D_1 has different KV values from D_3 placed after D_2, since attention for each token looks at the preceding tokens. The same document at two different tree positions is therefore stored as two separate nodes, e.g., nodes marked "Different" in Figure 8.38.

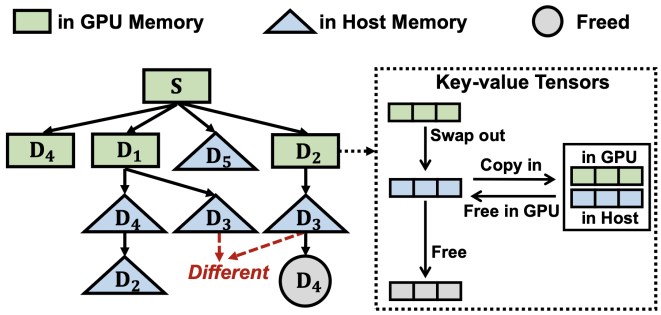


Figure 8.38: Knowledge tree; Image credit: [Jin et al., 2025]

Placing the Cache Across GPU and Host Memory

Typically, the KV cache of documents is large, and GPU memory is small compared with the number of documents in a knowledge base. RAGCache therefore spreads the knowledge tree over a memory hierarchy [Jin et al., 2025]. Nodes that are used often stay in GPU memory, nodes used less often live in the larger but slower host memory, and the rest are dropped. Because a node's KV values depend on its parent, a node in host memory can only have a parent in GPU or host memory, never the other way round. Eviction therefore always removes leaves first (Figure 8.38).

Further, ordinary eviction policies such as LRU ignore that documents differ in size and in the cost of recomputing them. RAGCache uses a prefix-aware variant of Greedy-Dual-Size-Frequency (PGDSF), which gives each node a priority

$$\text{Priority} = \text{Clock} + \frac{\text{Frequency} \times \text{Cost}}{\text{Size}}.$$

Nodes with the lowest priority, computed as shown above, are evicted first. In the above formula, "Frequency" counts how often the document was retrieved, and "Size" is its number of tokens. "Clock" is a logical timestamp that grows as nodes are evicted, so recently used nodes rank higher. "Cost"

is the time needed to recompute the document's KV tensors. This is "prefix-aware" because the time depends on how much of the preceding sequence is already cached. For example, computing D_2 after a cached $[S, D_1]$ is cheaper than computing it when only S is cached. RAGCache estimates this time from offline measurements of prefill time for different cached and non-cached lengths. Finally, a node is copied to host memory only the first time it is evicted from the GPU. Later evictions simply free the GPU copy, which avoids repeated transfers over the slow PCIe link.

Reordering to Create Prefix Matches

Even with a good cache, a prefix match only happens if the prompt starts with a cached sequence. Two kinds of reordering help with this. They work at different levels and can be combined.

The first is reordering requests. When many requests arrive, the order in which the scheduler processes them affects how often the cache is useful. If requests that need different documents alternate, a small cache keeps evicting what the next request needs. RAGCache therefore keeps a priority queue and favors requests with a large cached part relative to the part that must be computed,

$$\text{OrderPriority} = \frac{\text{Cached Length}}{\text{Computation Length}}.$$

A time window bounds how long any request can be postponed, so none is starved.

An example shows why this helps. Suppose the cache can hold four documents, and two requests are waiting. Request A has three documents cached and needs one more computed, while request B has one document cached and needs three computed. If B runs first, its three new documents do not fit without evicting A's cached documents, so A must later recompute all four, for a total of $3+4=7$ document computations. If A runs first, it computes only one document, and B then computes its four, for a total of $1+4=5$. The priority formula gives A a score of $3/1=3$ and B a score of $1/3$, so the scheduler picks A first. Running the request with the larger reusable part first protects the cache contents that give the largest saving.

The second is reordering the documents inside a request. A RAG prompt contains the same set of documents regardless of the order, and the order is chosen by retrieval rank. If this order is changed so that the documents that are already cached come first, the prompt can match an existing prefix. CacheWeaver [Tan et al., 2026] does this with a greedy walk over its trie (Figure 8.39). Starting at the root, it looks for any retrieved document that continues a cached path, places it next, and moves down the tree. When no retrieved document continues a cached path, the remaining documents are appended in their original rank order. If nothing is cached, the original order is kept.

The above two ways of reordering, across requests in the scheduler queue, and across document snippets appended to the user prompt, enable better prefix sharing and KV cache reuse with RAG workloads, thereby improving performance.

Reusing Documents That Cannot Form a Prefix

Reordering only helps when the documents whose KV cache we wish to reuse can be arranged to form a shared prefix. However, many prompts contain several reused documents that cannot all be

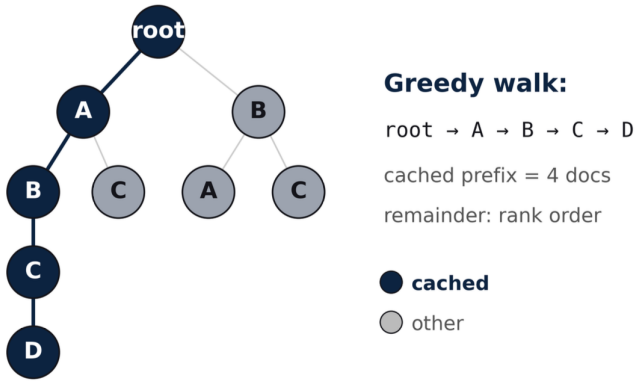


Figure 8.39: Dark nodes are cached; gray nodes are evicted or on other branches. Given candidate set {A,B,C,F,D}, the greedy walk follows root→A→B→C→D (the longest cached path), yielding a 4-document prefix hit; the remaining document (F) is appended in retrieval-rank order; Image credit: [Tan et al., 2026]

rearranged as a prefix. The natural idea is to cache the KV tensors of each document on its own and simply concatenate them, whatever the position. This is cheap, but it causes a correctness problem that prefix caching never has. To see why, recall that when the LLM processes a prompt, attention in each layer looks at all earlier tokens. If a document's KV cache was computed in isolation, it was computed as if no earlier document existed. It therefore lacks the cross-attention between its tokens and those of the documents being placed before it. Figure 8.40 gives an example. One snippet states how many goals Messi scored, a second states Ronaldo's total, and the question asks who scored more. Computing the whole prompt from scratch (full recompute) gives the right answer, because the model can attend across the two snippets. Reusing the two independent KV caches (full reuse) gives a confused answer, because that link was never inferred by the model. Figure 8.41 shows the cause. The cross-attention region of the attention matrix is missing in full reuse, and this changes the attention that the last tokens pay to the context, which directly affects the generated text. So there is a trade-off between the two extremes. Prefix caching and full recompute are correct but slow when documents are not prefixes. Full reuse of precomputed KV cache is fast but may be wrong due to missing cross attention.

Selective Recomputation to Fuse Cached Documents

CacheBlend [Yao et al., 2025] looks for the middle ground in this tradeoff between full KV cache recomputation and reuse. It reuses the stored KV cache of most shared tokens in a document, but recomputes the KV values of a small subset of tokens, namely those that are most affected by the

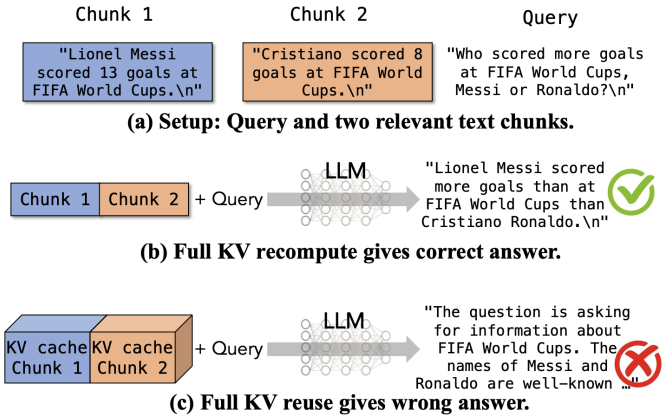


Figure 8.40: Reusing independently computed KV caches can give a wrong answer; Image credit: [Yao et al., 2025]

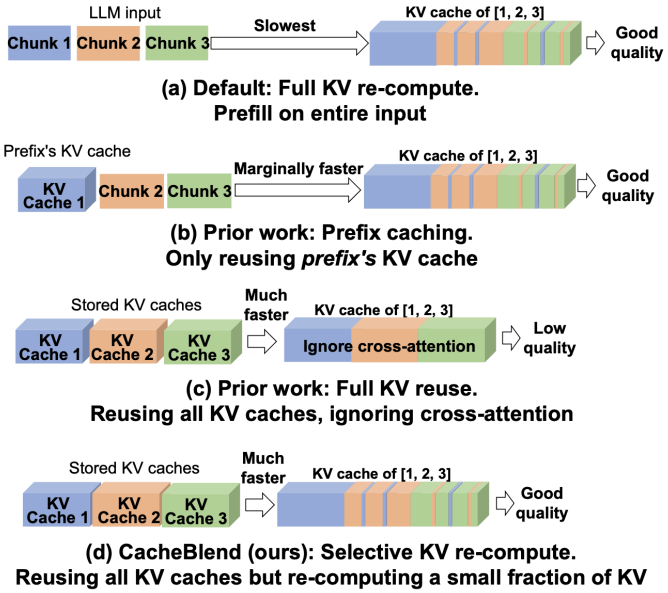


Figure 8.41: Full reuse misses the cross-attention block, which changes the forward attention that drives generation; Image credit: [Yao et al., 2025]

missing cross-attention shown in Figure 8.41. Recomputing only a small subset is enough because the attention matrix is sparse. Most tokens attend strongly to only a few other tokens, so for most tokens the isolated KV values are already close to the values that full recompute would give. Before this step, the keys of each stored document are also adjusted for their new position in the prompt. With rotary position embeddings this is a single rotation applied once.

This method, however, needs a way to find the tokens whose KV values are wrong the most due to missing cross attention. Define the KV deviation of a token in a layer as the difference between its stored KV values and the values it would have if it had seen the preceding documents. The tokens with the highest deviation are called HKVD tokens. Recomputing exactly these tokens reduces the error in the attention matrix the most. The difficulty is that the true deviation is known only after doing the full computation, which is what we want to avoid. CacheBlend relies on an observation — the tokens with high deviation in one layer tend to also have high deviation in the next layer, because the representation of each token changes slowly from layer to layer. So the choice of tokens can be carried forward instead of being rediscovered at every layer.

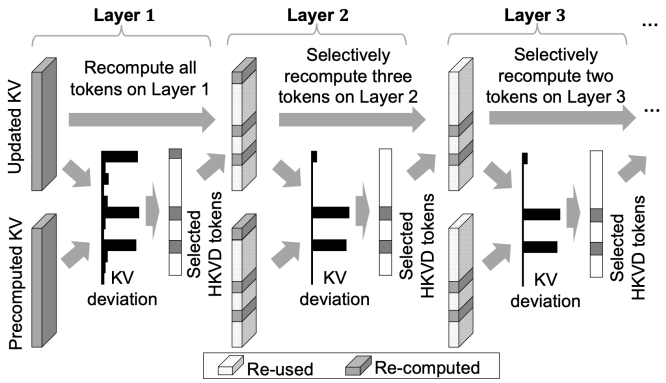


Figure 8.42: The high-deviation tokens of one layer become the candidates for the next layer, and the set shrinks with depth; Image credit: [Yao et al., 2025]

Figure 8.42 shows the resulting procedure, which is called gradual filtering. In the first layer, all tokens are recomputed, and the deviation of each token is measured by comparing the new KV values with the stored ones. The tokens with the highest deviation are kept as candidates. In the second layer, only these candidates are recomputed. Their deviations are measured, and a smaller number of the highest ones is kept for the third layer, and so on. The candidate set therefore only shrinks as we go deeper, and each layer needs deviation values only for the tokens it recomputes anyway. Gradual filtering also makes the selection more robust. A token reaches the later layers only if its deviation stays among the highest at every earlier layer, so the final set does not depend on a single layer's measurement, which could be noisy.

Within every layer, the attention computation is done only for the selected tokens (Figure 8.43). The layer's input is reduced to these tokens, their queries, keys and values are computed, and the new keys and values are placed among the reused KV cache of all other tokens. Attention is then computed between the selected tokens and the whole context. Therefore, the cost of a layer's computation becomes proportional to the number of selected tokens alone.

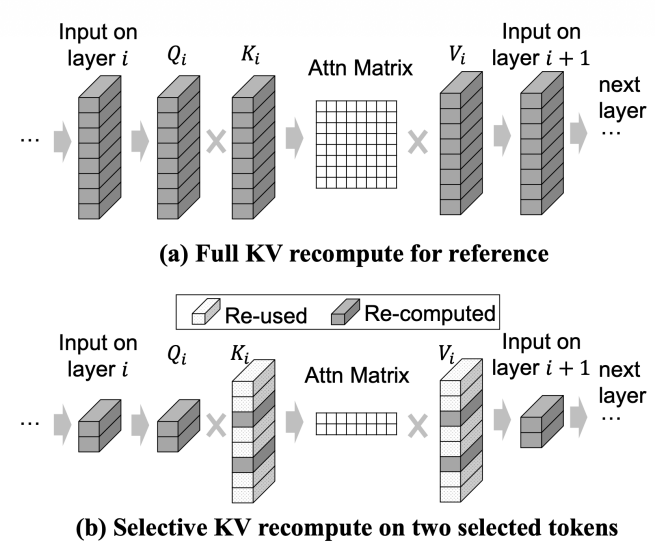


Figure 8.43: Selective recomputation inside one layer; Image credit: [Yao et al., 2025]

Practice Problem 8.6

Consider an inference engine that processes requests in batches. Four requests arrive in the order R_1 to R_4. Each request has input prompt tokens as follows: $R_1(A, B, C, D)$, $R_2(E, F)$, $R_3(A, B, C, G)$, $R_4(A, C)$, where A through G represent distinct tokens. The inference engine batches requests to process them together. The processing of a request involves the prefill step, and a decode step to generate exactly one output token per request. Every request is allocated memory to store KV cache for all input tokens and the one output token that will be generated. The GPU has limited memory and can store KV caches for at most 8 tokens at a time.

Requests are added to a batch sequentially, until there is enough GPU memory remaining for all the KV cache tokens (input + output) of a request. Once the batch is full, a batch iteration executes the prefill and decode steps of all the requests together on the GPU and returns, after which the next batch begins. The system supports prefix caching, meaning that all requests sharing a common prefix can reuse previously computed KV caches within the same batch. Note that after each batch completes, the GPU memory is fully cleared (i.e., all KV caches are discarded) and cannot be reused across batches.

(i) Consider the First-Come-First-Served (FCFS) scheduling policy. How many batch iterations are required to

complete processing all requests? List the requests processed in each batch as well.
Solution: Under FCFS, requests are added in arrival order R_1, R_2, R_3, R_4. Each request requires GPU memory for all input tokens plus one output token. In the first batch we add R_1, which needs $4+1=5$ tokens. R_2 needs $2+1=3$ tokens, so we will add it too in the first batch, after which we don't have any space left. In the second batch, we add R_3 which needs $4+1=5$ tokens, then we add R_4 which needs $1+1=2$ tokens (because of prefix caching, KV cache of A is already available). So, two batch iterations are needed to complete processing all the requests.

(ii) For FCFS, what is the total number of tokens for which KV cache must be computed, across all requests in all batch iterations?
Solution: The first batch requires 8 tokens, and the second requires 7 tokens, so, in total, 15.

(iii) Now, consider a different scheduling policy that constructs a radix tree (prefix tree) for the given set of requests based on their token sequences, and uses a depth-first-search (DFS) traversal of this tree to order requests during batch formation. This algorithm ensures that requests have good enough overlap in their tokens where possible, and can better reuse KV caches. Now, assume that all the requests are available upfront (i.e., no arrival order constraint). Draw the radix tree formed by all the requests.
Solution:

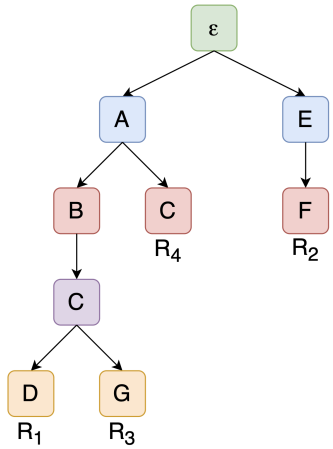


Figure 8.44: Radix tree

(iv) For this DFS-based scheduling policy, how many batch iterations are required to complete processing all requests? List the requests processed in each batch as well. Assume that the DFS traversal starts with adding request R_1 first.
Solution: The DFS traversal starting from R_1 visits requests in the order: R_1, R_3, R_4, R_2. In the first batch, we add R_1, which takes $4+1=5$ tokens, then we add R_3 which requires $1+1=2$ tokens because $A-B-C$ is shared. In the second batch, we add R_4 which requires $2+1=3$ tokens, followed by R_2 which takes $2+1=3$ tokens. So, two batch iterations are needed to complete processing all the requests.

(v) For the above DFS-based policy, what is the total number of tokens for which KV cache must be computed,

across all requests in all batch iterations?

Solution: The first batch requires 7 tokens, and the second requires 6 tokens, so, in total, 13.

(vi) Based on all the previous parts, can you come up with one advantage of DFS-based scheduling over FCFS?

Solution: Schedules requests that share prefixes together, so better KV cache reuse.

Practice Problem 8.7

Answer the following questions based on CacheBlend.

(i) Which of the following statements is/are true?

(A) Because the prefix's KV cache is independent of the succeeding texts, prefix caching does not hurt generation quality.

(B) Many applications, such as RAG, include multiple text chunks, rather than one, in the LLM input to provide all necessary contexts to ensure good response quality.

(C) Naive prefix caching can reuse all the individual chunks' KV caches, combine them without cross-attention for a prefill with all the chunks concatenated together.

(D) Prefill latency is also termed as TPOT (time-per-output-token).

Solution: (A), (B)

(A) True. With causal attention, a prefix token's KV values depend only on earlier tokens, so reusing the prefix's cache gives exactly the same result as recomputing it.

(B) True. RAG-style applications retrieve and concatenate several chunks into the prompt so the model has all the context it needs.

(C) False. Naive prefix caching can only reuse the KV cache of a shared prefix; reusing every chunk's independent cache without cross-attention is what "Full KV Reuse" does, and it loses quality.

(D) False. Prefill latency corresponds to TTFT (time to first token); TPOT measures the per-token latency of the decode phase.

(ii) Which of the following correctly represent the limitations and characteristics of the "Full KV Reuse" approach?

(A) It correctly preserves the positional embedding information of non-prefix text chunks.

(B) It fully computes the cross-attention between a chunk and its preceding unknown text chunks to ensure mathematical correctness.

(C) The disparity in response quality between full prefill and full KV reuse becomes more pronounced as the number of related chunks increases.

(D) It guarantees same output as full prefill.

Solution: (A), (C)

(A) True. Each chunk's KV cache is precomputed with a dummy prefix so that its positional embeddings match its position in the final concatenated input.

(B) False. Full KV Reuse skips cross-attention between chunks entirely, which is its main limitation.

(C) True. More related chunks means more missing cross-attention, so the quality gap relative to full prefill grows.

(D) False. Since cross-attention is ignored, the KV cache differs from full prefill and the output is not guaranteed to be the same.

(iii) Which of the following statements is/are true?

(A) To concatenate the KV caches of chunks C1 and C2, "Full KV Reuse" first needs to precompute the KV cache of C2 by running prefill on a hypothetical input that prepends C2 with a dummy prefix of length greater or equal to C1. This is done to preserve positional embedding.

(B) Increasing the number of retrieved text chunks continuously improves generation quality linearly without any degradation.

(C) With a larger number of chunks, the amount of cross-referencing and interdependency (cross-attention) between different parts of the input decreases.

(D) Including too many chunks can eventually hurt generation quality due to the lost-in-the-middle issue.

Solution: (A), (D)

(A) True. The dummy prefix shifts C2's tokens to the positions they will occupy after C1, so the positional embeddings stay correct when the caches are concatenated.

(B) False. Quality does not improve linearly; it saturates and can degrade when too many chunks are included.

(C) False. More chunks means more cross-references and interdependencies between different parts of the input, not fewer.

(D) True. With very long inputs, models tend to overlook information placed in the middle, which can reduce response quality.

(iv) When CacheBlend performs selective KV recompute on a specific layer, what specific steps does it execute?

(A) It skips the computation of the non-selected tokens completely, dropping their previously computed KV cache entries.

(B) It applies a mask to reduce the input of the layer to only a subset of selected tokens before transforming them into Q, K, and V vectors.

(C) It expands the K and V vectors by reusing the KV cache entries associated with the un-selected tokens on that layer.

(D) It recomputes the cross-attention by computing Q, K, and V vectors for all tokens from scratch.

Solution: (B), (C)

(A) False. The KV entries of non-selected tokens are not dropped; they are kept from the precomputed cache and reused.

(B) True. Masking the layer input to only the selected tokens means Q, K, and V are computed for just those tokens, saving computation.

(C) True. The newly computed K and V of the selected tokens are combined with the reused cached K and V of the other tokens, so attention sees the full context.

(D) False. Recomputing Q, K, and V for all tokens is just a full prefill, which is the cost CacheBlend is designed to avoid.

(v) Which of the following statements is/are true?

(A) High attention typically occurs only between a small number of tokens and their preceding tokens.

(B) A token requires KV recomputation primarily when it has a low KV deviation compared to the ground truth full KV recompute.

(C) CacheBlend leverages the insight that tokens with the highest KV deviations on one layer are highly likely to have the highest KV deviations on the next layer.

(D) High KV Deviation tokens are identical across all layers.

Solution: (A), (C)

(A) True. Attention is sparse, so only a small fraction of tokens have strong attention to earlier tokens, and only those need their cross-attention restored.

(B) False. Recomputation is needed for tokens with a high KV deviation, since their reused KV differs most from the full-prefill values.

(C) True. The high-deviation tokens are strongly correlated between consecutive layers, so CacheBlend can use one layer's deviations to pick the tokens to recompute in the next.

(D) False. The correlation across layers is strong but not perfect, so the set of high-deviation tokens is similar and not identical.

Programming Assignment 8.1: Prefix Sharing in nanoGPT

In this assignment, you will extend the KV caching idea to a setting where many requests share the same starting text, such as a chatbot where every user query begins with the same long system prompt. Your job is to modify a simplified version of nanoGPT [Karpathy, 2023] so that it finds the longest prefix shared by all prompts in a batch, computes its keys and values only once, and then reuses (a copy of) this shared cache when generating tokens for each sequence. Only the prefix is cached — the remaining tokens are recomputed at every step, so that you can measure how much prefix sharing alone helps. To make this work, you will need to handle attention when queries and keys have different lengths, and make sure positional embeddings reflect the true position of each token in the full sequence. The generated text must be identical to the version without prefix sharing, while being noticeably faster for long shared prefixes. The assignment, along with a detailed `README` is available at:

`https://github.com/mythilivutukuru/SysMLBook/tree/main/prefix_sharing_nanogpt`

Programming Assignment 8.2: Radix Tree Prefix Caching in nanoGPT

In this assignment, you will extend KV caching to a setting where many prompts share parts of their starting text. Your job is to modify a simplified version of nanoGPT [Karpathy, 2023] so that it tracks which prompts share prefixes using a radix tree, where each node stores a contiguous run of tokens along with the keys and values computed for them. For each new prompt, you will walk down the tree to find the longest prefix already seen, start from the cached keys and values for that prefix, and compute the cache only for the remaining distinct suffix and the generated tokens. The newly computed suffix cache is then inserted back into the tree so that future prompts can reuse it. To make this work, you will need to handle attention when queries and keys have different lengths, using a rectangular causal mask for the suffix, and make sure positional embeddings reflect the true position of each token in the full sequence. The generated text must be identical to processing each prompt independently, while being noticeably faster when prompts share long prefixes. The exact design of the tree and the cache management is left to you. The assignment, along with a detailed `README` is available at:

`https://github.com/mythilivutukuru/SysMLBook/tree/main/radix_tree_nanogpt`

8.5 CPU Offloading

So far in this chapter we have assumed that everything an inference server needs, whether the model weights or the KV cache of every active request in a batch, all live in GPU memory. This assumption is convenient, because GPU high bandwidth memory (HBM) is extremely fast, but it is also the biggest constraint on how large a batch we can run and how long a context we can serve. GPU memory capacity grows slowly from one hardware generation to the next relative to compute capacity, while model sizes and context lengths have been growing much faster. For example, an NVIDIA H100 has 80 GB of HBM, roughly the same capacity as the previous generation A100, even though the compute throughput of the H100 is close to three times higher. This memory bottleneck has constrained the deployment of larger models and longer contexts.

This section begins with the observation that modern servers connected to GPUs often have spare memory and compute, e.g., a terabyte or more of DRAM, and this DRAM is orders of magnitude cheaper per unit than the GPU's HBM. It is therefore tempting to use the CPU's memory, and sometimes even the CPU's compute, as a second tier of resources that the GPU can fall back on when it runs out of room. This general idea is called offloading — moving some of the data or computation of LLM inference away from the GPU and onto the host CPU.

Offloading is not a new idea in deep learning systems; it has long been used during training to swap activations and optimizer state to host memory when a model does not fit on the GPU. What makes offloading during LLM inference distinctive is the presence of the KV cache. Unlike model weights, which are fixed in size once a model is chosen, the KV cache grows linearly with the number of tokens in flight, that is, with the sequence length and the batch size together. As we saw in §8.2, this is exactly why techniques like PagedAttention were introduced to manage KV cache memory efficiently. Offloading pushes this idea one step further — instead of only managing the KV cache carefully within the GPU, we ask whether some of it can live outside the GPU altogether, and what techniques are required to manage this offloaded memory efficiently.

8.5.1 Why Offload, and What Can We Offload?

Before looking at specific systems, it is useful to be clear about what exactly can be moved off the GPU, and why we would want to do so. The simplest thing to offload is the model's parameters. If a model does not fit in GPU memory at all, part of the weights can be kept in CPU memory and streamed over to the GPU layer by layer during the forward pass. This is the strategy used by early offloading systems such as FlexGen [Sheng et al., 2023], and it lets us run models that are simply too large for the available GPU, at the cost of paying a PCIe transfer for every layer, every time we use it. The second, and the more pressing candidate to offload for long-generation workloads, is the KV cache. As discussed earlier in this chapter, the KV cache size grows with sequence length and batch size, and for long outputs or large batches it can exceed the size of the model weights themselves. Because the KV cache is read every single decoding step, offloading it to the CPU means we must repeatedly transfer it back to the GPU, which is exactly the tension this section explores.

A more aggressive form of offloading does not just move data to the CPU, but offloads some of the actual arithmetic as well. The decoding-stage attention operation, in particular, turns out to be a reasonable candidate for this offload, because it is bound by memory bandwidth rather than by raw compute throughput, and CPUs are not nearly as far behind GPUs in memory bandwidth as they are in compute FLOPs.

The common thread behind all of these designs is that they trade cheap, abundant host resources for scarce GPU memory. Since the batch size is usually capped by how much KV cache fits in HBM, and a small batch leaves the GPU's compute units idle, absorbing part of the memory pressure on the CPU raises the effective batch size and therefore the throughput, without buying a larger GPU.

Of course, offloading is not free. The CPU is connected to the GPU through a relatively narrow and high-latency link, typically PCIe, whose bandwidth is one or two orders of magnitude lower than GPU HBM bandwidth. Any offloading design therefore has to answer three questions carefully — how much data actually needs to cross this link, how that transfer can be made as efficient as possible, and how the resulting latency can be hidden behind useful computation rather than sitting on the critical path. The rest of this section looks at a few systems, each of which answers these questions differently.

8.5.2 Offloading the KV Cache

Let us first consider only offloading the KV cache, while keeping all computation on the GPU. Figure 8.45 sketches four ways of running a single Transformer block under this design. In the first case, the entire KV cache stays resident in GPU memory. Loading the cache for a block is then just a read from HBM, which is so fast that it barely shows up in the timeline. The catch is that this only works as long as the KV cache is small enough to fit on the GPU alongside the model weights, which limits how many requests we can batch together and how long a sequence we can generate.

In the second case, we move the KV cache to CPU memory to escape this capacity limit. Now, before we can run attention for a block, we must first copy the relevant keys and values for every token generated so far from CPU memory to the GPU over PCIe. Because this load happens before the block's computation can start, and because the amount of data can run into the tens or hundreds of gigabytes for large batches and long sequences, this load latency dominates the total time per block. The attention and feed-forward computation, which used to be the expensive part, looks quite insignificant compared to the time spent in waiting for data to arrive.

A natural fix is prefetching — while the GPU is busy computing block $i-1$, we can start loading the KV cache needed for block i in the background, so that by the time block $i-1$ finishes, at least part of the data for block i has already arrived. This is the third case in the figure. Prefetching genuinely helps, but only up to a point, because the compute time of a single Transformer block is usually much shorter than the time needed to transfer the entire KV cache of a long sequence. Even with perfect overlap, the load simply cannot be hidden behind one block's worth of computation, and the system remains, in practice, bound by PCIe bandwidth rather than by GPU compute.

Prefetching is the idea behind most practical systems that spill the KV cache out of the GPU.

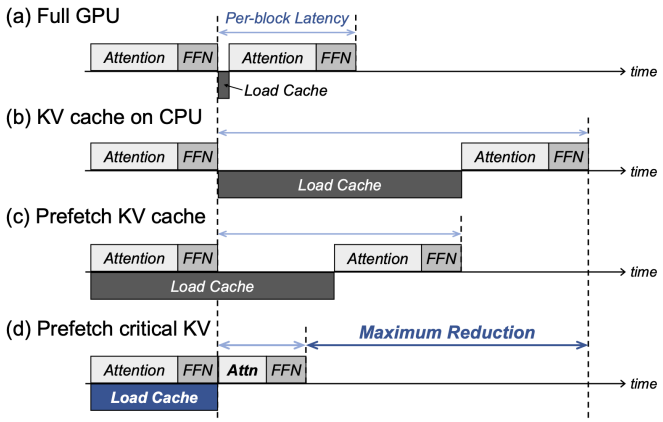


Figure 8.45: Comparison between different execution styles of Transformer blocks; Image credit: [Lee et al., 2024]

FlexGen [Sheng et al., 2023] treats GPU memory, host DRAM, and disk as three tiers and overlaps transfers with computation. CachedAttention [Gao et al., 2024] and Pensieve [L. Yu et al., 2025] apply the same idea to multi-turn serving. They keep the KV cache of finished turns in DRAM or SSD instead of discarding it, load it layer by layer so that layer $i + 1$ is fetched while layer i computes, and write new KV entries back in the background. FlashGen [Jeong and Ahn, 2025] adds scheduling on top. Instead of running requests in arrival order, it reorders execution based on whether a request's KV cache has already been loaded, so the GPU works on ready requests while the others are still being fetched. Two lessons from these systems carry over to the rest of this section. First, the scheduler must know where each request's cache lives, because this decides how long the request will wait. Second, time spent waiting in the request queue is free time that can be used to start loading the KV cache into GPU memory.

All of these designs, however, still move the entire cache. The fourth case in the figure hints at a different fix — instead of trying to hide the transfer of the full KV cache, we should question whether we need to transfer the full KV cache at all. If we could identify, ahead of time, which tokens are actually going to matter for the next attention computation, we could fetch only those and leave the rest sitting in CPU memory untouched. Because the softmax inside attention concentrates most of the probability mass on a handful of tokens, this is not an unreasonable hope, and it is the idea that InfiniGen [Lee et al., 2024] turns into a working system.

Selecting Only the Critical Tokens: InfiniGen

InfiniGen is a KV cache management system built to work alongside offloading-based inference engines. Its central idea is simple to state — keep the full KV cache in CPU memory, but for

each decoding step and each layer, predict which tokens will actually receive significant attention weight, and fetch only those tokens' keys and values to the GPU. Everything else about the layer's computation, like the attention and the feed-forward network, proceeds exactly as before, just with a much smaller amount of data having crossed PCIe. Crucially, a token that is not fetched is skipped only for this layer at this step, but stays in CPU memory and remains a candidate for every later step and layer. The CPU pool is itself trimmed only when it reaches its capacity, by dropping the entries of tokens that are rarely selected.

This is in contrast to KV cache eviction methods, which keep the cache small by permanently discarding tokens that look unimportant at some point, so that a discarded token can never be attended to again. Such permanent eviction has two problems. The first is that attention patterns are not static across decoding steps. A token that receives negligible attention weight at one step can become important again several hundred steps later, once the generation has moved on to a different part of the discussion; if it has already been discarded, it is gone. The second is that the number of tokens that account for most of the attention weight is not a fixed percentage of the sequence length; it varies enormously from one layer to another, and even from one query token to the next within the same layer. Some layers, especially early ones, attend broadly across many tokens, so a small fixed budget badly under-represents them. Other layers are highly peaked, with only a handful of tokens carrying almost all the weight, so a large fixed budget wastes memory bandwidth on them. InfiniGen's design avoids the first problem by keeping the full cache and making the selection temporary, and it addresses the second by choosing the number of fetched tokens dynamically per layer and per step.

Speculating the Next Layer's Attention Pattern

Given that InfiniGen wants to fetch only the important tokens, it needs some way to know, before it starts loading data, which tokens are going to be important. Computing the exact attention scores requires the full keys, which is exactly what we are trying to avoid transferring from the CPU. InfiniGen's solution is to speculate — use cheap, approximate information that is already sitting on the GPU to guess which tokens will matter, and only fetch those.

The key empirical observation behind this speculation is that the input to one Transformer layer is highly similar to the input of the next layer. This happens because of two components of Transformers — residual connections and layer normalization. Because the output of the attention and feed-forward sub-layers is added back onto the block's input through a residual connection, and because that output is comparatively small in magnitude after layer normalization, the block's input tensor changes only gradually from one layer to the next. InfiniGen exploits this by computing, at layer $i-1$, a cheap approximate version of the attention that will happen at layer i, using layer $i-1$'s own input together with a small piece of layer i's query weights and key cache. This is the minimal rehearsal — the attention of layer i, performed one layer ahead of schedule, so that its result can decide what to prefetch while layer $i-1$ is still computing.

With an approximate attention score for the upcoming layer in hand, InfiniGen selects tokens

whose score is close enough to the maximum score, using a threshold expressed relative to the top score rather than as a fixed count of tokens. Because attention weights go through a softmax, a token whose raw score is far below the maximum contributes a vanishingly small amount to the final output, so it can be safely skipped without noticeably changing the result. This threshold-based rule is what allows the number of prefetched tokens to expand or shrink automatically layer by layer and step by step, addressing the second problem described above.

8.5.3 Offloading Compute

InfiniGen reduces how much data needs to cross PCIe, but it still assumes that all computation, including attention, happens on the GPU. A more aggressive strategy, taken by the system NEO [Jiang et al., 2024], is to also offload the computation of decoding attention to the CPU, along with the KV cache it depends on. If attention for some requests runs entirely on the CPU using the KV cache that already lives there, the GPU never has to fetch that cache at all, sidestepping the PCIe bottleneck altogether for those requests. The cost, naturally, is that the CPU now has real work to do, and CPUs are far weaker than GPUs at raw arithmetic.

Why Decoding Attention Is a Reasonable Thing to Offload?

Decoding attention has a low ratio of floating point operations to bytes read, because for each new query token it must read the keys and values of every previous token but perform relatively little arithmetic on each one. As we saw in Chapter 2, operations with this property are called memory-bandwidth bound — their speed is set by how fast data can be streamed out of memory, not by how many floating point units are available. This matters because, while GPUs vastly outperform CPUs in raw compute throughput, often by two orders of magnitude, the gap in memory bandwidth is much smaller, typically only a factor of two to four. A GPU might offer on the order of 600 GB/s of memory bandwidth against roughly 200 GB/s on a capable CPU, whereas the corresponding compute gap can be a hundred-fold or more. Since decoding attention cares mostly about bandwidth, moving it to the CPU is a far less painful trade than moving, say, the dense linear layers, which are compute bound and would run drastically slower on a CPU. In addition, offloading decoding attention avoids repeatedly moving the KV cache back and forth between CPU and GPU, since the computation that needs the cache now happens where the cache already lives.

Simple Offloading

The most direct way to offload compute is to take the decoding attention operation out of the GPU pipeline entirely and run it on the CPU for every request, while everything else, like the prefilling attention and the linear layers (i.e., the projections and the feed-forward network), stays on the GPU. Figure 8.46 shows the resulting timeline. The problem is immediate — whenever the GPU is busy with the linear layers, the CPU sits idle waiting for the next batch of queries, keys, and values to arrive, and whenever the CPU is computing attention, the GPU sits idle waiting for the attention output to come back before it can continue with the post-projection and feed-forward computation.

Neither device is used while the other is working, so this simple offloading strategy wastes exactly the resources that made offloading attractive in the first place.

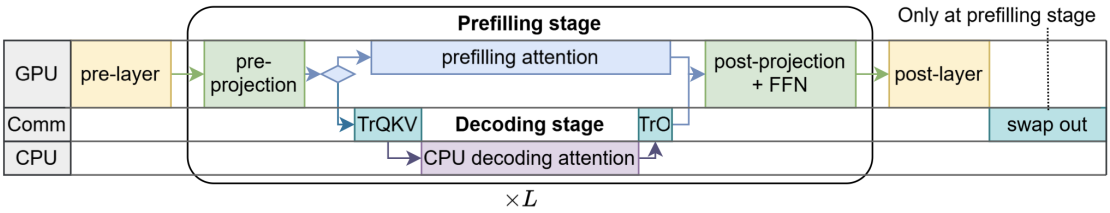


Figure 8.46: Simple offloading; "Comm" stands for GPU-CPU communication; "TrQKV" means transferring Q,K,V tensors to CPU; "TrO" means transferring attention output to GPU; Image credit: [Jiang et al., 2024]

Symmetric Pipelining

The natural fix is to introduce pipelining — split the decoding batch into two halves, and while the GPU computes the linear layers of one half, have the CPU simultaneously compute the attention of the other half, then swap roles. This is called symmetric pipelining, since the two sub-batches are meant to be equal in size. It does noticeably better than simple offloading, but NEO's analysis points out a few remaining problems. First, this scheme wastes GPU memory. Because the whole point is to keep the KV cache off the GPU, none of the GPU's spare HBM, which would otherwise have held the KV cache, is being put to any use. Second, achieving good overlap between the two sub-batches is harder than it sounds. The scheme ignores the prefilling stage and the time needed to swap the freshly computed KV cache of new requests out to the CPU, during both of which the CPU sits idle. Third, splitting a batch into two genuinely equal halves is difficult in practice, because real requests differ in input length and generate outputs of unpredictable length. Any mismatch between the two sub-batches shows up as idle bubbles in the pipeline, further eroding the benefit.

Asymmetric Pipelining

NEO's answer to these problems is to stop insisting that the two sub-batches be symmetric, and instead to deliberately make them different in a way that plays to each device's strengths. This design, called asymmetric pipelining, is shown in Figure 8.48. NEO first splits the KV cache system itself into a GPU-side cache and a CPU-side cache. Once a request has been prefilled, its entire KV cache lives either fully on the GPU, making it a GPU-request, or fully on the CPU, making it a CPU-request. Requests are preferentially kept on the GPU as long as there is room, which fixes the wasted-memory problem of symmetric pipelining, since GPU memory that is not needed for offloading is used to hold as many GPU-requests as possible. The two sub-batches are then formed asymmetrically. For example, Batch-0 holds all the prefilling requests and GPU-requests plus a few

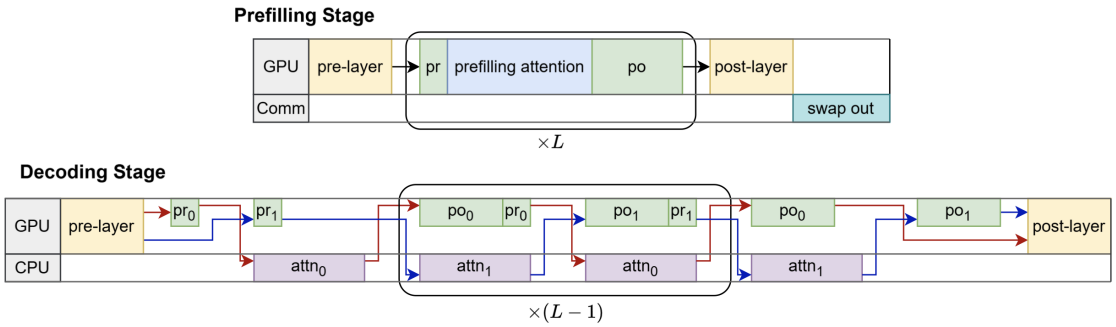


Figure 8.47: Symmetric pipelining; Image credit: [Jiang et al., 2024]

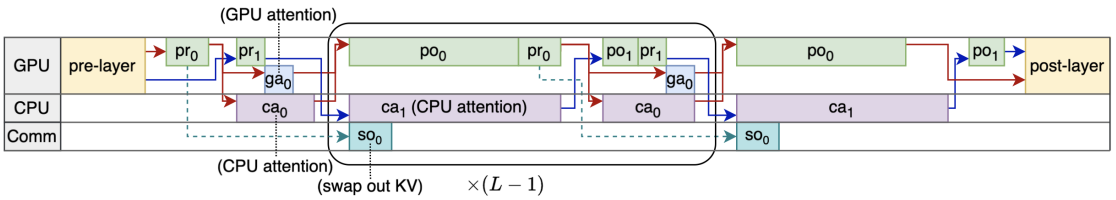


Figure 8.48: Asymmetric pipelining; Image credit: [Jiang et al., 2024]

CPU-requests, so it has a long linear stage on the GPU but little CPU attention work. Batch-1 holds most of the CPU-requests, so it has a long attention stage on the CPU but only a short linear stage. In Figure 8.48, execution alternates within each layer — while the GPU runs the long linear stage of batch-0, the CPU runs the attention of batch-1; the GPU then runs the short linear stage of batch-1 and the GPU-side attention of batch-0, while the CPU handles the few CPU-requests in batch-0. Because the long GPU stage hides the long CPU stage, neither device sits idle for long, and folding prefilling into batch-0 keeps the CPU busy during prefill too.

8.5.4 Offloading for Long-Context Serving

The offloading techniques discussed so far, like InfiniGen's token selection and NEO's compute offloading, were designed with a particular workload in mind — relatively short prompts followed by long decoding. Long-context serving stresses the memory hierarchy in a different way. Here the prompt itself, not the generated output, is the dominant cost. A single request may carry hundreds of thousands of tokens of context, and the same context is often reused across many requests, for example when several questions are asked about the same document, or when an agent repeatedly consults the same history. This reuse is what makes caching KV states across the memory hierarchy so attractive — instead of offloading only to survive a single request, systems now offload to preserve

a working set of contexts across many requests, spilling from GPU HBM to CPU DRAM, and sometimes further to local SSD or a remote memory pool.

Earlier in this chapter we treated the cost of moving KV cache data from CPU to GPU as something that could either be hidden behind computation, or reduced by fetching a smaller set of tokens. Both of these assumptions become shaky once the context being loaded is very long. First, the amount of data that must move across PCIe for a single request can now run into the tens of gigabytes, since practically the whole prompt may already be sitting in CPU memory from an earlier turn. Second, because we generally do want to reuse the entire cached prefix, rather than only its most attended tokens, the kind of selective fetching used by InfiniGen is not directly applicable here; the goal is not to attend to a subset of a long history during decoding, it is to bring back a long prefix during prefill so that it does not have to be recomputed. As a result, the time needed to load the cached prefix can, for long contexts, exceed the time needed to run the prefill computation on the newly arrived tokens.

The Page Size Dilemma

A more subtle problem comes from how the KV cache is laid out in memory. As we saw in §2.2, systems that manage GPU memory efficiently, following PagedAttention, divide the KV cache into small fixed-size pages so that memory can be allocated and reused flexibly. Typical page sizes are very small, often just a handful of tokens, because small pages let the system match cached prefixes precisely and avoid wasting memory on partially used pages. This is good for the GPU-resident cache, but it is a poor fit for data transfer. A basic fact about interconnects such as PCIe is that achieving good bandwidth utilization needs fairly large, contiguous transfers, typically in the range of a megabyte or more. Small KV cache pages, each only a few kilobytes, fall far short of this, so each transfer wastes most of the available bandwidth.

The obvious fix is to use larger pages so each transfer carries more data, but this creates a new problem. Larger pages make prefix matching coarser, since a request now only counts as a cache hit if it matches an entire page, not just a few tokens. This lowers the effective cache hit rate, which in turn increases how often the system falls back to recomputation. Choice of page size has a trade-off — small pages are good for cache hit rate but bad for transfer efficiency. Systems that pick one fixed page size are stuck accepting a compromise on one side or the other.

Delay Hits

The second family of problems has less to do with data movement itself and more to do with how requests are scheduled around it. Consider what happens when several requests that need the same long context arrive close together in time. If the first request causes a cache miss and its context has to be loaded or recomputed, and a second request for the same context arrives before that work finishes, most schedulers have no way of knowing this and will treat the second request as a cache miss as well, triggering a second, entirely redundant, load or recomputation. This effect is called a delay hit. Delay hits become more likely as request rates grow and as the time needed to resolve a

cache miss grows, both of which are exactly the conditions found in high-throughput, long-context serving. They are made worse by the asynchronous batch schedulers used by modern serving engines, which prepare the next batch of requests before the current one has finished executing, effectively widening the window during which a delay hit can occur.

Strata: A Hierarchical Context Caching System

Strata [Xie et al., 2026] is a serving system that addresses both of these problems together. It is organized around two components (Figure 8.49). A cache controller manages the data plane, i.e., the actual movement and placement of KV cache pages across GPU HBM, CPU DRAM, and external storage. A scheduler manages the control plane, i.e., the decision of which requests to batch together and when, taking cache loading cost explicitly into account. The two pieces share a data structure called the HiRadixTree, an extension of SGLang's existing radix tree for prefix matching, which additionally tracks, for every cached span of tokens, where its data currently lives in the hierarchy and how heavily it has been reused. This tree acts much like a page table, letting the scheduler reason about loading costs before a batch is even dispatched to the GPU.

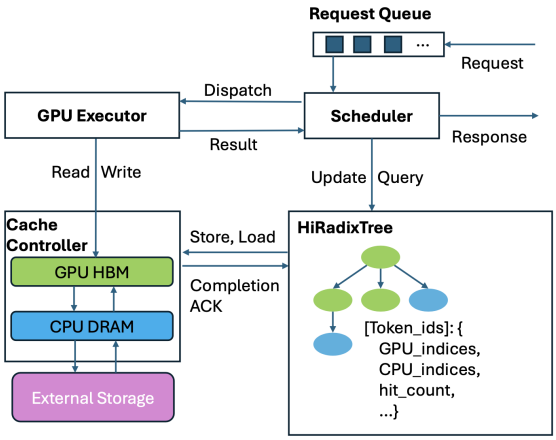


Figure 8.49: System architecture of Strata; Image credit: [Xie et al., 2026]

Strata's Data Plane: GPU-Assisted I/O and Decoupled Layouts

Resolving the page size dilemma first requires understanding why small pages transfer so badly. Strata explains this with Little's law — the sustained throughput of an I/O path is $X = C \cdot S/L$, where C is the number of transfers in flight, S the size of each, and L the latency of one transfer. The standard way of moving data between CPU and GPU, repeated calls to `cudaMemcpyAsync` that engage the GPU's DMA engine, does poorly on all three counts when the data is a collection of small pages. Each call carries a fixed CPU–GPU launch and scheduling overhead (large L), the number of

calls in flight is capped by CPU-side parallelism and driver queues (small C), and the page size pins S to a few kilobytes. The obvious lever is to raise S by enlarging pages, but that is exactly what hurts the hit rate. Strata instead attacks C and L directly, so that S can stay small.

Its mechanism is GPU-assisted I/O. Rather than issuing many copy calls, Strata launches a CUDA kernel whose thousands of threads each move a small chunk of data between GPU memory and pinned host memory, staging it through registers. This has three advantages. It provides high concurrency, since a GPU can keep thousands of such operations in flight against only tens on a CPU. It is efficient for small transfers, since a chunk of only 128 bytes is enough to use the hardware efficiently, far below the size of a page. And it is flexible about layout — because each thread computes its own source and destination address, the kernel can write data in a different layout than it read, at the cost of one extra address calculation.

The price is that an I/O kernel competes with the model's own kernels for SMs, registers, and cache. Strata limits this interference by launching only a few large thread blocks, which nudges the GPU's hardware scheduler to confine the kernel to a small set of SMs, and by using instructions that bypass the cache so the copies do not evict useful data.

The mechanism of GPU-assisted I/O solves a related problem about the choice of the optimal memory layout. The layout that is best for computation and the layout that is best for transfer no longer need to be the same, as long as converting between them is cheap, and GPU-assisted I/O makes the conversion essentially free. On the GPU, attention kernels want a layer-first layout, where all tokens of one layer are contiguous, since this is the order in which a layer reads keys and values. This layout is poor for transfer, because the keys and values of one page, which span all layers of the model, are scattered into one small fragment per layer. A page-first layout, where all layers of a page are stored together, is the opposite — a page becomes one contiguous block, but a GPU kernel would need an extra level of indirection to use it. Strata therefore keeps the GPU cache in the compute-friendly layer-first layout and stores host memory, and any storage behind it, in the transfer-friendly page-first layout (Figure 8.50). The I/O kernel converts between the two as it copies.

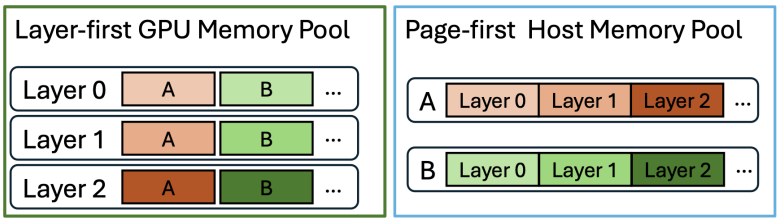


Figure 8.50: Layer first vs. page first layout; Image credit: [Xie et al., 2026]

Together, these pieces resolve the page size dilemma. Small pages are kept on the GPU, preserving fine-grained prefix matching and a high hit rate. In host memory and on SSD, a page is one large contiguous object spanning all layers, and the I/O kernel moves data between the two layouts

efficiently, so transfer efficiency no longer depends on the page size.

Strata handles tiers beyond host memory with the queue-based prefetching idea described earlier in this section. When a request's context is found on SSD, the cache controller immediately starts loading it into host memory, overlapping the load with the request's time in the queue. Once the scheduler dispatches the request, any prefetch still in flight is cancelled and the request uses whatever is already in host or GPU memory. This best-effort policy reflects the high and unpredictable latency of SSDs, unlike host-to-GPU transfers, which Strata overlaps layer by layer.

Strata's Control Plane: Cache-Aware Scheduling

Faster transfers help, but a long cached context can still take longer to load than the new tokens take to compute. The scheduler therefore treats PCIe bandwidth as a resource, just like GPU compute. It relies on the HiRadixTree, which tells it, for every queued request, how many tokens are already on the GPU, how many sit in host memory and must be loaded, and how many are new and must be computed. From these counts it estimates each request's load and compute cost before a batch is launched, and schedules them using several principles — defer requests that would repeat work already in progress, form batches where loads hide behind computation, share context where possible, and fill any remaining stalls with decoding.

Figure 8.51 illustrates these principles on a short queue, with a FIFO scheduler on top and Strata below. Each request is named by its context and an index, so A0 and A1 both use context A. In the delay hit case, FIFO puts A0 and A1 in one batch, so context A is computed twice, while Strata defers A1 until A is cached. Further, FIFO puts C0 and D0 together, so both contexts must be loaded and the GPU waits, while Strata pairs C0 with F0, whose context is already on the GPU, so the load hides behind computation. To load context once and share across requests, D0 and D1 are scheduled together, as a bundle hit. Finally, G0 has a long cached context and little new text, so no pairing helps, and Strata runs a decoding batch while G loads.

There are several principles behind the scheduling in Strata. The first one is to avoid delay hits. To spot requests that would wait on work already in progress, the HiRadixTree also holds temporary transient nodes. When the scheduler scans the queue, it adds a node marked in-queue for the uncached tokens of each request. When that request is dispatched, its nodes are marked in-flight, and when the computation finishes they become normal nodes that point to cached data. If a later request matches a transient node on enough tokens, it would repeat the same work, so the scheduler defers it to the next round. It also moves the request to the front of the queue, so it runs right after the cache becomes ready and its latency barely suffers.

The second principle is to form balanced batches. A batch is loading-bound when the ratio of tokens to load to new tokens to compute exceeds a threshold. The threshold depends on the hardware and model. The scheduler walks the queue in order, always starting from the first request so that no request starves. It admits a request if the ratio of load to compute for that batch stays below the threshold. Otherwise it moves the request to a deprioritized list in its original order, which is used at the end to fill any space left in the batch. Whenever a request is admitted, the scheduler also

scans the queue for other requests that share its context and admits them too. Such a bundle hit is the best case for performance because the shared context is loaded once and held once in GPU memory for the whole group, so the extra requests add computation without adding load.

Sometimes no arrangement avoids a stall, for example when a single request has a very long cached context and little new text. Here the scheduler holds back the prefill batch and runs a decoding batch while the context loads. This overlaps well because decoding mostly uses GPU memory bandwidth, while loading uses PCIe bandwidth, so the two barely compete.

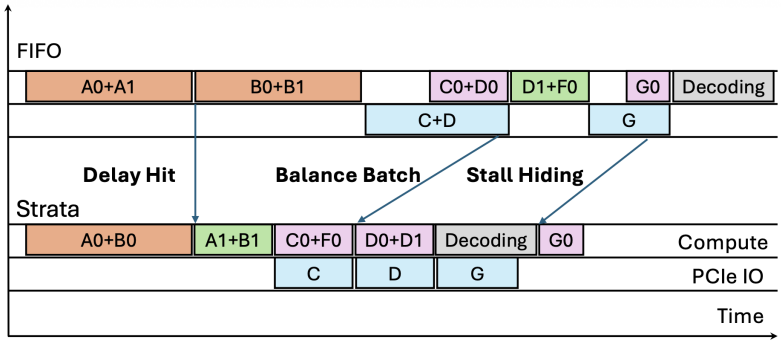


Figure 8.51: Scheduling policies in Strata. Orange: prefill with cache miss; green: cache hit on GPU; purple: cache hit in host memory; blue: data transfer; gray: decoding batch. *Xn* is the *n*-th request that uses context *X*. Image credit: [Xie et al., 2026]

All the systems in this section trade abundant host memory for scarce GPU memory, and all of them are limited by the narrow link between the two. Earlier we asked three questions — how much data must cross the link, how efficiently it can cross, and how its latency can be hidden. Each system answers a different one. InfiniGen reduces the amount of data by fetching only the tokens that matter. NEO avoids the transfer altogether by running decoding attention on the CPU, where the KV cache already lives, and keeps both devices busy with asymmetric pipelining. Strata makes the transfers themselves efficient with GPU-assisted I/O and decoupled layouts, and hides the remaining latency through cache-aware scheduling.

8.6 KV Cache Compression and Sparsity

In the earlier sections of this chapter, we saw how the KV cache lets us avoid recomputing the keys and values of past tokens at every decoding step, and how systems like PagedAttention manage this cache efficiently in GPU memory. Those techniques treat the size of the KV cache as fixed — for a given model and a given sequence length, the number of bytes needed to store keys and values is determined by the model architecture, and the job of the serving system is simply to allocate and move that memory around efficiently. This section looks at a complementary question — can we make the KV cache itself smaller? If we can store the same information using fewer bytes, then every technique we have discussed so far — batching, paging, scheduling — gets to work with a smaller memory footprint. This directly increases the number of requests we can batch together, which is usually the biggest lever for improving GPU throughput.

8.6.1 Why the KV Cache is Worth Compressing?

Recall that the KV cache grows linearly with sequence length, batch size, and model depth (number of layers and heads). For long-context workloads, or when serving many concurrent users, the KV cache can easily become larger than the model weights themselves. As we saw before, this leads to two problems. First, GPU high-bandwidth memory (HBM) is limited. Once the KV cache and model weights fill up available memory, the serving system simply cannot admit more requests into a batch, no matter how much spare compute capacity the GPU has. Second, decoding is a memory-bandwidth-bound operation — at every decode step, the GPU must read the entire KV cache of every sequence in the batch just to produce one new token per sequence. A larger KV cache therefore means more bytes to move per token generated, which directly slows down decoding. Reducing the size of the KV cache attacks both problems at once — it frees up memory for larger batches, and it reduces the amount of data that must be read from memory at each step, speeding up decoding. This is why KV cache compression has become one of the active areas of systems research for LLM inference.

Broadly speaking, there are two places where we can intervene to shrink the KV cache. Model-level techniques change the architecture of the model itself, usually by reducing the number of distinct key and value vectors the model computes and stores in the first place. These changes are decided at training time and are baked into the model architecture. Runtime (data-level) techniques keep the model architecture unchanged, but compress the KV cache values themselves at serving time, for example through quantization, or by identifying and discarding less important entries. These two strategies are not mutually exclusive. In fact, most production serving stacks today combine an architectural choice such as grouped-query attention with an additional runtime compression or eviction scheme.

8.6.2 Model-Level Compression: Grouped-Query Attention

In standard multi-head attention, every attention head has its own query, key, and value projection. If a model has H heads, it stores H separate key vectors and H separate value vectors per token, per layer. This is great for model quality, since each head can attend to a different aspect of the context, but it is expensive for the KV cache — the cache size scales directly with the number of heads.

An earlier idea, multi-query attention (MQA) [Shazeer, 2019], pushes this to the other extreme. Instead of giving every head its own key and value projection, all query heads share a single key head and a single value head. Only the query projections remain per-head. This shrinks the KV cache by a factor equal to the number of heads, and correspondingly speeds up decoding, since far fewer bytes need to be read from memory at each step. The catch is that sharing a single key/value head across all queries removes a lot of representational capacity, and MQA models tend to lose some accuracy as compared to full multi-head attention models.

Grouped-query attention (GQA) [Ainslie et al., 2023] is designed as a middle ground between multi-head attention and multi-query attention. Instead of every head having its own key/value pair (multi-head) or all heads sharing one key/value pair (multi-query), GQA divides the query heads into a small number of groups, and each group shares a single key head and a single value head. This is illustrated in Figure 8.52. With G groups, we get an attention mechanism that behaves like multi-head attention when G equals the number of heads H, and like multi-query attention when G equals 1. Any value of G in between interpolates smoothly between the two, letting a designer trade off KV cache size against model quality.

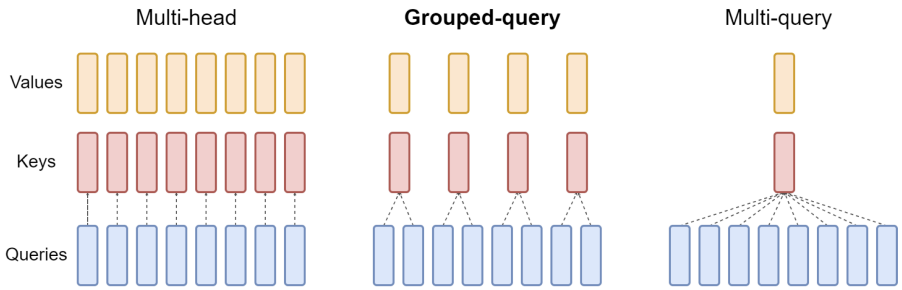


Figure 8.52: Overview of grouped-query method; Image credit: [Ainslie et al., 2023]

A second, equally important contribution of the paper is a recipe for obtaining a GQA model cheaply, without training from scratch. Rather than training a new model with fewer key/value heads from the start, they show that an existing multi-head checkpoint can be uptrained into a GQA (or MQA) model. The key/value heads within each group are combined, for example by mean-pooling the existing per-head projections, to initialize the smaller set of shared key/value projections. The model is then fine-tuned for a short additional period, using only about 5% of the original

pre-training compute. This means that a system does not need to choose between a high-quality multi-head model and a fast multi-query model at training time; it can train the best multi-head model it can afford, and then cheaply convert it into a GQA model for efficient serving.

8.6.3 Runtime Compression: Differentiated KV Cache Management

Model-level techniques like GQA reduce the KV cache uniformly, by the same factor for every token and every request. But not all parts of the KV cache are equally important at serving time. Some tokens contribute much more to the model's output than others, some attention heads attend broadly across the whole context while others focus narrowly on a few tokens, and keys and values do not affect the attention computation in the same way. Runtime compression techniques try to exploit these differences to compress the cache, without retraining the model. DiffKV [Y. Zhang et al., 2025] is a good example of a system built with this paradigm. It is built around the observation that KV cache entries differ from one another along multiple separate axes.

Keys vs. Values

The keys are used to compute attention scores, which then decide how much weight is placed on each value. Perturbing a key can shift the entire attention distribution for a token, while perturbing a value only changes the contribution of that one token to the output. Empirically, this means keys are more sensitive to precision loss than values. DiffKV therefore compresses keys and values with different precision instead of always using the same precision for both, keeping keys more precise while compressing values more aggressively.

Token Importance

Not every token in the context matters equally for future predictions. Some tokens (for example, the first few tokens of a sequence, or tokens that are frequently attended to) receive consistently high attention scores, while many others contribute very little. DiffKV uses attention scores to estimate token importance, and allocates more memory/precision to important tokens while compressing or dropping the less important ones more aggressively, rather than applying one compression level uniformly to every token in a sequence.

Compressing keys and values by different, dynamically varying amounts sounds straightforward from a modeling point of view, but it creates a genuine systems problem. In the memory management techniques we studied earlier in this chapter, such as PagedAttention, the KV cache is organized into fixed-size blocks precisely because every sequence, and every head within a sequence, needs the same amount of memory. Once we allow different heads and different tokens to use different, changing amounts of memory, this regularity disappears. Some heads need a large block, others a tiny one, and the amount needed for a given head can grow or shrink as generation proceeds and importance scores are updated. If handled naively, this leads to memory fragmentation — memory is freed in irregular chunks scattered throughout GPU memory, and over time it becomes hard to find enough contiguous space for new allocations even though the total free memory is sufficient.

Fragmentation wastes exactly the memory that compression was supposed to save, and can also slow down memory management itself.

DiffKV's answer to this problem is a memory manager that performs compaction, the process of moving live data together so that free memory forms contiguous regions, directly on the GPU and in parallel (Figure 8.53). Rather than compacting memory with a single sequential pass, DiffKV designs the compaction procedure so that many threads can pack fragmented entries into contiguous regions at the same time. This keeps the overhead of managing highly irregular, per-head, per-token memory allocations low enough that the compression gains actually translate into real improvements.

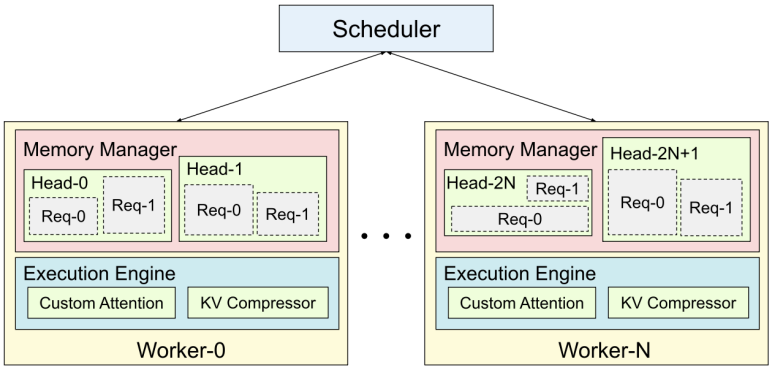


Figure 8.53: DiffKV architecture; Image credit: [Y. Zhang et al., 2025]

8.6.4 Sparse Attention and the KV Cache

So far in this section we have looked at ways to make each KV cache entry smaller, either by changing the model architecture, as in GQA, or by compressing entries at runtime with different precision for different keys, values, and tokens, as in DiffKV. There is a third, complementary idea for making long-context inference cheaper, called sparse attention. Sparse attention is based on the observation that at any given decoding step, a token usually does not need to attend to every single token that came before it. Instead, only a small subset of past tokens, say the top few thousand out of a hundred thousand, actually contribute meaningfully to the attention output. If we can identify this small subset in advance, we only need to compute attention scores and read KV cache entries for those tokens, instead of for the whole context. Dense attention compares the new token with all n previous tokens, so one decoding step costs $O(n)$ and generating a whole sequence costs $O(n^2)$. If we instead select a fixed number k of tokens, with $k \ll n$, each step reads only k keys and values. The expensive part of attention then stops growing with context length, and total cost grows roughly linearly with n. This is what makes very long context windows practical.

The simplest sparse attention methods are training-free — they take an already-trained dense

model and, at inference time, estimate which tokens are important using some cheap approximation of the attention scores, then only compute exact attention over the top-scoring tokens. This is easy to deploy, since it needs no retraining, but it tends to hurt model accuracy, especially on tasks that require reasoning over the full context, because the approximation used to pick important tokens is often crude. To avoid this accuracy loss, more recent models bake sparsity into training itself, so that the model learns to work well even though it only ever sees a sparse subset of the context during both training and inference. DeepSeek Sparse Attention, used in DeepSeek-V3.2 [DeepSeek-AI et al., 2025], is a prominent example of this trained approach — it adds a small extra module called an indexer, which computes a lightweight importance score for every past token, and only the top-k scoring tokens are then used in the full, expensive attention computation. Sparse attention reduces how much computation and how many KV cache bytes need to be touched at each step, but it does not shrink the KV cache that must be stored. Every token's key and value still has to be kept somewhere, since a token that looks unimportant now might be selected as important at some later decoding step.

Most sparse attention methods share the same building blocks. A budget k is the number of tokens each step is allowed to attend to. A scorer (often called an indexer) gives each past token an easy-to-compute importance score, and the top-k highest-scoring tokens are chosen for the exact attention computation. Different methods differ mainly in when and how they choose the top-k tokens. Static sparsity decides once which tokens to keep, and permanently drops the rest, so the same subset of tokens are used for all later steps. Dynamic sparsity keeps every token around, but re-selects the top-k at every decoding step, based on the current query. The difference matters for memory — static methods shrink the stored KV cache, while dynamic methods do not.

H2O [Z. Zhang et al., 2023] is an example of a static sparsity method. It observes that a small set of "heavy hitter" tokens receives most of the attention weight. It keeps a running sum of the attention each token has received, then stores only the heavy hitters plus the most recent tokens, using the same budget in every layer. Everything else is evicted for good. This genuinely shrinks the KV cache, but an evicted token can never come back, so accuracy drops if a token that looked unimportant earlier becomes important later.

Quest [Tang et al., 2024] is a dynamic method built on the idea that token importance depends on the current query. It keeps the full KV cache but splits it into small blocks (pages) and stores the element-wise minimum and maximum of the keys in each page. At every step, it uses the current query together with these min/max values to estimate the highest score any token in a page could get, and then reads only the top-k pages. Quest does not save memory, but it avoids reading most of the cache at each step and loses much less accuracy than static eviction.

8.6.5 ECHO: Offloading the KV Cache of Sparse-Attention Models

Because the KV cache memory of a sparse-attention model still grows linearly with context length, GPU high-bandwidth memory quickly becomes the limiting resource in long-context serving, regardless of how sparse the attention computation itself is. For example, serving DeepSeek-V3.2 on eight

NVIDIA H20 GPUs leaves room for only a few hundred thousand tokens of KV cache once model weights are accounted for, which allows just three or four concurrent hundred-thousand-token requests per worker. This severely limits how many long-context requests a sparse-attention model can serve at once, defeating much of the purpose of using sparse attention in the first place.

As we saw before, the natural way around this wall is to offload most of the KV cache to host DRAM, which is much larger than GPU memory, and bring back only the keys and values of the tokens that sparse attention actually selects at each step, a process usually termed "recall". Two problems stand in the way of doing this well in a real serving system. First, decoding in production serving frameworks almost always runs inside GPU graphs, such as CUDA graphs, so that the overhead of launching many small GPU kernels one at a time is avoided. But the bookkeeping needed for offloading, such as deciding which cache slots are free and updating them as tokens are evicted or recalled, is usually implemented with dynamically sized data structures, which cannot be captured inside a fixed graph. This forces existing offloading systems to fall back on CPU control at every decoding step, undoing much of the benefit that graph execution was meant to provide. Second, moving keys and values from host memory to the GPU over a link such as PCIe is one to two orders of magnitude slower than reading from GPU memory directly, so recall can easily become the new bottleneck. Existing systems try to hide this cost by prefetching tokens ahead of time, guessing which ones will be needed based on similarity between layers or between decoding steps. This guesswork is not always accurate, and when it is wrong, it silently degrades the model's output quality, which defeats the point of using a trained sparse attention method rather than a cheap training-free one.

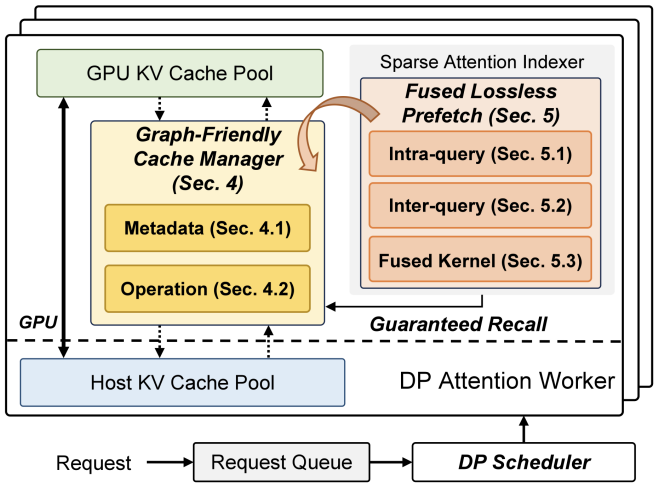


Figure 8.54: Overview of the ECHO design; Image credit: [G. Liu et al., 2026]

ECHO [G. Liu et al., 2026] tackles the first problem with what it calls a "graph-friendly" cache

manager (Figure 8.54). Rather than representing the state of the KV cache pool using data structures that change size at runtime, ECHO stores all of its bookkeeping, such as which GPU slots are free and which host-memory token maps to which GPU slot, as fixed-length integer tensors on the GPU. The basic cache operations such as allocating space, freeing space, and recalling tokens from host memory, are then implemented as fully parallel GPU kernels that operate on these fixed-length tensors, for instance by having every GPU thread atomically claim a free slot in parallel. Because none of these operations change the shape of any tensor, the entire decoding step, cache management included, can be captured inside a single GPU graph, removing the CPU-side overhead that limited earlier offloading systems.

ECHO tackles the second problem with what it calls lossless prefetching. The key observation is that computing the importance scores for all preceding tokens takes measurably longer than transferring the tokens that will eventually be selected, since the scoring work grows with the full context length n while the number of tokens that need to be recalled stays fixed at k. ECHO exploits this timing gap by predicting, using a simple moving average over previous decoding steps, the score threshold a token will need to clear to end up among the top-k selected tokens. As soon as a token's partially computed score crosses this predicted threshold, its key and value can be fetched from host memory immediately, while the indexer is still busy scoring the remaining tokens. Since the prediction only affects when a transfer is started, and the final top-k selection used for attention is still computed exactly once the indexer finishes, this prefetching never changes the model's actual output, unlike the earlier similarity-based schemes.

8.6.6 Communicating KV Cache in Disaggregated Serving

The techniques discussed above, GQA, DiffKV, and ECHO, all assume that the KV cache, however it is managed, stays on machines that are cooperating closely, whether that means one GPU or a GPU together with its own host memory. Modern serving deployments increasingly break this assumption. As we saw before, to scale beyond a single node, production systems often separate prefill and decode onto different GPUs, so that each stage can be scaled and optimized on its own. Systems also increasingly move the KV cache into a separate storage tier or a shared remote pool, so that it can be reused across requests, for example in retrieval-augmented generation or multi-turn conversations (Figure 8.55). In both designs, the KV cache stops being something that simply lives in GPU memory and instead becomes a payload that has to be sent across a network or an I/O link. Because these links are often far slower than GPU memory bandwidth, this transfer can dominate end-to-end latency.

KV cache compression, of the kind discussed earlier in this section, is the obvious way to shrink this transfer, since a smaller cache takes less time to send. The trouble is that almost every existing compression method is set up with one fixed configuration and a fixed quantization precision, chosen once and reused for every request. This turns out to be a bad assumption in two separate ways. First, the best compression method depends on the workload. A method that gives the best accuracy on one kind of task, such as long-document question answering, can rank near the bottom on another

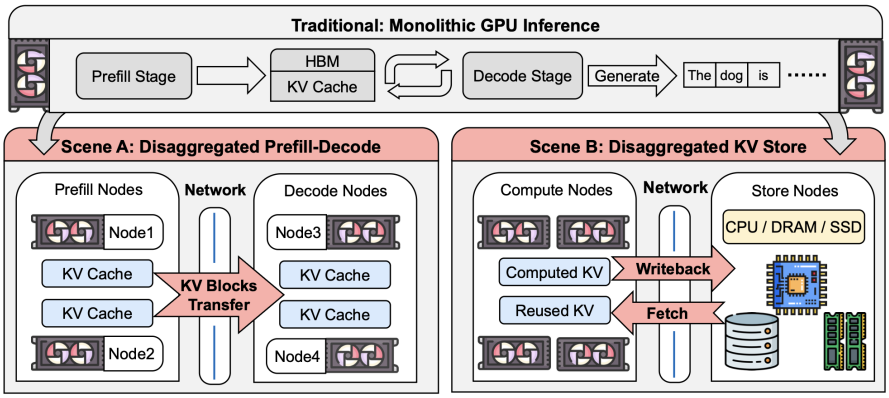


Figure 8.55: Disaggregated serving systems; Image credit: [Z. Liu et al., 2026]

kind of task, such as code generation, because different tasks produce KV caches with very different statistical structure. Second, the best method also depends on the network bandwidth available at that moment. At low bandwidth, spending extra compute to compress harder pays off because it saves a lot of transfer time. As bandwidth increases, though, the fixed cost of compressing and decompressing the cache can outweigh the shrinking transfer-time savings, so a method that helped at low bandwidth can end up making latency worse at high bandwidth. Moreover, searching every possible combination of transform, quantization setting, and codec parameter is far too slow to do online, since the number of combinations grows roughly exponentially with the number of tunable choices. A transform is a reversible preprocessing step applied before quantization. KV values often contain a few extreme outliers that force a quantizer to use a wide range and waste its limited bits. Transforms, such as rotating the vectors or rescaling channels, spread these outliers out so that the values fit a narrow range and lose less information when rounded to fewer bits. The codec stage is then ordinary lossless entropy coding (like Huffman or arithmetic coding), which gives frequent values shorter bit patterns.

KVServe [Z. Liu et al., 2026] treats KV compression as a decision that should be made per request based on current conditions. It starts by decomposing existing compression methods into a three-stage pipeline — a transform stage that reshapes the KV distribution to make it easier to quantize, a quantizer stage that reduces width, possibly using different precision for different attention heads, and a codec stage that entropy-codes the result to shrink it further (Figure 8.56). Searching this large strategy space exhaustively is not practical, so KVServe instead uses Bayesian optimization to predict which untested configurations are likely to be both accurate and highly compressive. Choosing among these candidates at request time still requires knowing the current bandwidth, the latency budget, and the quality requirement. KVServe uses a simple latency model to show that, for a fixed compression profile, whether compression helps or hurts depends only on whether the available

bandwidth is below a threshold computed from that profile's compression ratio and its encoding and decoding speed.

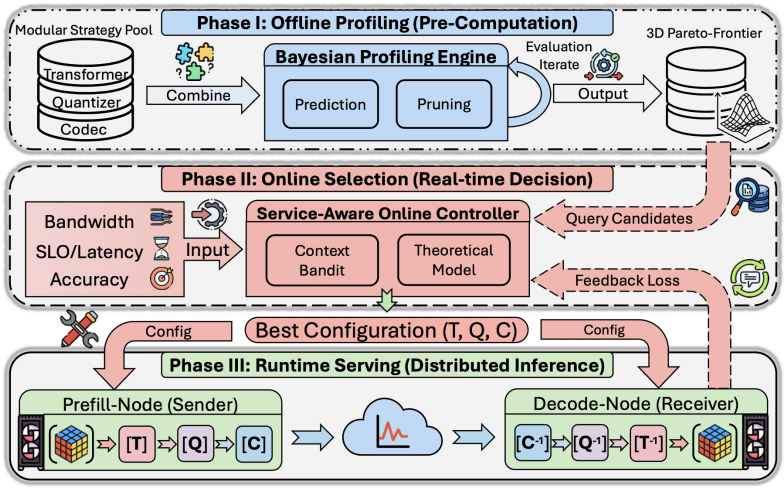


Figure 8.56: Overview architecture of KVServe; Image credit: [Z. Liu et al., 2026]

This section looked at the KV cache compression problem from four angles. Model-level changes like GQA shrink the cache for every token. Runtime methods like DiffKV shrink it further by quantizing keys, values, and tokens differently, at the cost of harder memory management. Sparse attention does not necessarily shrink the cache, but it cuts how much of it is read per step; when the full cache no longer fits on the GPU, systems like ECHO offload it and recall only what is needed for sparse attention. Finally, once the cache has to travel across machines, KVServe shows that the best compression choice depends on the workload and on the network. In practice, these ideas are combined, and each one targets a different part of the same bottleneck — the memory and bandwidth cost of the KV cache.

8.7 An Inference Engine in Practice: Nano-vLLM

The previous sections described the ideas that make LLM serving efficient — KV caching, batching and scheduling, memory management with paged attention, prefix sharing, CPU offloading, and KV cache compression. These ideas do not live in isolation; a real system has to combine all of them inside a single piece of software called an inference engine. In this section we look at the concrete structure of such an engine. We first describe what components any LLM inference engine needs. We then study a small, readable, open source implementation called Nano-vLLM [GeeeekExplorer, 2025], and walk through its engine, scheduler, and block manager to see how the ideas from earlier sections turn into actual classes and functions. Nano-vLLM is a good teaching example because it implements KV caching, paged attention, prefix caching, continuous batching, tensor parallelism, and CUDA graphs, in about twelve hundred lines of Python, while still reaching a throughput comparable to vLLM itself.

8.7.1 Core Components of an Inference Engine

Almost every LLM serving engine, however large or small, is built around the same handful of components, connected in the same basic way (Figure 8.57). It helps to think of the engine as a small operating system for language model requests — users submit requests, the engine decides which requests to run and when, and a worker process actually runs the model.

The engine front end is the part that faces the user. It accepts a prompt and a set of sampling parameters, turns the prompt into token ids using a tokenizer, and wraps the result in an internal object called a sequence. A sequence records the prompt tokens, the tokens generated so far, and the status of the request, e.g., waiting to start, running, or finished. In this section, the terms "request" and "sequence" mean the same thing. Engines receive requests in one of two ways. In offline mode, the user hands over a list of prompts and waits until all outputs are ready. In online mode, requests arrive over time, for example through an HTTP server, and results are streamed back as they are produced. Nano-vLLM supports only offline mode, which keeps its code small; the scheduler and the other components work the same way in both modes.

The scheduler decides, at every step of generation, which requests should be part of the next batch that is sent to the GPU. As discussed in the batching section, the scheduler must balance two competing goals — keeping GPU utilization high by batching many requests together, and keeping latency low by not making new requests wait too long behind large ones. The scheduler usually maintains at least two queues, one for requests that are waiting to be admitted and one for requests that are already running, and it moves requests between these queues as resources become available or run out.

The memory manager, often called a block manager, is responsible for the KV cache. Because the KV cache grows with every generated token, and because different requests finish at different times, the engine cannot simply allocate one large contiguous buffer per request ahead of time without wasting memory. Instead, as we saw in the memory management section, the KV cache is split

into fixed size blocks, and the block manager hands out blocks to requests on demand, reclaims them when a request finishes, and, if prefix caching is supported, keeps track of which blocks can be shared between requests that happen to share a prefix.

The model runner, or executor, is the component that actually owns the model weights and runs the forward pass on the GPU. It takes the batch of requests chosen by the scheduler, converts them into the tensors the model expects, such as input token ids and position ids, runs the transformer layers, and produces logits for the next token of every sequence in the batch. When the engine uses tensor parallelism, there is one model runner per GPU, each holding a shard of the weights, and they must be coordinated so that every runner executes the same batch at the same time.

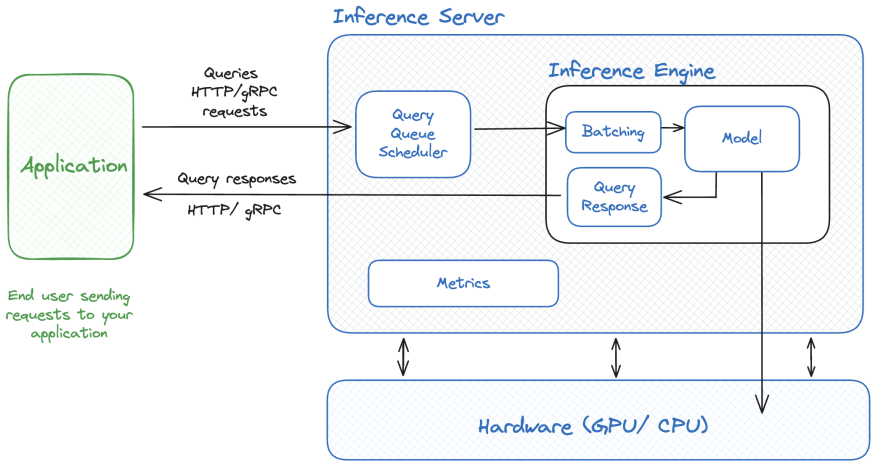


Figure 8.57: LLM inference engine; Image credit [Koyeb, 2024]

The attention backend sits inside the model runner and is where paged attention actually happens. It is responsible for writing newly computed keys and values into the right slots of the KV cache, and for computing attention scores against the cached keys and values, whether the request is in the prefill phase, where a whole prompt is processed at once, or the decode phase, where only a single new token is attended over.

Finally, the sampler turns the logits produced by the model into an actual next token, using the sampling parameters attached to each request, such as temperature. Once a token has been sampled, it is appended to the corresponding request, and the engine checks whether the request is finished, for example because an end of sequence token was produced or a maximum length was reached, and returns the result to the user if so.

As long as there are unfinished requests, the engine schedules a batch, runs the model, samples output tokens, updates states of requests, and repeats the process. This loop is often called a step, and

the throughput of the whole engine is essentially the number of tokens produced per step divided by the time each step takes. Inference engines typically measure several such metrics like token throughput, time to first token (TTFT), and time between tokens (TBT).

Nano-vLLM is a lightweight reimplementation of the essential ideas of vLLM. It supports offline batch inference, prefix caching, tensor parallelism, and CUDA graph capture. Because it is small enough to read in full, and because it mirrors the design of a production system, it is an excellent way to see how the six components described above turn into real code.

8.7.2 Starting the Engine

The user facing class is `LLMEngine`. Its constructor is short, but it already reveals the process structure of the whole system.

Code 8.1: LLMEngine Initialization

```
class LLMEngine:

    def __init__(self, model, **kwargs):
        config_fields = {field.name for field in fields(Config)}
        config_kwargs = {k: v for k, v in kwargs.items() if k in config_fields}
        config = Config(model, **config_kwargs)
        self.ps = []
        self.events = []
        ctx = mp.get_context("spawn")
        for i in range(1, config.tensor_parallel_size):
            event = ctx.Event()
            process = ctx.Process(target=ModelRunner, args=(config, i, event))
            process.start()
            self.ps.append(process)
            self.events.append(event)
        self.model_runner = ModelRunner(config, 0, self.events)
        self.tokenizer = AutoTokenizer.from_pretrained(config.model,
    use_fast=True)
        config.eos = self.tokenizer.eos_token_id
        self.scheduler = Scheduler(config)
        atexit.register(self.exit)
```

If tensor parallelism is not used, the whole engine lives in a single process, and this process owns one `ModelRunner` that holds the entire model. If the tensor parallel size is greater than one, the constructor spawns one extra operating system process for every additional GPU rank, and each of those processes builds its own `ModelRunner`, holding only its own shard of the weights. The

main process also builds a `ModelRunner` for rank zero, and in addition owns the tokenizer and the scheduler, since only one process needs to turn text into tokens and decide what to schedule. The `Event` objects are used to signal the worker processes that a new command from the main process is ready to be executed.

8.7.3 Scheduler

The `Scheduler` owns the block manager and the two queues introduced earlier. The `waiting` queue holds requests that have not yet been admitted, and the `running` queue holds requests that currently own KV cache blocks and are being decoded.

Code 8.2: Scheduler State

```
class Scheduler:

    def __init__(self, config: Config):
        self.max_num_seqs = config.max_num_seqs
        self.max_num_batched_tokens = config.max_num_batched_tokens
        self.eos = config.eos
        self.block_manager = BlockManager(config.num_kvcache_blocks,
    config.kvcache_block_size)
        self.waiting: deque[Sequence] = deque()
        self.running: deque[Sequence] = deque()
```

`max_num_seqs` bounds how many requests may run at once, and `max_num_batched_tokens` bounds the total number of tokens that may be processed in a single step (token budget), which matters most during prefill, where a single request can contribute many tokens at once.

The Block Manager

The block manager is the part of Nano-vLLM that implements paged attention style memory management, together with prefix caching. It works with two kinds of blocks: logical blocks, which are lightweight Python objects that only store metadata about which tokens live in a block, and physical blocks, which are slices of one large preallocated tensor that actually stores keys and values on the GPU. The size of this tensor is not configured by the user. At start-up the model runner performs a warm-up forward pass at the largest allowed batch, measures the peak memory used by weights and activations, and assigns all remaining memory (up to a configured utilization fraction of the GPU) to the KV cache. Dividing this budget by the size of one block gives the number of blocks the block manager has to manage.

Code 8.3: A Logical Block and the Block Manager

```
class Block:

    def __init__(self, block_id):
        self.block_id = block_id
        self.ref_count = 0
        self.hash = -1
        self.token_ids = []

    def update(self, hash: int, token_ids: list[int]):
        self.hash = hash
        self.token_ids = token_ids

    def reset(self):
        self.ref_count = 1
        self.hash = -1
        self.token_ids = []


class BlockManager:

    def __init__(self, num_blocks: int, block_size: int):
        self.block_size = block_size
        self.blocks: list[Block] = [Block(i) for i in range(num_blocks)]
        self.hash_to_block_id: dict[int, int] = dict()
        self.free_block_ids: deque[int] = deque(range(num_blocks))
        self.used_block_ids: set[int] = set()
```

Every block carries a reference count, since more than one sequence may share it once prefix caching is taken into account, and a hash. The hash of a block is computed from the token ids it contains together with the hash of the block before it, so that two blocks can only be considered identical if their entire prefix, not just their own content, matches (Figure 8.19).

When a new prefill request is admitted, `allocate` walks through the blocks the request will need, one block worth of tokens at a time, and for each one checks whether an identical block, with an identical prefix, already exists in `hash_to_block_id`. If it does, the existing block is reused and its reference count is increased, its KV cache values are simply read again rather than recomputed, and `num_cached_tokens` on the sequence is advanced. The moment a block fails to match, every later block in the same request is guaranteed to miss as well, since its prefix has already diverged, so the loop stops trying to match and switches to allocating fresh blocks for the rest of the request.

Code 8.4: Allocating Blocks with Prefix Cache Matching

```
def allocate(self, seq: Sequence):
    assert not seq.block_table
    h = -1
    cache_miss = False
    for i in range(seq.num_blocks):
        token_ids = seq.block(i)
        h = self.compute_hash(token_ids, h) if len(token_ids) == self.block_size
    else -1
        block_id = self.hash_to_block_id.get(h, -1)
        if block_id == -1 or self.blocks[block_id].token_ids != token_ids:
            cache_miss = True
        if cache_miss:
            block_id = self.free_block_ids[0]
            block = self._allocate_block(block_id)
        else:
            seq.num_cached_tokens += self.block_size
            if block_id in self.used_block_ids:
                block = self.blocks[block_id]
                block.ref_count += 1
            else:
                block = self._allocate_block(block_id)
        if h != -1:
            block.update(h, token_ids)
            self.hash_to_block_id[h] = block_id
        seq.block_table.append(block_id)
```

Scheduling Requests

The scheduler decides, once per step, which sequences form the next batch. Nano-vLLM follows a prefill-first policy — any waiting prefill requests are always scheduled before decode requests are allowed to continue, and if the system runs out of KV cache blocks, running decode requests can be preempted, that is, evicted back into the waiting queue, to free space for others.

The function returns a pair — the list of sequences to run in this step, and a boolean flag that is true for a prefill step and false for a decode step. The model runner needs the flag because the two kinds of steps prepare their inputs differently. The block manager helpers are simple — `can_allocate` checks that there are enough free blocks for the whole prompt, and `can_append` checks that the sequence can store its next token. A new block is needed only when the next token starts a new block, so `can_append` usually succeeds without free space.

Code 8.5: The Scheduling Loop

```
def schedule(self) -> tuple[list[Sequence], bool]:
    # prefill
    scheduled_seqs = []
    num_seqs = 0
    num_batched_tokens = 0
    while self.waiting and num_seqs < self.max_num_seqs:
        seq = self.waiting[0]
        if num_batched_tokens + len(seq) > self.max_num_batched_tokens or not
    self.block_manager.can_allocate(seq):
            break
        num_seqs += 1
        self.block_manager.allocate(seq)
        num_batched_tokens += len(seq) - seq.num_cached_tokens
        seq.status = SequenceStatus.RUNNING
        self.waiting.popleft()
        self.running.append(seq)
        scheduled_seqs.append(seq)
    if scheduled_seqs:
        return scheduled_seqs, True

    # decode
    while self.running and num_seqs < self.max_num_seqs:
        seq = self.running.popleft()
        while not self.block_manager.can_append(seq):
            if self.running:
                self.preempt(self.running.pop())
            else:
                self.preempt(seq)
                break
        else:
            num_seqs += 1
            self.block_manager.may_append(seq)
            scheduled_seqs.append(seq)
    self.running.extendleft(reversed(scheduled_seqs))
    return scheduled_seqs, False
```

The first loop takes sequences from the front of the waiting queue. For each one it checks two things — does the sequence fit in the token budget, and does the block manager have enough free blocks? The token check is conservative, because it counts the full prompt length before knowing

how much is cached; the budget is then charged only for the uncached tokens. If both checks pass, the sequence gets its blocks, moves to the running queue, and joins the batch. As soon as at least one sequence is scheduled, the function returns with the prefill flag set, and decoding waits for the next step. This is what gives prefill priority.

The second loop runs only when no prefill was scheduled, that is, when the waiting queue is empty or its first sequence does not fit. It takes sequences from the front of the running queue and tries to reserve room for the token each will produce. If there is no room, it preempts sequences from the back of the queue, so the oldest sequences keep making progress. For example, let the running queue be $[A, B, C]$ with no free blocks. We take A, see that it cannot append, and preempt C, which frees its blocks. Now A can append and is scheduled. If the queue had run out, A would preempt itself. In Python, the `else` of a `while` loop runs only if the loop ended without `break`, so a sequence is scheduled only if room was found. Preempting a sequence frees all its blocks and puts it at the front of the waiting queue. Its tokens are kept, so when it is scheduled again, the whole text so far is prefilled again. Prefix caching can make this cheaper if the old blocks have not been overwritten.

8.7.4 Model Runner, Attention, and CUDA Graphs

The batch chosen by the scheduler is executed by the `ModelRunner`. Each GPU rank owns exactly one `ModelRunner`, which runs in its own process. The main process sends it commands, such as "run this batch", by writing them to shared memory and setting the rank's `Event`, so all ranks execute the same batch at the same time. At construction time it initializes the distributed process group, loads its shard of the weights, runs the warm-up pass used to size the KV cache, preallocates that cache, and, unless eager execution is requested, captures CUDA graphs for decoding.

Paged Attention

Before running the model, the runner packs the batch into flat tensors and stores per-step metadata in a small global context object, holding the prefill flag, the cumulative sequence lengths, the slot each new token will occupy in the KV cache, and the block tables. Every attention layer reads this context instead of receiving extra arguments. The layer first writes the new keys and values into their cache slots with a small Triton kernel, then calls a FlashAttention kernel. For prefill, all prompts are concatenated and attention is computed causally within each sequence only; if prefix caching applies, only uncached tokens appear as queries while keys and values are read from the cache via the block table. For decode, each sequence contributes one query token and the kernel gathers that sequence's blocks directly from the paged cache.

CUDA Graphs

During prefill, tensor shapes change at every step, which rules out graph capture. During decode, every sequence contributes exactly one token and the KV cache has a fixed shape, so only the batch size varies. The runner therefore captures one CUDA graph per representative batch size, from 1 up to 512, which replaces many small kernel launches with a single graph replay. At inference time a

batch is rounded up to the nearest captured size, copied into static input tensors, and replayed.

8.7.5 One Full Step of Generation

We can now put the pieces together. The top level loop that drives generation for a batch of prompts looks like this.

Code 8.6: The Engine Step and the Outer Generation Loop

```
def step(self):
    seqs, is_prefill = self.scheduler.schedule()
    token_ids = self.model_runner.call("run", seqs, is_prefill)
    self.scheduler.postprocess(seqs, token_ids)
    outputs = [(seq.seq_id, seq.completion_token_ids) for seq in seqs if
    seq.is_finished]
    num_tokens = sum(len(seq) for seq in seqs) if is_prefill else -len(seqs)
    return outputs, num_tokens

def generate(self, prompts, sampling_params):
    for prompt, sp in zip(prompts, sampling_params):
        self.add_request(prompt, sp)
    outputs = {}
    while not self.is_finished():
        output, num_tokens = self.step()
        for seq_id, token_ids in output:
            outputs[seq_id] = token_ids
    outputs = [outputs[seq_id] for seq_id in sorted(outputs.keys())]
    return [{"text": self.tokenizer.decode(t), "token_ids": t} for t in outputs]
```

Every call to `step` performs the whole loop described at the start of this section in miniature — the scheduler chooses a batch and secures KV cache blocks for it, the model runner prepares tensors, runs the model, possibly by replaying a captured CUDA graph, and samples the next token for every sequence in the batch. Finally, `postprocess` appends the new tokens to their sequences, checks each one for completion, and, for any sequence that has just finished, releases its KV cache blocks back to the block manager.

This loop runs until every submitted prompt has finished generating. A single call to `step` is either a prefill step for newly admitted requests or a decode step for the running ones, but because the batch is chosen fresh at every step, new requests can join and finished ones can leave at any time. This is exactly the continuous batching behaviour described in the batching and scheduling section — the running set is a mixture of requests at different stages of generation, rather than fixed for the lifetime of a static batch.

Code 8.7: Post Processing After a Model Step

```
def postprocess(self, seqs: list[Sequence], token_ids: list[int]):
    for seq, token_id in zip(seqs, token_ids):
        seq.append_token(token_id)
        if (not seq.ignore_eos and token_id == self.eos) or
    seq.num_completion_tokens == seq.max_tokens:
            seq.status = SequenceStatus.FINISHED
            self.block_manager.deallocate(seq)
            self.running.remove(seq)
```

Although Nano-vLLM is a simplified system, and leaves out features found in production engines, such as more elaborate scheduling policies, speculative decoding, or CPU offloading of cold KV cache blocks, its structure — one engine process coordinating a scheduler, a block manager, and one model runner per GPU, connected through a per-step batch of sequences and a paged KV cache — is essentially the same structure used by large scale serving systems in production. Reading through a codebase this size is a practical way to see how the abstract ideas of the previous sections, like key value caching, paged memory, prefix sharing, and batching, come together into a single working system.

Programming Assignment 8.3: Decode-Priority Scheduling in Nano-vLLM

In this assignment, you will explore how the scheduling policy of an LLM inference engine shapes the latency and throughput it delivers. Nano-vLLM [GeeeekExplorer, 2025], a small open source reimplementation of vLLM, currently follows a prefill-first policy — whenever new prompts are waiting, they are processed before running requests are allowed to generate their next token. Your job is to modify the scheduler so that it follows the opposite, decode-first policy — as long as any request is running, the engine decodes it, and new prompts are prefilled only when the running queue is empty. To do this correctly, you will need to handle the case where preemption under KV cache pressure empties the running queue, so that the scheduler never returns an empty batch and no request is lost. You will then run the same workloads under both policies and compare throughput, time to first token, and per-token latency, and explain the trade-offs you observe. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/decode_prio_nanovllm

8.8 Summary

In this chapter, we began by understanding what inference is, and how a trained large language model generates text one token at a time. We saw that the model does not output words directly. Instead, it produces a probability distribution over the vocabulary, and a decoding strategy such as greedy decoding, beam search, temperature sampling, top-k, or top-p is used to pick the next token. Since each new token depends on all the tokens before it, generation is autoregressive, and naively reprocessing the entire sequence at every step would make the compute cost grow quadratically.

We then looked at how real systems split generation into two phases. The prefill phase processes the entire prompt at once and is compute bound, while the decode phase generates tokens one at a time and is memory-bandwidth bound. Key-Value (KV) caching avoids recomputation by storing the key and value tensors of all previous tokens, so that each decode step only needs to compute them for the newest token. Since decode performs very little computation per byte of data read from memory, batching the decode steps of multiple requests together is essential for good GPU utilization.

Next, we studied how GPU memory is managed during inference. Model parameters and the KV cache together take up most of the GPU memory, so the number of requests that can be batched depends on how efficiently the KV cache is stored. Traditional systems allocate one contiguous region per request, sized for the maximum possible sequence length, which causes both internal and external fragmentation. PagedAttention fixes this by borrowing the idea of virtual memory paging from operating systems. It divides the KV cache into small fixed-size blocks that are allocated on demand and mapped to physical memory through a block table, so the blocks of a request need not be contiguous. This block-level indirection also allows memory to be shared across sequences, since sequences with a common prefix can point to the same physical blocks.

We then moved on to batching and scheduling. We saw that request-level scheduling wastes GPU capacity, because a batch must wait for its longest request to finish before any new request can be admitted. Orca addresses this with iteration-level scheduling and continuous batching, where the batch is reformed after every token generation step, and requests can join and leave the batch as required. We also compared decode-prioritizing and prefill-prioritizing schedules, and saw that they trade off throughput and time to first token against the time between tokens. Sarathi-Serve resolves this tension through chunked prefill, where a prompt is split into chunks, and the chunks are then batched along with ongoing decodes under a fixed token budget. This gives us stall-free token generation, and uniform iteration times. POD-Attention fuses prefill and decode attention kernels using SM-aware CTA scheduling, which makes sure that prefill and decode run side by side on the same SM, using complementary resources. Finally, we looked at DistServe, which questions the premise of sharing a GPU between prefill and decode altogether and disaggregates them onto separate GPU instances, so that each can be scaled and parallelized independently while only the KV cache is transferred between them.

Next, we studied prefix sharing, which avoids recomputing the KV cache for prefixes that are common across requests, such as system prompts, few-shot examples, and chat histories. SGLang's

RadixAttention stores KV caches in a radix tree, which finds the longest cached prefix of a new request, so that only the remaining tokens need prefill. Cache-aware scheduling then visits requests in depth-first order over the tree, scheduling requests with shared prefixes close together in time. We also saw how FastTree groups queries that share a context so that the shared KV cache is loaded once and attention becomes a matrix-matrix computation, and how Feather shows that prefix homogeneity of a batch can matter as much as its size. Finally, we looked at retrieval-augmented generation (RAG), where plain prefix caching works poorly because different requests retrieve overlapping documents in different orders. RAGCache handles this with a knowledge tree over documents, placed across GPU and host memory, while reordering requests and documents creates more prefix matches. CacheBlend goes further and reuses the KV cache of documents at any position, recomputing only a small set of tokens most affected by the missing cross-attention.

We then turned to CPU offloading, which uses the cheaper and larger host memory, and sometimes the host compute, to relax the limits of GPU memory. For KV cache offloading, prefetching alone cannot hide the transfer of a long cache, so InfiniGen speculates which tokens will matter in the next layer and fetches only those from the CPU. NEO instead offloads decoding attention to the CPU, since it is memory-bandwidth bound and the CPU-GPU gap in bandwidth is much smaller than in compute. For long-context serving, we looked at Strata, which addresses the page size dilemma by using different KV cache layouts for compute and for transfer, and which avoids redundant work caused by delay hits through cache-aware scheduling.

We then looked at KV cache compression, which makes the KV cache itself smaller so that larger batches fit in memory and less data is read at each decode step. At the model level, grouped-query attention lets groups of query heads share a single key and value head. At runtime, DiffKV compresses the cache using different precision for keys and values, and for tokens of different importance. We also studied sparse attention, where each decoding step attends only to the top-scoring tokens, as in DeepSeek Sparse Attention. Since this does not reduce the stored cache, ECHO offloads the cache to host memory using a graph-friendly cache manager, and uses lossless prefetching to hide the recall latency without changing the output. Finally, we saw that when prefill and decode are disaggregated, the KV cache must be sent over a network. KVServe handles this by choosing a compression configuration per request, based on the workload and the available bandwidth.

Finally, we saw how all these ideas come together in an inference engine, using Nano-vLLM as a concrete example. We described the core components that any engine needs — a front end, a scheduler, a block manager, a model runner, an attention backend, and a sampler, which together run in a loop of scheduling a batch, running the model, and sampling tokens.

In this chapter, we saw how the ideas of memory management, batching, scheduling, caching, and offloading come together to make LLM inference efficient. This also brings us to the end of this book. Over the chapters, we built up the stack of systems for machine learning, from the hardware and programming models that power it, to the techniques used to train and serve large models at scale. We hope this foundation helps you understand, build, and improve the systems that will run the next generation of ML models.

References

Agrawal, Amey, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee (2024). "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. URL: https://www.usenix.org/conference/osdi24/presentation/agrawal.

Agrawal, Amey, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee (2024). "Taming throughput-latency tradeoff in LLM inference with sarathi-serve". In: *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*.

Ainslie, Joshua, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai (2023). "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. URL: https://aclanthology.org/2023.emnlp-main.298/.

DeepSeek-AI et al. (2025). *DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models*. URL: https://arxiv.org/abs/2512.02556.

Gao, Bin, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo (2024). "Cost-efficient large language model serving for multi-turn conversations with CachedAttention". In: *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*.

GeeeekExplorer (2025). *Nano-vLLM: A lightweight vLLM implementation built from scratch*. URL: https://github.com/GeeeekExplorer/nano-vllm.

Jeong, Jinwoo and Jeongseob Ahn (2025). "Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. URL: https://doi.org/10.1145/3676641.3716245.

Jiang, Xuanlin, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu (2024). *NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference*. URL: https://arxiv.org/abs/2411.01142.

Jin, Chao, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin (2025). "RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation". In: URL: https://doi.org/10.1145/3768628.

Kamath, Aditya K., Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar (2025). "POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. URL: https://doi.org/10.1145/3676641.3715996.

Karpathy, Andrej (2023). *nanoGPT*. https://github.com/karpathy/nanogpt. GitHub repository.

Koyeb (2024). *Best LLM Inference Engines and Servers to Deploy LLMs in Production*. Koyeb. URL: https://www.koyeb.com/blog/best-llm-inference-engines-and-servers-to-deploy-llms-in-production.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica (2023). "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. New York, NY, USA. URL: https://doi.org/10.1145/3600006.3613165.

Lee, Wonbeom, Jungi Lee, Junghwan Seo, and Jaewoong Sim (2024). "InfiniGen: efficient generative inference of large language models with dynamic KV cache management". In: *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*.

Liu, Guangda, Wenhao Chen, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Quan Chen, Shixuan Sun, Jieru Zhao, and Minyi Guo (2026). "ECHO: Efficient KV Cache Offloading with Lossless Prefetching for Serving Native Sparse Attention LLMs". In: *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. URL: https://www.usenix.org/conference/osdi26/presentation/liu-guangda.

Liu, Zedong, Xinyang Ma, Dejun Luo, Hairui Zhao, Bing Lu, Wenjing Huang, Yida Gu, Xingchen Liu, Zheng Wei, Jinyang Liu, Dingwen Tao, and Guangming Tan (2026). "KVServe: Service-Aware KV Cache Compression for Communication-Efficient Disaggregated LLM Serving". In: *Proceedings of the ACM SIGCOMM 2026 Conference*. New York, NY, USA. URL: https://doi.org/10.1145/3789240.3829139.

Pan, Zaifeng, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding (2025). "FastTree: Optimizing Attention Kernel and Runtime for Tree-Structured LLM Inference". In: *Proceedings of Machine Learning and Systems*. MLSys. URL: https://proceedings.mlsys.org/paper_files/paper/2025/file/96894468eb44631a32d7ebd56f9892c7-Paper-Conference.pdf.

Rathi, Saksham, Preeti, and Mythili Vutukuru (2026). *Requests of a Feather Must Flock Together: Batch Size vs. Prefix Homogeneity in LLM Inference*. URL: https://arxiv.org/abs/2605.06046.

Shazeer, Noam (2019). *Fast Transformer Decoding: One Write-Head is All You Need*. URL: https://arxiv.org/abs/1911.02150.

Sheng, Ying, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang (2023). "FlexGen: high-throughput generative inference of large language models with a single GPU". In: *Proceedings of the 40th International Conference on Machine Learning*.

Tan, Kaizhen, Rong Gu, and Mingyuan Li (2026). *CacheWeaver: Cache-Aware Evidence Ordering for Efficient Grounded RAG Inference*. URL: https://arxiv.org/abs/2606.19667.

Tang, Jiaming, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han (2024). "QUEST: query-aware sparsity for efficient long-context LLM inference". In: *Proceedings of the 41st International Conference on Machine Learning*.

Xie, Zhiqiang, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis (2026). "Strata: Hierarchical Context Caching for Long Context Language Model Serving". In: *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. URL: https://www.usenix.org/conference/osdi26/presentation/xie-zhiqiang.

Yao, Jiayi, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang (2025). "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion". In: *Proceedings of the Twentieth European Conference on Computer Systems*. URL: https://doi.org/10.1145/3689031.3696098.

Yu, Gyeong-In, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun (2022). "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. URL: https://www.usenix.org/conference/osdi22/presentation/yu.

Yu, Lingfan, Jinkun Lin, and Jinyang Li (2025). "Stateful Large Language Model Serving with Pensieve". In: *Proceedings of the Twentieth European Conference on Computer Systems*. URL: https://doi.org/10.1145/3689031.3696086.

Zhang, Yanqi, Yuwei Hu, Runyuan Zhao, John C. S. Lui, and Haibo Chen (2025). "DiffKV: Differentiated Memory Management for Large Language Models with Parallel KV Compaction". In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. URL: https://doi.org/10.1145/3731569.3764810.

Zhang, Zhenyu, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen (2023). "H2O: heavy-hitter oracle for efficient generative inference of large language models". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*.

Zheng, Lianmin, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng (2024). "SGLang: efficient execution of structured language model programs". In: *Proceedings of the 38th International Conference on Neural Information Processing Systems*.

Zhong, Yinmin, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang (2024). "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association. URL: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.