LLM Inference

Systems for Machine Learning

Chapter 8

https://www.cse.iitb.ac.in/~mythili/sysml/

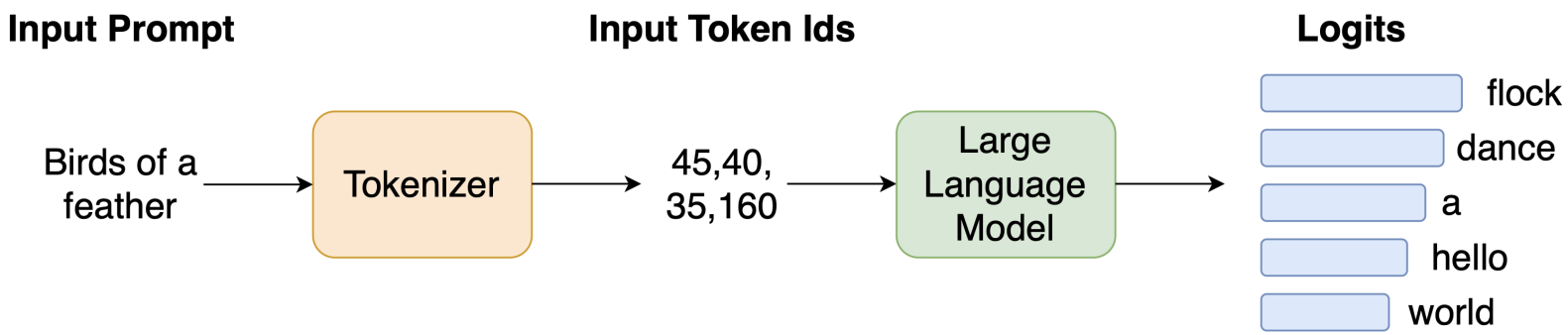
What is Inference?

What is Inference?

- Uses a **trained** model to produce output for a **new input**
- Executes the forward pass of the model for the input
- Generates one token at a time

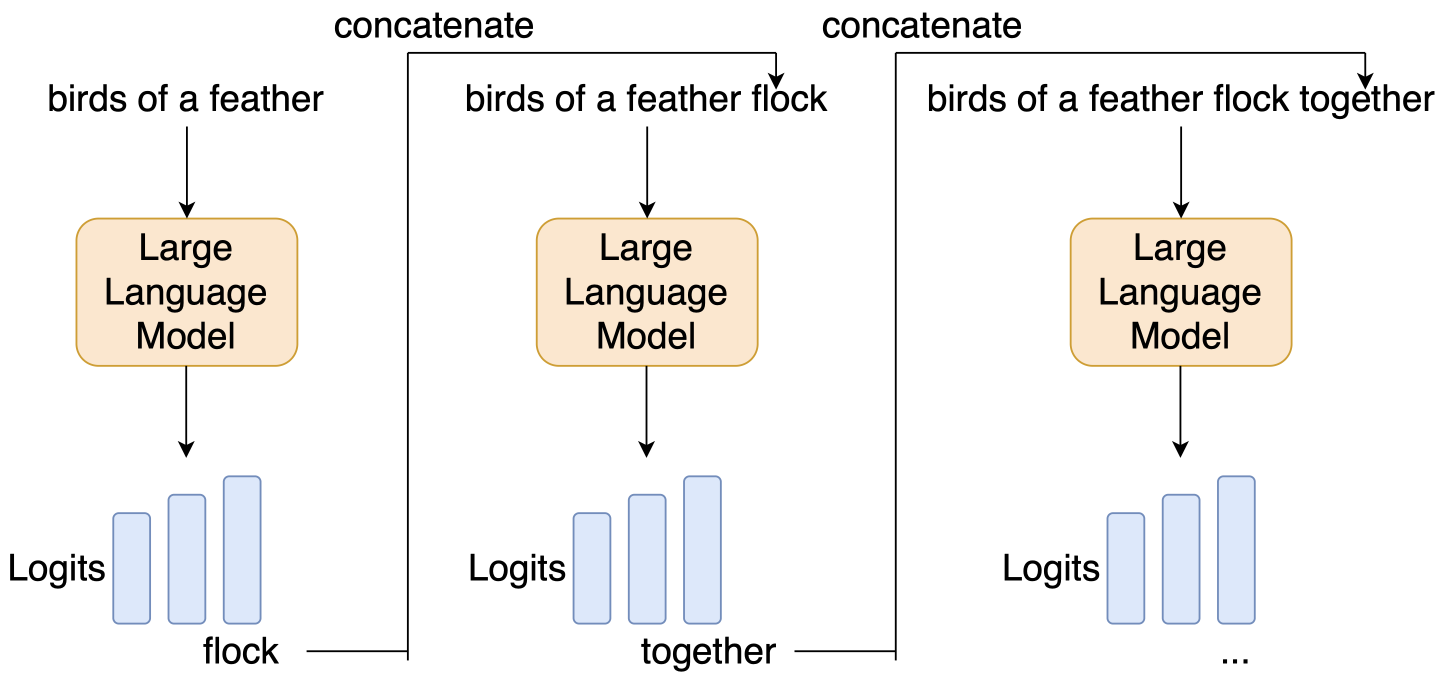
Output Generation

- Input text broken into tokens by a **tokenizer**
- Tokens mapped to **embedding** vectors
- **Unembedding** step after the final layer
- Next token chosen from the distribution, appended to the input, and process **repeats**



Output Generation

- Several **decoding strategies** – greedy, beam search, temperature sampling, top-k samping, and top-p sampling

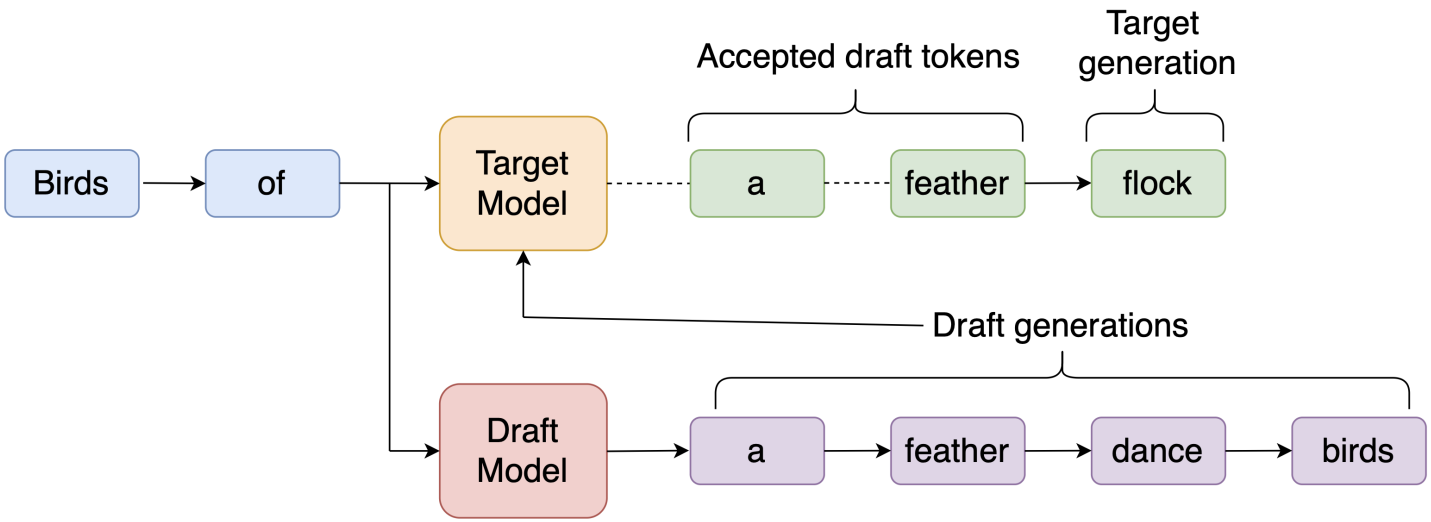


Speculative Decoding

- Uses a small, **draft model** alongside the target model
- Draft model proposes several consecutive tokens quickly
- Target model checks proposed tokens in **a single forward pass**, in parallel
- Target model compares the probability assigned to each proposed token
  - Token accepted if $P(\text{Target}) \geq P(\text{Draft})$
  - Otherwise accepted with probability $P(\text{Target})/P(\text{Draft})$

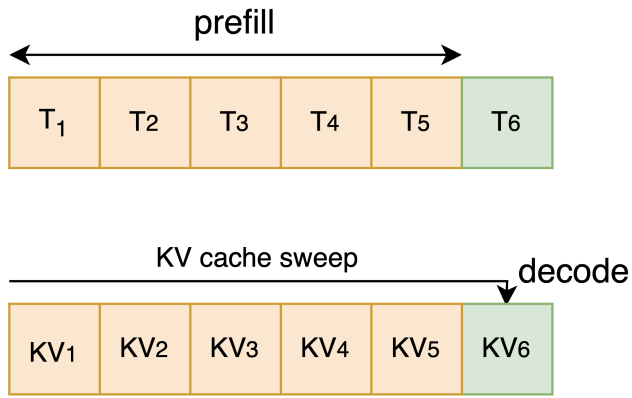
Speculative Decoding

- If **rejected**, new token resampled from target model's distribution output
- As soon as one proposed token is rejected, every token after it, is **discarded**



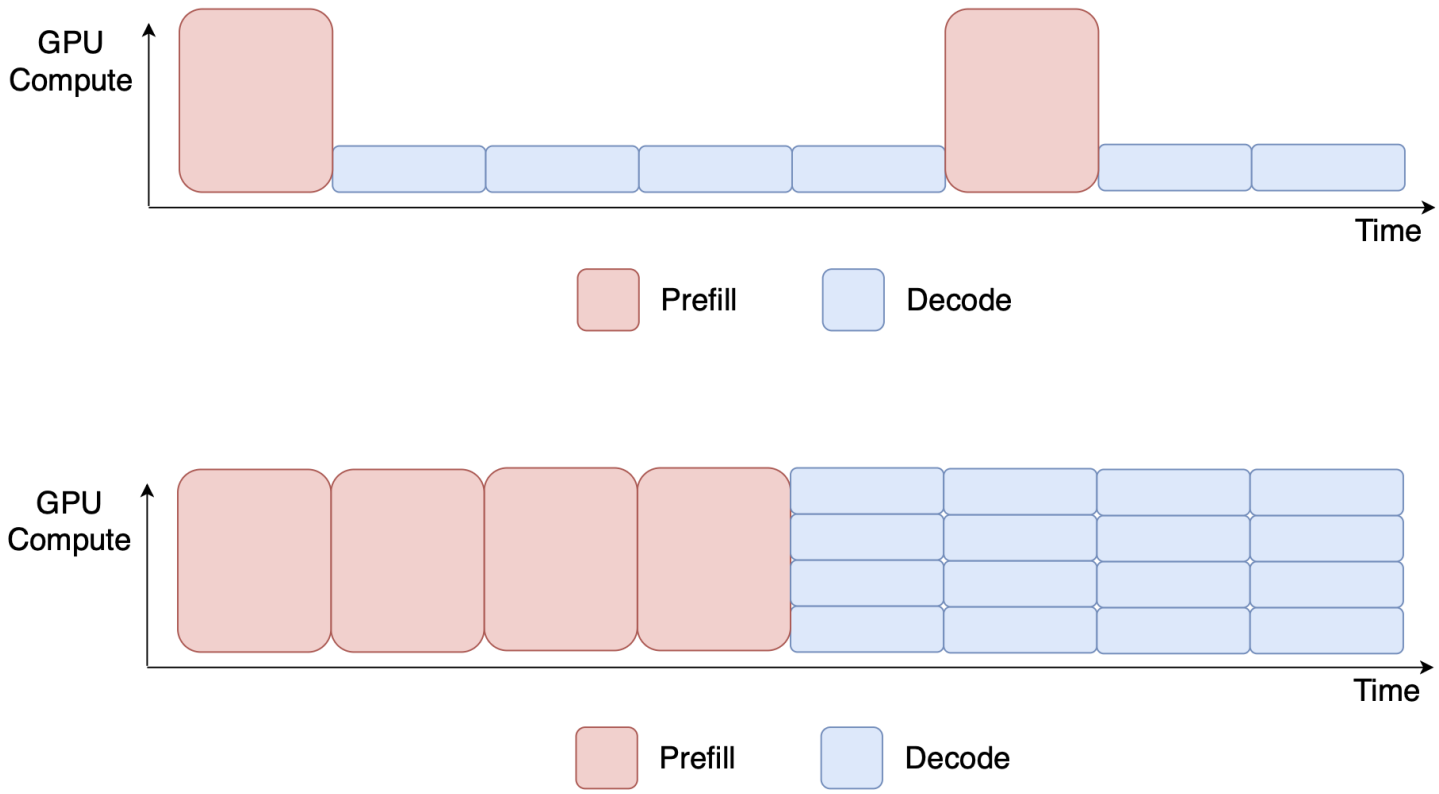
Prefill, Decode, and KV Caching

- Generation splitted into two distinct phases
- **Prefill** – entire prompt in a single forward pass
- **Decode** – output tokens generated one at a time
  - Computes attention scores by reading keys and values from KV cache
- Prefill has higher arithmetic intensity than decode



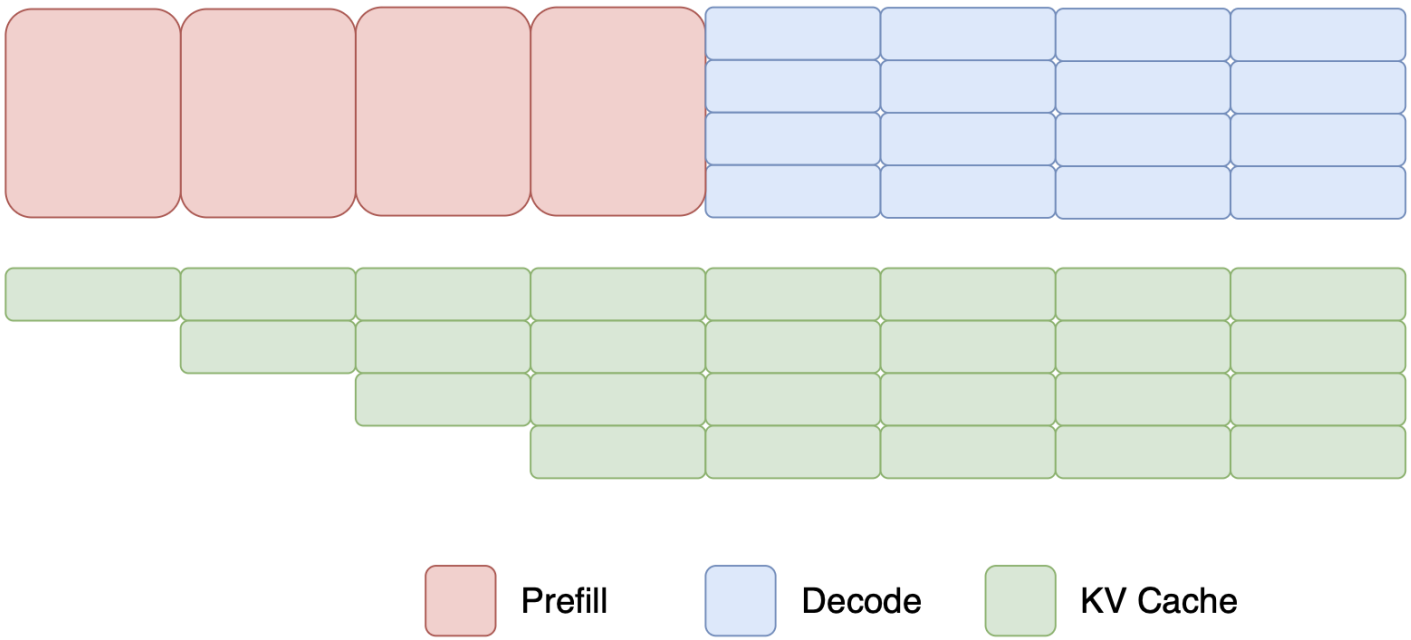
Batching

- **Batching** multiple requests together ensures many decode steps run in parallel



Batching and KV Caching

- KV caches of all the requests of a batch need to reside in GPU memory
- Places a hard limit on the **max batch size** a system can support during decode



Performance Metrics

- Time To First Token (**TTFT**) – time taken to complete the prefill phase and generate the first token
- Time Per Output Token (**TPOT**) – time of decode step
  - Also called Time Between Tokens (**TBT**)
- Theoretical min decode step time is given by

$$\frac{\text{Batch Size} \times \text{KV Cache Size} + \text{Parameter Size}}{\text{Total Memory Bandwidth}}$$

- **Throughput** – tokens processed/generated per second

Performance Metrics

- Prefill throughput limited by GPU's peak **compute**
- Decode throughput limited by **memory bandwidth**
- Batching more beneficial for decode than for prefill

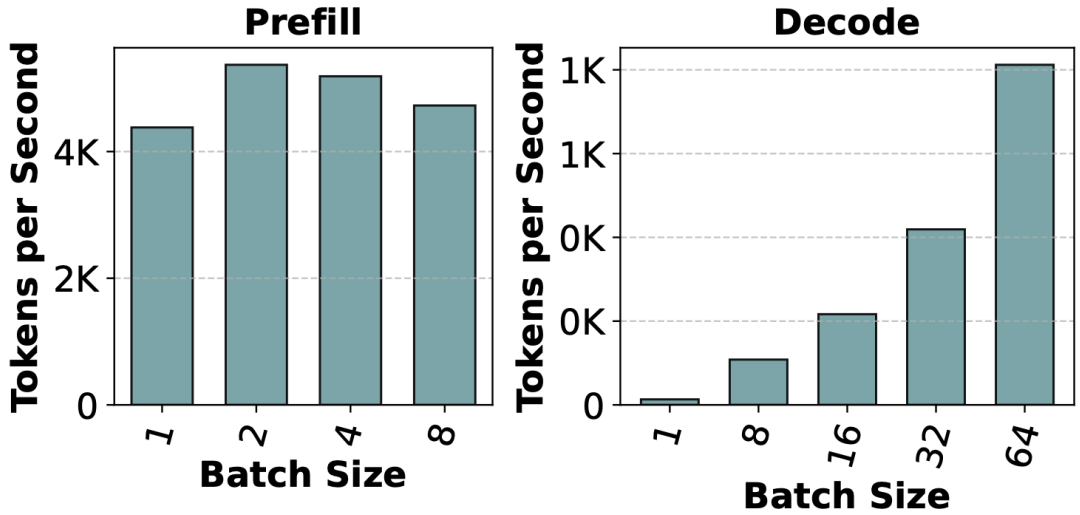


Image Credit: https://dl.acm.org/doi/10.5555/3691938.3691945

Memory Management

Traditional Memory Management for KV Caches

- GPU memory consumed by three things: **model parameters**, **KV cache**, and **activation memory**
- Goal is to pack as many requests' KV cache as possible to maximize batch size

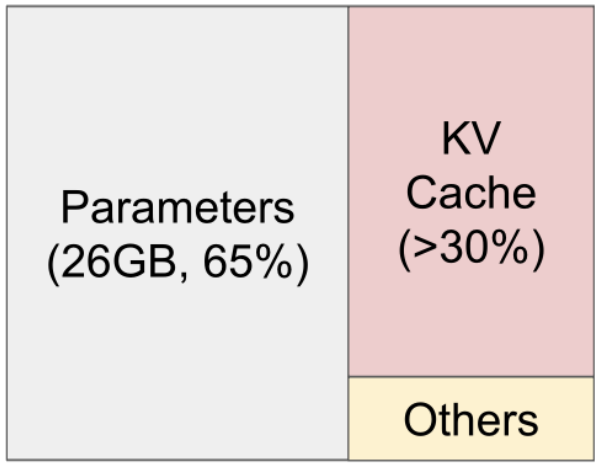


Image Credit: https://doi.org/10.1145/3600006.3613165

Traditional Memory Management for KV Caches

- Allocate one **contiguous** block of memory, large enough to hold the maximum possible sequence length any request could ever reach
- **Internal fragmentation** – reserves space for tokens never generated
- **External fragmentation** – many small gaps between allocated regions

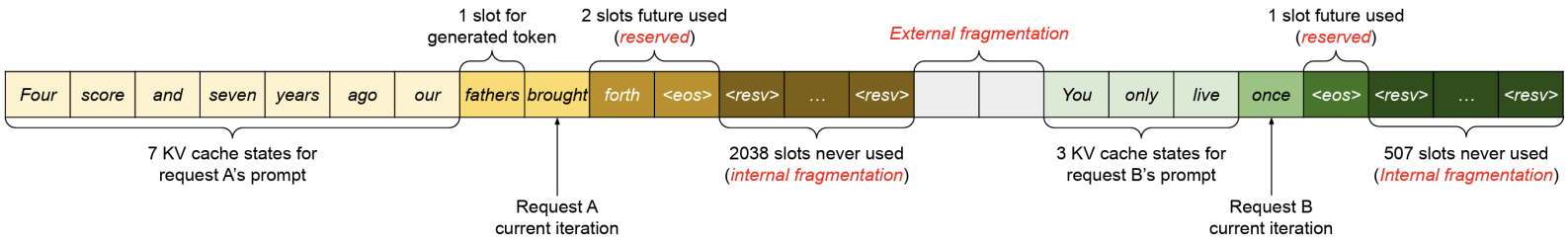


Image Credit: https://doi.org/10.1145/3600006.3613165

PagedAttention

- Divides KV cache into fixed-size **blocks**, each holding KV vectors for a small, fixed number of tokens
- Blocks are allocated on demand, hence no internal fragmentation
- Fixed sized blocks and non-contiguous allocation, hence no external fragmentation

PagedAttention

- Key block $K_j = (k_{(j-1)B+1}, \ldots, k_{jB})$ and value block $V_j = (v_{(j-1)B+1}, \ldots, v_{jB})$

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

where $A_{ij} = (a_{i,(j-1)B+1}, \ldots, a_{i,jB})$ is the row vector of attention scores on the i-th query and j-th KV block

PagedAttention

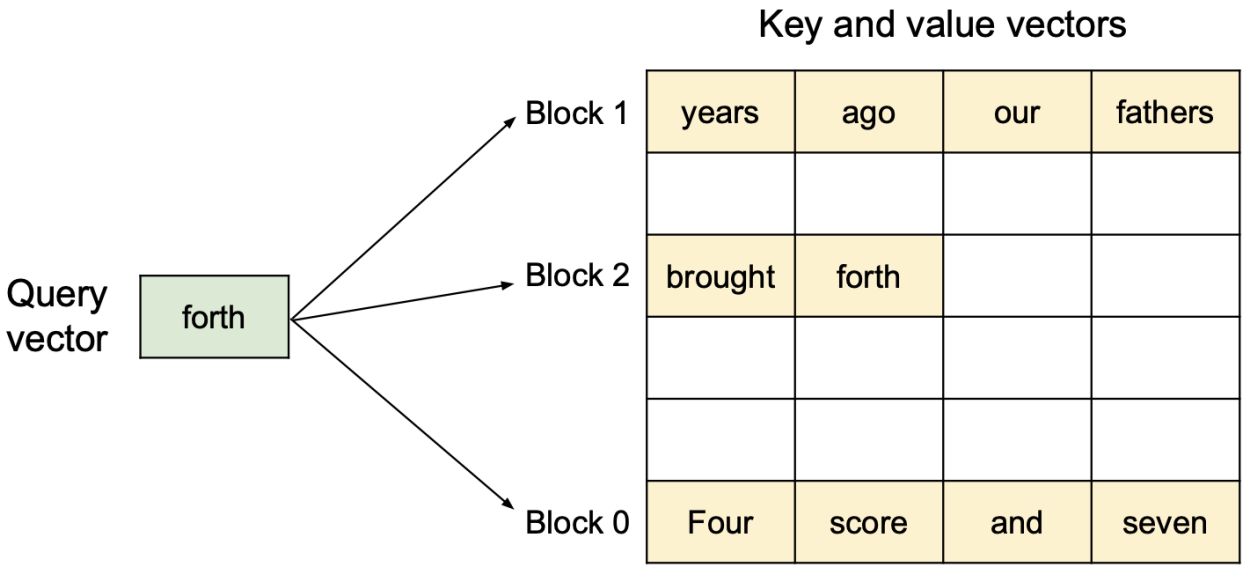


Image Credit: https://doi.org/10.1145/3600006.3613165

PagedAttention

- Each request has its own sequence of **logical** blocks
- **Block table** maps each of these logical block numbers to the physical block in GPU memory

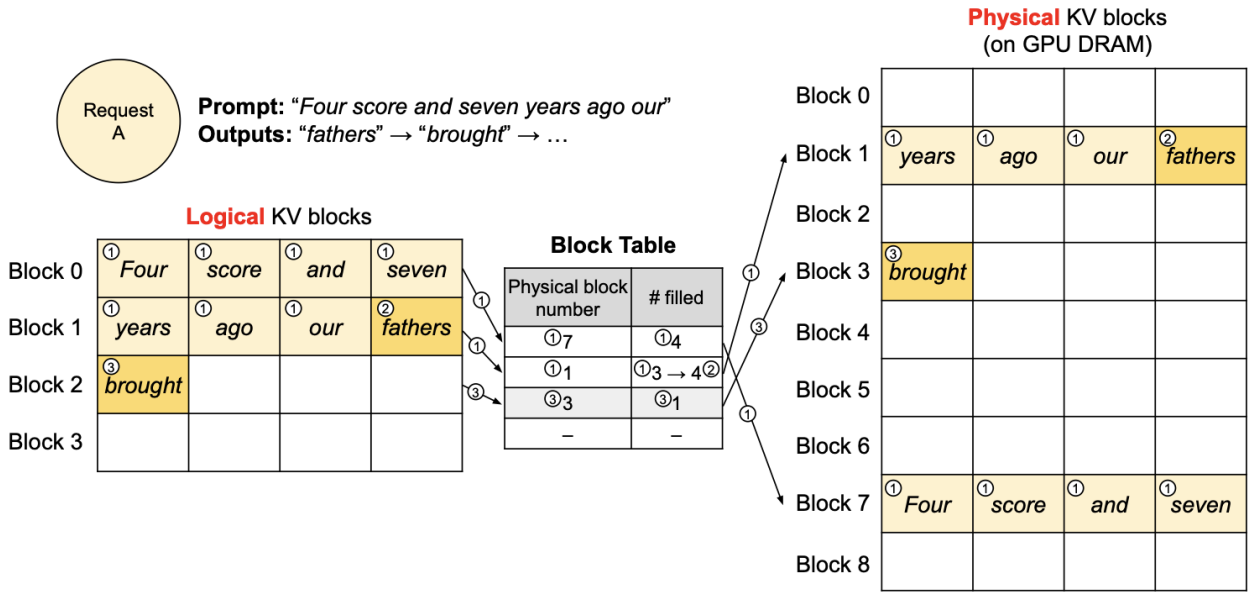


Image Credit: https://doi.org/10.1145/3600006.3613165

PagedAttention

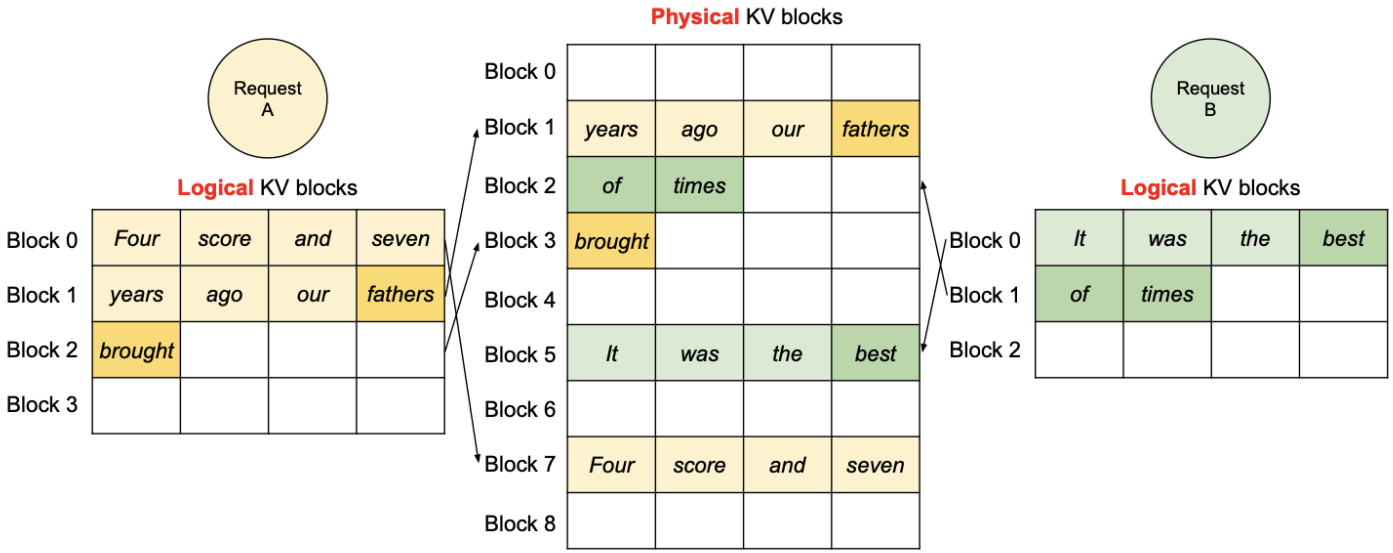


Image Credit: https://doi.org/10.1145/3600006.3613165

PagedAttention

- **KV cache manager** delegates the bookkeeping of free and used physical blocks to separate allocators for GPU memory and for CPU memory

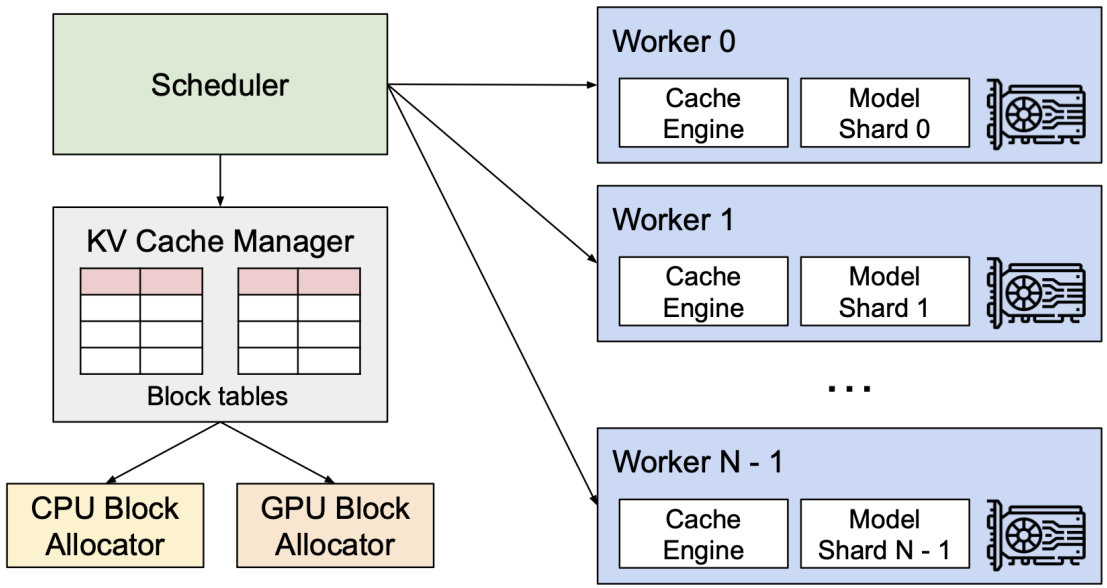


Image Credit: https://doi.org/10.1145/3600006.3613165

PagedAttention

- When the GPU runs out of blocks:
  - Pause the request needing a new block
  - Or **swap** out existing blocks to CPU memory
- Choosing the block size involves a trade-off
  - **Larger blocks** improve the efficiency of the block-wise attention kernel, but increase internal fragmentation
  - **Smaller blocks** reduce the effectiveness of memory coalescing inside the attention kernel

Sharing Memory Across Sequences

- When multiple sequences being processed **share** common tokens, the logical KV cache blocks can point at the same physical block
- To share KV cache blocks, requests need to share **prefixes** and not arbitrary chunks in middle
  - Because key and value vectors computed for a token depend not just on that token and its position, but on its entire preceding context

Parallel Sampling

- Single prompt used to generate several independent completions
- **Copy-on-write** strategy followed

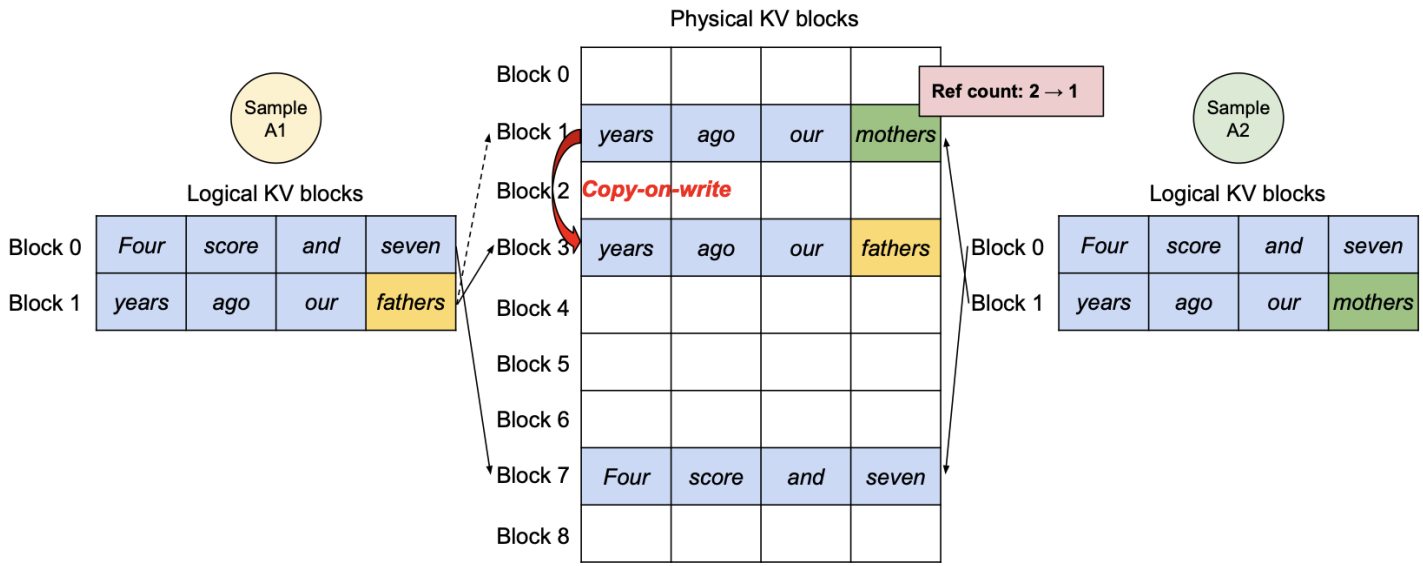


Image Credit: https://doi.org/10.1145/3600006.3613165

Beam Search

- **Beam search** also benefits from the same mechanism

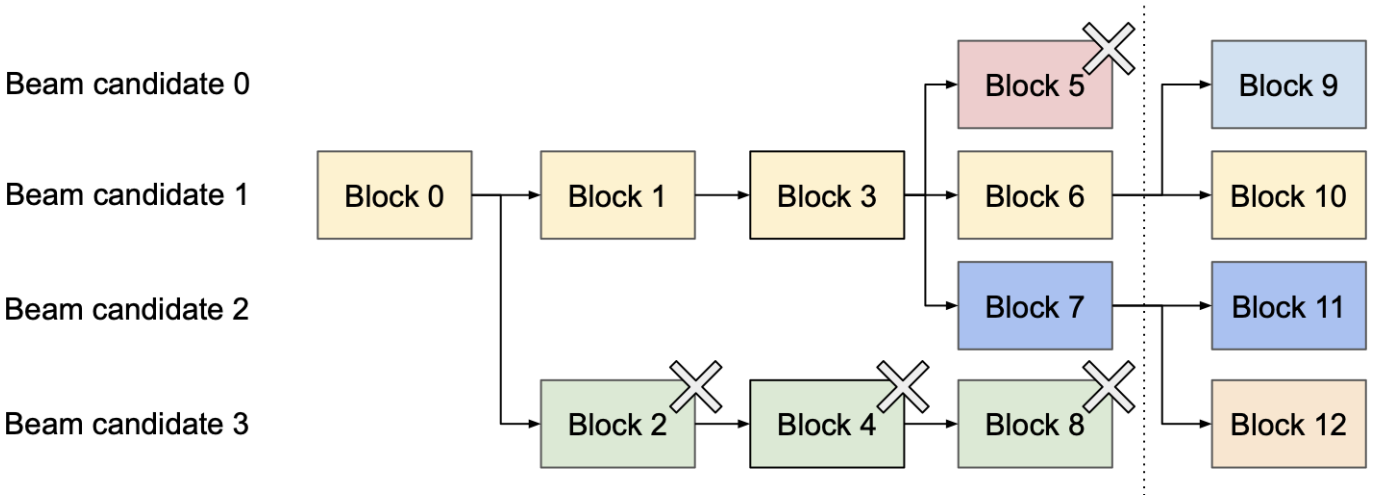


Image Credit: https://doi.org/10.1145/3600006.3613165

Prefix Sharing Across Different Requests

- Common when requests are built from the same template

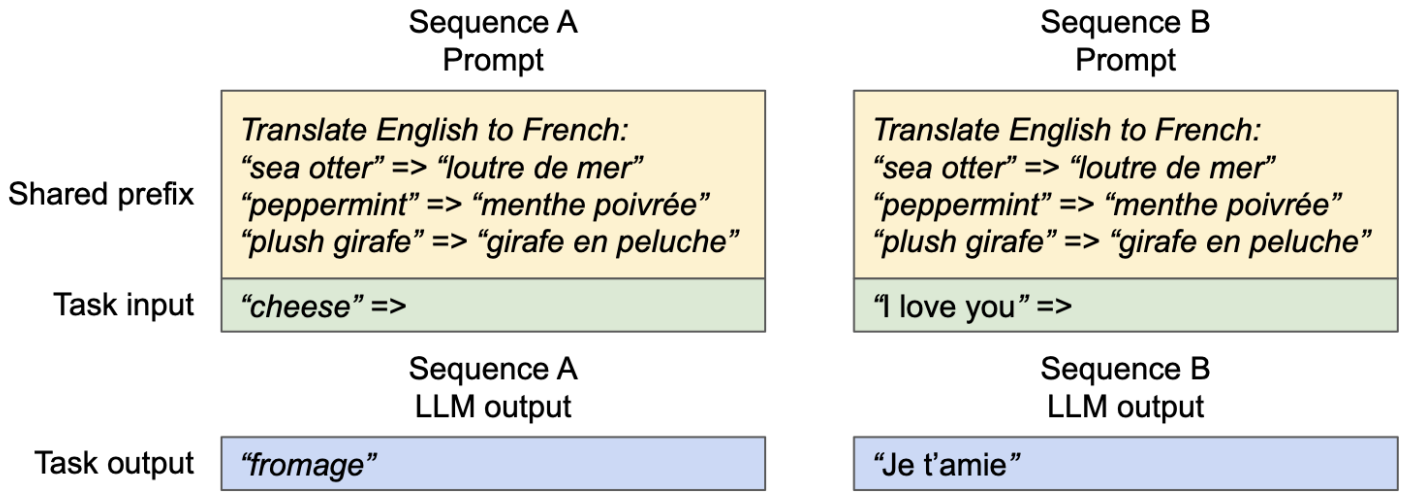
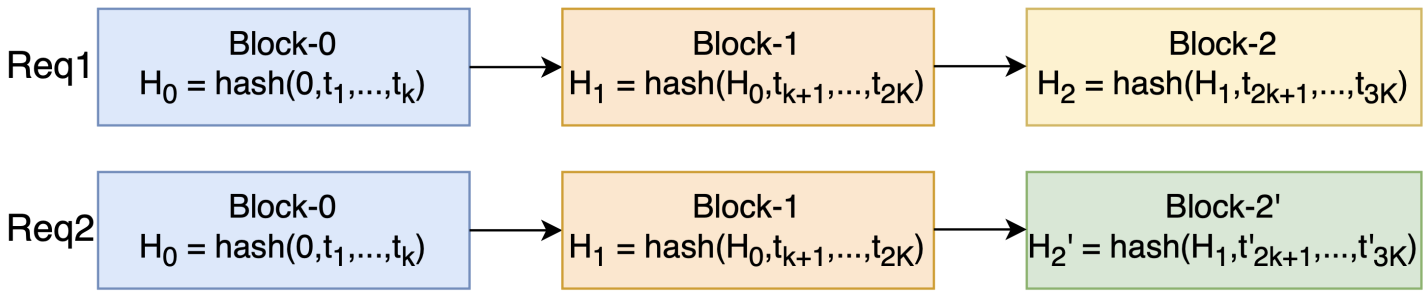


Image Credit: https://doi.org/10.1145/3600006.3613165

Detecting Sharing

- Compute the **hash** of each block's contents
- Incorporate the hash of the **preceding** block also
- When a new request arrives, the manager checks whether a block with a matching hash already exists in the cache



Batching and Scheduling

Request-Level Scheduling

- Batch is considered complete only once every request in it has finished generating
- Different requests in a batch finish at **different times**
- Slot vacated by the finished request is left idle or padded
- New requests must wait for the entire batch to drain
- Sub-optimal GPU utilization due to wait for **longest** request to complete

Iteration-Level Scheduling

- Orca uses **iteration-level scheduling**
- Make a fresh decision after every iteration
- **Continuous batching** – a request can leave running batch upon EOS token, new request can join anytime

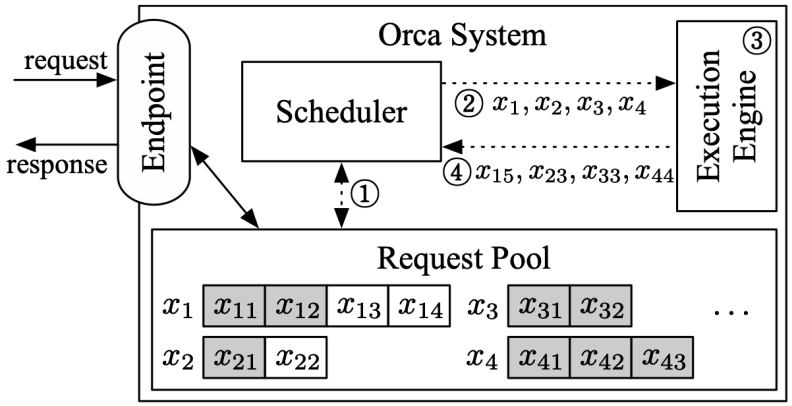


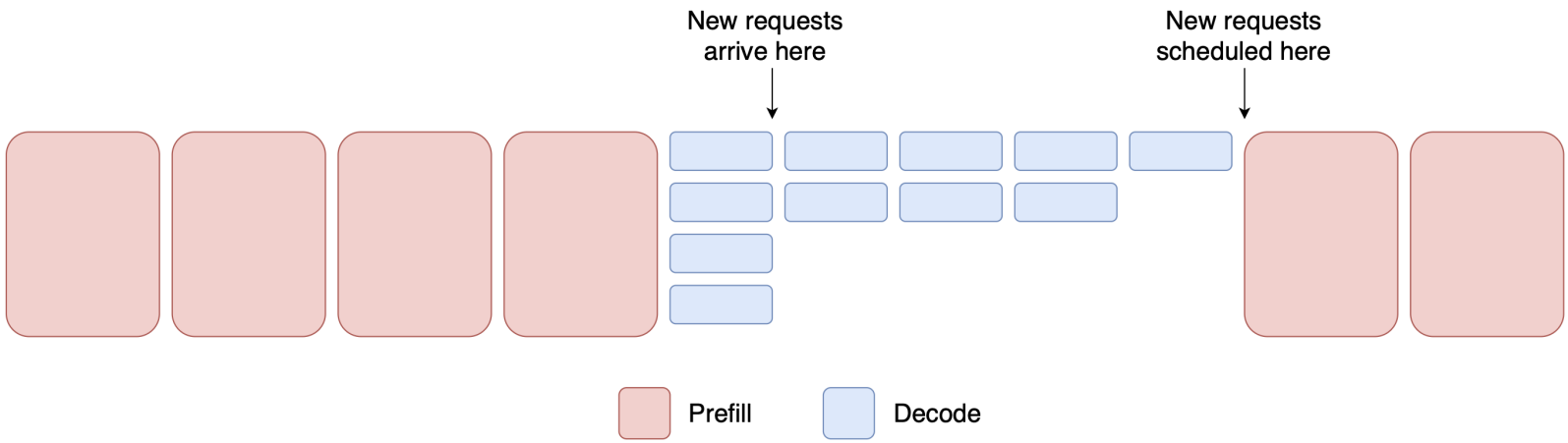
Image Credit: `https://www.usenix.org/conference/osdi22/presentation/yu`

Iteration-Level Scheduling

- **Selective batching** – linear operations batched, attention operation per request
- Requests in prefill and decode phases can have different lengths, padding may be required
- vLLM uses first-come-first-served (**FCFS**) scheduling
- Two ways of preempting requests to free up space:
  - Evict the request's KV cache to CPU memory
  - Evict completely and recompute later

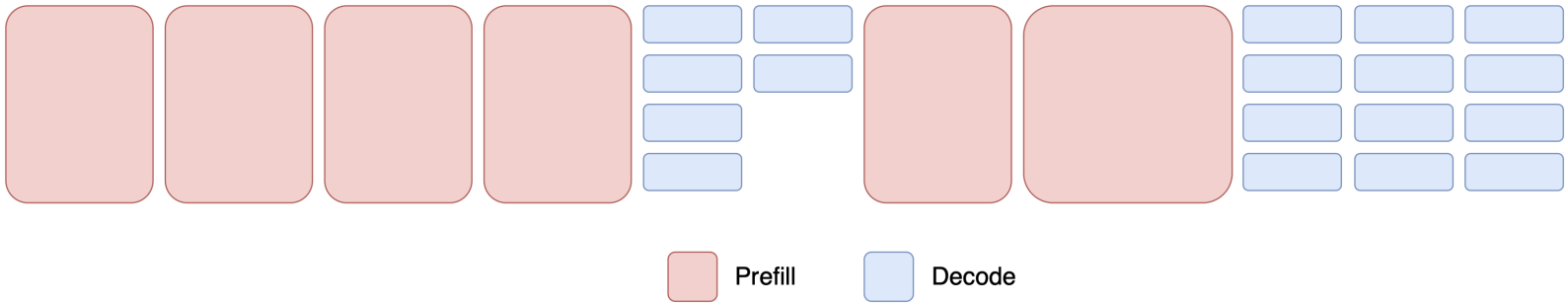
Decode Prioritizing Schedule

- A request that arrives while a batch is already decoding is not admitted immediately
- Once a request enters its decode phase, nothing else is allowed to interrupt it, so **TBT** stays predictable and low
- As requests leave, batch continues running with fewer requests



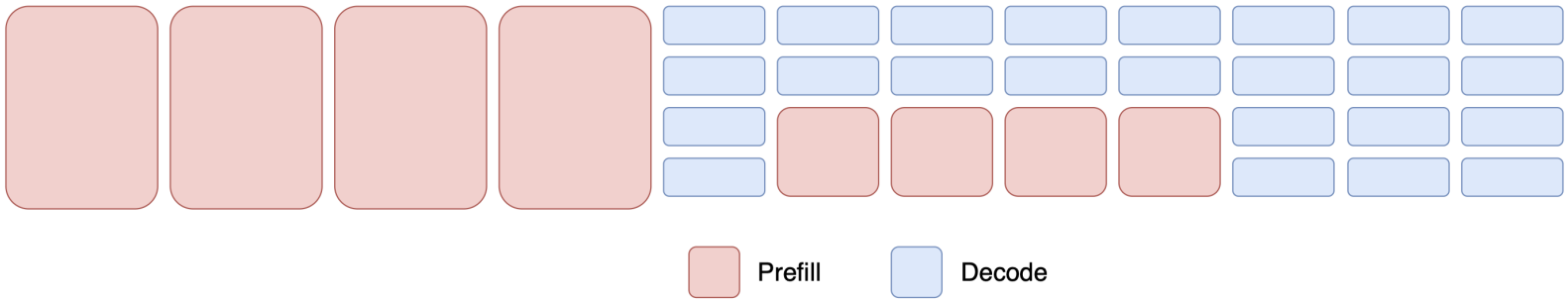
Prefill Prioritizing Schedule

- As soon as a new request arrives, prefill is scheduled
- When no prefills left, scheduler falls back to batching decodes
- Produces larger decode batches, so good choice for **throughput**, and optimizes **TTFT**
- Can produce generation stalls



Chunked Prefills in Sarathi-Serve

- Splits prefill into sequence of roughly equal-sized **chunks**
- Builds a **hybrid** batch on every iteration
- First fill with decodes, and then with chunked prefills
- Relies on a **token budget** – counting both decode and prefill chunk tokens



Chunked Prefill and Pipeline Parallelism

- **Pipeline parallelism** works best when every stage takes roughly the same amount of time to process a microbatch
- With Sarathi Serve's token budget, every iteration is built to consume close to the same number of tokens

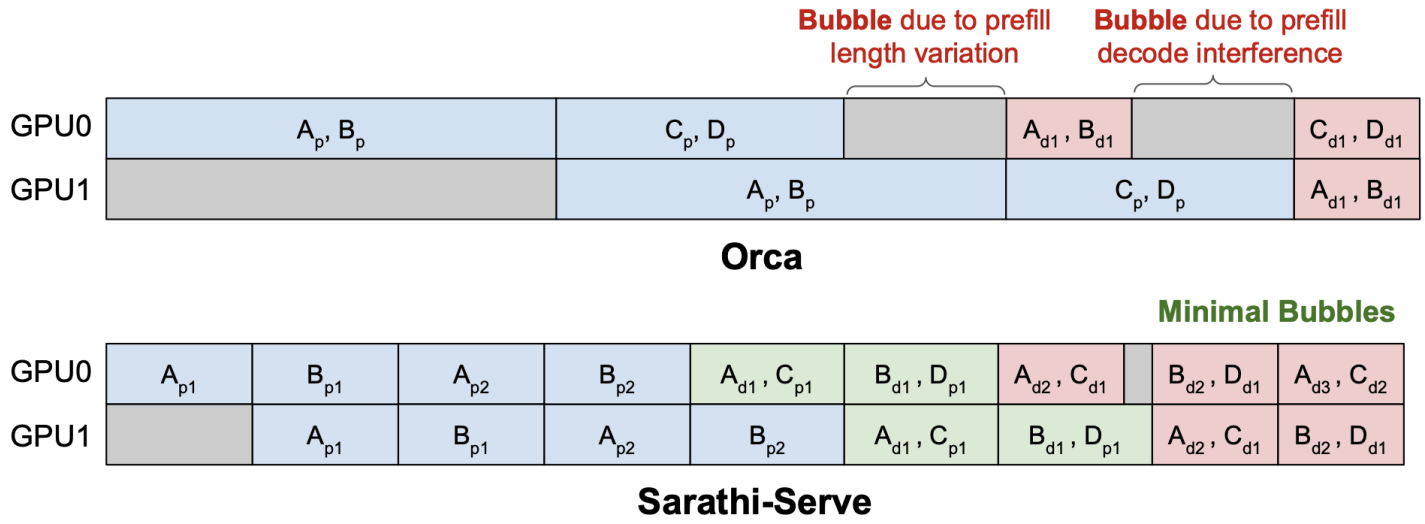


Image Credit: https://dl.acm.org/doi/10.5555/3691938.3691945

Compute-Memory Balance

- Chunk that is too small leaves the batch memory-bound
- Chunk that is too large pushes the batch toward being compute-bound, and introduces stalls

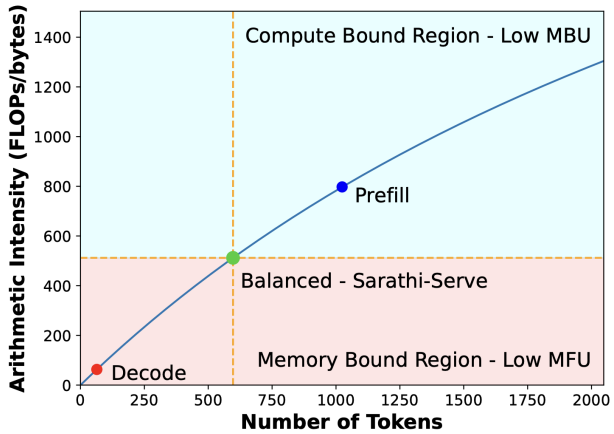


Image Credit: https://dl.acm.org/doi/10.5555/3691938.3691945

Issues with Hybrid Batching at the Attention Layer

- Hybrid batching effective with **linear operations**, because they are independent and can be merged
- For attention, prefill and decode are separate kernel calls
- Does not always lead to a balanced utilization of GPU compute and memory bandwidth

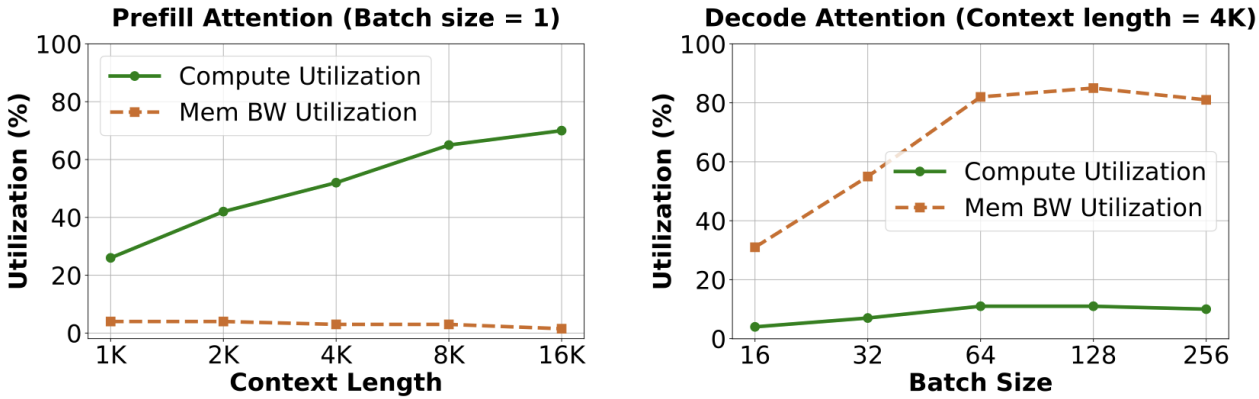


Image Credit: https://dl.acm.org/doi/10.5555/3691938.3691945

Issues with Chunking

- Query tokens of i-th chunk must read all $K_{<i}$ and $V_{<i}$ from slow HBM
- Total bytes read grows **quadratically** with the number of chunks
- Shrinking the token budget reduces TBT but increases the number of chunks a long prompt is split into

Fusing Prefill and Decode Attention in POD-Attention

- Prefill and decode kernels not guaranteed to run on the **same SM**, no complementary resource usage
- POD-Attention **fuses** prefill and decode CTAs into a single kernel call
- SMs maintain **counters** on how many prefill and decode CTAs assigned

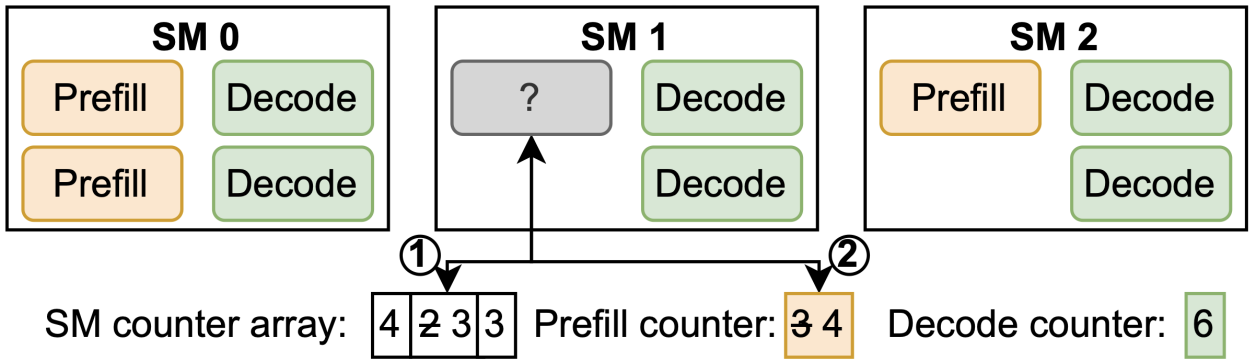


Image Credit: https://doi.org/10.1145/3676641.3715996

Disaggregating Prefill and Decode

- **DistServe** runs prefill and decode on different GPUs
- Helps to fulfill both TTFT and TPOT SLOs
- Worthwhile only if moving KV cache is cheap enough

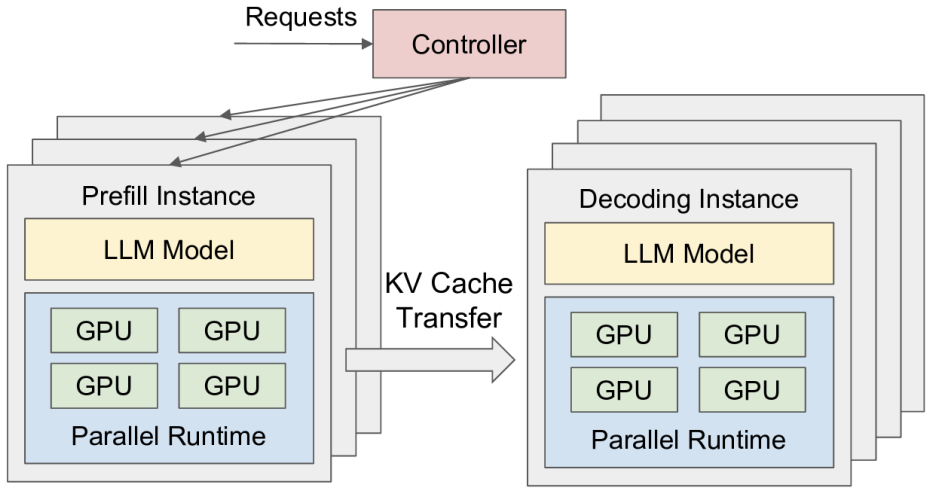


Image Credit: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmi

Prefix Sharing

Prefix Sharing

- Occurs in few-shot and multi-turn conversation settings
- **SGLang** uses **RadixAttention** to avoid recomputation
- **Radix tree** for finding cached prefix
- Every node maintains a **ref count** of how many requests are using it, evicted when count drops to 0
- Eviction starts from leaves

Radix Tree

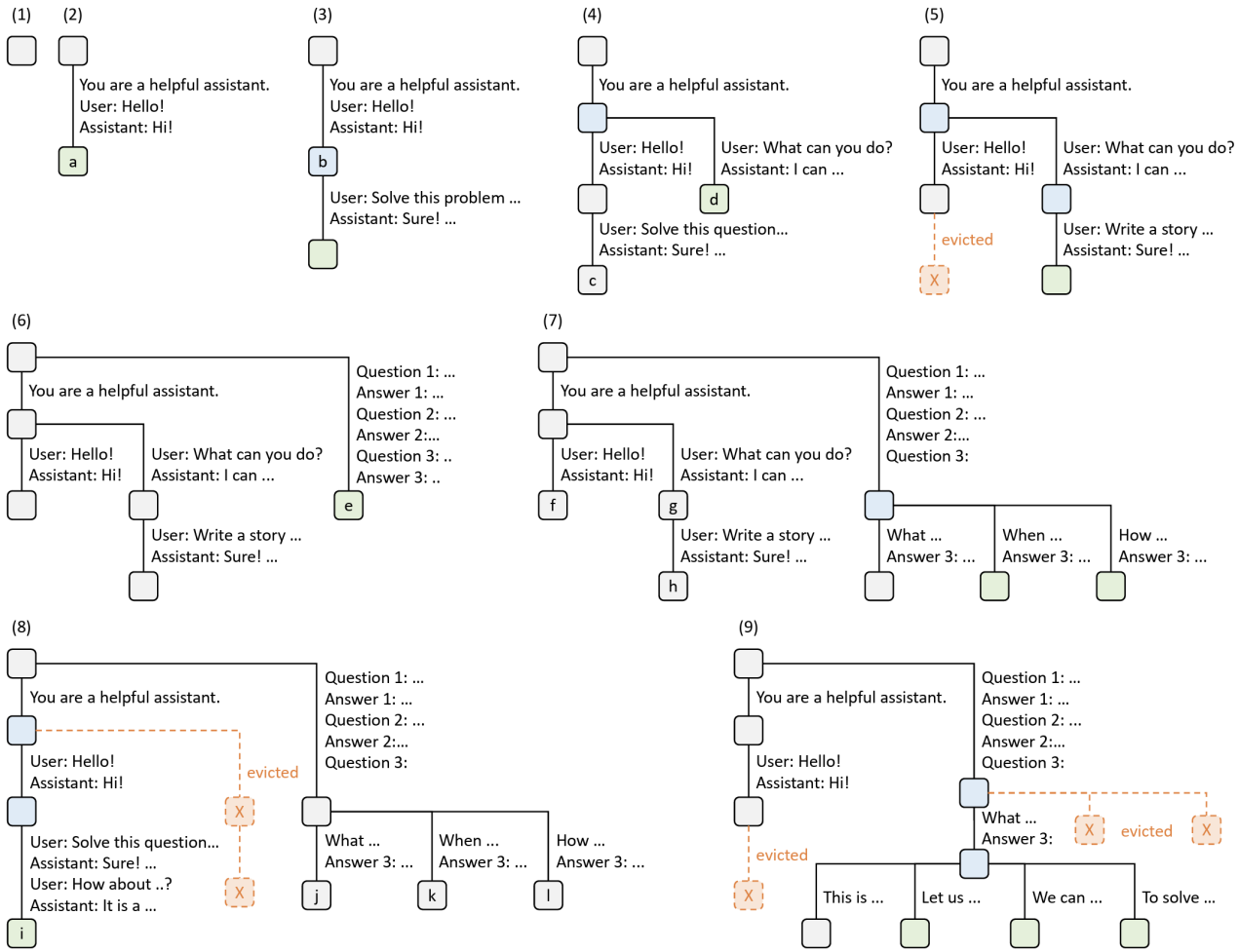


Image Credit: https://dl.acm.org/doi/10.5555/3737916.3739916

Patterns of KV Cache Sharing

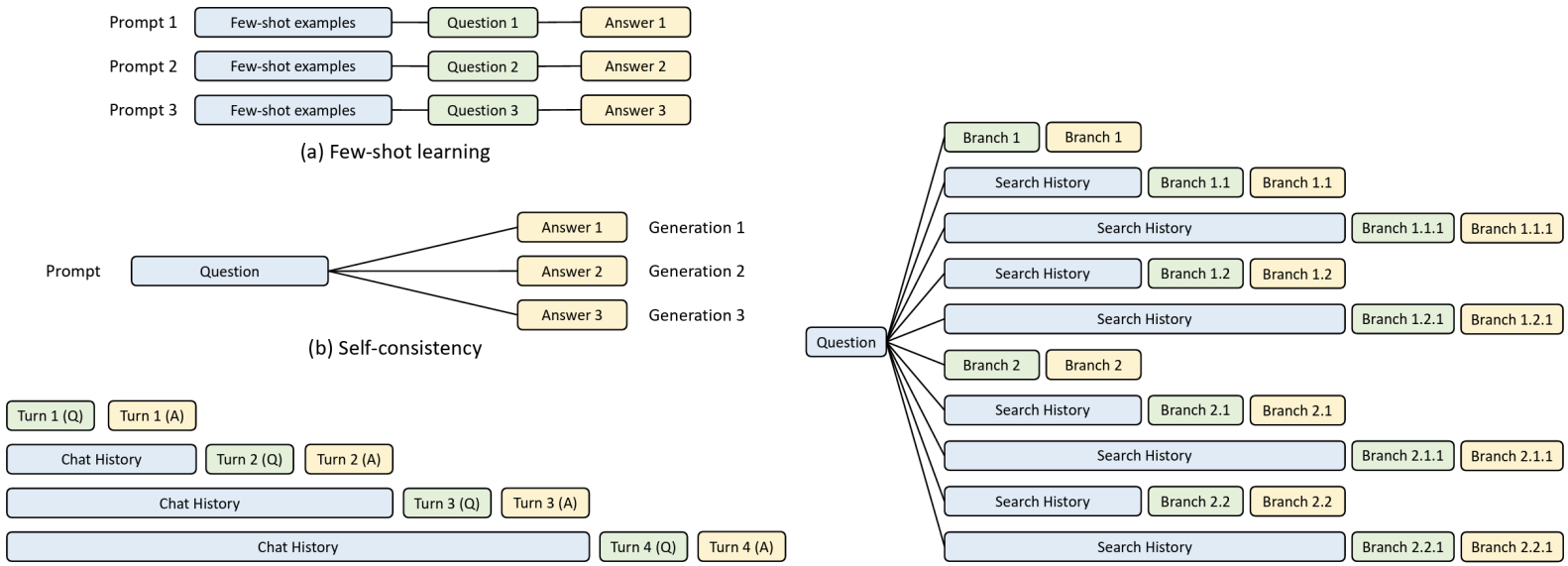
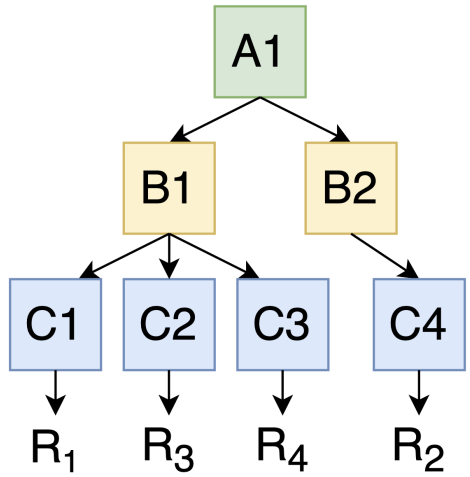


Image Credit: https://dl.acm.org/doi/10.5555/3737916.3739916

Cache-Aware Scheduling

- Walk radix tree in **depth-first order**
- **FCFS order** – R_1, R_2, R_3, R_4
- **DFS order** – R_1, R_3, R_4, R_2
- Avoids evicting $B1$, which is a shared prefix of R_1, R_3, R_4



Tree-Structured Attention Kernels

- Queries sharing context still load KV blocks redundantly into GPU cache
- During decoding, in $QK^\top$, Q is vector, **matrix-vector multiplication** cannot run efficiently on tensor cores

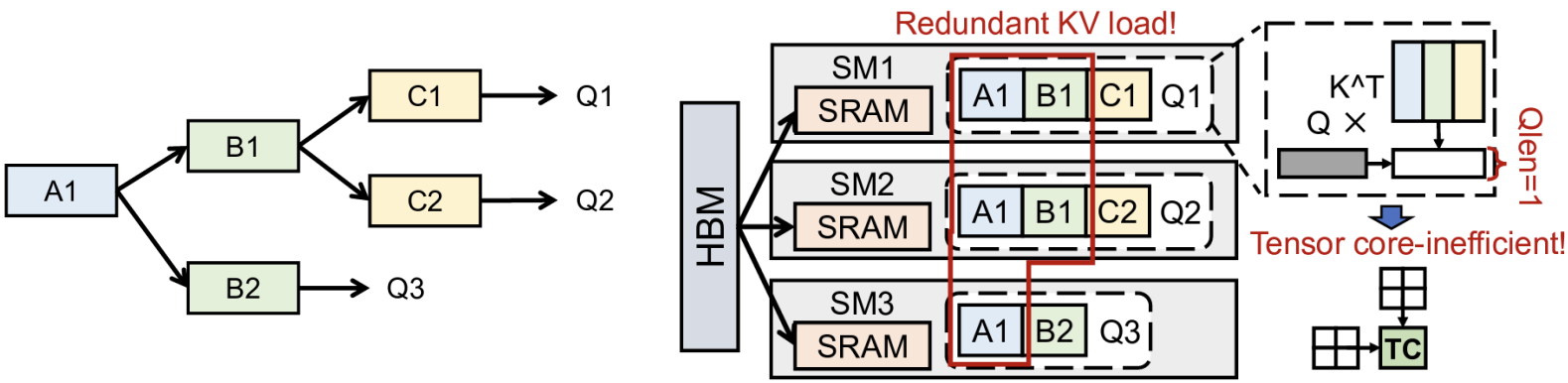


Image Credit: https://proceedings.mlsys.org/paper_files/paper/2025/file/96894468eb44631a32d7ebd56f9892c7-Paper-Conference.pdf

FastTree

- **FastTree** groups requests with shared context on a single kernel invocation

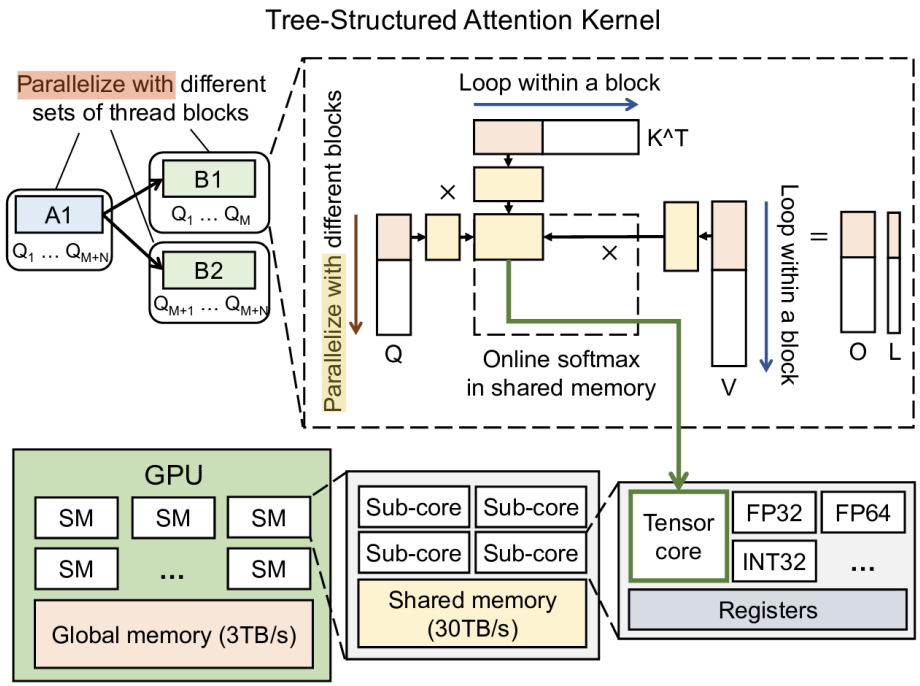
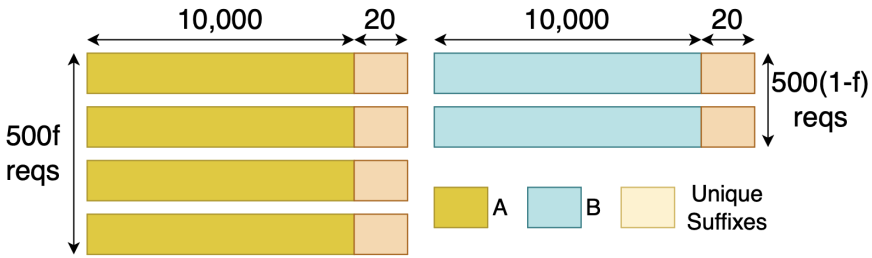


Image Credit: https://proceedings.mlsys.org/paper_files/paper/2025/file/96894468eb44631a32d7ebd56f9892c7-Paper-Conference.pdf

Feather

- **Prefix homogeneity** – extent to which all requests of a batch share a common prefix
- Helps in spatial and temporal **locality**, as requests sweep the same KV cache region
- Tradeoff between batch size and prefix homogeneity – larger batches utilize GPU better but lesser locality gain
- FEATHER uses RL for learning optimal batch size



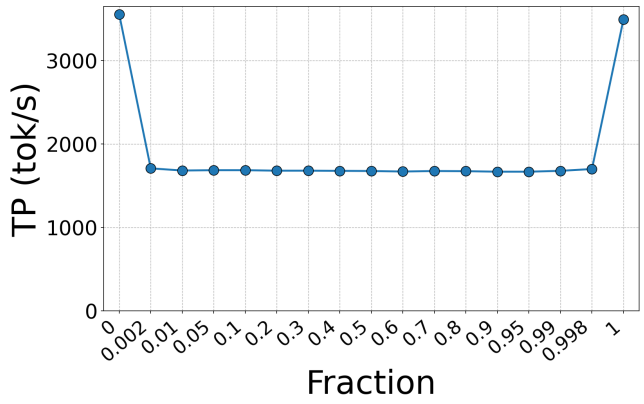


Image Credit: https://arxiv.org/abs/2605.06046

Retrieval-Augmented Generation

- Keeps a large external knowledge base, retrieves snippets, adds before the LLM query

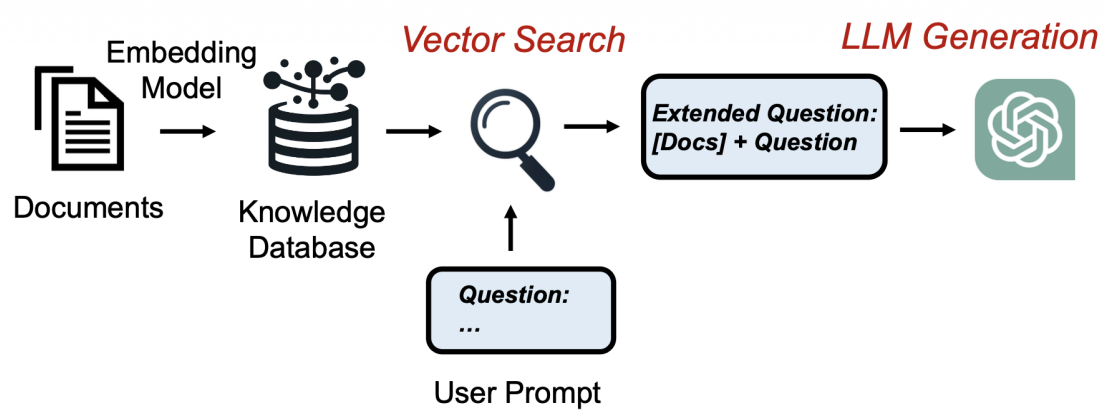


Image Credit: https://doi.org/10.1145/3768628

Why RAG Breaks Prefix Sharing?

- Snippets can come in **any order**
- Prefix need not be shared any longer

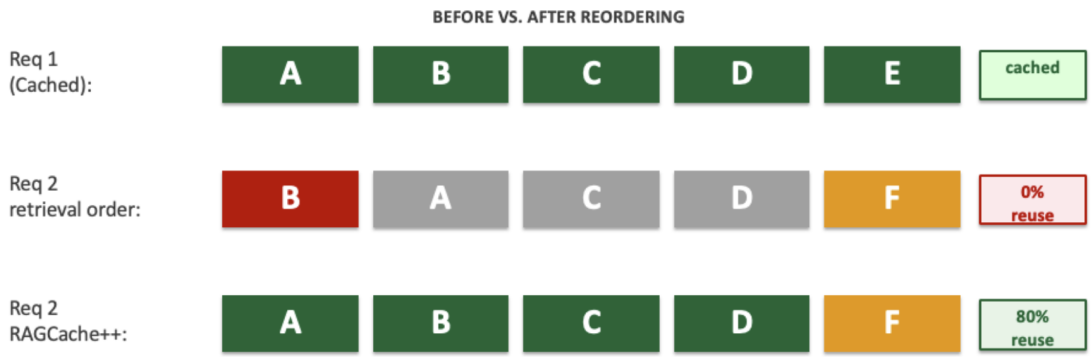


Image Credit: `https://arxiv.org/abs/2606.19667`

Caching Knowledge as a Tree of Documents

- **RAGCache** stores the KV tensors of documents in a knowledge tree
- Nodes that are used often stay in GPU memory, others in host memory

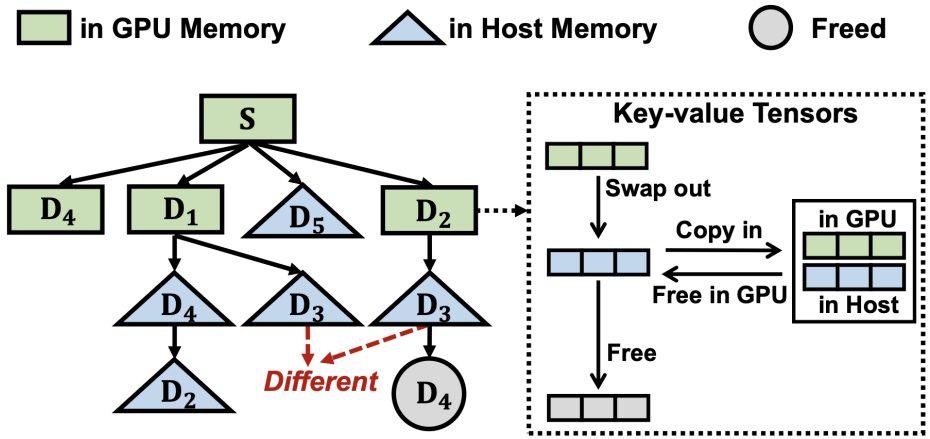


Image Credit: https://doi.org/10.1145/3768628

Reordering to Create Prefix Matches

- Requests can be reordered to favour ones with a large cached part, reduces evictions
- Documents can be reordered too to match a larger prefix
  - **CacheWeaver** uses a greedy walk for this

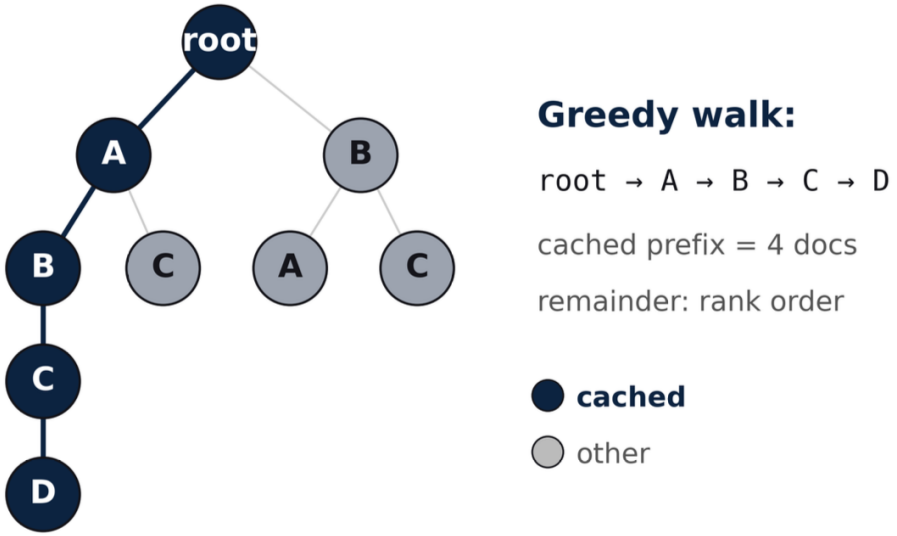


Image Credit: https://arxiv.org/abs/2606.19667

Reusing Documents That Cannot Form a Prefix

- Simply concatenating individual KVs does not work, because it **lacks cross-attention**

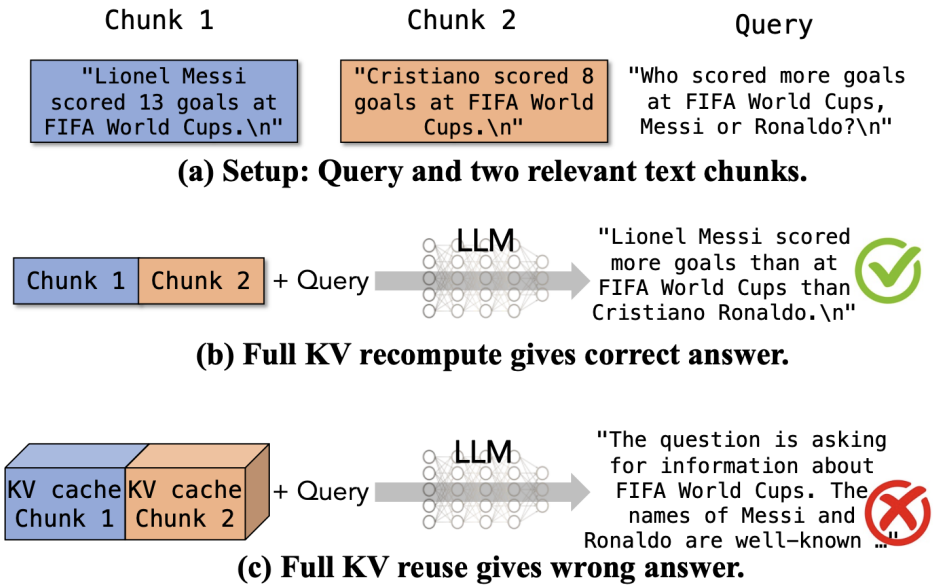


Image Credit: https://doi.org/10.1145/3689031.3696098

Reusing Documents That Cannot Form a Prefix

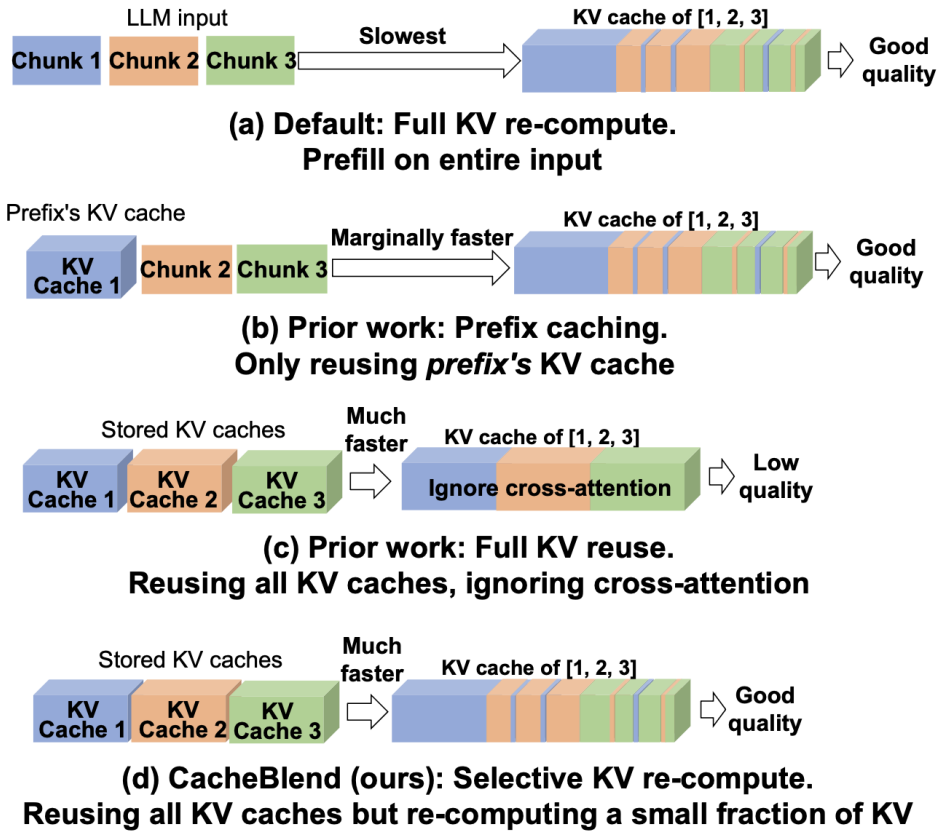


Image Credit: https://doi.org/10.1145/3689031.3696098

Selective Recomputation

- **CacheBlend** recomputes the KV values of a small subset of tokens that are most affected
- Identifies tokens with highest KV deviation (**HKVD**)
- HKVD tokens of one layer tend to have high deviation in next layer too

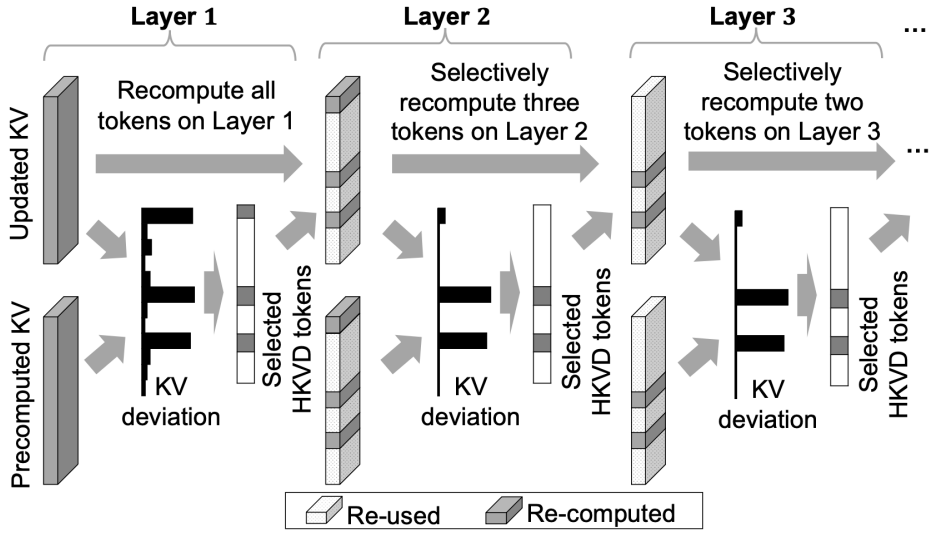
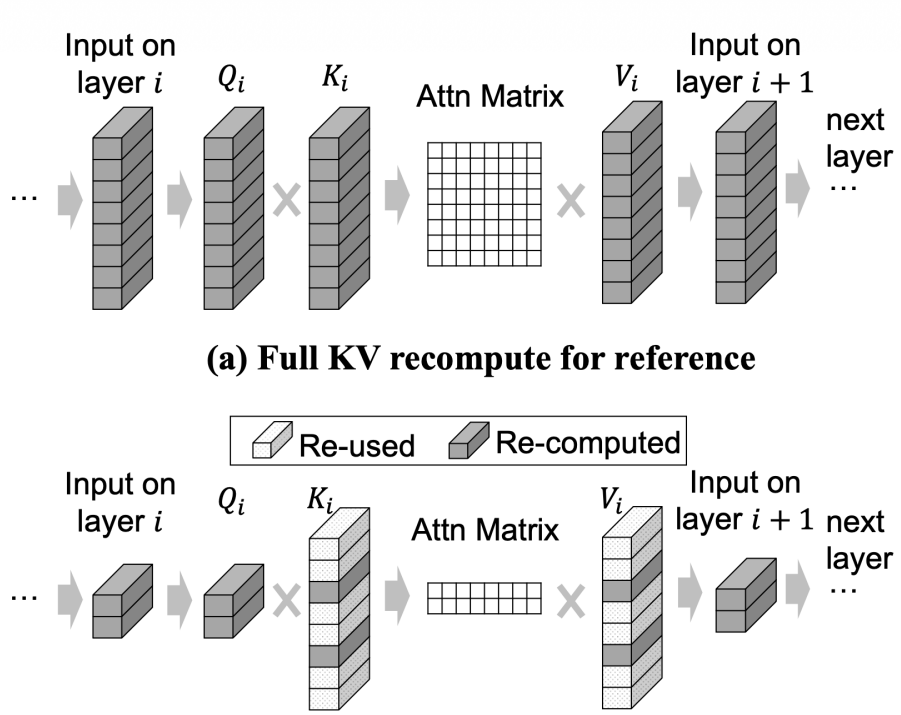


Image Credit: https://doi.org/10.1145/3689031.3696098

Selective Recomputation

- All tokens in the first layer are recomputed, and then **gradual filtering** is used



(a) Full KV recompute for reference

(b) Selective KV recompute on two selected tokens

Image Credit: https://doi.org/10.1145/3689031.3696098

CPU Offloading

CPU Offloading

- Offload **model parameters** and **KV cache** to host memory
- Offload **compute** to host CPU
- Uses cheap and abundant **host** resources
- Increases effective batch size
- **PCIe**, however, has much lower bandwidth than HBM

Offloading the KV Cache

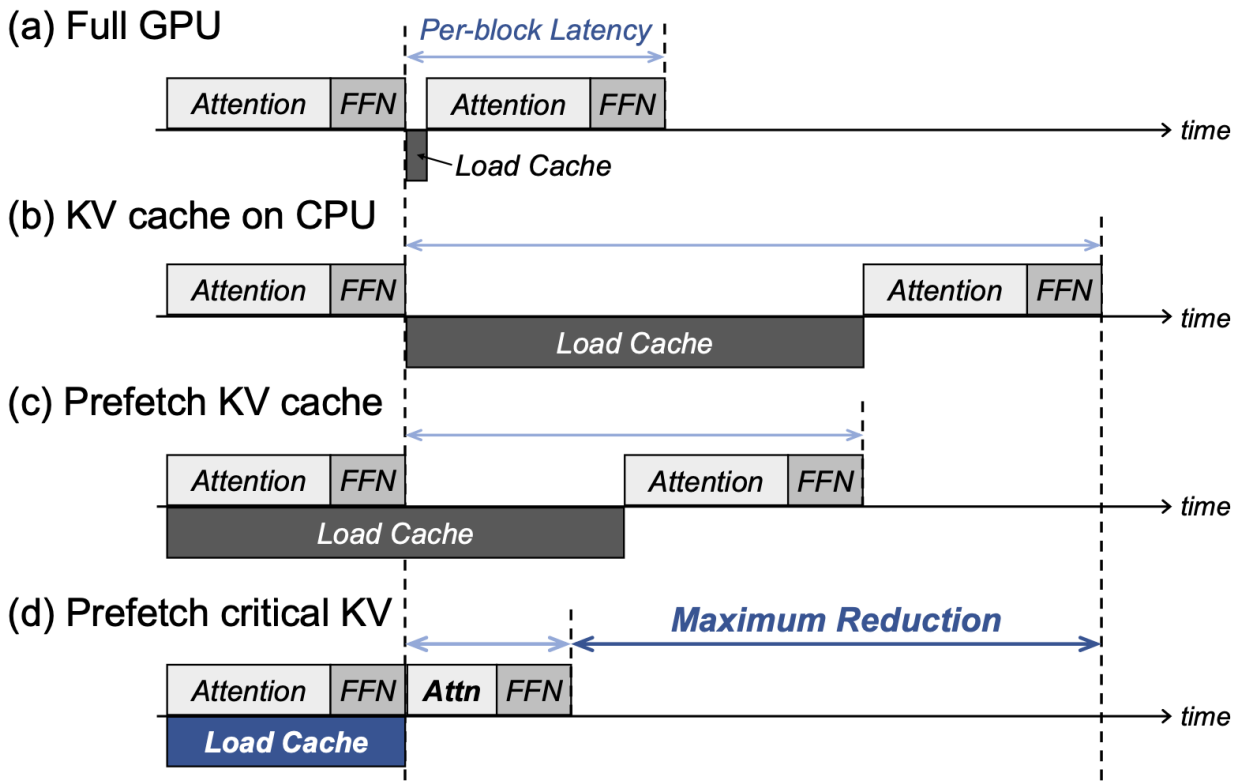


Image Credit: https://dl.acm.org/doi/10.5555/3691938.3691947

InfiniGen

- **Infinigen** predicts tokens which will actually receive significant attention weight
- Tokens stay in CPU memory and not evicted, until free space is available
- Consecutive layers are similar
  - At a previous layer, compute approx attention weights for the next layer and select tokens

Offloading Compute

- Gap in memory bandwidth between CPU and GPU is smaller than gap in compute
- Memory-bandwidth **decode** can be offloaded
- CPU idle in simple offloading

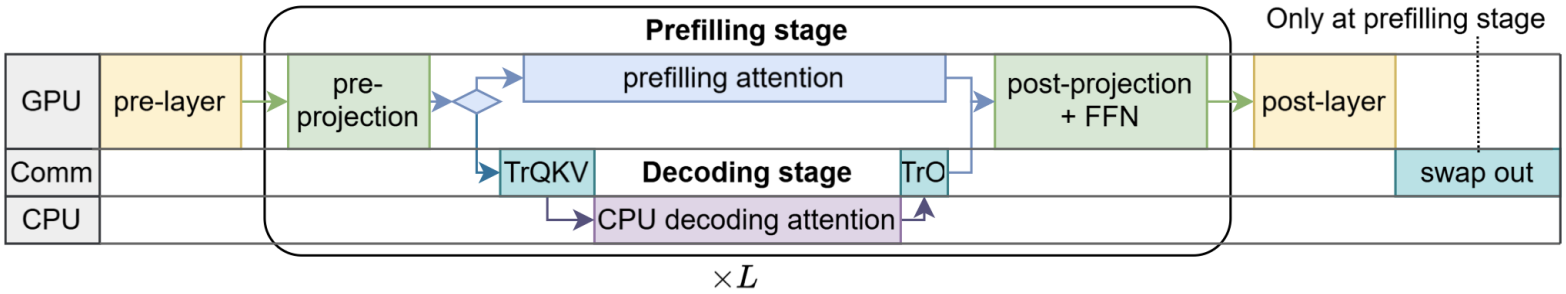
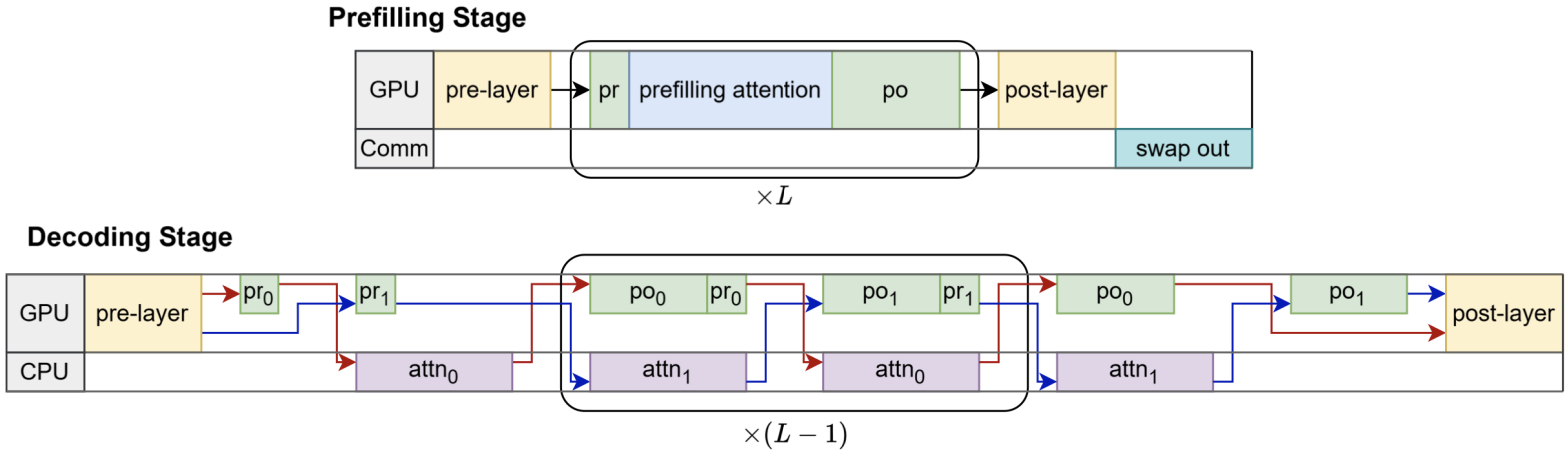


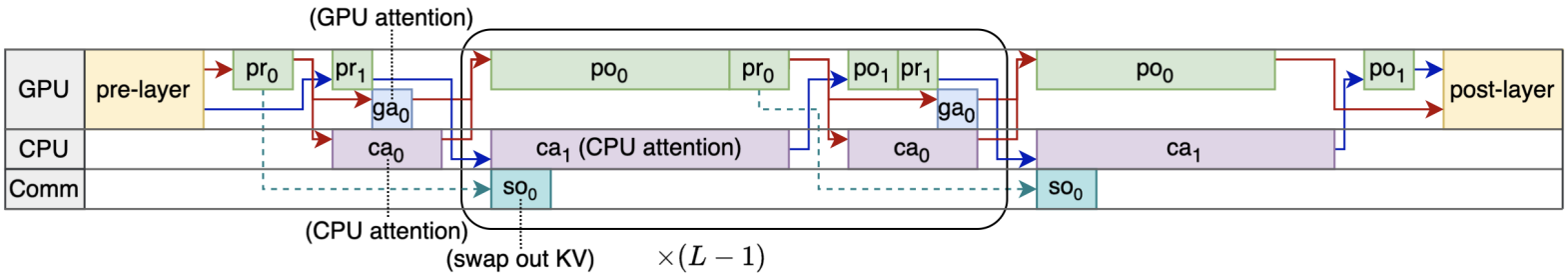
Image Credit: https://arxiv.org/abs/2411.01142

Pipelining During Compute Offload

- **Symmetric** pipelining



- **Asymmetric** pipelining



Offloading for Long-Context Serving

- PCIe transfers cannot be hidden behind computation for long context requests
- Small pages are bad for utilizing PCIe bandwidth, larger ones reduce cache hit rates
- Requests sharing context might load it multiple times if they do not arrive together (delay hits)

Strata

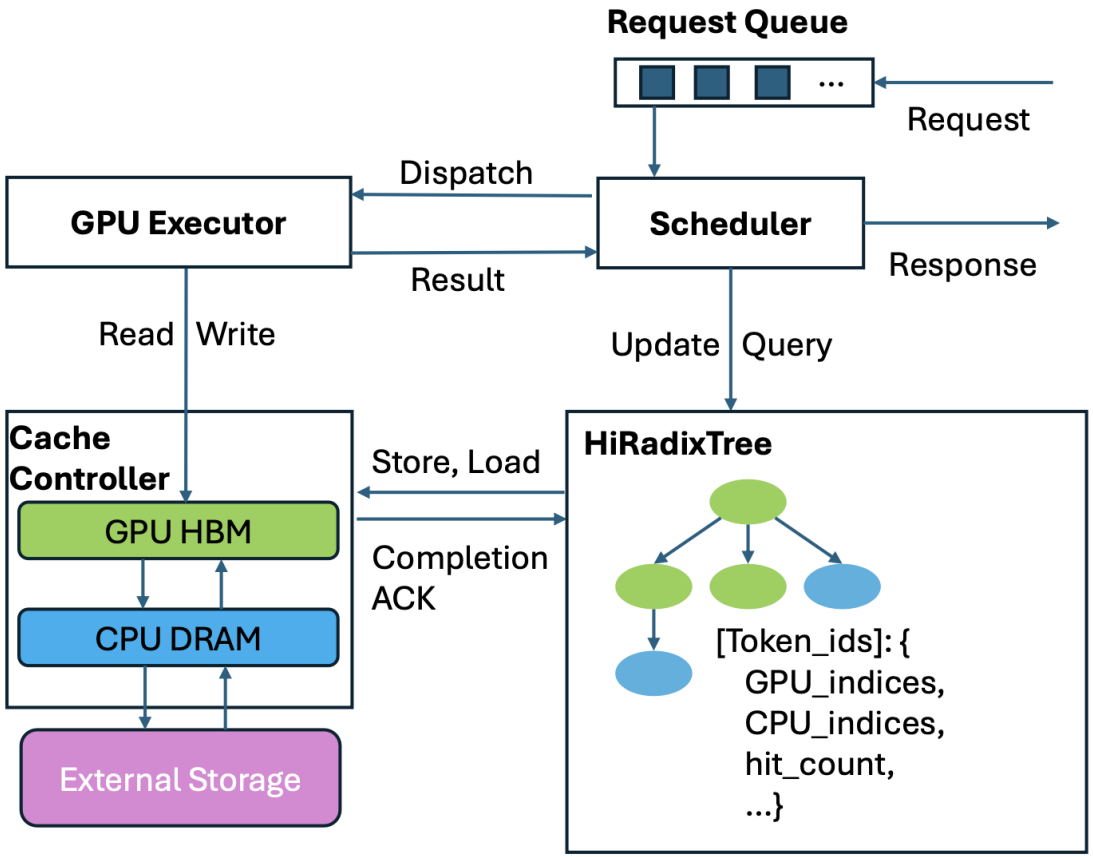


Image Credit: https://www.usenix.org/conference/osdi26/presentation/xie-zhiqiang

Strata's Data Plane

- Launches a cuda kernel with 1000s of threads for efficient GPU-assisted IO transfers
- Different **layouts** for computation and transfers

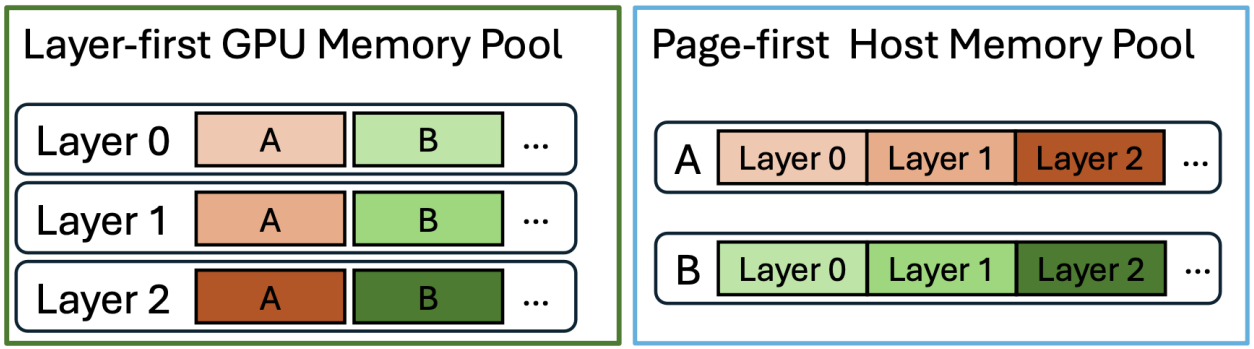


Image Credit: https://www.usenix.org/conference/osdi26/presentation/xie-zhiqiang

Strata's Control Plane

- Uses **HiRadixTree** for each request's load and compute cost
- Avoids delay hits, balances load time and compute time

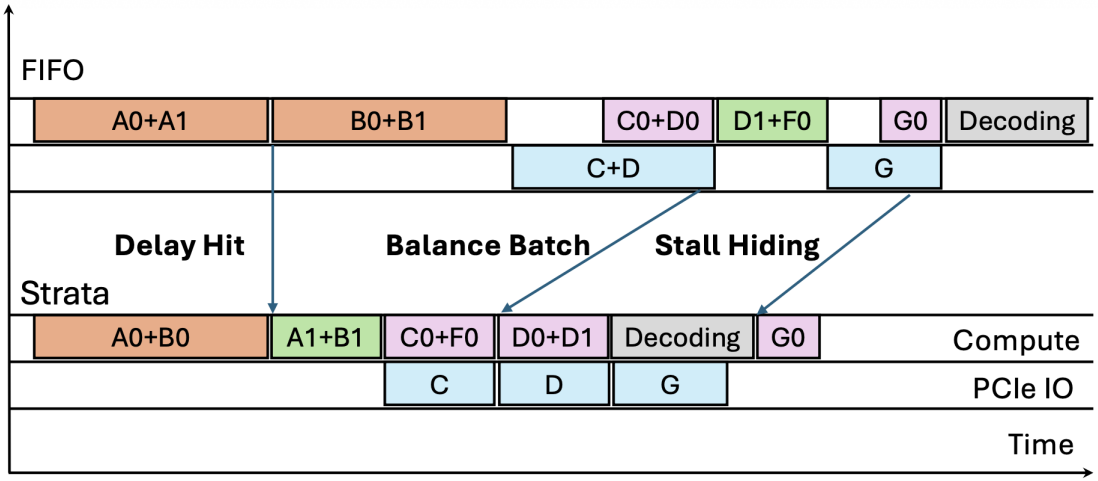


Image Credit: https://www.usenix.org/conference/osdi26/presentation/xie-zhiqiang

KV Cache Compression and Sparsity

KV Cache Compression

- KV cache **grows** linearly with sequence length, batch size, and model depth, can become larger than model weights
- GPU's HBM is limited and decode is memory-bandwidth bound
- Model-level and data-level (runtime) techniques

Grouped-Query Attention

- **Standard MHA** – every head has its own key and vaule projections
- **Multi-query attention** – all heads share a single key and value matrix
- **Grouped-query attention** – groups of query heads share key and value heads
- Existing multi-head checkpoint can be uptrained into a GQA (or MQA) model, without training from scratch

Grouped-Query Attention

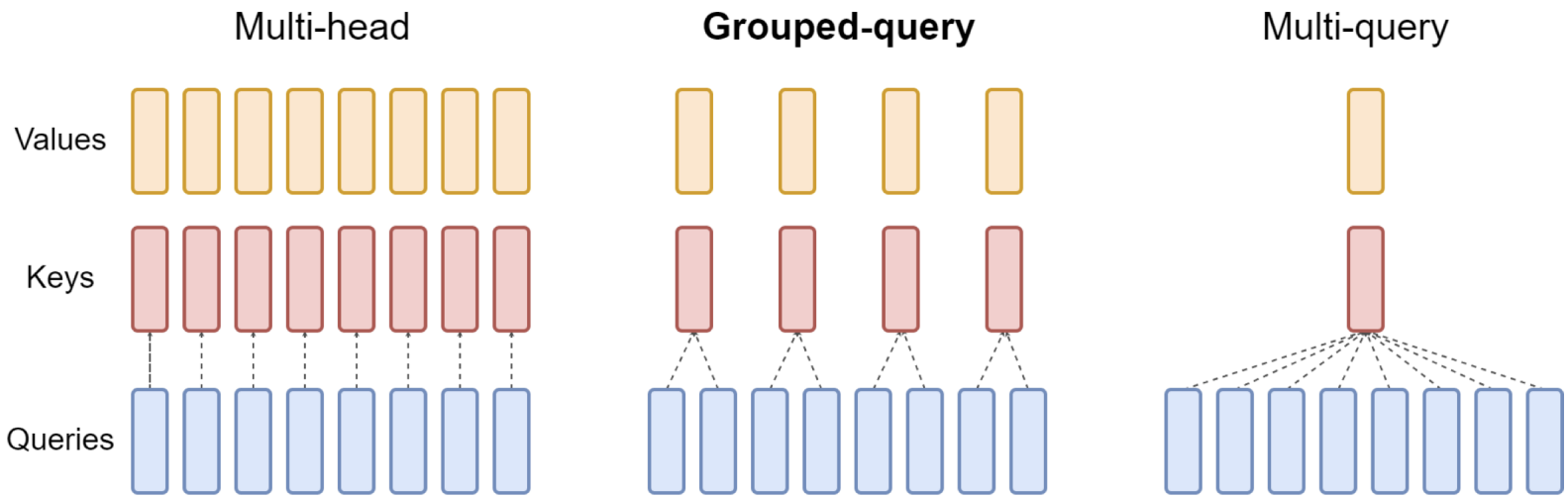


Image Credit: https://aclanthology.org/2023.emnlp-main.298/

Differentiated KV Cache Management

- Not all parts of KV cache are equally important
- Keys are more **sensitive** to precision loss than values
- Starting tokens receive high attention scores
- Dynamic compression removes fixed page size regularity, can lead to fragmentation
- **DiffKV** uses a memory manager that performs compaction with many threads in parallel

Differentiated KV Cache Management

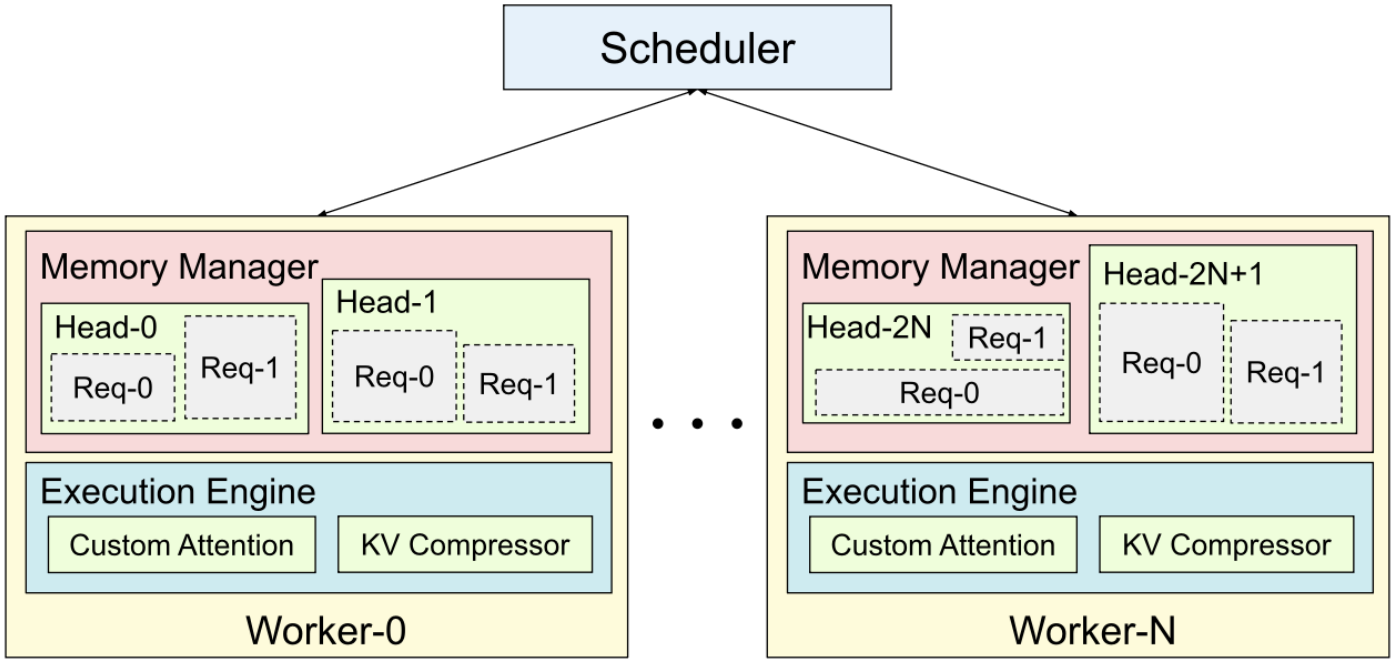


Image Credit: https://doi.org/10.1145/3731569.3764810

Sparse Attention

- Only a small **subset** of past tokens actually contribute to attention output
- If we select k tokens, attention cost grows **linearly** instead of quadratically
- Training-free approaches use indexer to select top-k tokens for approximate attention scores
- **DeepSeek** bakes sparsity in training itself

ECHO

- KV cache still grows with context length, HBM is limited
- **Prefetching** done to hide PCIe transfer latencies
- **ECHO** predicts top-k tokens using moving average heuristic over preceeding tokens

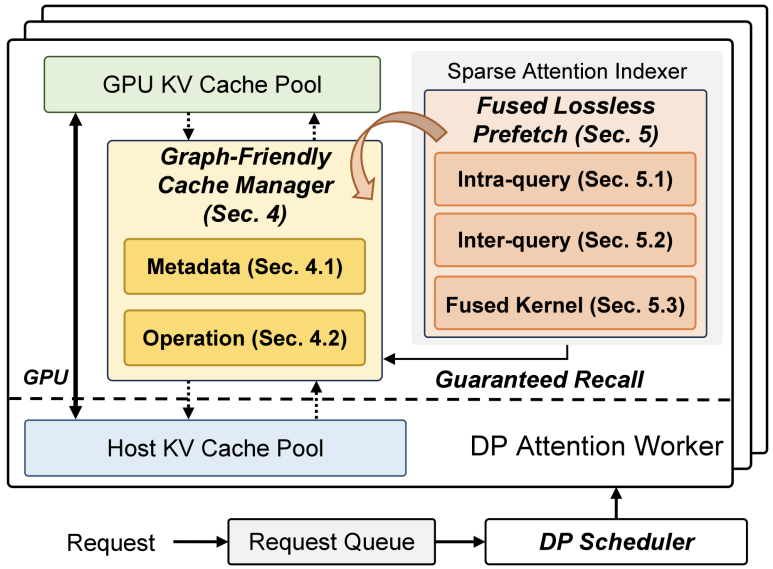


Image Credit: https://www.usenix.org/conference/osdi26/presentation/liu-guangda

Communicating KV Cache in Disaggregated Serving

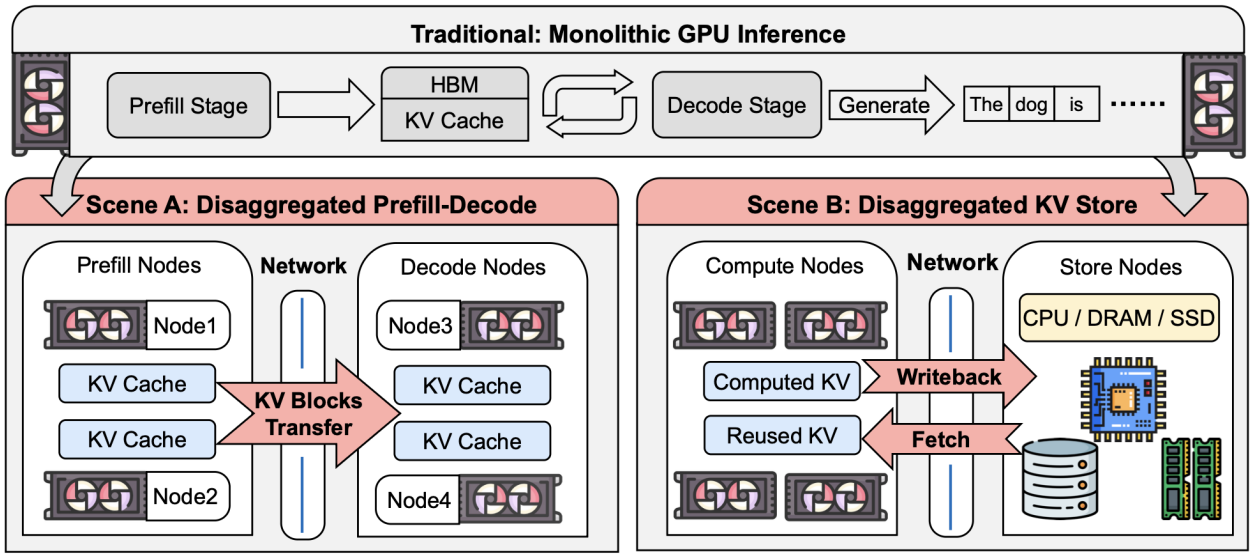


Image Credit: https://doi.org/10.1145/3789240.3829139

KVServe

- **KVServe** uses workload specific compression techniques

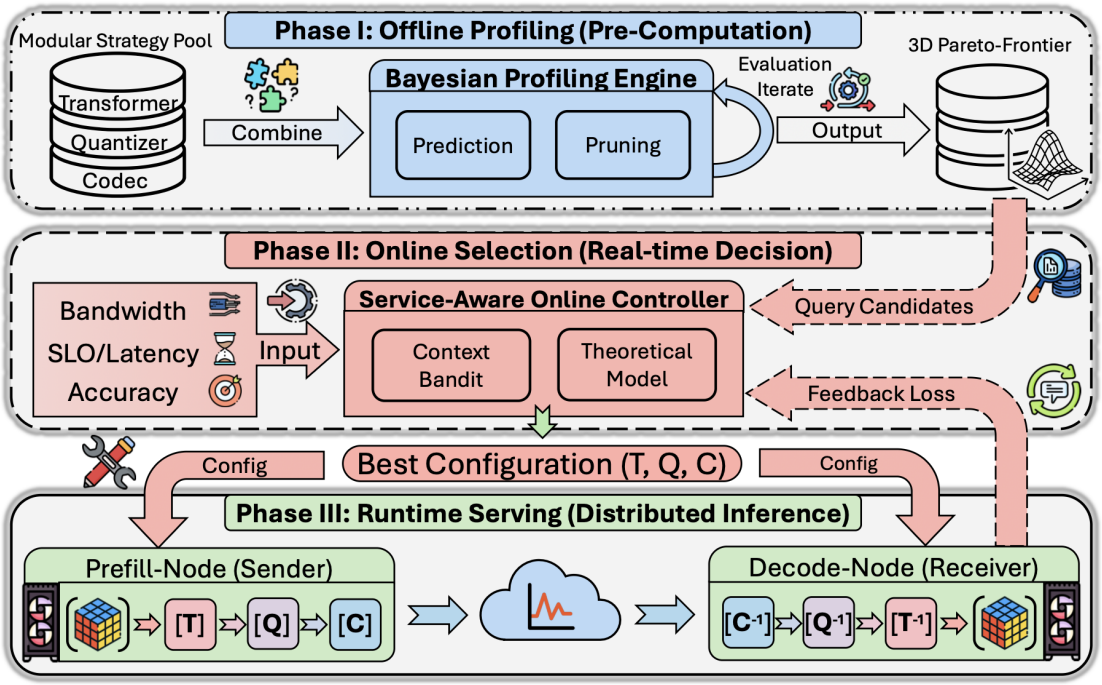


Image Credit: https://doi.org/10.1145/3789240.3829139

Inference Engine

Inference Engine

- Small operating system for language model requests
- **Frontend** accepts prompt and sampling parameters
- **Scheduler** decides which requests will go in a batch
- **Block manager** responsible for KV cache
- **Model runner** owns model weights, runs forward pass
- Nano-vLLM is a good readable example

Inference Engine

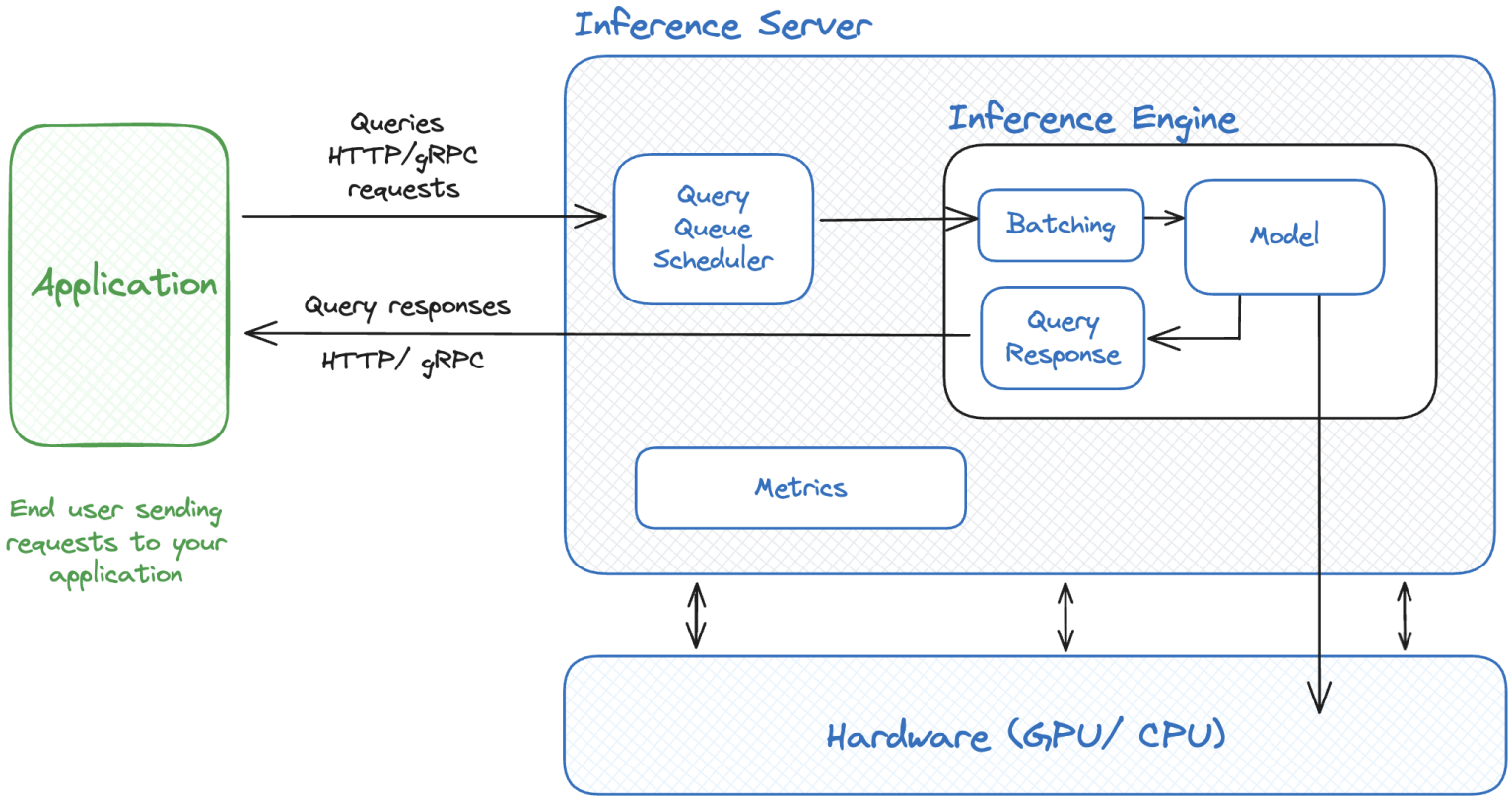


Image Credit: https://www.koyeb.com/blog/best-llm-inference-engines-and-servers-to-deploy-llms-in-production