

Paged Attention & vLLM for Efficient LLM Inference Engine

Adapted from the guest lecture by Woosuk Kwon



Language Technologies Institute | Carnegie Mellon University School of Computer Science

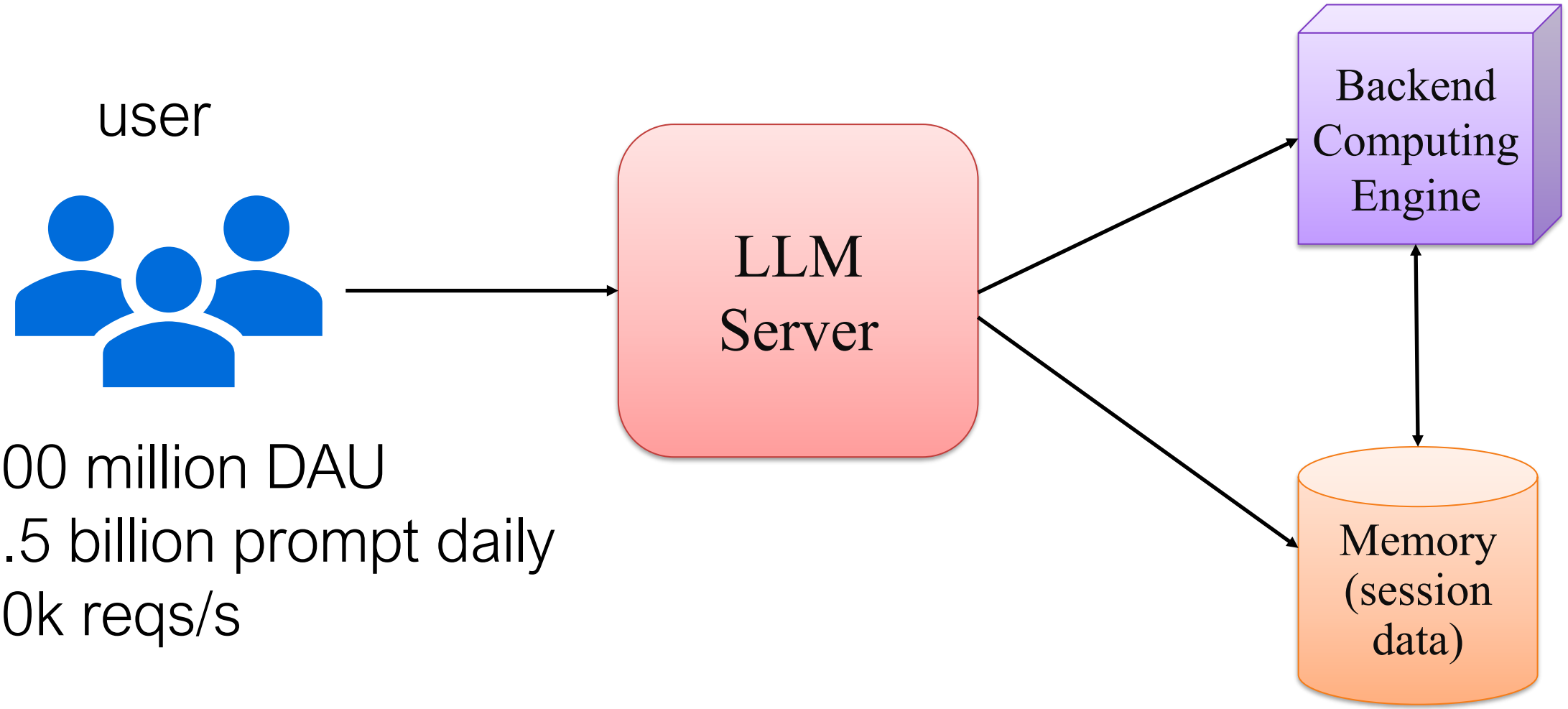
LLM Serving Problem

- LLMs typically run on high-end GPUs (e.g., NVIDIA H100)
- Yet, each GPU can only serve a handful of QPS; even simple queries (with reasoning) can take minutes





LLM Serving



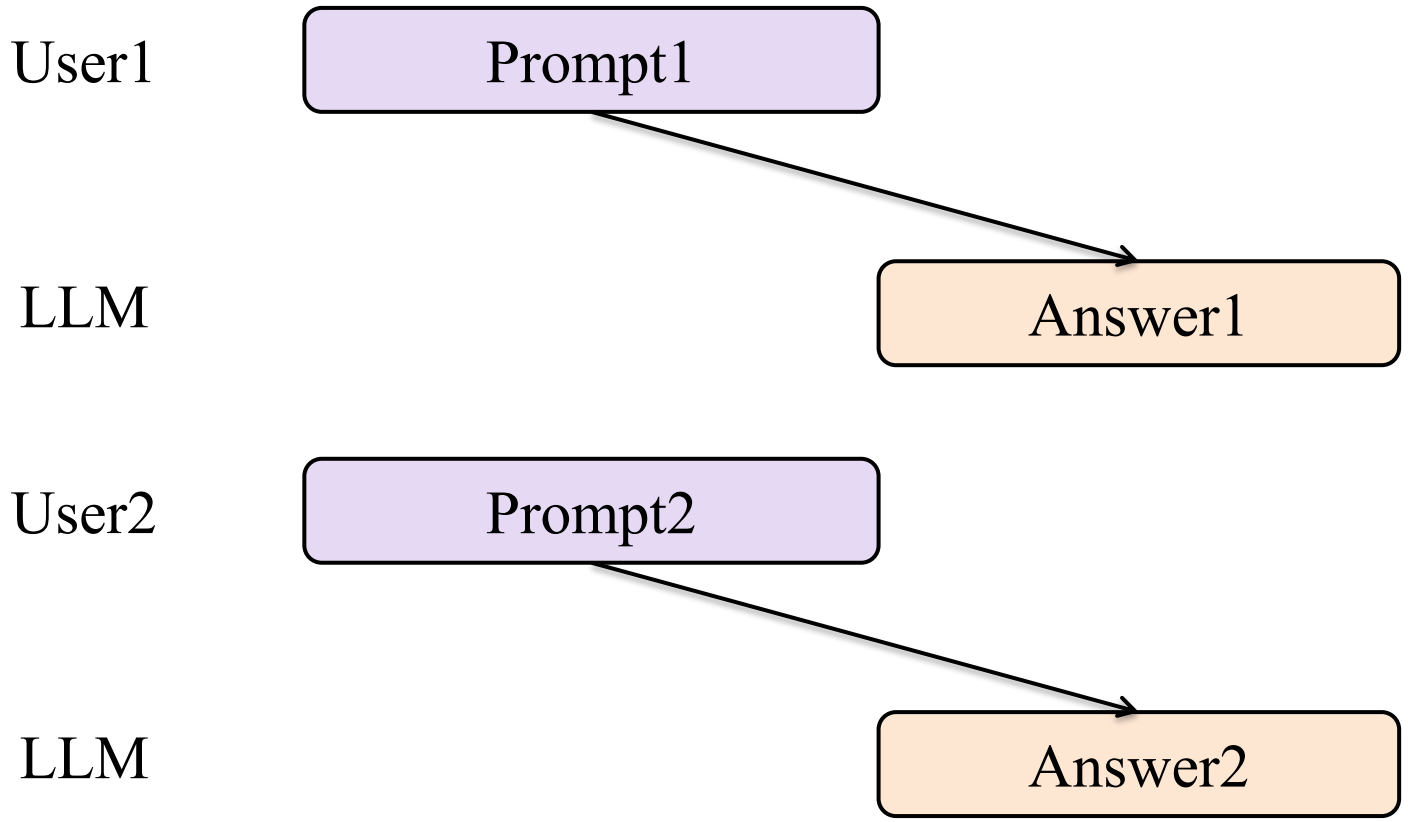
200 million DAU
2.5 billion prompt daily
30k reqs/s

Inference vs Serving

- Inference:
  - Usually offline: batch throughput is important
  - Single user: no need to maintain session
- Serving:
  - Online
  - Multi-user
  - Latency!

LLM Serving Pattern

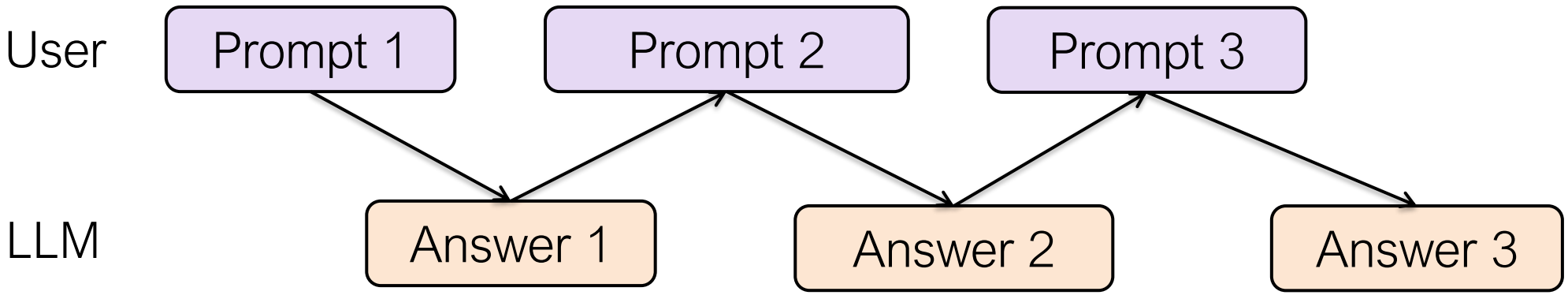
Multi-user Single Round Generation



Different users may issue same prompts

LLM Serving Pattern

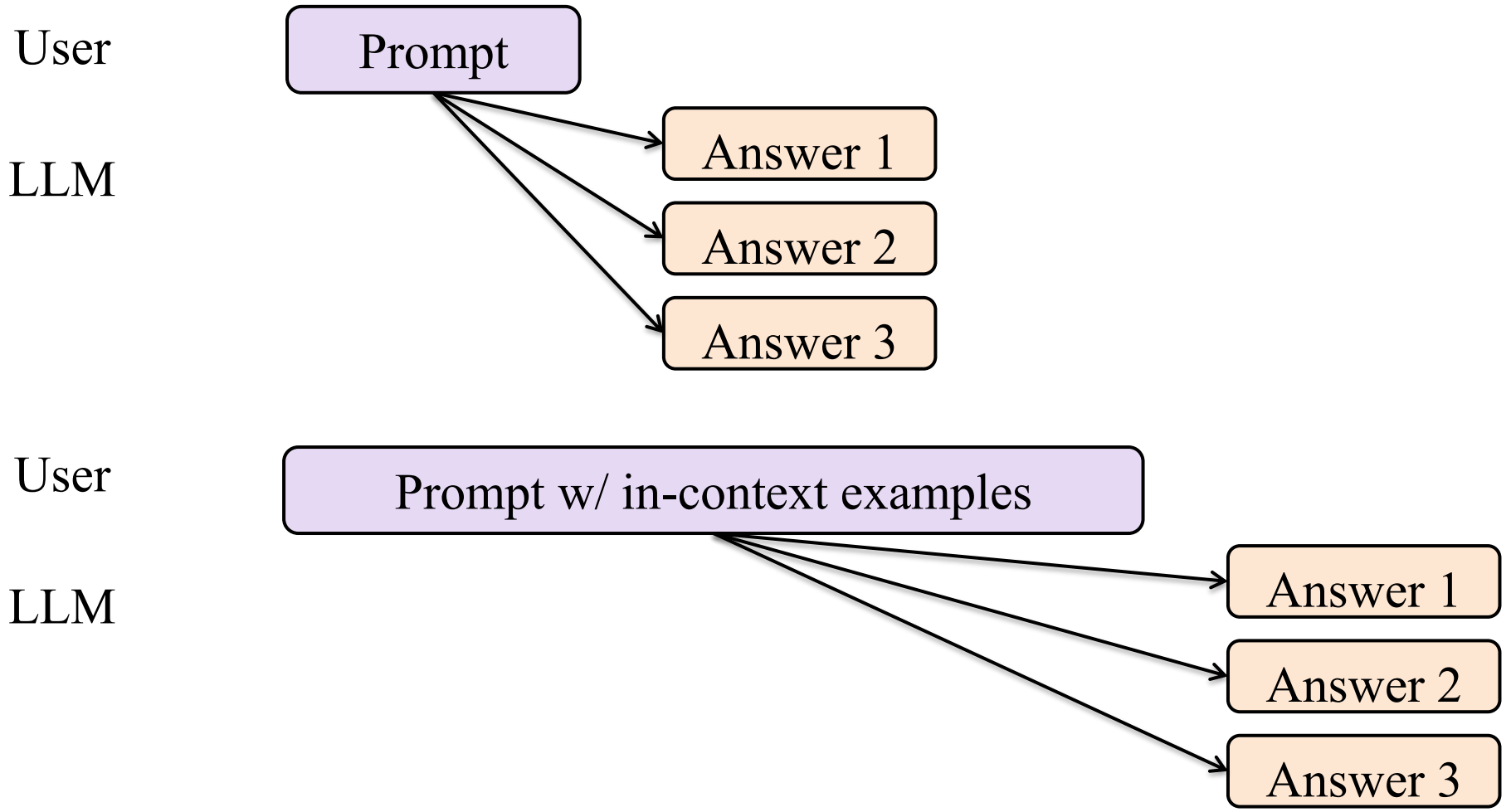
Multi-round Chat



Should LLM server keep the user chat history in GPU memory and wait for next turn?

LLM Serving Pattern

Multi-Generation

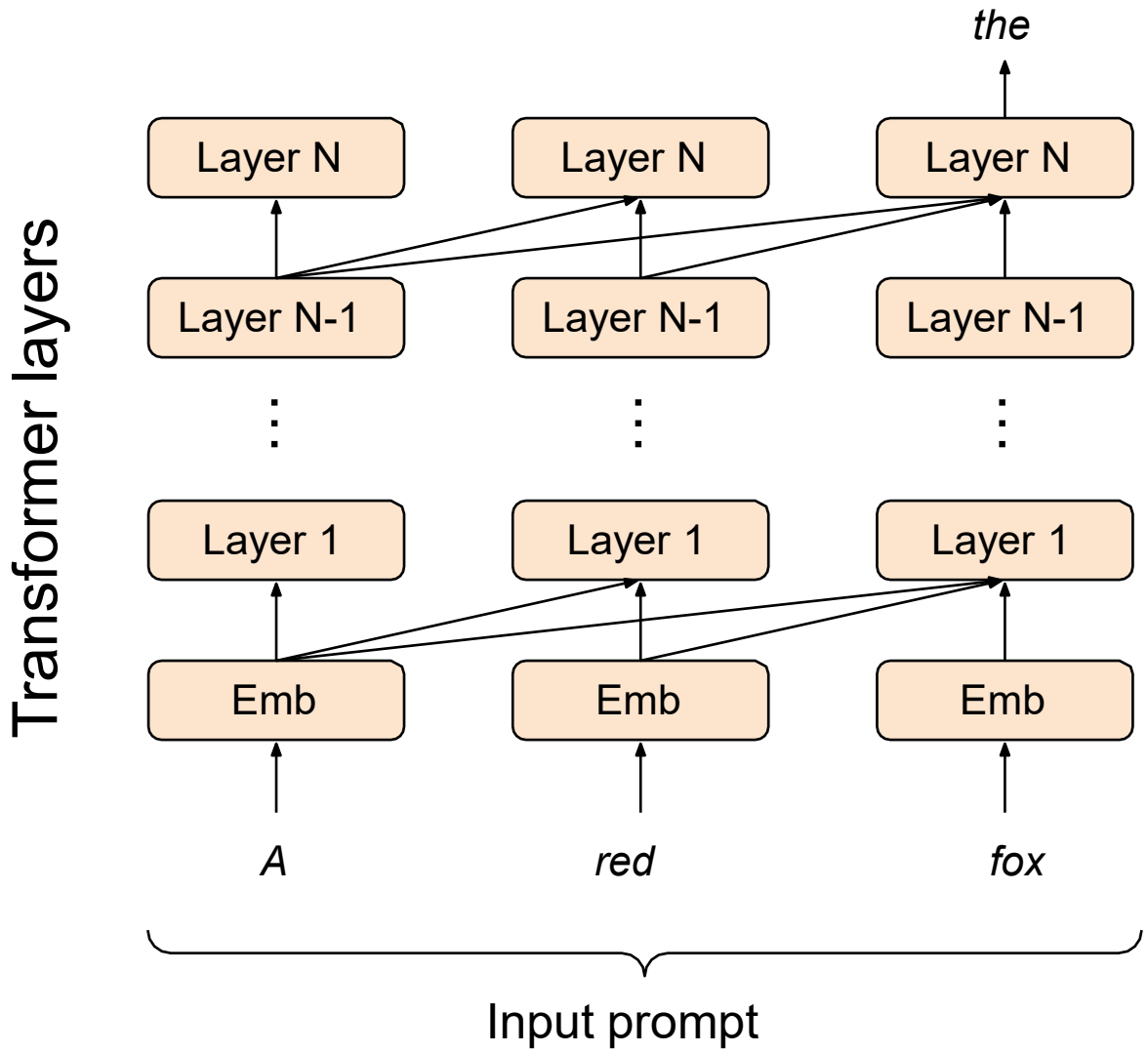


Execute the LLM generation three times?

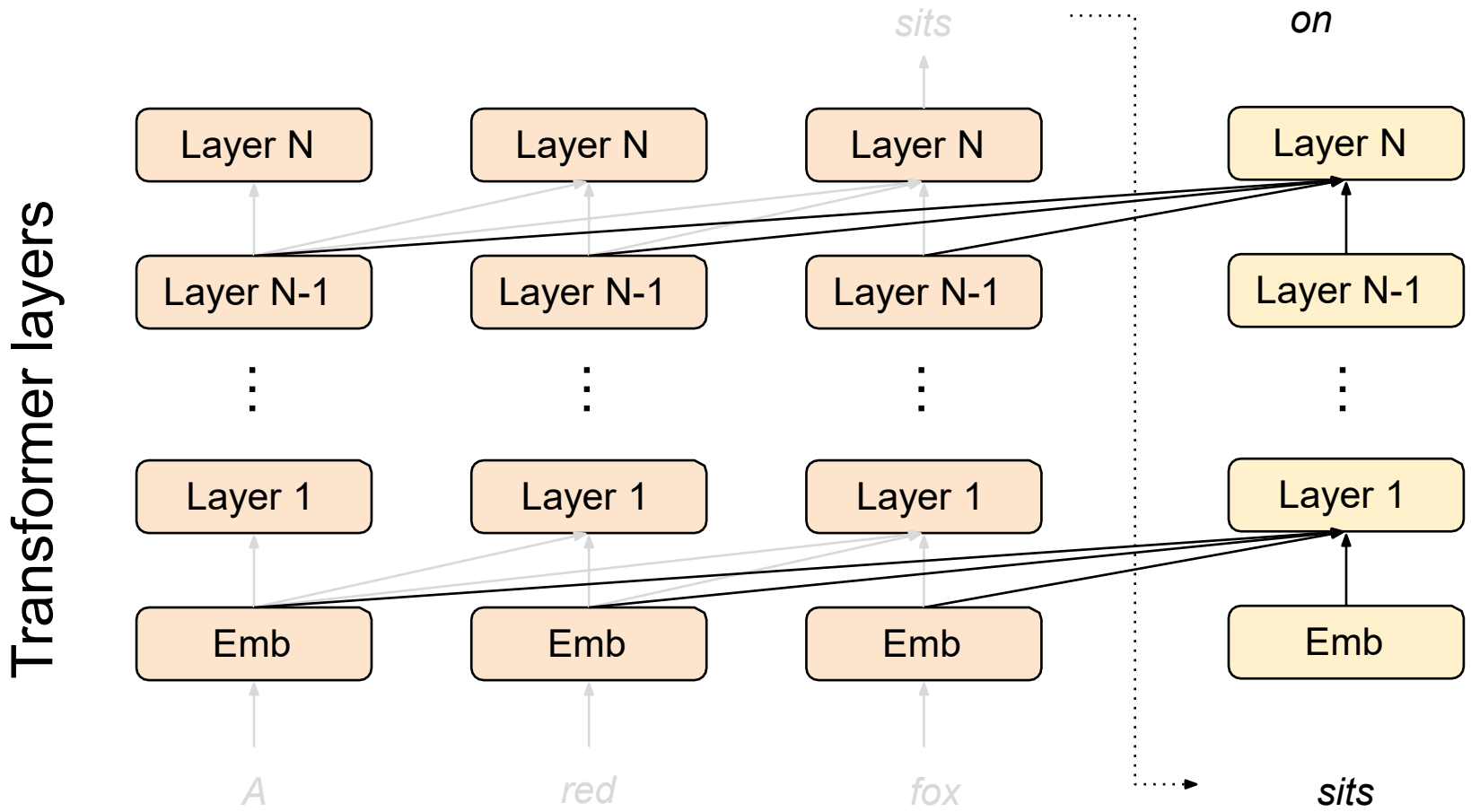
Goals and Challenges of Efficient LLM Server

- Maintain multi-user sessions
- Handle massive user requests at one time
- Enable Large Model and Long sequence generation
- Optimizing for Throughput and Latency
- Utilize heterogeneous devices (CPU/GPU)

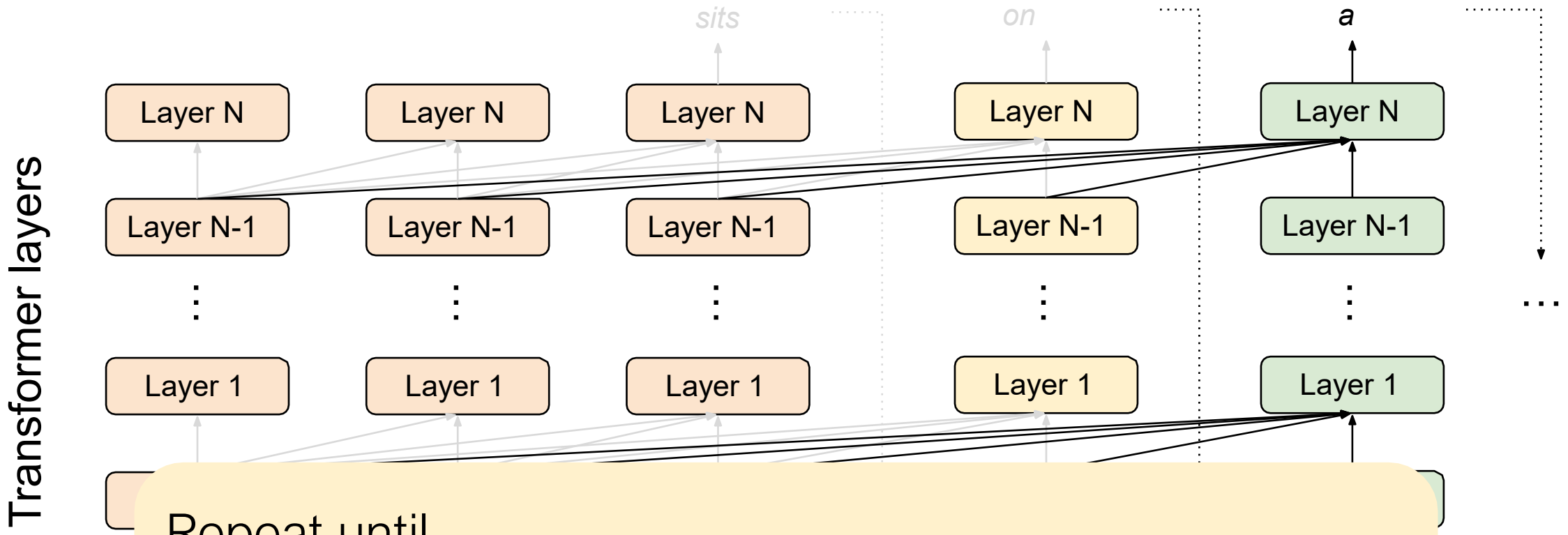
LLM Inference Process



LLM inference process



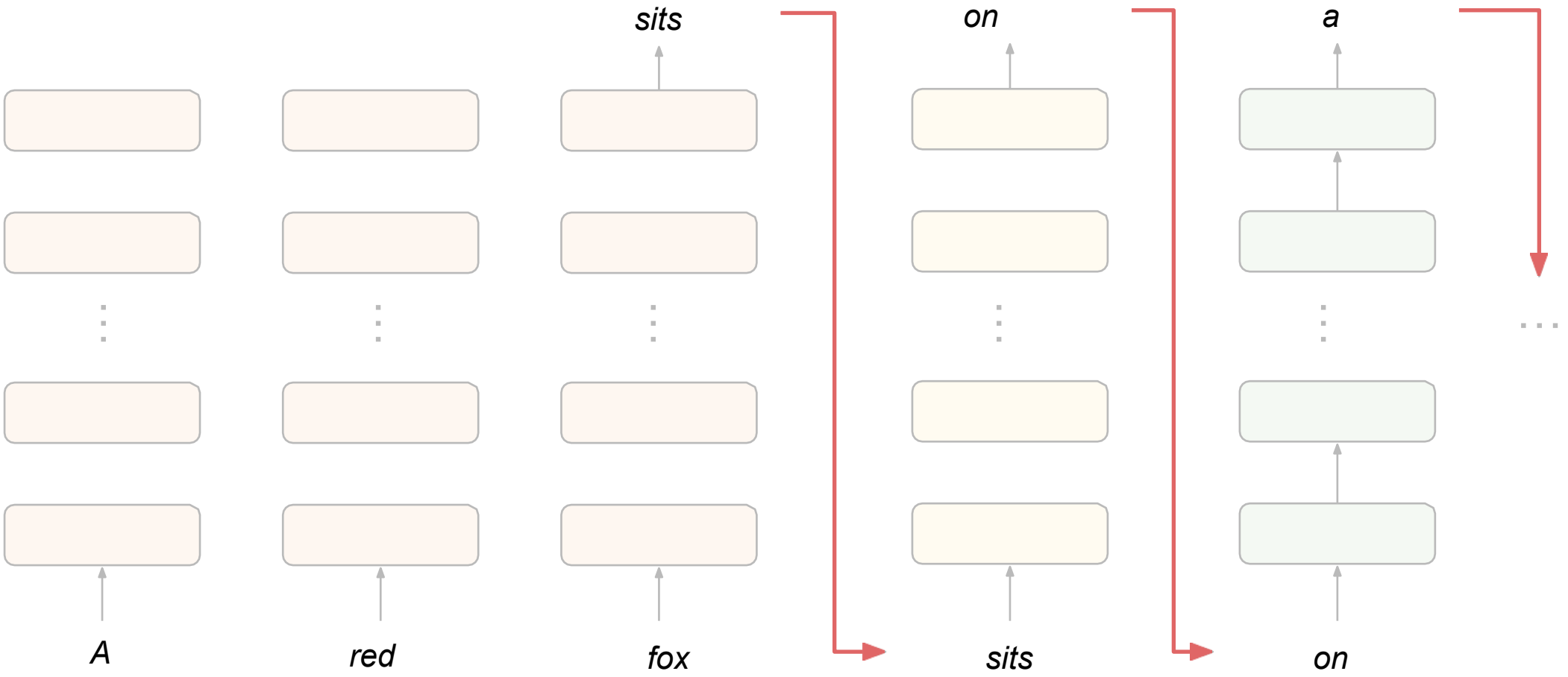
LLM inference process



Repeat until

- Reaches maximum length (e.g., 1M tokens)
- Generate certain tokens (e.g., [EOS], [EOT])
- ...

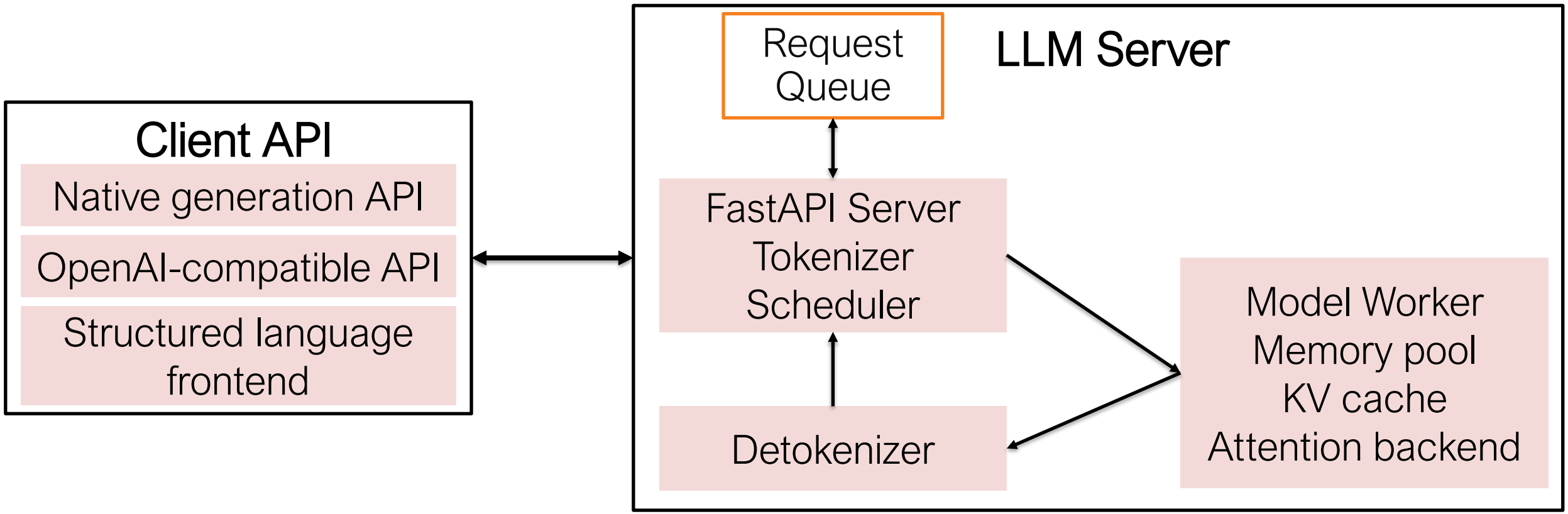
Latency in LLM Inference



Sequential dependency → Hard for GPU parallelization

LLM Server Architecture

SGLang / vLLM share similar arch



LLM Request Scheduler

1. Receive input messages
2. Stream model outputs
3. Check the stop conditions
4. Reorder requests and prepare a batch
5. Allocate memory for the next batch and running batch

```
while True:
    recv_reqs = recv_requests()
    process_input_requests(recv_reqs)
    batch = get_next_batch_to_run()
    result = run_batch(batch)
    process_batch_result(batch, result)
```

Server scheduler example

```
while True:
    recv_reqs = self.recv_requests()
    self.process_input_requests(recv_reqs)

    batch = self.get_next_batch_to_run()
    self.cur_batch = batch

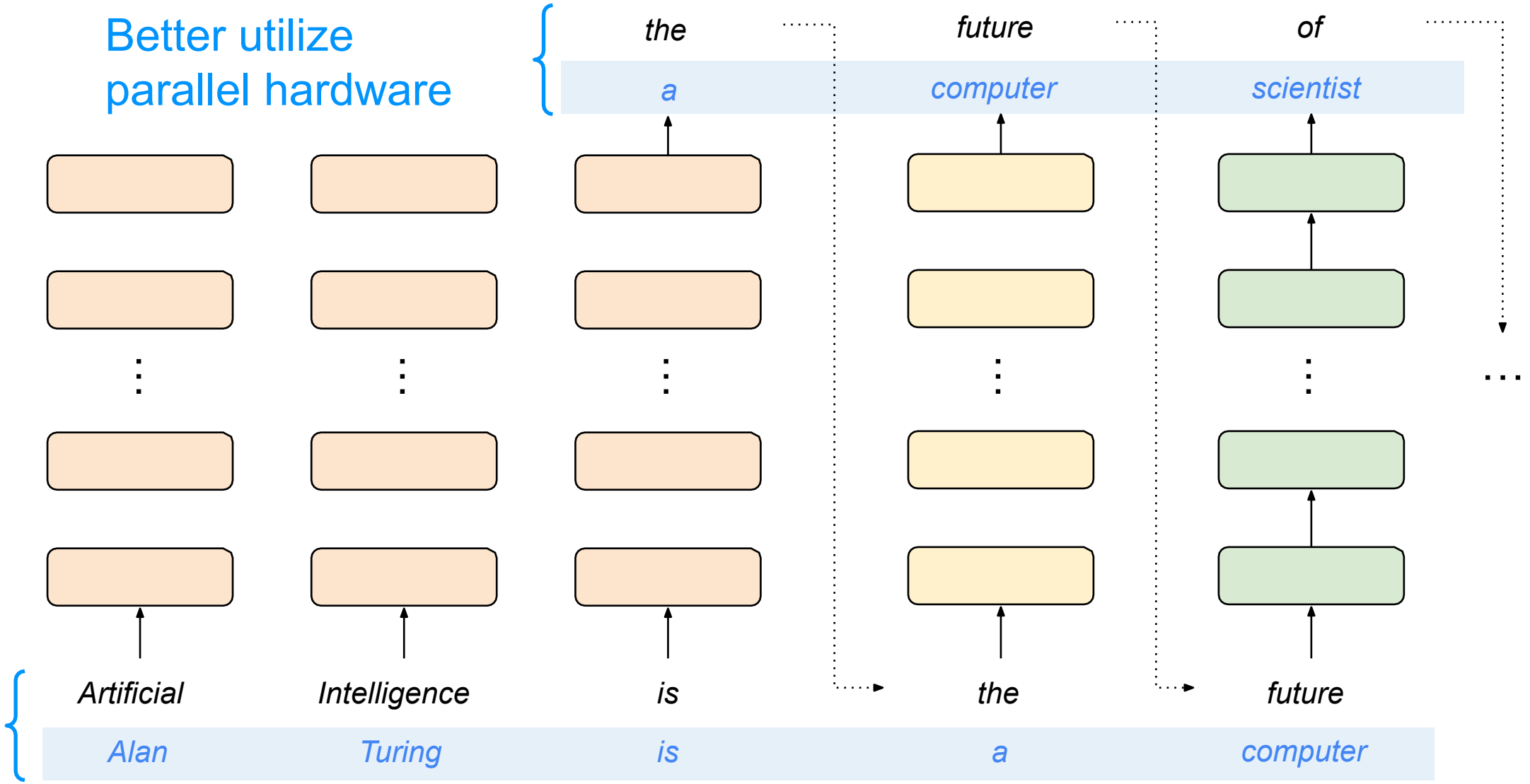
    if batch:
        result = self.run_batch(batch)
        self.result_queue.append((batch.copy(), result))

        if self.last_batch is None:
            # Create a dummy first batch to start the pipeline for overlap schedule.
            # It is now used for triggering the sampling_info_done event.
            tmp_batch = ScheduleBatch(
                reqs=None,
                forward_mode=ForwardMode.DUMMY_FIRST,
                next_batch_sampling_info=self.tp_worker.cur_sampling_info,
            )
            self.process_batch_result(tmp_batch, None)

    if self.last_batch:
        # Process the results of the last batch
        tmp_batch, tmp_result = self.result_queue.popleft()
        tmp_batch.next_batch_sampling_info = (
            self.tp_worker.cur_sampling_info if batch else None
        )
        self.process_batch_result(tmp_batch, tmp_result)
    elif batch is None:
        # When the server is idle, do self-check and re-init some states
        self.check_memory()
        self.new_token_ratio = self.init_new_token_ratio

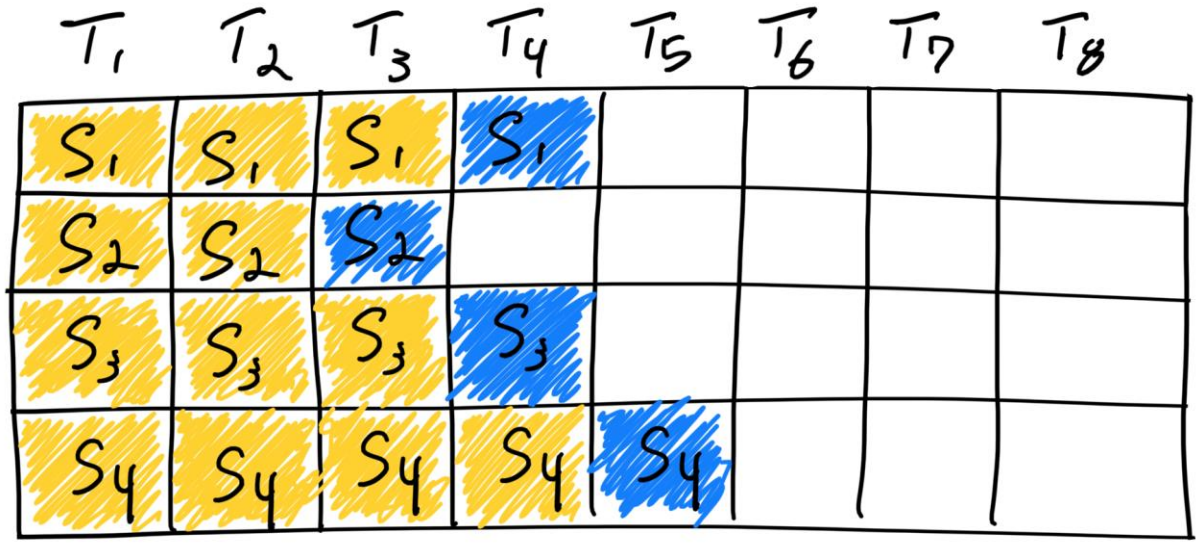
    self.last_batch = batch
```

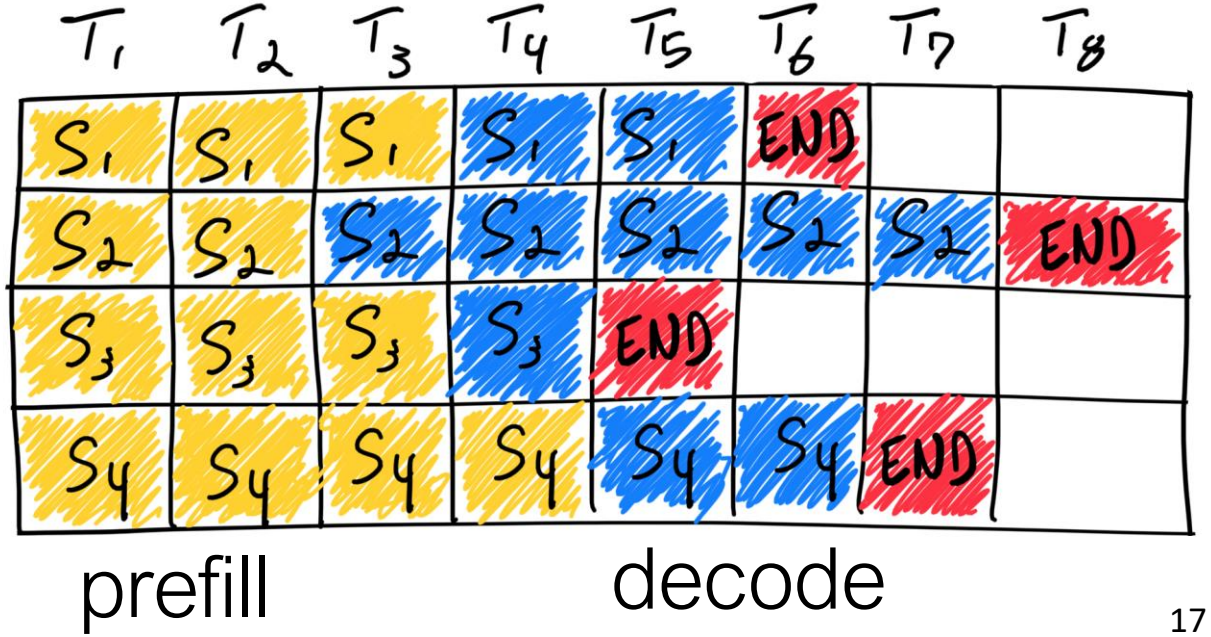
Batching


Better utilize
parallel hardware
the
future
of
a
computer
scientist
Artificial
Intelligence
is
the
future
Alan
Turing
is
a
computer

Serving Challenge: Waiting for Longest Gen

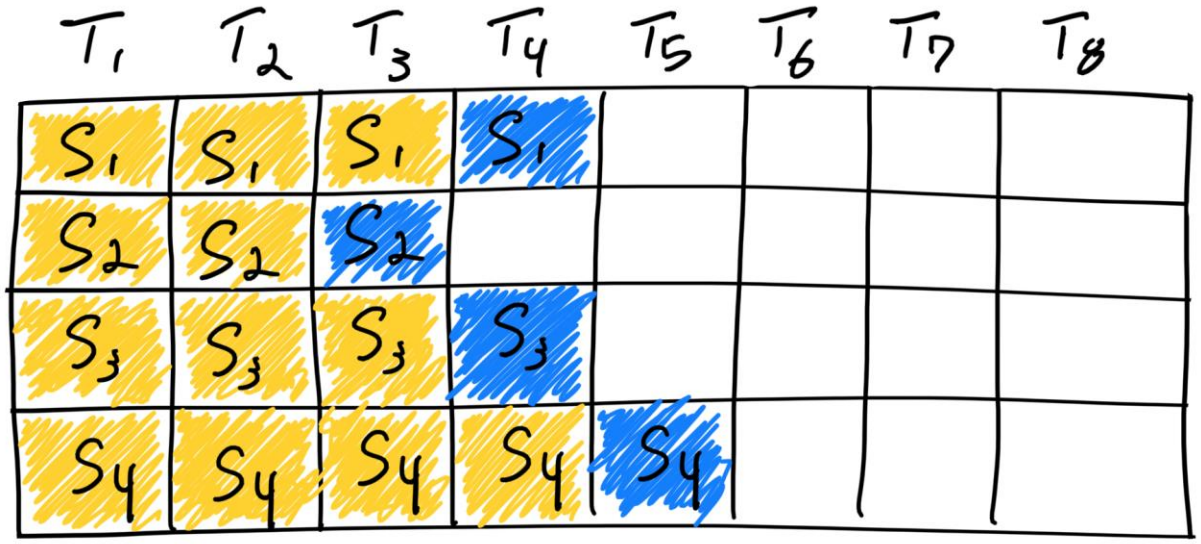
- A Request Batch may lead to different lengths.
- Naïve implementation needs to wait for the longest seq

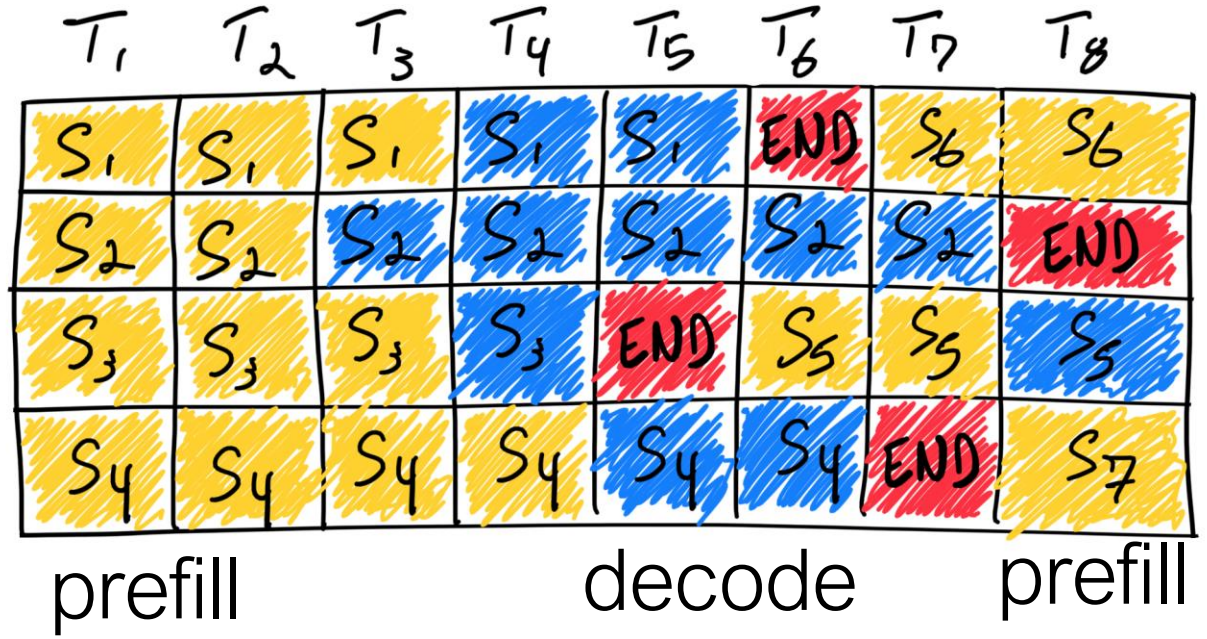




Serving with Continuous Batching

- Iteration-level: at each iteration (i.e. each token generation step) schedule new request whenever one request finishes in a batch

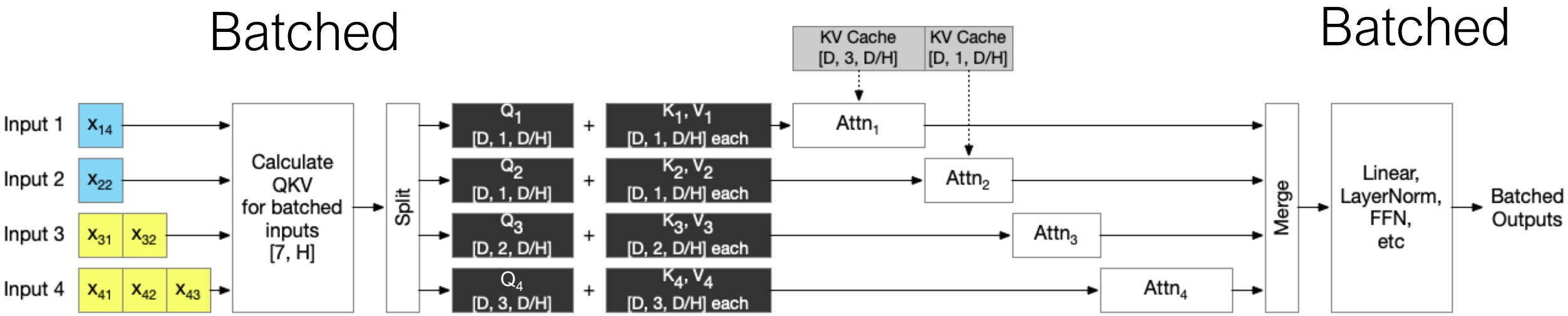




ORCA: A Distributed Serving System for Transformer-Based Generative Models. Yu et al. OSDI 2022

Serving with Selective Batching

- Selectively batch all non-attention operations (linear, layer norm, GeLU, etc)



prefill is grouped into one step

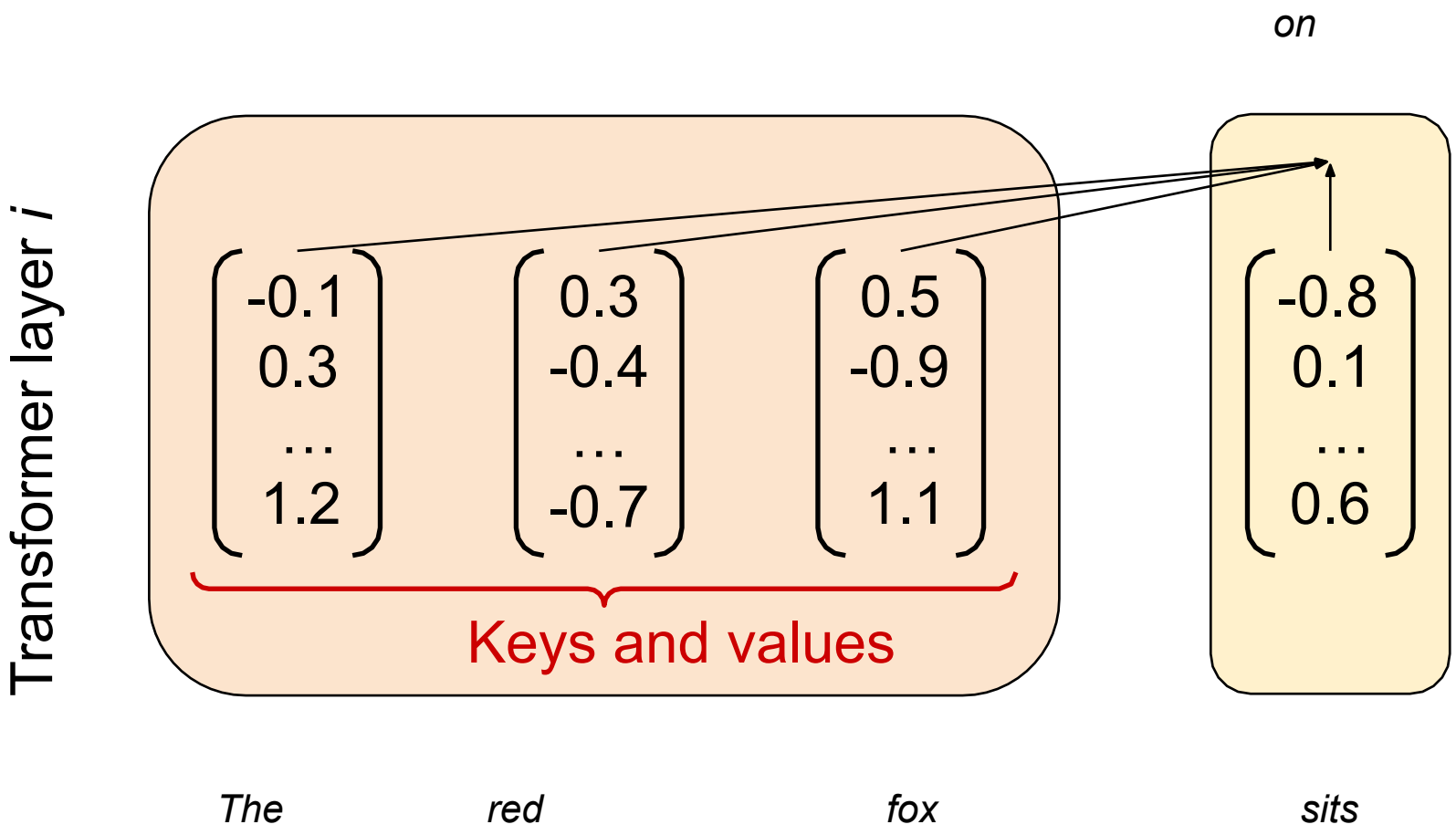
Executed by Attention Engine (e.g. PagedAttention, next)

ORCA: A Distributed Serving System for Transformer-Based Generative Models. Yu et al. OSDI 2022

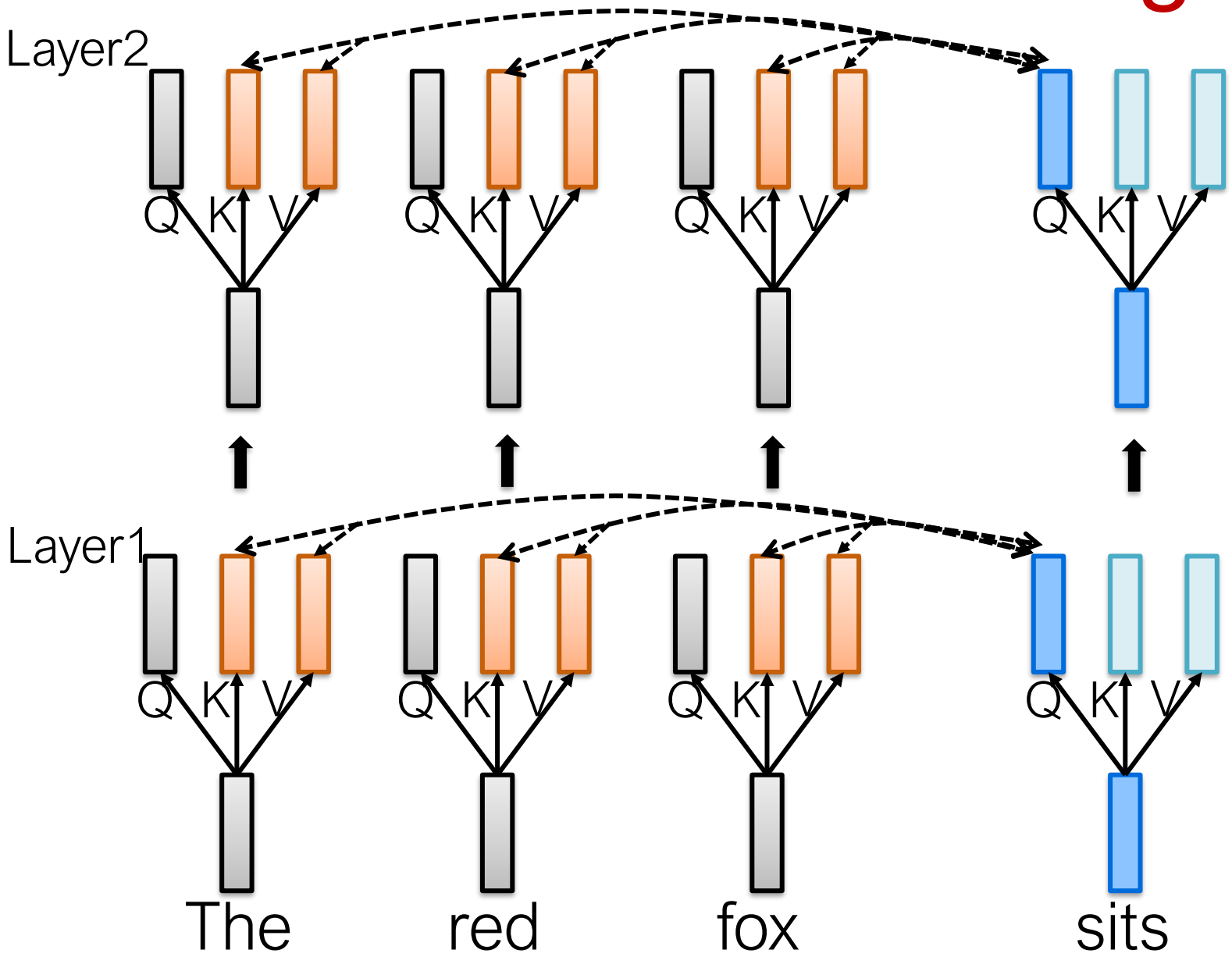
KV Cache Management

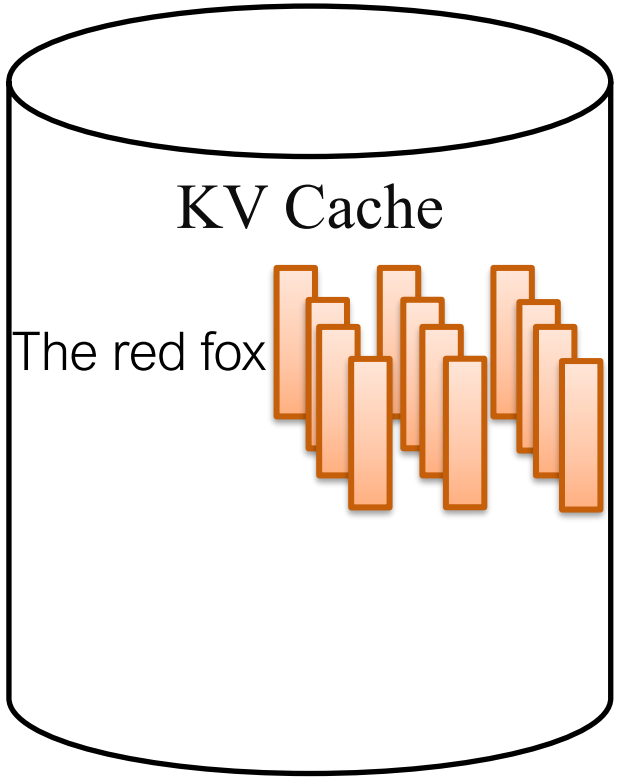
- Standard NN inference only stores the current layer's output, but do not store all intermediate results for each layer
- This is ok for LSTM/RNN, but it is increasingly slower for Transformer inference – generating each next token would require all previous keys and values for each Transformer layer.
- Store all keys, values for each layer in GPU memory ➔ KV Cache

Attention needs KV



KV Cache Management



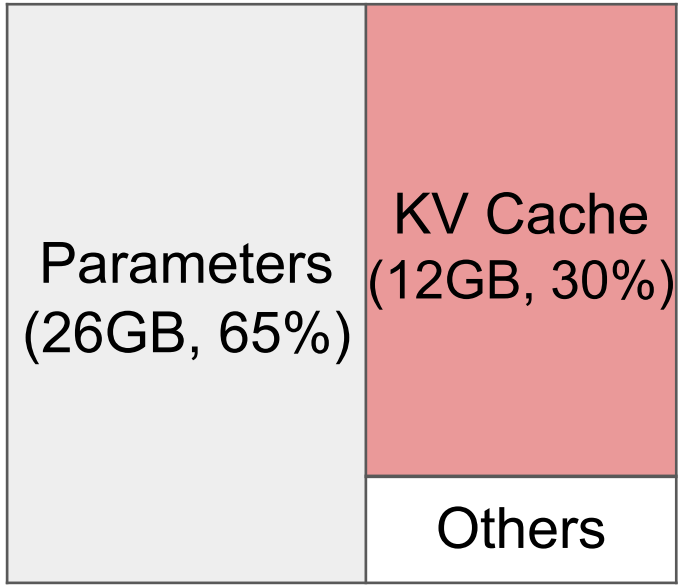


$$\text{Att} = \text{Softmax}(q \cdot K^T) \cdot V$$

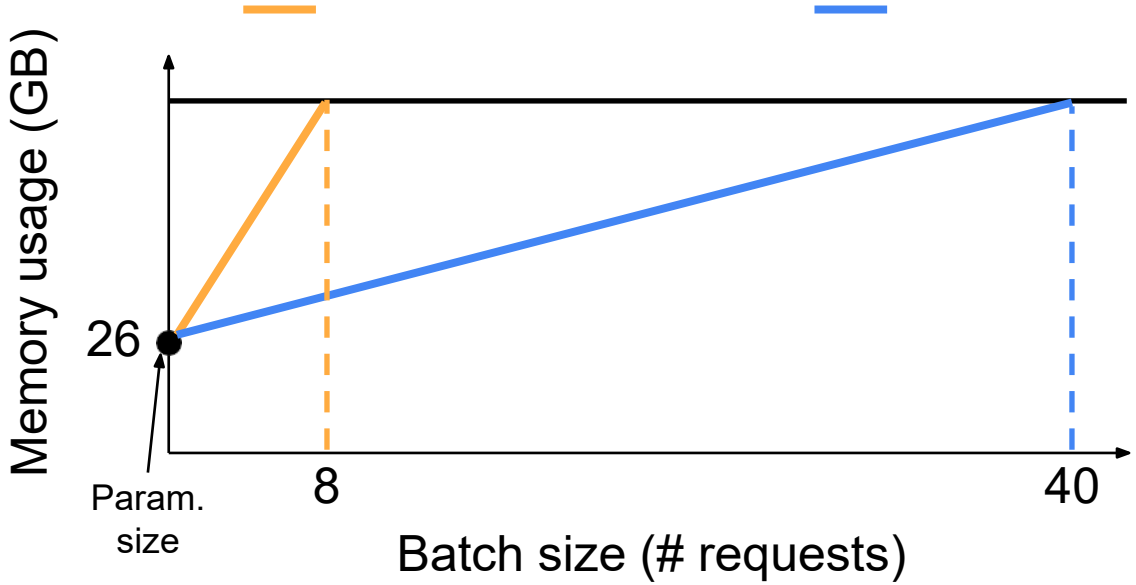
KV Cache take large memory!

- LLaMA3 70B
  - 80 Transformer layers
  - dimension: 8192
  - per token KV uses 2.5MB (stored in FP32)
  - context length 8k => 20GB per user request

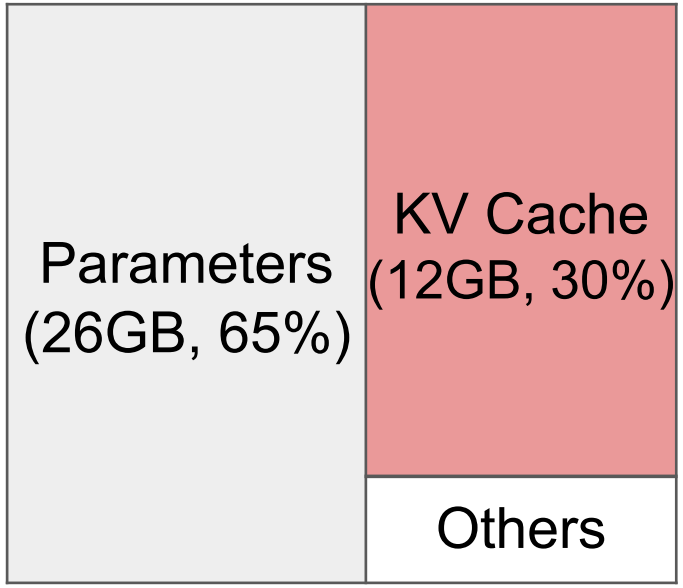
Efficient KV Cache management is crucial



13B LLM on A100-40GB

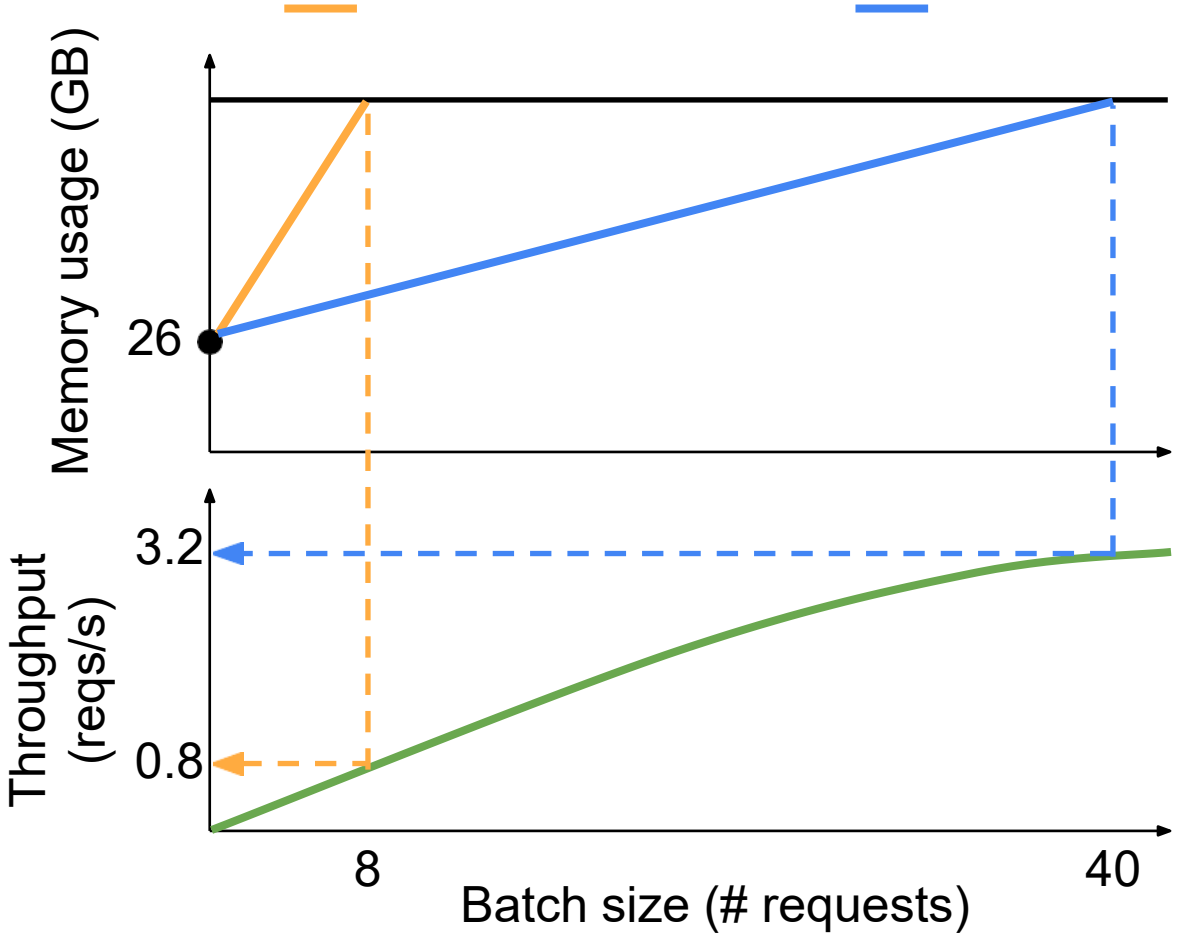


Efficient KV Cache management is crucial



13B LLM on A100-40GB*

*Common setup in 2023

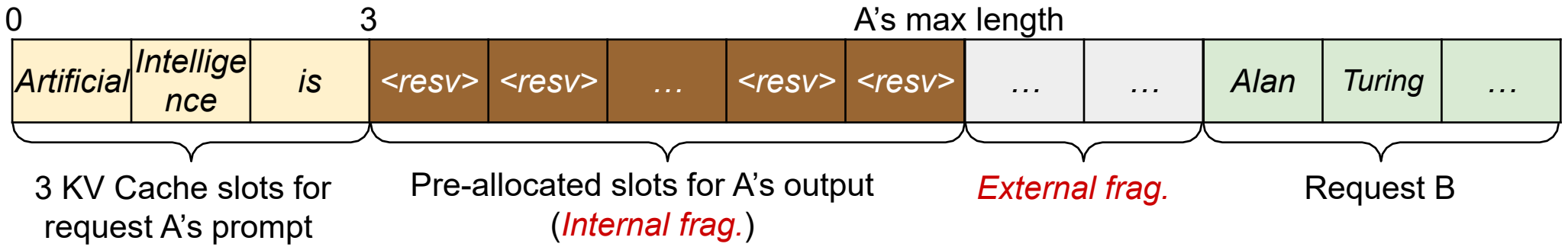


Challenge for MoE models

- Recent MoE models have high sparsity
  - DeepSeek V3: 32x (top k = 8, E = 256)
  - Kimi K2: 48x (top k = 8, E = 384)
- This means, each expert only handles 1/32 – 1/48 tokens in the batch

- to saturate GPU compute, MoE models require 32x–48x larger batch sizes than dense models
  - The demand for KV cache space gets even more significant

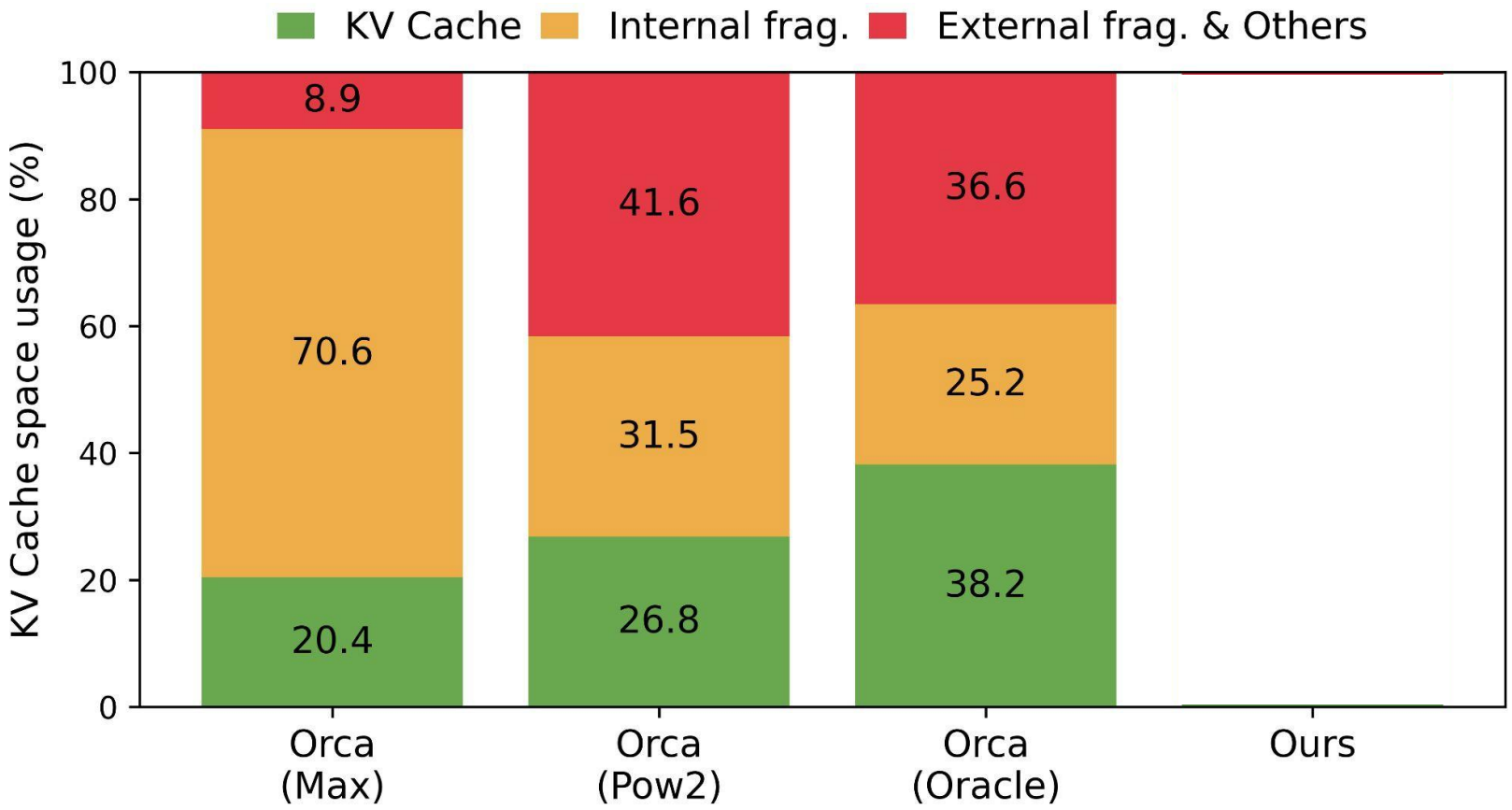
Pre-allocated Contiguous KV in Memory



- **Pre-allocates contiguous** memory to the request's maximum length
  - Convention in previous deep learning workloads with **static** input/output shapes
- Results in memory fragmentation
  - **Internal fragmentation** due to the **unknown** output length.
  - **External fragmentation** due to **non-uniform** per-request max lengths.

Significant memory waste in KV cache space

Only **20–40%** of KV Cache space is utilized to store actual token states

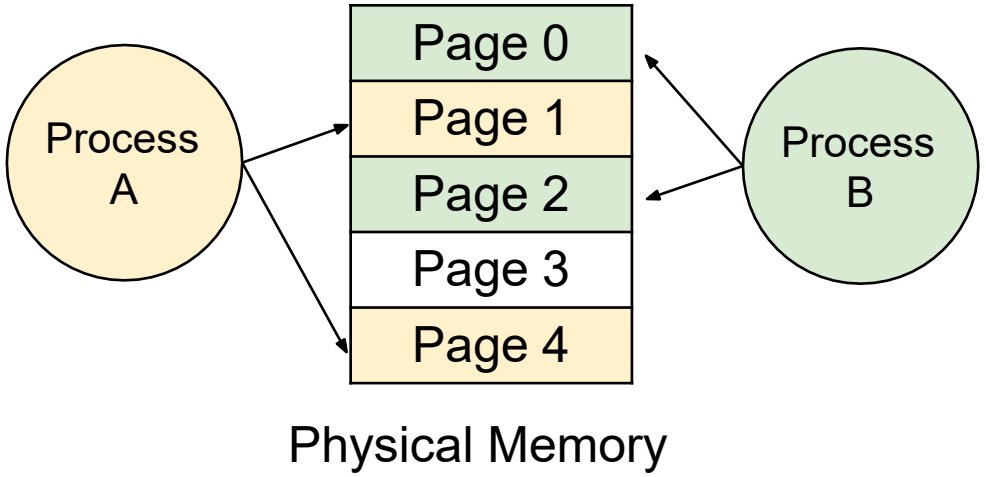


Yu et al. “Orca: A Distributed Serving System for Transformer-Based Generative Models” (OSDI 22).

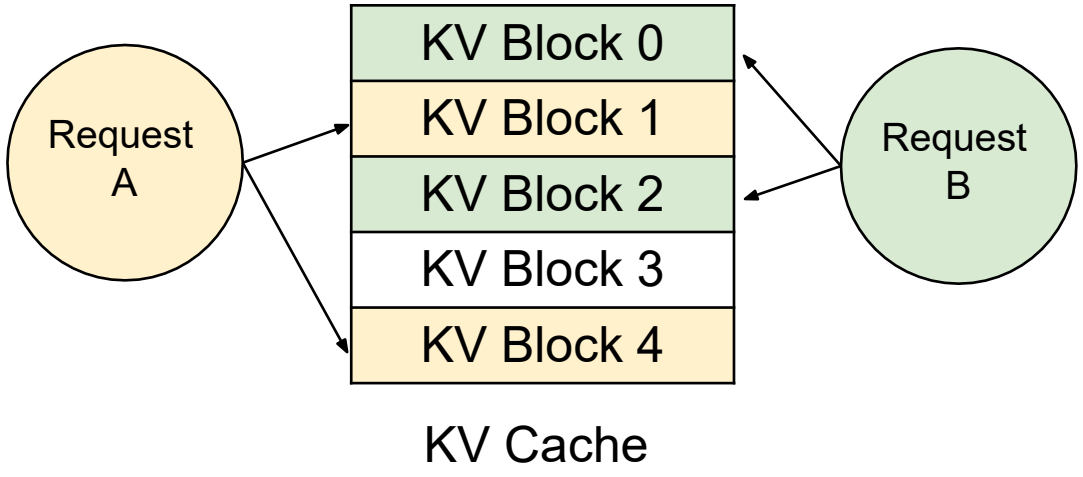
PagedAttention

Application-level memory paging and virtualization for attention KV Cache in GPU

Memory management in OS

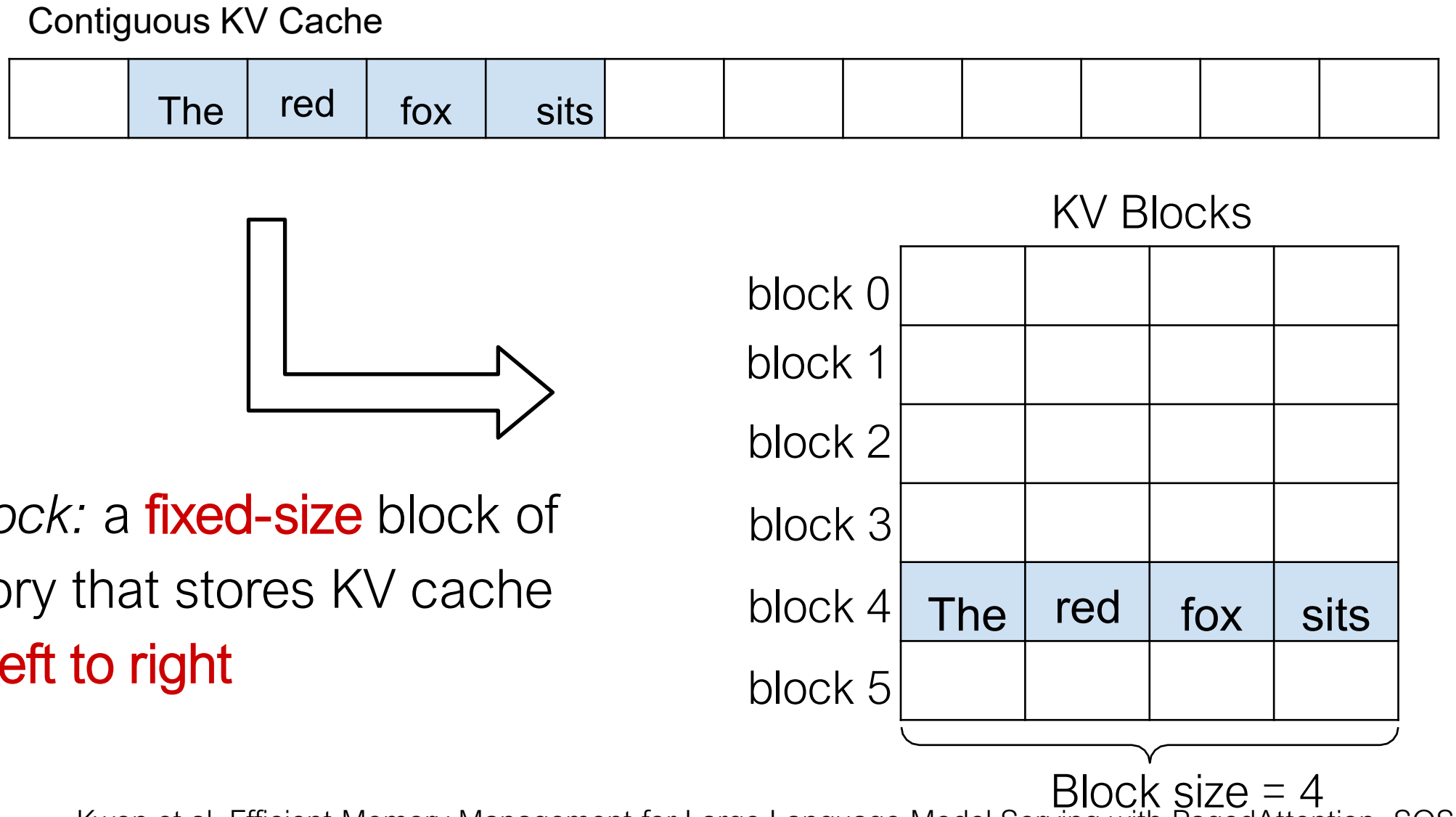


PagedAttention



Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

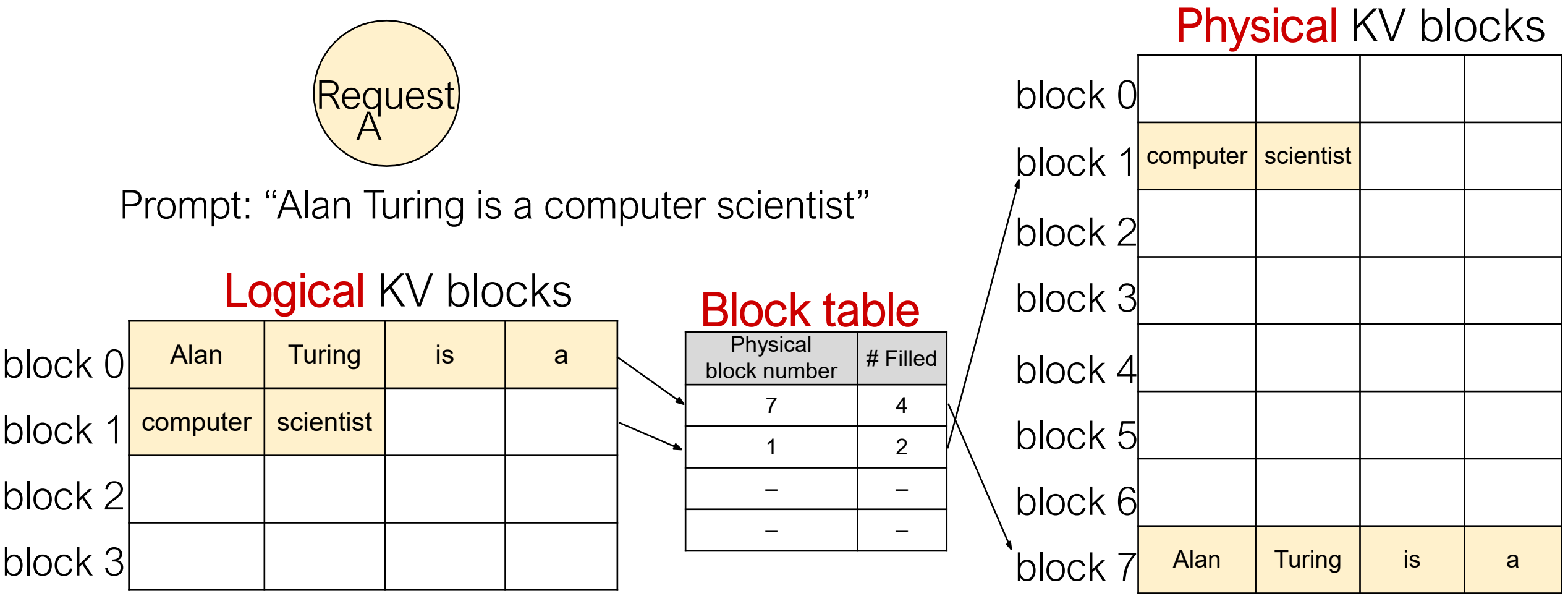
Paging KV cache space into KV blocks



KV block: a fixed-size block of memory that stores KV cache from left to right

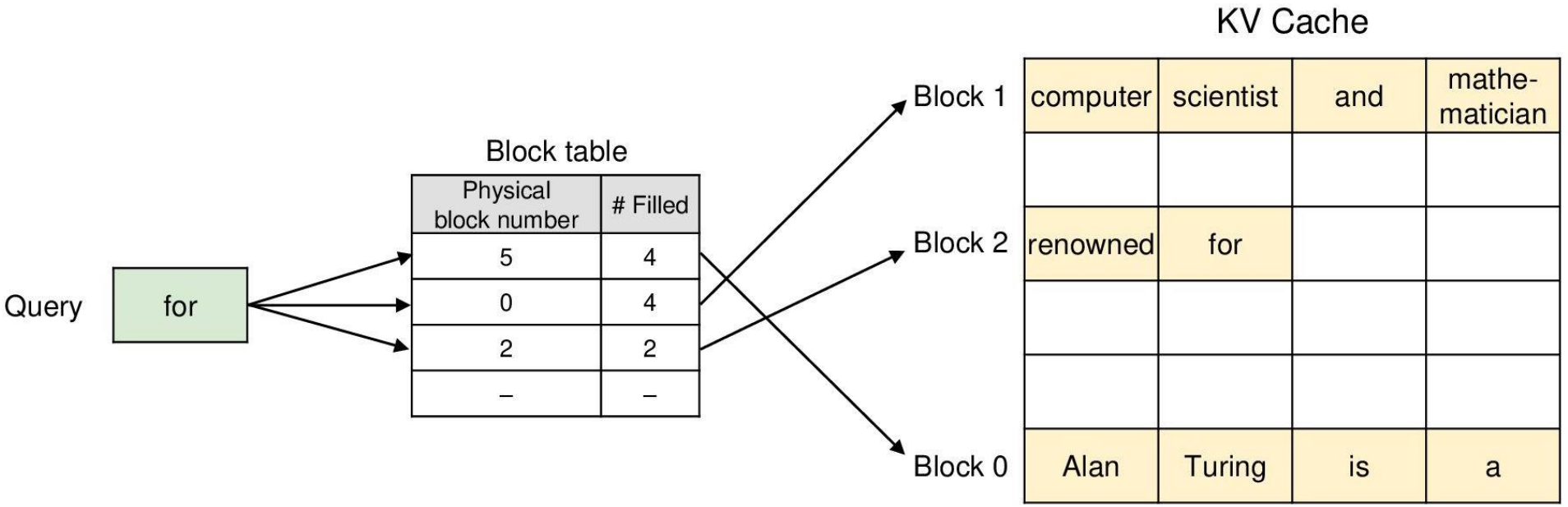
Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

Virtualizing KV Cache



Attention mechanism with virtualized KV cache

1. Fetch non-contiguous KV blocks using the block table
2. Apply attention operation on the fly



5-10% **slowdown** in GPU kernel latency due to memory indirection

Paged Attention Pseudocode

```
def normal_attn(
    query: torch.Tensor,         # [batch, 1, head_size]
    key_cache: torch.Tensor,     # [batch, max_kv_len, head_size]
    value_cache: torch.Tensor,   # [batch, max_kv_len, head_size]
    kv_len: int,
) -> torch.Tensor:               # [batch, 1, head_size]
    key = key_cache[:, :kv_len]
    value = value_cache[:, :kv_len]
    scores = torch.einsum("bqd,bkd->bqk", query, key)
    scores = scores.softmax(dim=-1)
    return scores @ value
```

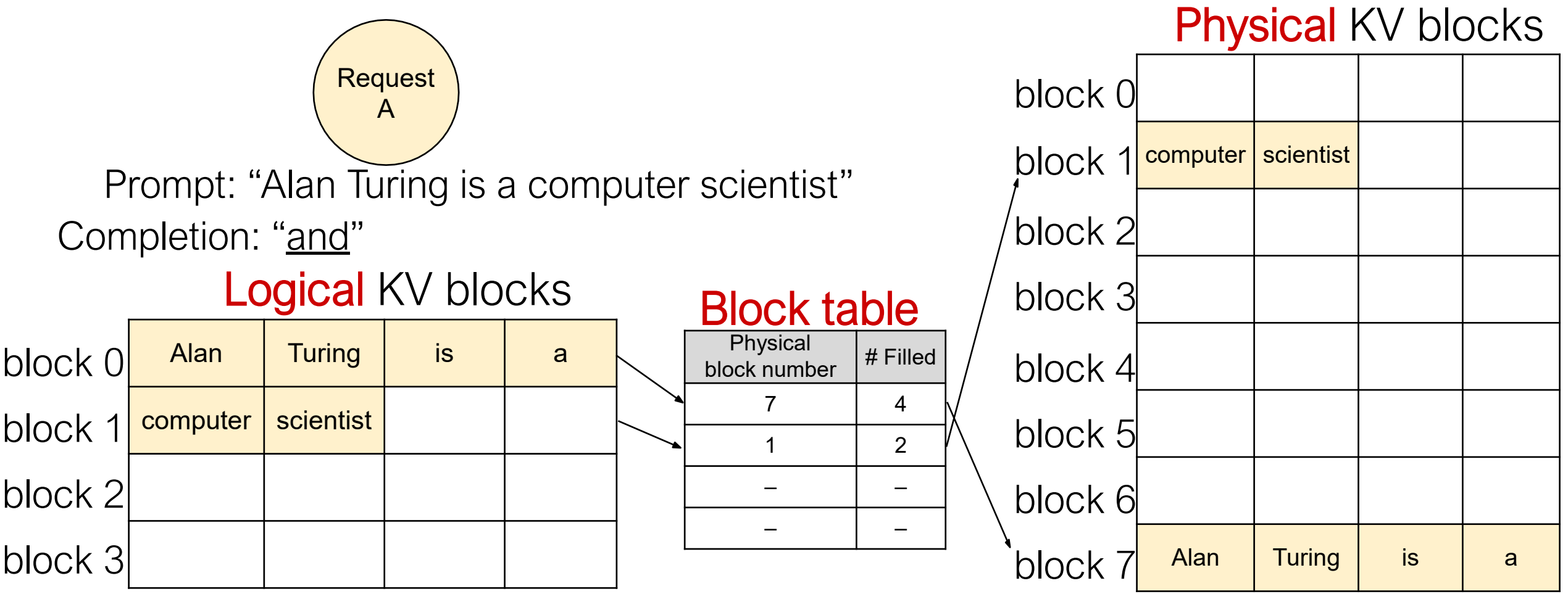
```
def paged_attn(
    query: torch.Tensor,         # [batch, 1, head_size]
    key_cache: torch.Tensor,     # [total_num_blocks, block_size, head_size]
    value_cache: torch.Tensor,   # [total_num_blocks, block_size, head_size]
    kv_len: int,
    block_table: torch.Tensor,   # [batch, num_blocks]
) -> torch.Tensor:               # [batch, 1, head_size]
    batch, _, head_size = query.shape

    key = key_cache[block_table]  # [batch, num_blocks, block_size, head_size]
    key = key.view(batch, -1, head_size)
    key = key[:, :kv_len]
    value = value_cache[block_table]  # [batch, num_blocks, block_size, head_size]
    value = value.view(batch, -1, head_size)
    value = value[:, :kv_len]

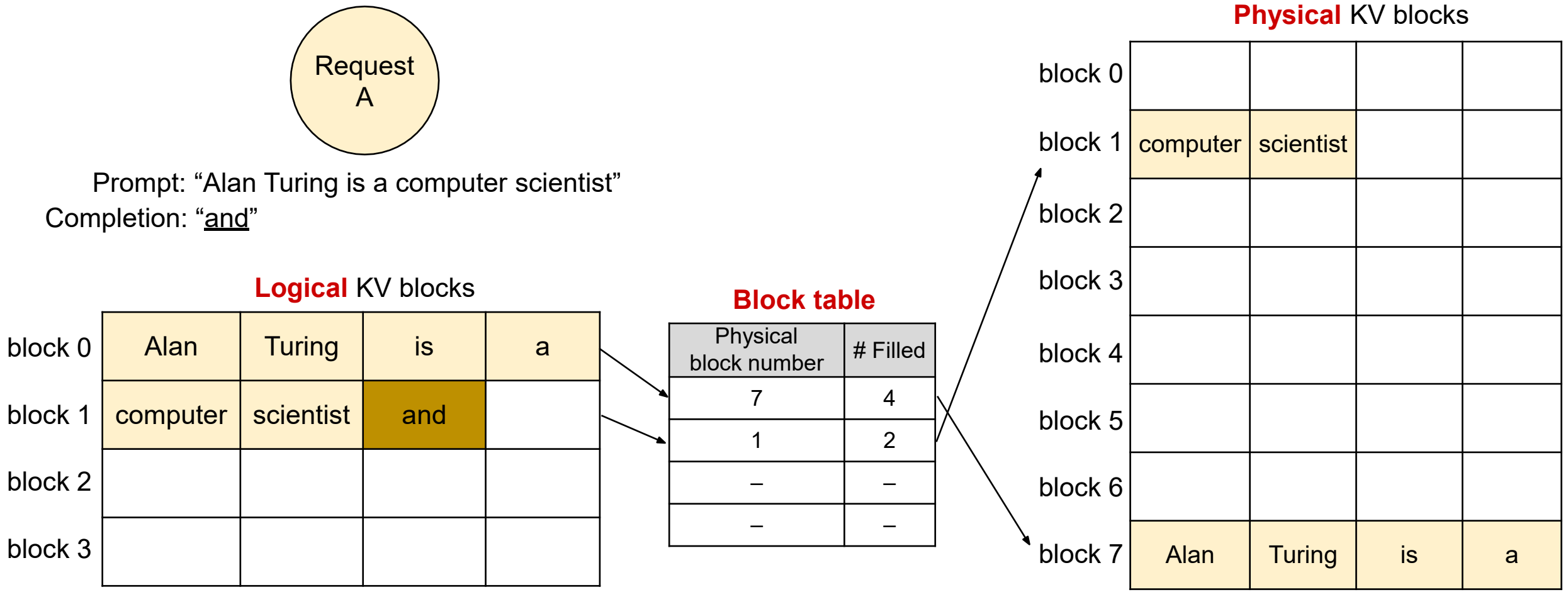
    scores = torch.einsum("bqd,bkd->bqk", query, key)
    scores = scores.softmax(dim=-1)
    return scores @ value
```

- In practice, PagedAttention is implemented as a custom GPU kernel to avoid materializing gathered keys and values
- Can be combined with FlashAttention

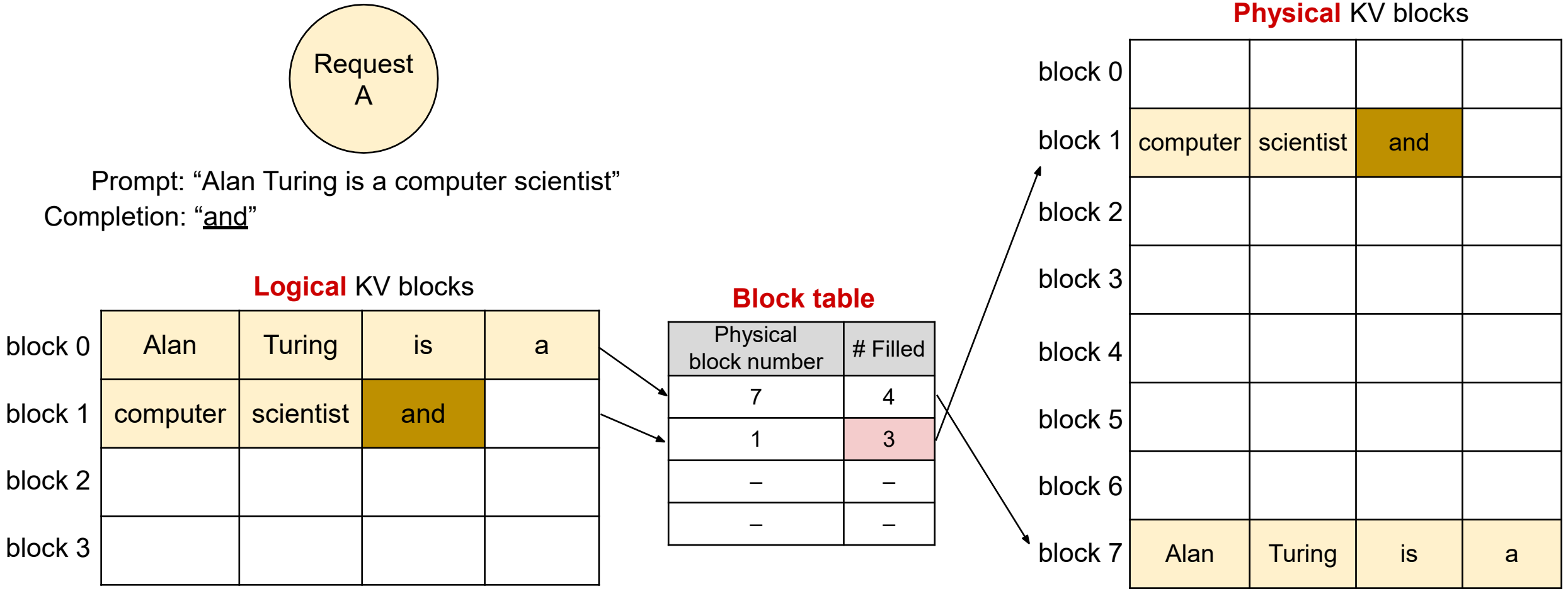
Memory management with PagedAttention



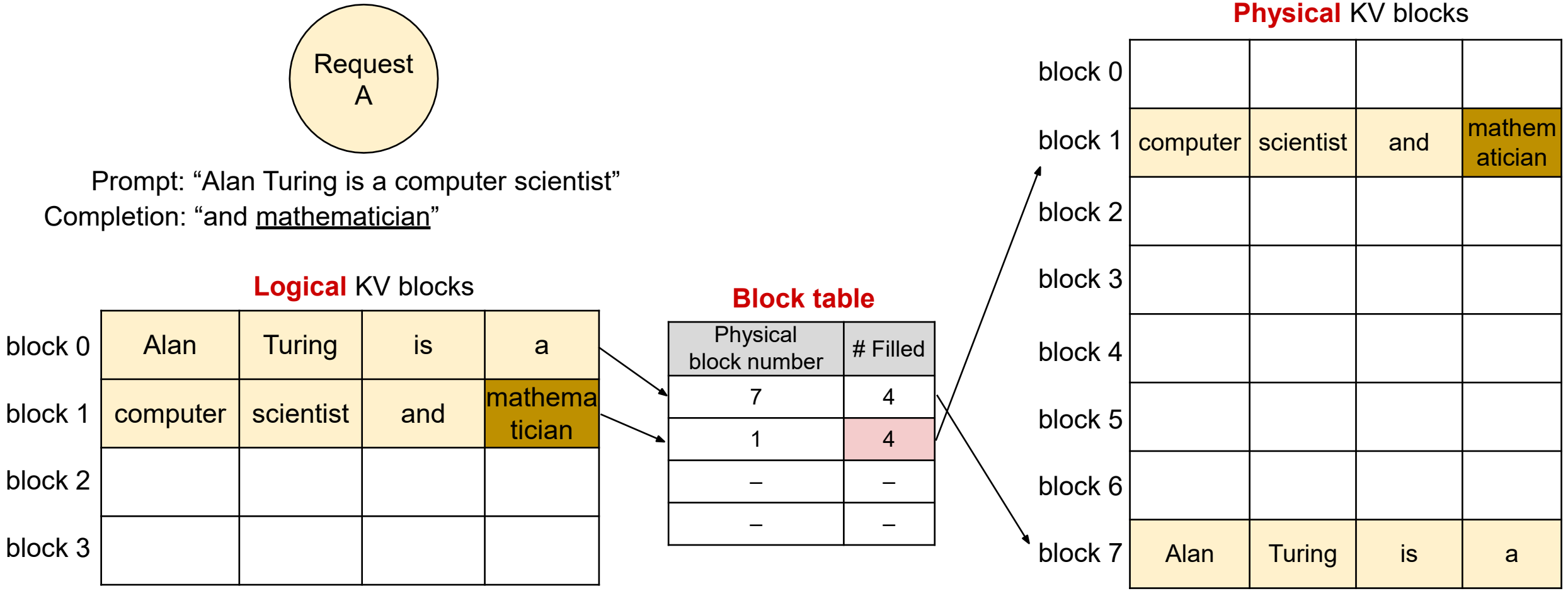
Memory management with PagedAttention



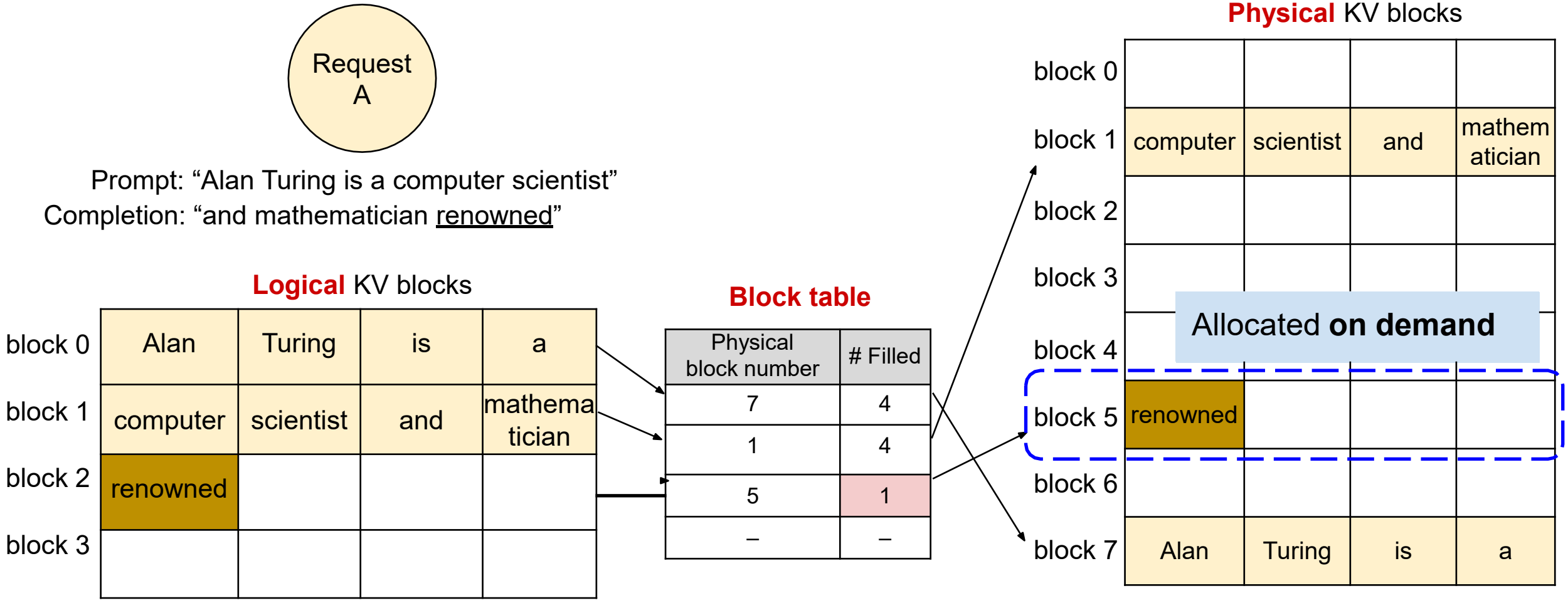
Memory management with PagedAttention



Memory management with PagedAttention



Memory management with PagedAttention

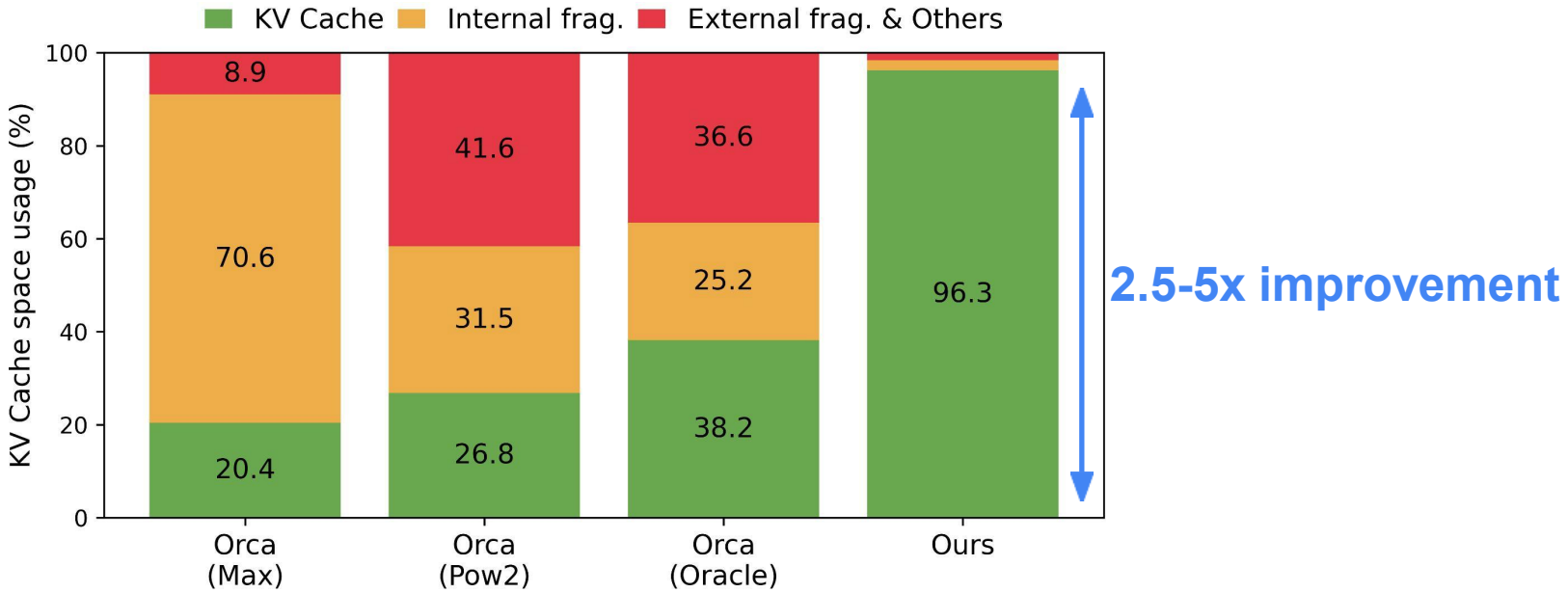


Memory efficiency of PagedAttention

- Minimal internal fragmentation
  - Only happens at the last block of a sequence
  - **# wasted tokens/seq < block size**
    - Sequence: ~1000 tokens
    - Block size: ~10 tokens
- No external fragmentation

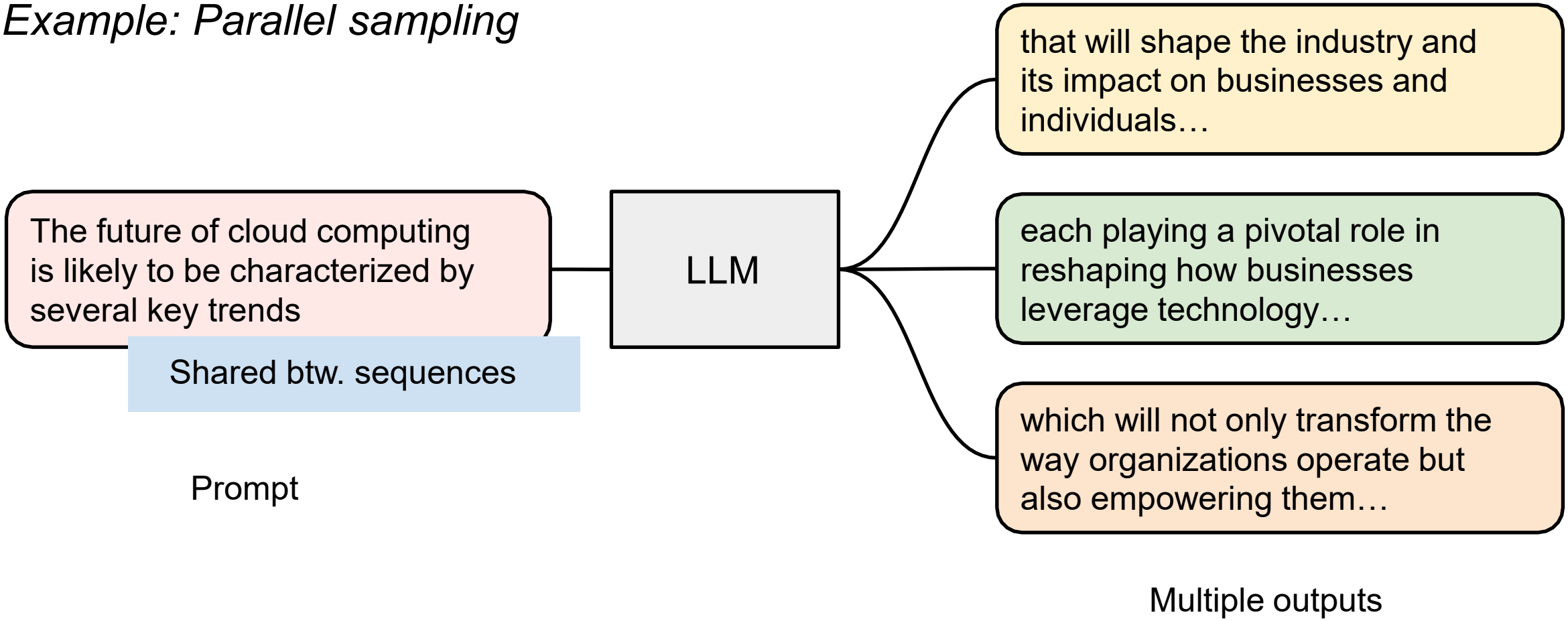
Alan	Turing	is	a
computer	scientist	and	mathematician
renowned			

Internal fragmentation



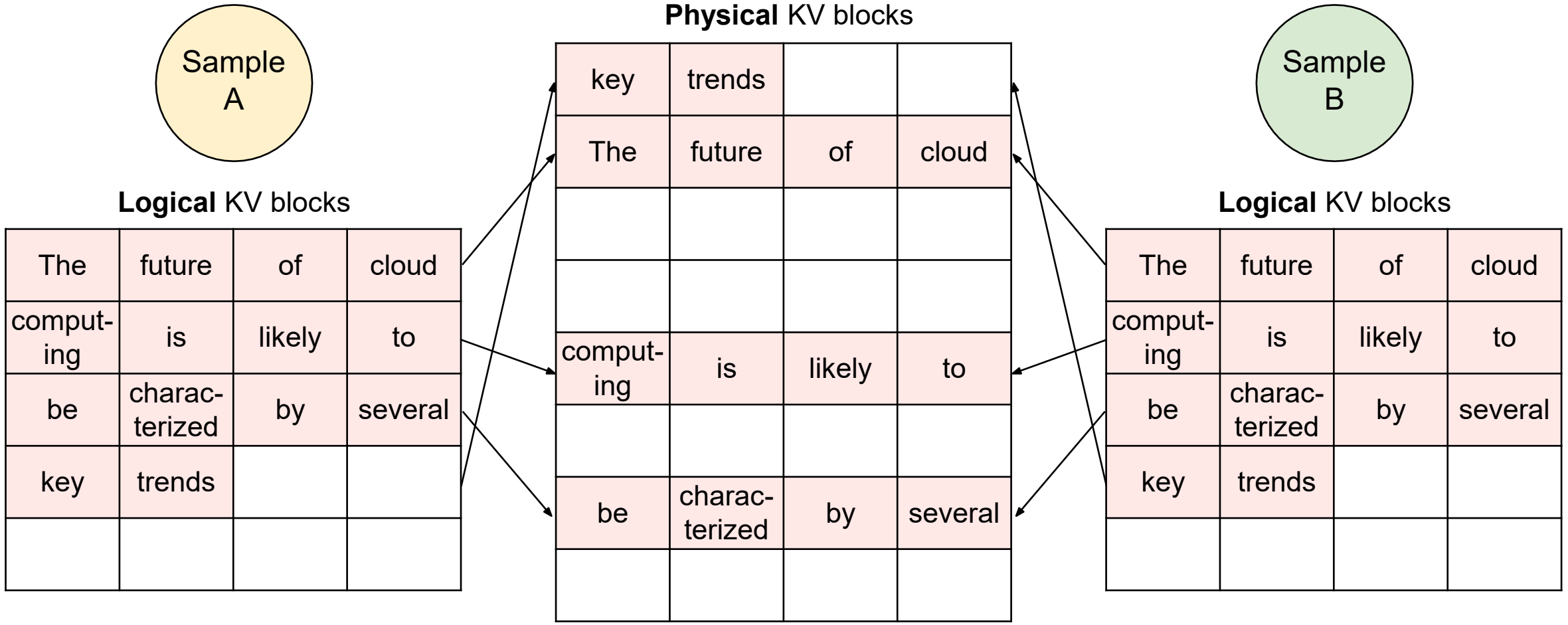
Paging enables sharing

Example: Parallel sampling

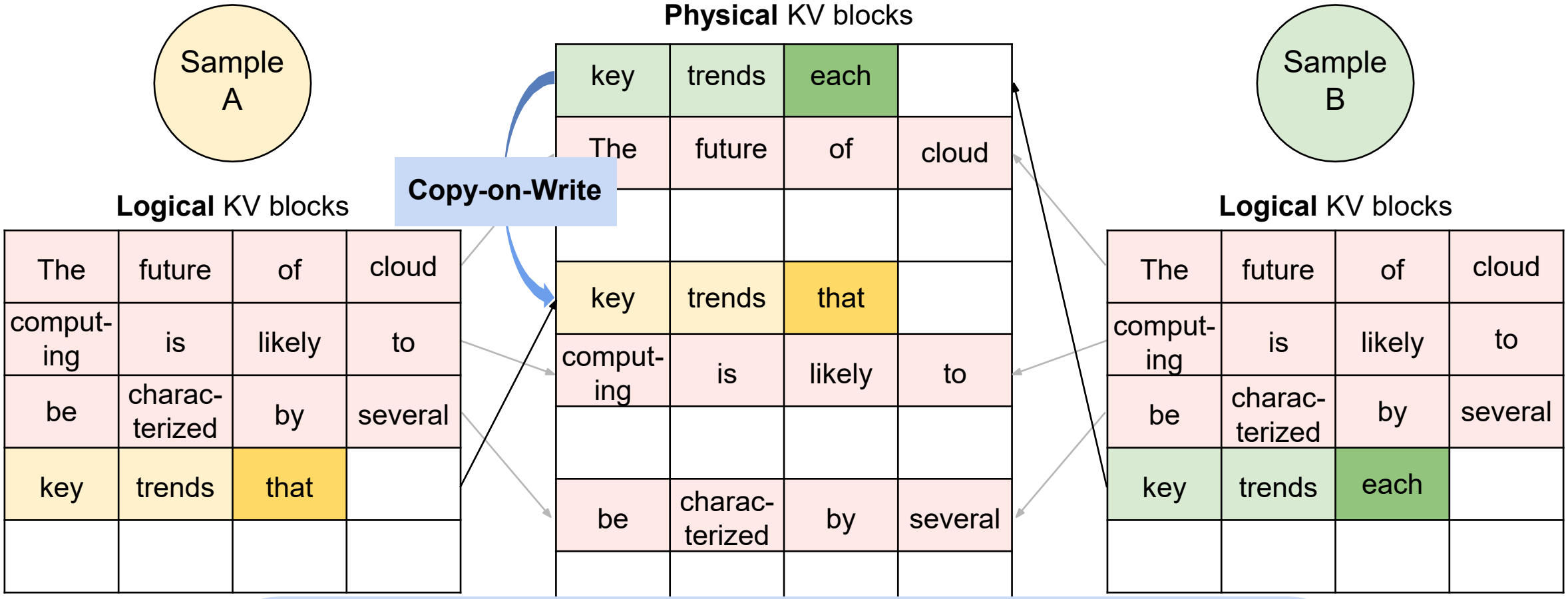


Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

Sharing KV blocks

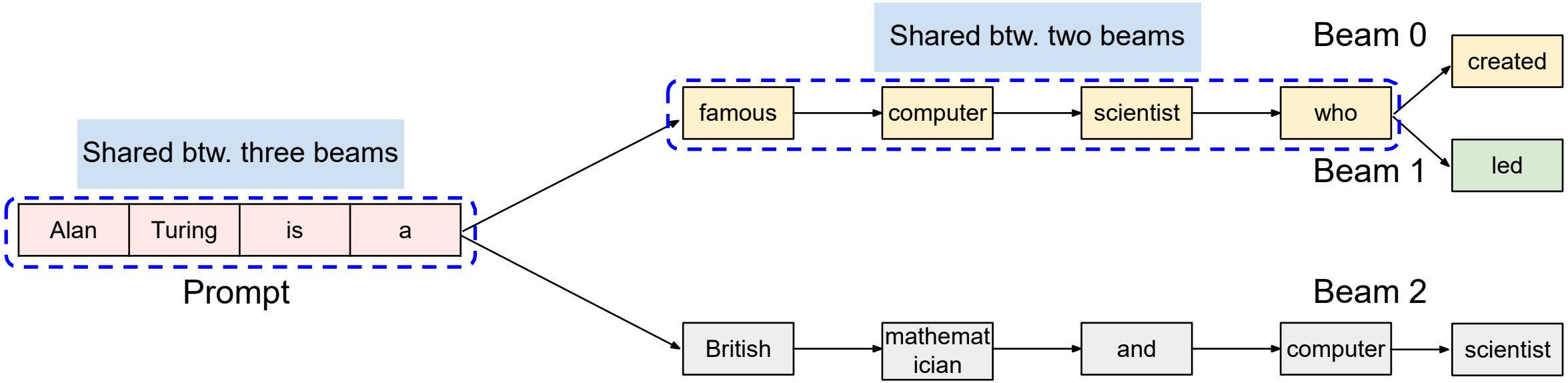


Sharing KV blocks



All prompt blocks (except the last) are *shared* across requests

More complex sharing: beam search



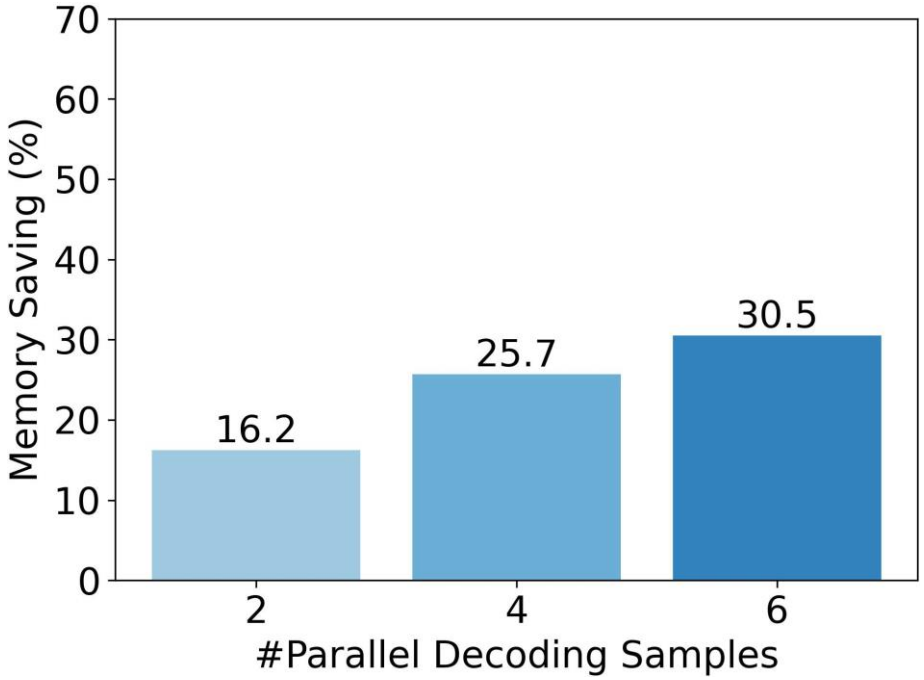
- Similar to process tree (fork & kill)
- Efficiently supported by paged attention and copy-on-write mechanism

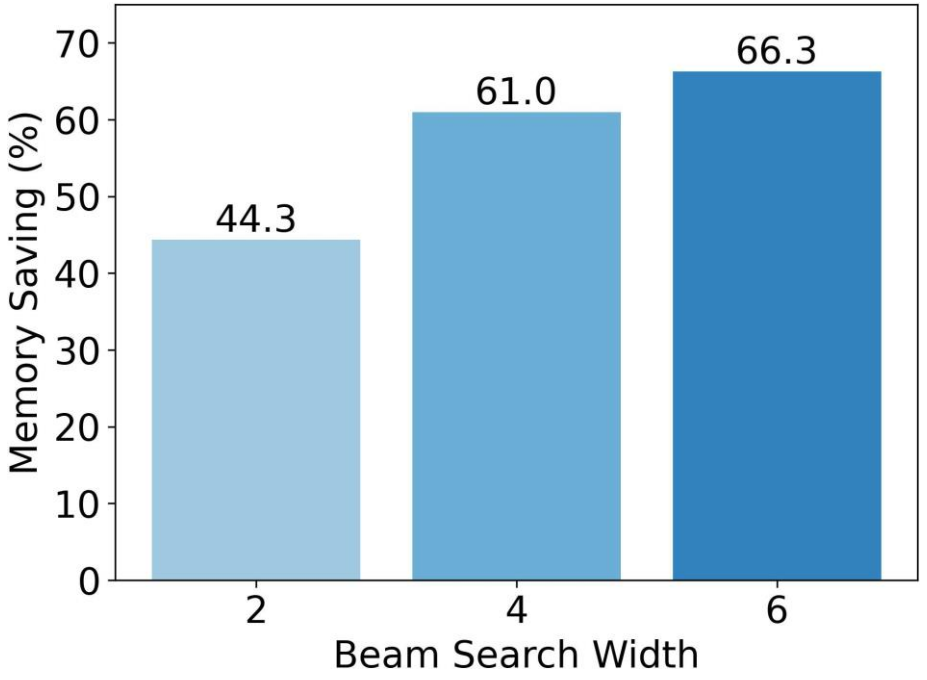
How does PagedAttention benefit LLM Serving?

Reduce memory fragmentation with paging

Further reduce memory usage with sharing

Memory saving via sharing

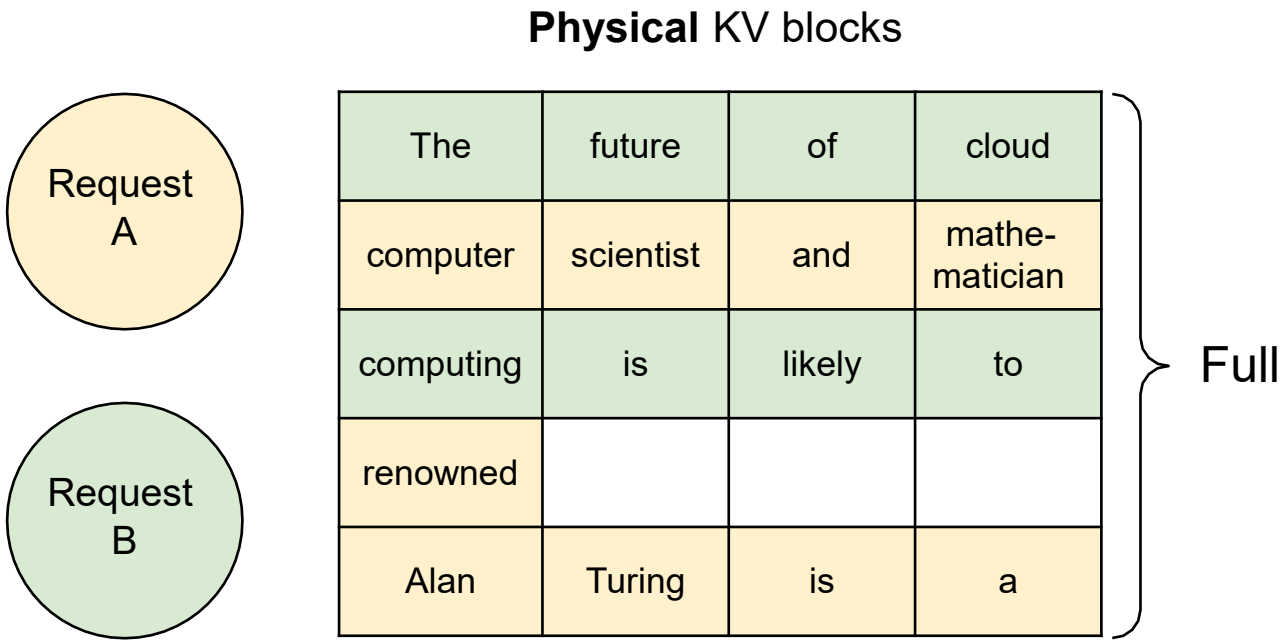




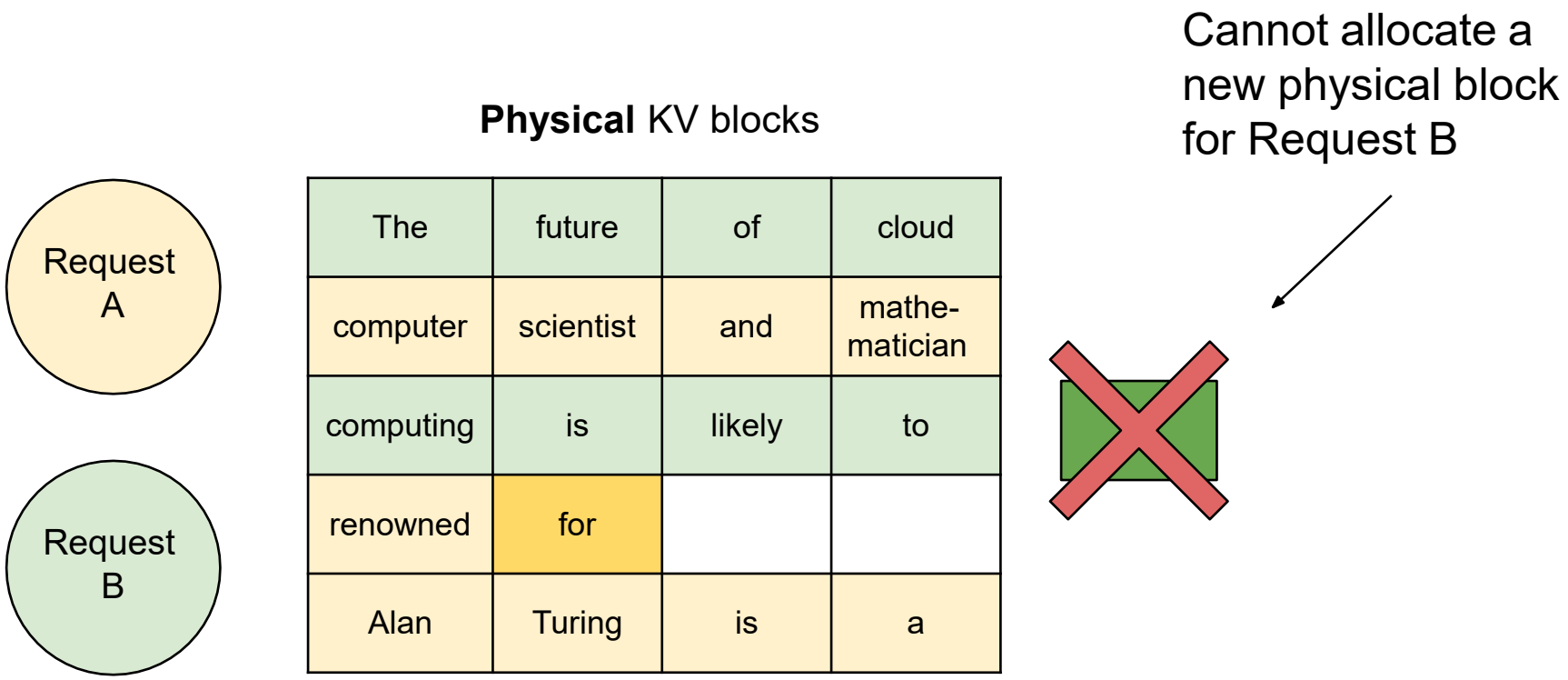
Percentage = (#blocks saved by sharing) / (#total blocks without sharing)
OPT-13B on 1x A100-40G with ShareGPT dataset

Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

Out of KV block memory



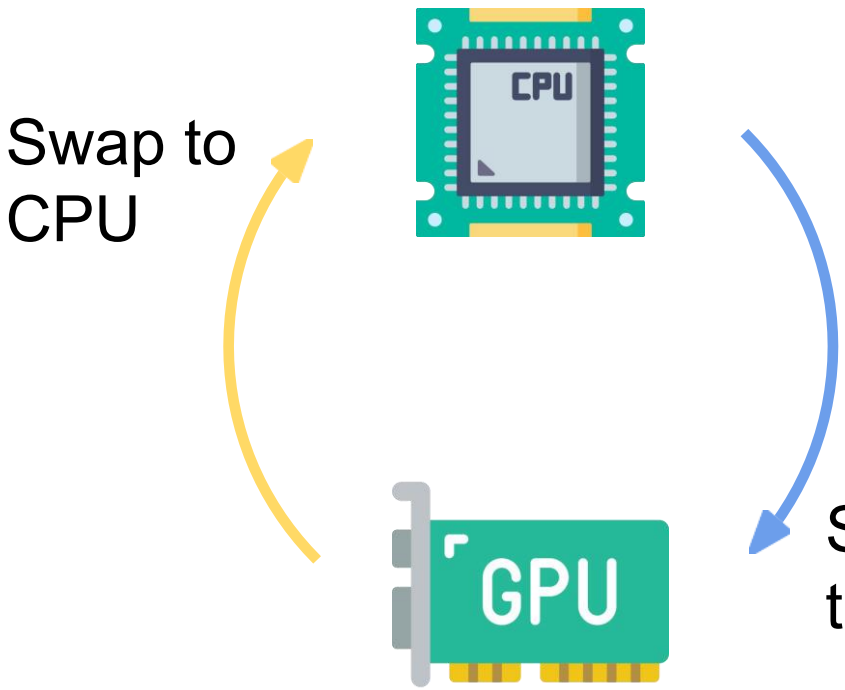
Out of KV block memory



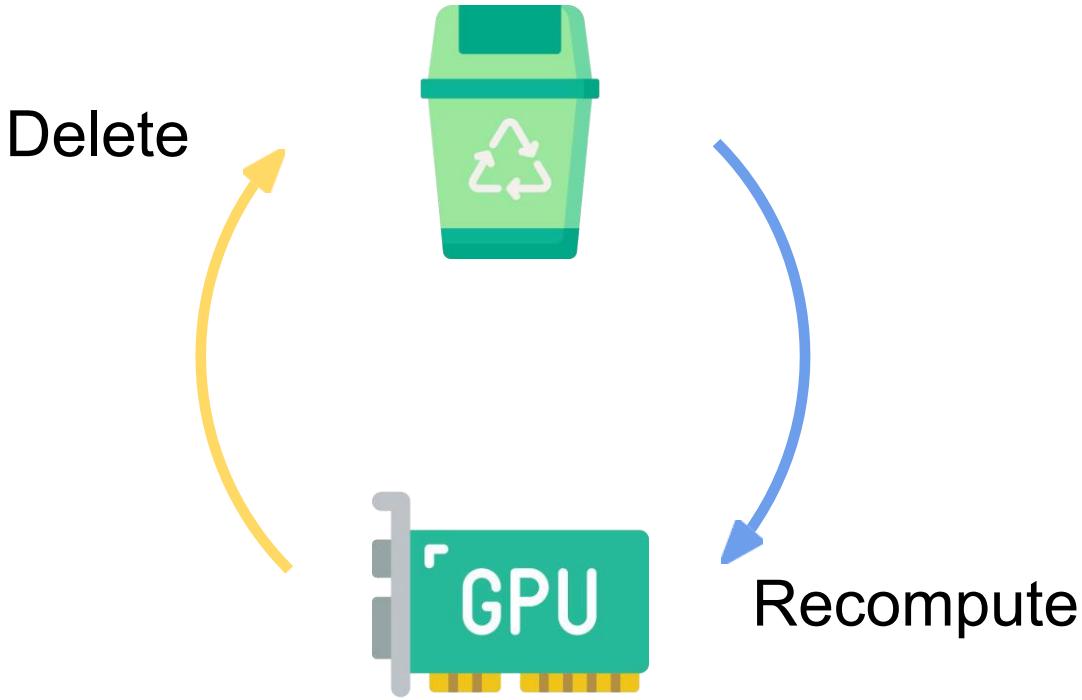
Request preemption & recovery

Goal: Free some requests' KV cache to let others run first.

Option 1: Swapping



Option 2: Recomputation



Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

Notes on preemption & recovery

Swap/recompute the whole request, since all previous tokens are required every step.

Swapping: smaller block sizes → higher overhead due to small data transfers.

Recomputation: surprisingly fast since all token's KV cache can be computed in parallel.

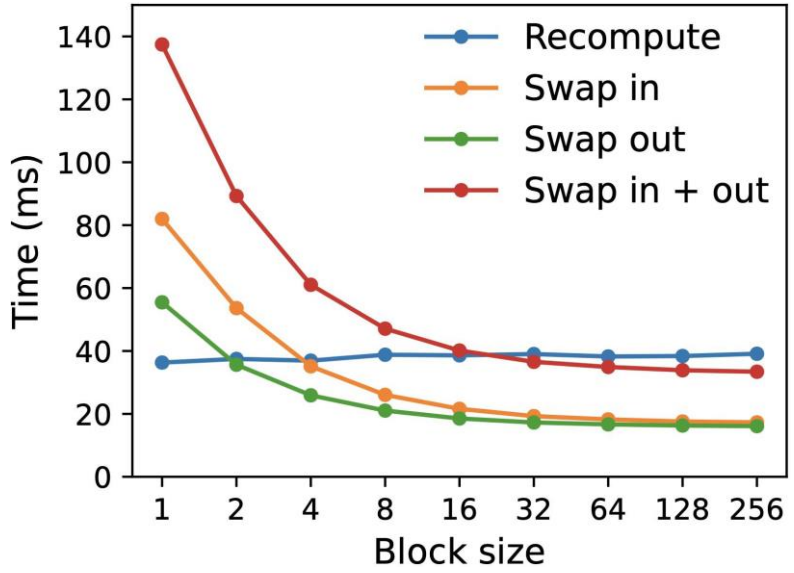


Figure: *Swap/Recomputation latency of 256 tokens.*

Our strategy: Use recomputation with FCFS policy

Comparison with operating system virtual memory

Analogies

OS pages ↔ KV blocks

- Reduce memory fragmentation

Shared pages across processes ↔ Shared KV blocks across samples

- Reduce memory waste via sharing

Differences

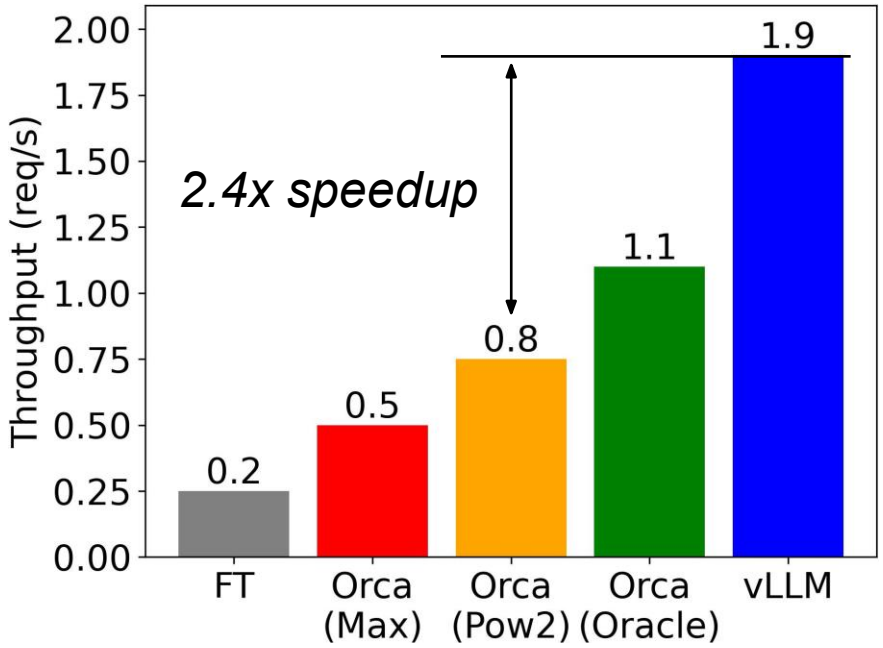
Single-level block table

- Block table is tiny compared to the actual data.

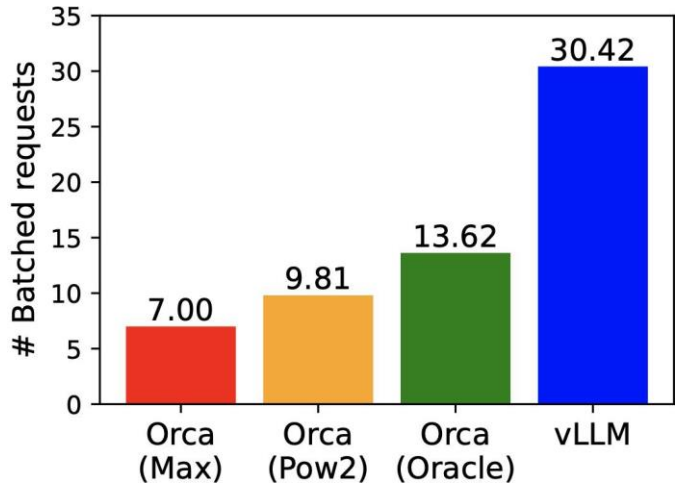
Preemption & Recovery

- Request-level preemption
- Recomputation-based recovery

PagedAttention Throughput – greedy decoding

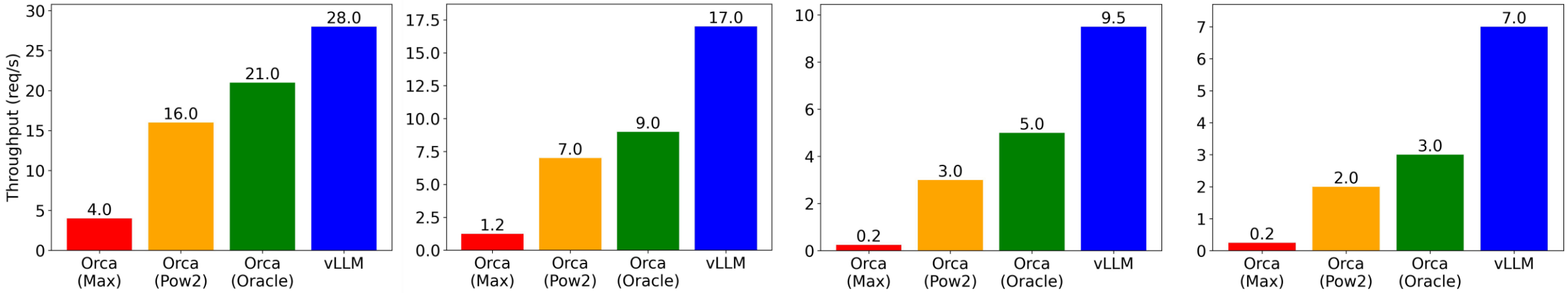


OPT-13B on 1xA100 40G with ShareGPT trace.



Average number of batched requests.

PagedAttention Throughput – beam search



No beam search
***1.8x** speedup*

Beam width = 2
***2.4x** speedup*

Beam width = 4
***3.2x** speedup*

Beam width = 6
***3.5x** speedup*

OPT-13B on 1xA100 40G with Alpaca trace.
Speedup: vLLM v.s. Orca(Pow2)

Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023.

Code Walkthrough: PagedAttention in vLLM

vLLM v0.1.7 (Sept 2023): the code behind the SOSP'23 PagedAttention paper

Concept	Source file (v0.1.7)	Code slides
KV blocks, block table, ref counts	vllm/block.py vllm/core/block_manager.py	Logical vs. physical blocks · BlockAllocator · BlockSpaceManager · append_slot & copy-on-write · fork / free
Scheduling and preemption	vllm/core/scheduler.py	Scheduler admission & preemption
Attention over the block table	tests/kernels/test_attention.py csrc/attention/attention_kernels.cu	PyTorch reference · CUDA kernel block lookup

Links point to a fixed commit (v0.1.7 = 90eb3f4). vLLM is Apache-2.0 licensed; excerpts are shortened with "...".

Code: Logical vs. Physical KV Blocks

vllm/block.py @ v0.1.7 · lines 9–63

```
class LogicalTokenBlock:
    ...
        self.token_ids = [_BLANK_TOKEN_ID] * block_size
        self.num_tokens = 0
    ...
    def is_full(self) -> bool:
        return self.num_tokens == self.block_size
    ...
class PhysicalTokenBlock:
    """Represents the state of a block in the KV cache."""
    ...
        self.device = device
        self.block_number = block_number
        self.block_size = block_size

        self.ref_count = 0
```

- LogicalTokenBlock is the sequence's view: block_size token-id slots; it holds no KV data
- PhysicalTokenBlock is just metadata: device, block_number (index into the KV cache tensors), ref_count
- ref_count = how many block tables point to the block → drives sharing, freeing, copy-on-write
- Default block_size: 16 tokens

Concept: logical vs. physical KV blocks

Code: BlockAllocator — Free List

vllm/core/block_manager.py @ v0.1.7 · lines 9–47

```python
class BlockAllocator:
    ...
    def allocate(self) -> PhysicalTokenBlock:
        if not self.free_blocks:
            raise ValueError("Out of memory! No free blocks are available.")
        block = self.free_blocks.pop()
        block.ref_count = 1
        return block

    def free(self, block: PhysicalTokenBlock) -> None:
        if block.ref_count == 0:
            raise ValueError(f"Double free! {block} is already freed.")
        block.ref_count -= 1
        if block.ref_count == 0:
            self.free_blocks.append(block)
```

- All physical blocks are created up front; free_blocks is a plain list
- allocate() pops a block and sets ref_count = 1; an empty list means out of memory
- free() only decrements ref_count; the block goes back to the list at 0
- The scheduler checks can_allocate() first, so the error is a safety net

Concept: physical KV blocks, ref counts

Code: BlockSpaceManager — Block Tables

vllm/core/block_manager.py @ v0.1.7 · lines 57–106

```
class BlockSpaceManager:
    ...
    def can_allocate(self, seq_group: SequenceGroup) -> bool:
        ...
        num_required_blocks = len(seq.logical_token_blocks)
        num_free_gpu_blocks = self.gpu_allocator.get_num_free_blocks()
        return (num_free_gpu_blocks - num_required_blocks >=
                self.watermark_blocks)

    def allocate(self, seq_group: SequenceGroup) -> None:
        ...
        block_table: BlockTable = []
        for _ in range(len(seq.logical_token_blocks)):
            block = self.gpu_allocator.allocate()
            block.ref_count = seq_group.num_seqs()
            block_table.append(block)
        for seq in seq_group.get_seqs():
            self.block_tables[seq.seq_id] = block_table.copy()
```

- block_tables maps each sequence to its block table: one physical block per logical block
- can_allocate(): admit a prompt only if its blocks fit and ≥ 1% of GPU blocks stay free (watermark)
- allocate() takes one physical block per logical block of the prompt and appends it to the table
- Each sequence gets its own copy of the list; ref_count counts the sequences sharing a block

Concept: block table, admission

Code: append_slot and Copy-on-Write

vllm/core/block_manager.py @ v0.1.7 · lines 115–139

```
def append_slot(self, seq: Sequence) -> Optional[Tuple[int, int]]:
    ...
    block_table = self.block_tables[seq.seq_id]

    if len(block_table) < len(logical_blocks):
        block = self.gpu_allocator.allocate()
        block_table.append(block)
        return None

    last_block = block_table[-1]
    assert last_block.device == Device.GPU
    if last_block.ref_count == 1:
        return None
    else:
        new_block = self.gpu_allocator.allocate()
        block_table[-1] = new_block
        self.gpu_allocator.free(last_block)
        return last_block.block_number, new_block.block_number
```

- Called every decode step to reserve the KV slot for each sequence's newest token
- Last block full → allocate a fresh physical block; nothing to copy
- Last block shared (ref_count > 1) → copy-on-write: new block for this sequence, one reference dropped
- The returned (src, dst) pair is copied by the worker before the model runs

Concept: copy-on-write

Code: fork() and free() — Sharing

vllm/core/block_manager.py @ v0.1.7 · lines 141–239

```
def fork(self, parent_seq: Sequence, child_seq: Sequence) -> None:
    src_block_table = self.block_tables[parent_seq.seq_id]
    self.block_tables[child_seq.seq_id] = src_block_table.copy()
    for block in src_block_table:
        block.ref_count += 1
...
def _free_block_table(self, block_table: BlockTable) -> None:
    for block in block_table:
        if block.device == Device.GPU:
            self.gpu_allocator.free(block)
        else:
            self.cpu_allocator.free(block)

def free(self, seq: Sequence) -> None:
    ...
    block_table = self.block_tables[seq.seq_id]
    self._free_block_table(block_table)
    del self.block_tables[seq.seq_id]
```

- fork() copies the parent's block table and bumps every ref_count: no new memory
- Used after sampling: parallel samples fork once after the prompt; beam search forks every step
- Sharers of a partly filled last block copy-on-write until ref_count is 1; the last one writes in place
- free() decrements; a shared block survives until its last owner frees it

Concept: sharing for parallel sampling and beam search

Code: Scheduler Admission & Preemption

vllm/core/scheduler.py @ v0.1.7 · lines 127–340

```
    if not self.swapped:
        ...
        while self.waiting:
            ...
            if not self.block_manager.can_allocate(seq_group):
                break
            ...
    while self.running:
        seq_group = self.running.pop(0)
        while not self.block_manager.can_append_slot(seq_group):
            if self.running:
                victim_seq_group = self.running.pop(-1)
                self._preempt(victim_seq_group, blocks_to_swap_out)
...
def _preempt(
    ...
        if seq_group.get_max_num_running_seqs() == 1:
            preemption_mode = PreemptionMode.RECOMPUTE
        else:
            preemption_mode = PreemptionMode.SWAP
```

- New prompts are admitted FCFS only when nothing is swapped out, until can_allocate() or the token/sequence budget fails
- pop(-1) takes the newest request off running (sorted oldest first) and preempts it until the current one can append
- One running sequence → RECOMPUTE (free blocks, requeue); several (e.g., beam search) → SWAP to CPU
- At this version a step is either all-prefill or all-decode

Concept: FCFS scheduling, preemption (recompute vs. swap)

Code: Paged Attention in PyTorch (Reference)

tests/kernels/test_attention.py @ v0.1.7 · lines 39–74

```
def ref_single_query_cached_kv_attention(
    ...
    for i in range(num_seqs):
        q = query[i].unsqueeze(0)
        block_table = block_tables[i]
        context_len = int(context_lens[i])
        ...
        for j in range(context_len):
            block_number = int(block_table[j // block_size])
            block_offset = j % block_size

            k = key_cache[block_number, :, :, block_offset, :]
            k = k.reshape(num_kv_heads, head_size)
            keys.append(k)

            v = value_cache[block_number, :, :, block_offset]
            values.append(v)
```

- The plain-PyTorch reference the CUDA kernel is tested against: one decode query per sequence
- Token j lives in physical block block_table[j // block_size], at slot j % block_size
- The gathered K/V then go through ordinary softmax attention
- The real kernel does the same lookup without materializing the gathered K/V

Concept: attention over a block table

Code: The CUDA Kernel's Block Lookup

csrc/attention/attention_kernels.cu @ v0.1.7 · lines 142–170

```
constexpr int x = 16 / sizeof(scalar_t);
...
const int* block_table = block_tables + seq_idx * max_num_blocks_per_seq;
const int context_len = context_lens[seq_idx];
const int num_blocks = (context_len + BLOCK_SIZE - 1) / BLOCK_SIZE;

// Iterate over the key blocks.
for (int block_idx = warp_idx; block_idx < num_blocks; block_idx += NUM_WARPS) {
  const int physical_block_number = block_table[block_idx];
  ...
  for (int i = 0; i < NUM_TOKENS_PER_THREAD_GROUP; i++) {
    const int physical_block_offset = (thread_group_idx + i * WARP_SIZE) % BLOCK_SIZE;
    const int token_idx = block_idx * BLOCK_SIZE + physical_block_offset;
    ...
    for (int j = 0; j < NUM_VECS_PER_THREAD; j++) {
      const scalar_t* k_ptr = k_cache + physical_block_number * kv_block_stride
                                      + kv_head_idx * kv_head_stride
                                      + physical_block_offset * x;
```

- One thread block per (head, sequence); each warp walks the sequence's logical blocks
- physical_block_number = block_table[block_idx]: the only indirection vs. a contiguous-KV kernel
- x = 16 / sizeof(scalar_t) elements, so each thread group loads 16 bytes at a time
- Blocks can sit anywhere in the preallocated KV cache pool

Concept: PagedAttention kernel

Topics

- Paged Attention: Efficient memory management for KV cache

- **vLLM: A real-world open-source inference engine**

vLLM API (1): LLM Class

A Python interface for offline batched inference

```python
from vllm import LLM

# Example prompts.
prompts = ["Hello, my name is", "The capital of France is"]
# Create an LLM with HF model name.
llm = LLM(model="Qwen/Qwen3-VL-8B-Instruct")
# Generate texts from the prompts.
outputs = llm.generate(prompts)
```

vLLM API (2): OpenAI-compatible Server

A FastAPI-based server for online serving

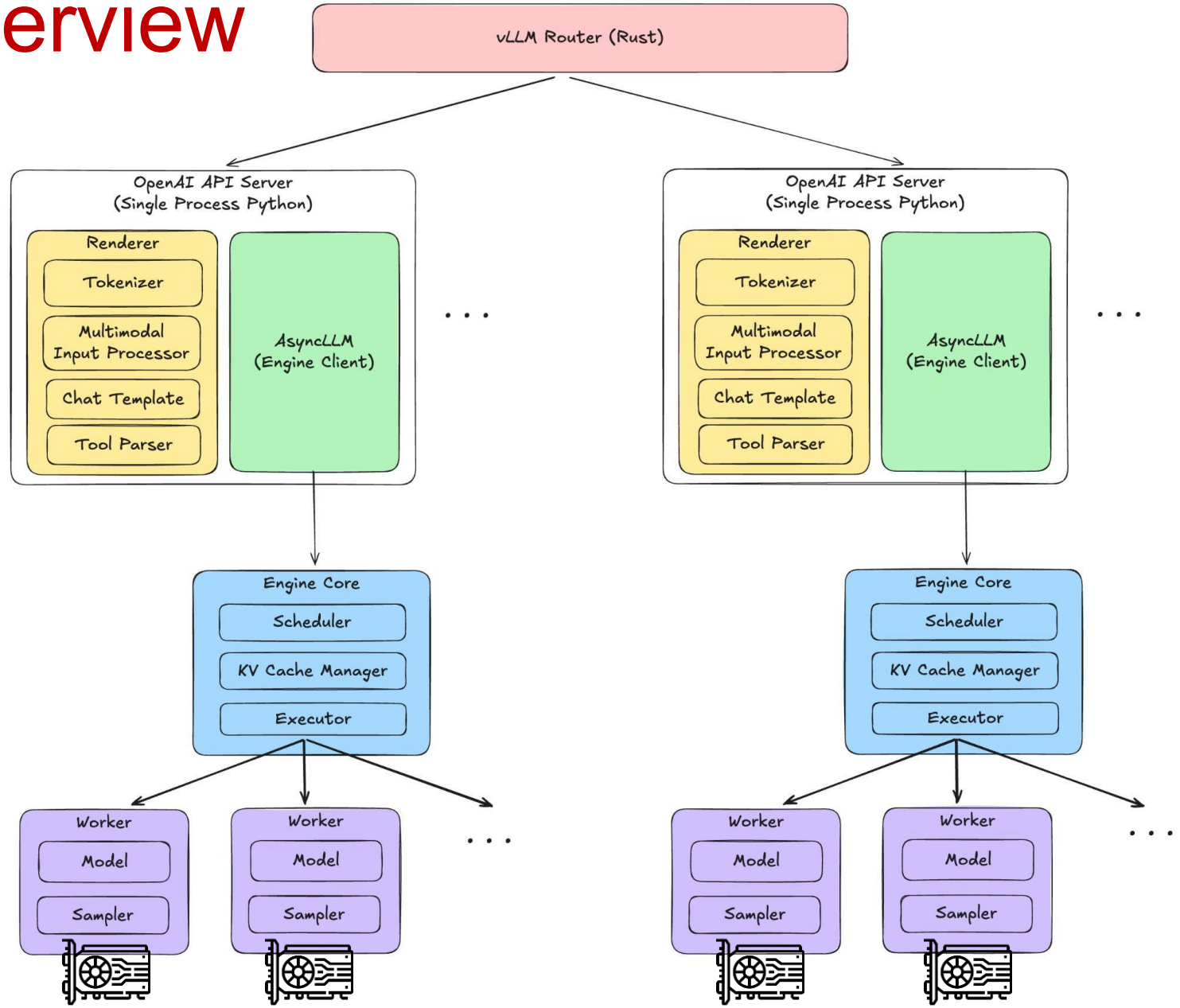
Server

```
$ vllm serve Qwen/Qwen3-VL-8B-Instruct
```

Client

```
$ curl http://localhost:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "Qwen/Qwen3-VL-8B-Instruct",
        "prompt": "San Francisco is a",
        "max_tokens": 7,
        "temperature": 0
    }'
```

vLLM Overview



How does vLLM optimize LLM inference?

1. Minimizing CPU overheads
2. Efficient GPU kernels
3. Model parallelisms
4. Efficient memory management & caching

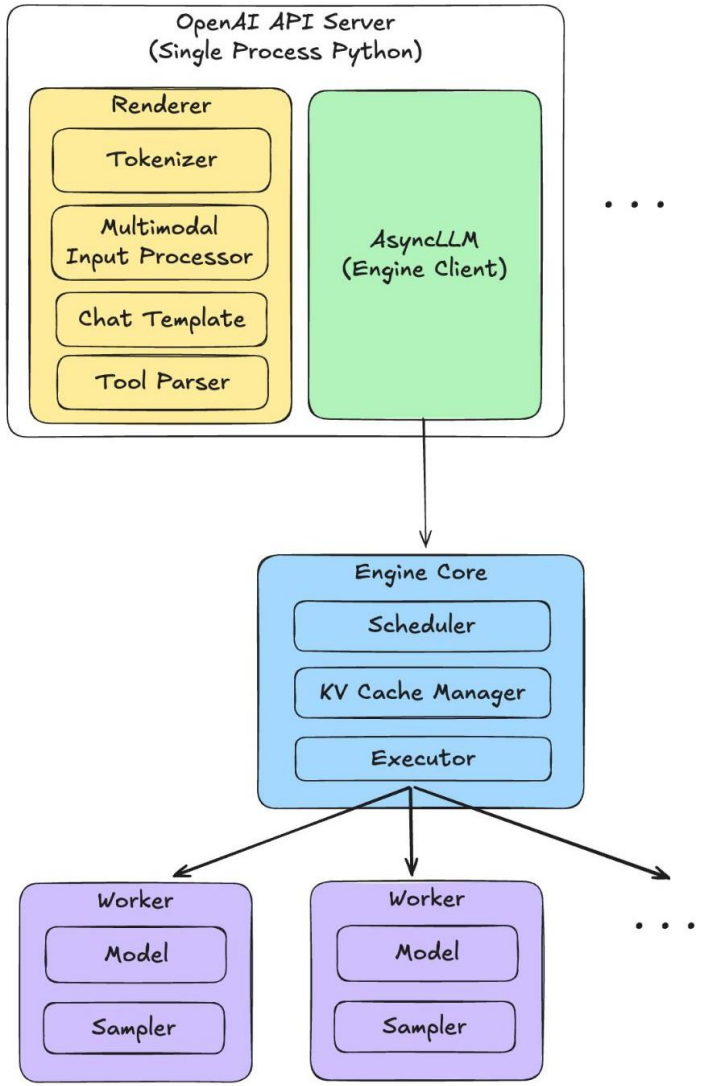
How does vLLM optimize LLM inference?

1. **Minimizing CPU overheads**
2. Efficient GPU kernels
3. Model parallelisms
4. Efficient memory management & caching

Why does CPU Overhead matter in LLM inference?

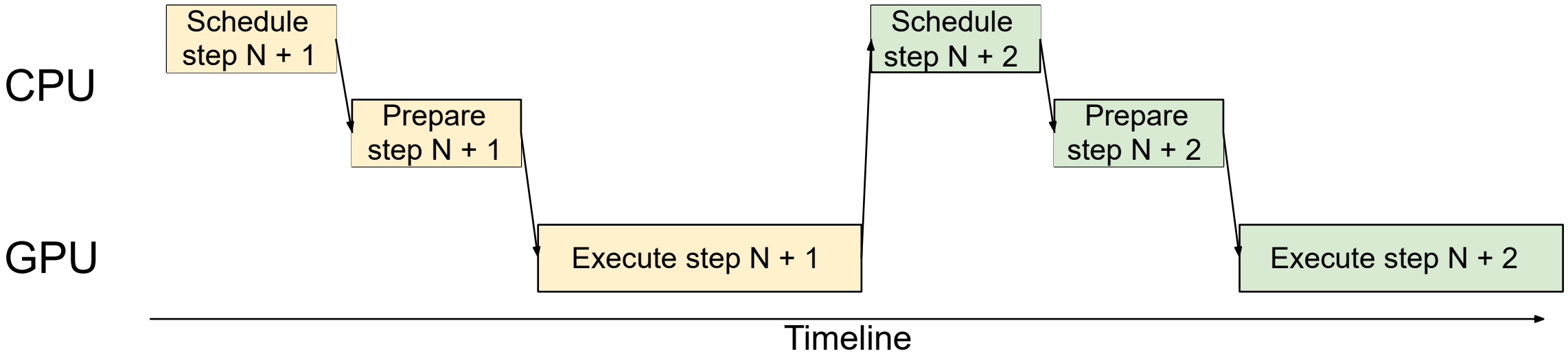
- In LLM inference, each step is very fast; just **5-10 ms** (100-200 tok/s)
  - c.f.) In training, each step is much slower; typically 100 ms - 1+ s
- This means even a small overhead matters
  - Just 1 ms extra CPU overhead can cut performance by 20%
- Unfortunately, **1 ms overhead is pretty easy to hit in Python**

Minimizing CPU Overheads



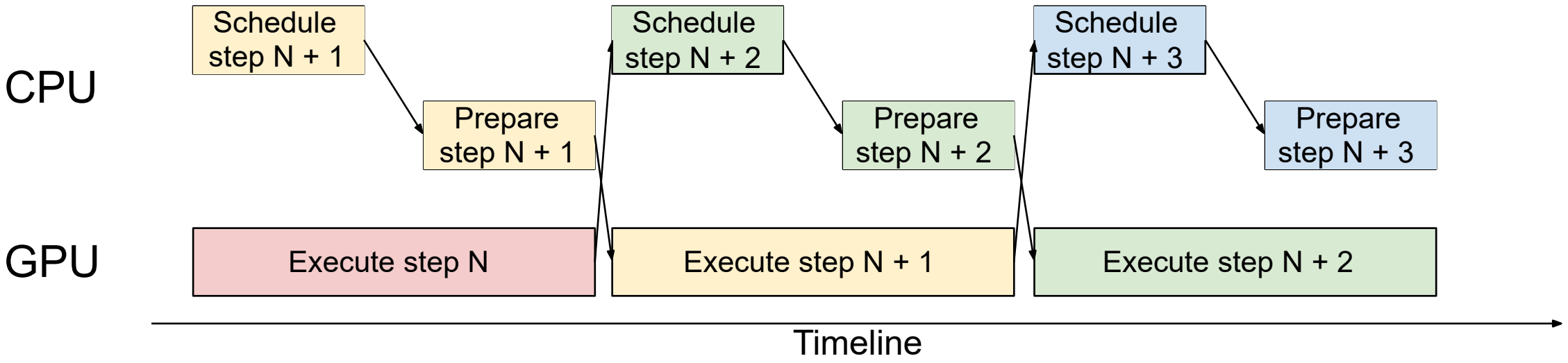
- API Server: Rewrite Python to Rust
- Scheduler & KV Cache Manager
  - **Async Scheduling**
  - Async De-tokenization
- Worker
  - **GPU-native Input Preparation**
  - **CUDA Graphs**

Synchronous Scheduling



- Scheduling & input preparation happens on the critical path, making GPU idle

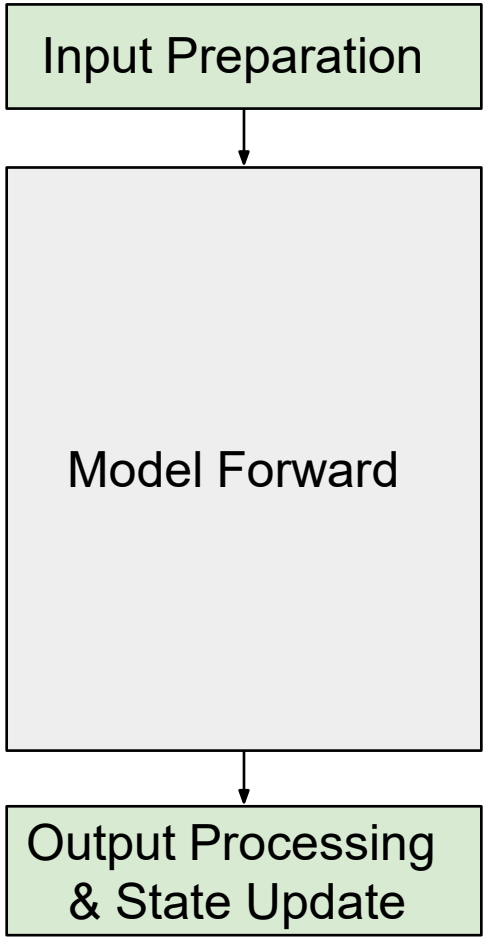
Asynchronous Scheduling



- vLLM overlaps the host overheads with model execution
  - Core idea: “schedule & prepare the next batch one step ahead”
- vLLM ensures *zero synchronization*, so the GPU is never stalled by the CPU in *any cases*

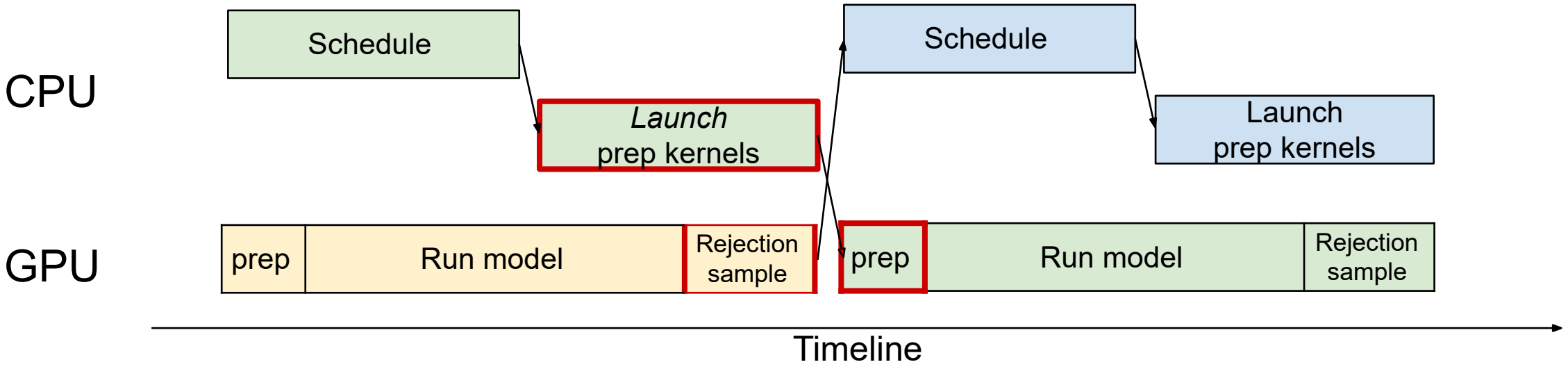
Zhu et al, “NanoFlow: Towards Optimal Large Language Model Serving Throughput“

GPU-Native Input Preparation



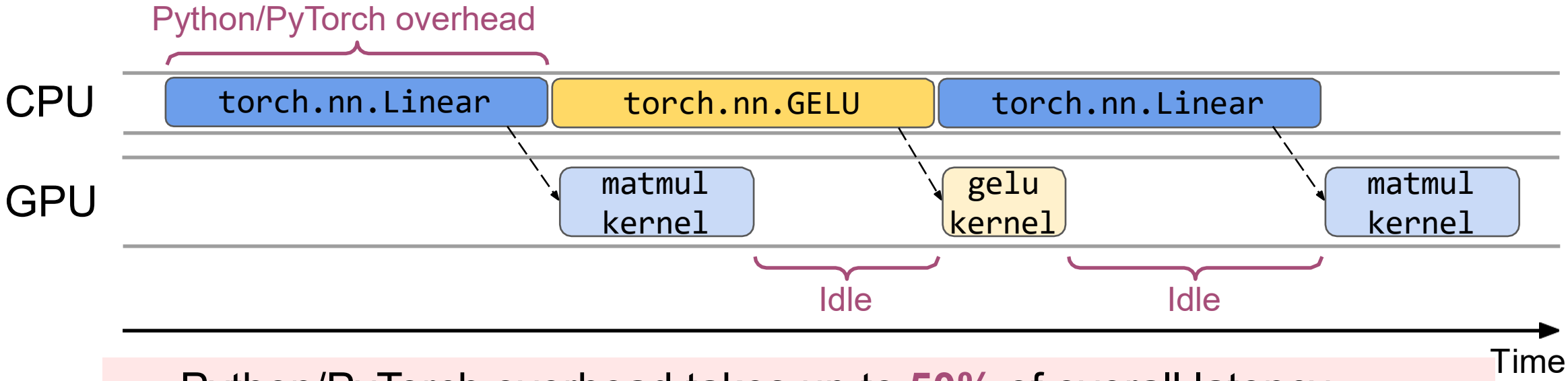
- vLLM involves complex bookkeeping for batching, paged attention, and sampling parameters
- Previously, this logic was implemented with many small PyTorch ops, mostly executed on CPU, which is slow and complex
- MRV2 replaces this with custom Triton kernels, which is fast and clean

Async Scheduling + Spec Decoding

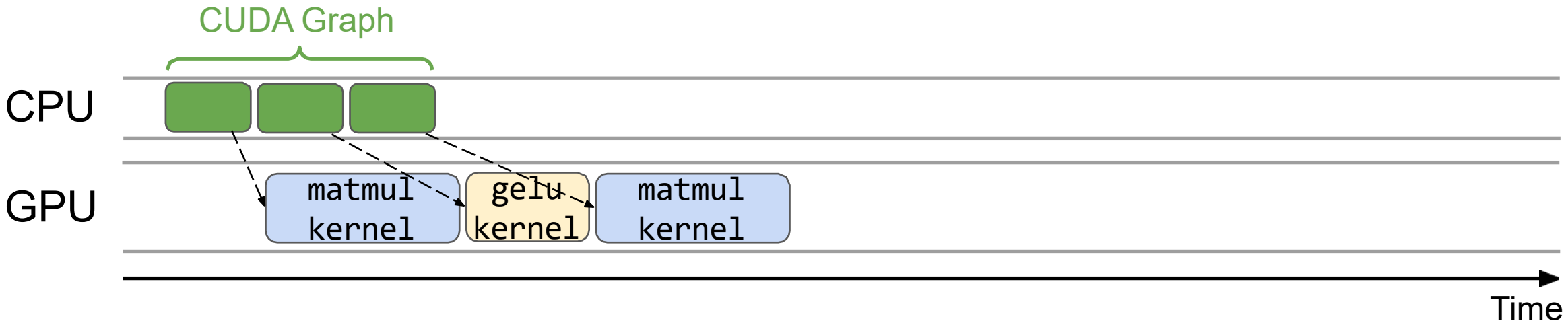


- GPU-native input preparation naturally makes async scheduling and spec decoding compatible
  - The input preparation kernels can see the results of rejection sampling

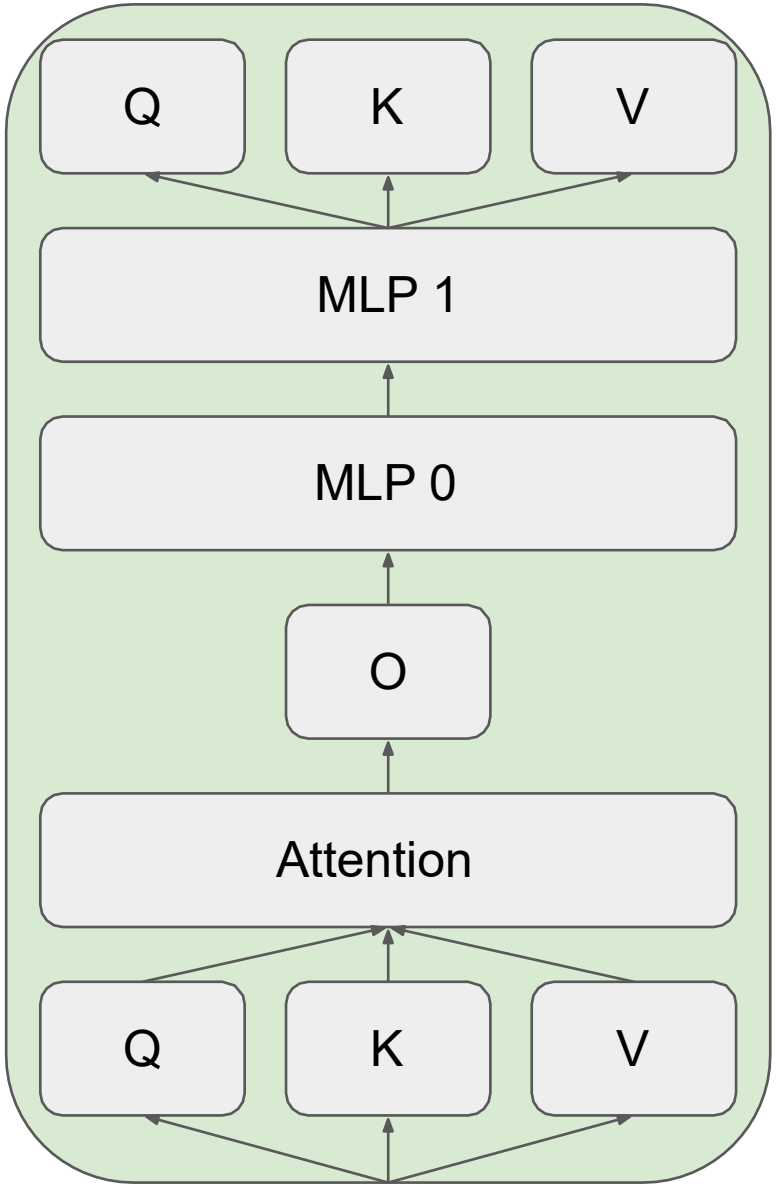
CUDA Graph for Minimizing CPU Overheads



Python/PyTorch overhead takes up to **50%** of overall latency



Full CUDA Graph



- We can capture the **entire** model into a **single** CUDA graph
- Pros: Minimal CPU overheads in model execution
- Cons: Limited flexibility
  - Static shapes are required
  - No CPU operations are allowed

→ Increased development burden

Dynamisms in LLM Inference

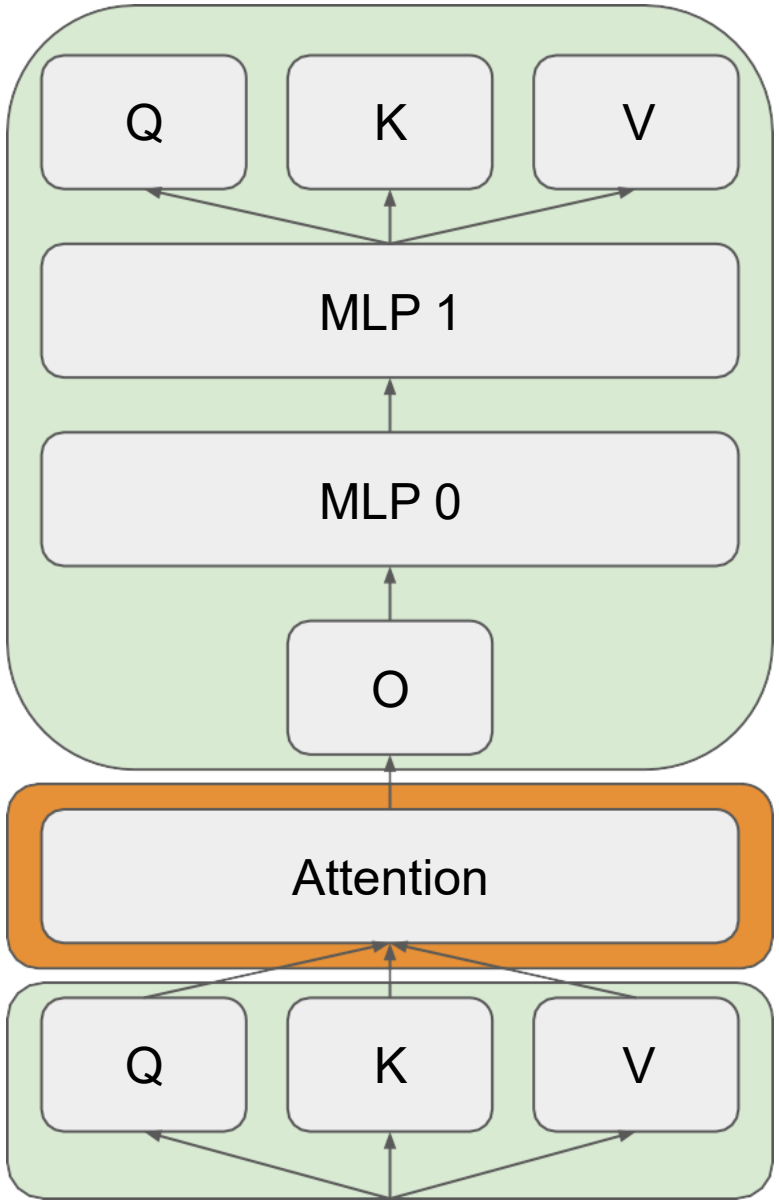
- Dynamic scheduling
  - The model needs to handle an arbitrary mix of prefills and decodes in the same batch
- Kernel heuristics
  - E.g., Cascade Attention
  - Rely on run-time heuristics that cannot be predetermined ahead of time
- CPU offloading
  - Requires CPU operations during the model execution

With full CUDA graph, our options were either

- Abandon CUDA graph and use PyTorch eager (**sacrificing performance**), or
- Develop a special GPU kernel to work around the issue (**sacrificing simplicity**)

Ye et al., “Cascade Inference: Memory Bandwidth Efficient Shared Prefix Batch Decoding”

Our solution: Piecewise CUDA Graphs

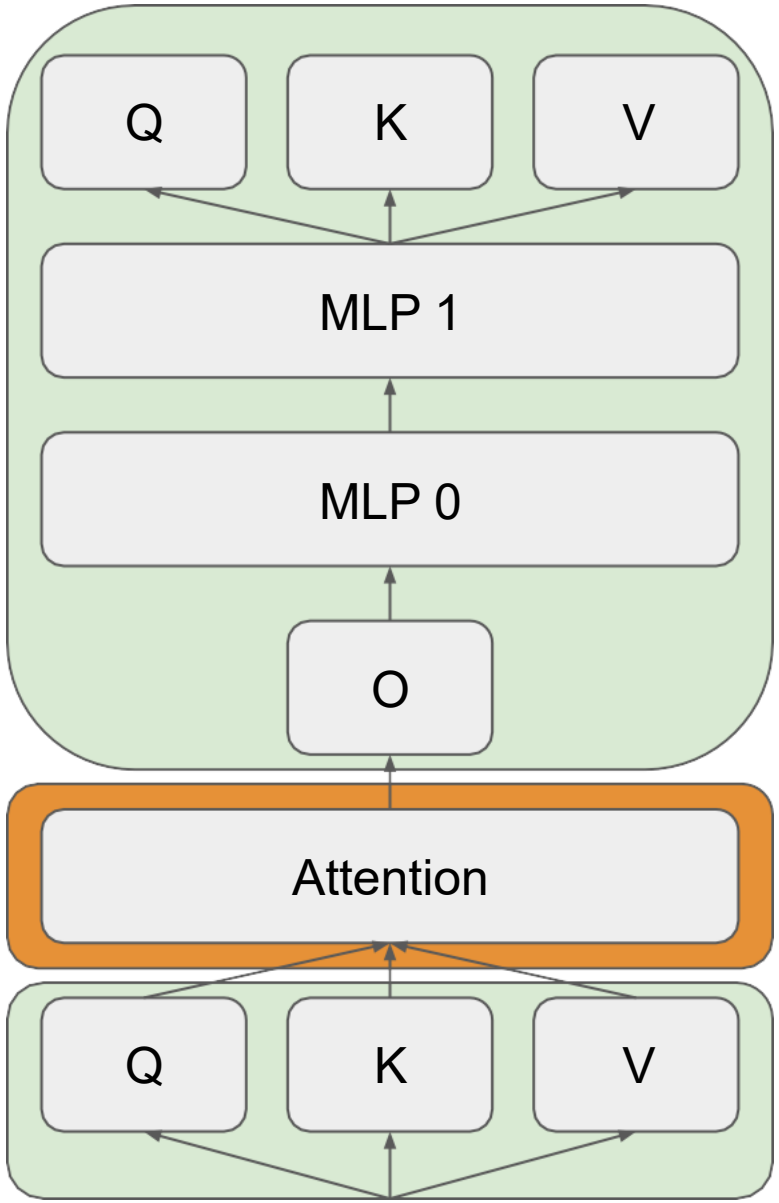


Token-wise Operations
(Stateless & Easy to deal with)

Dynamic & Stateful & Complex

Token-wise Operations
(Stateless & Easy to deal with)

Our solution: Piecewise CUDA Graphs



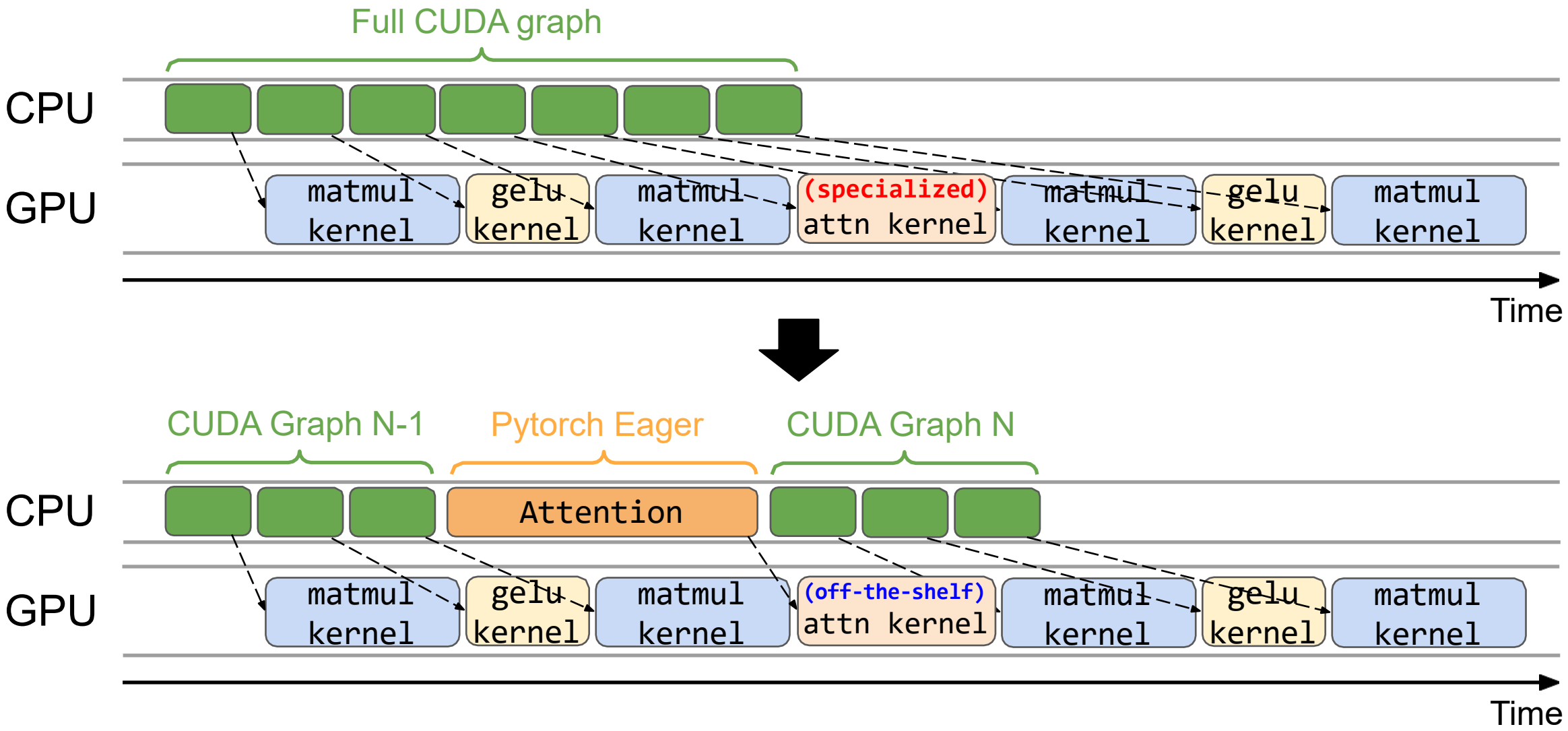
CUDA graph N

PyTorch Eager

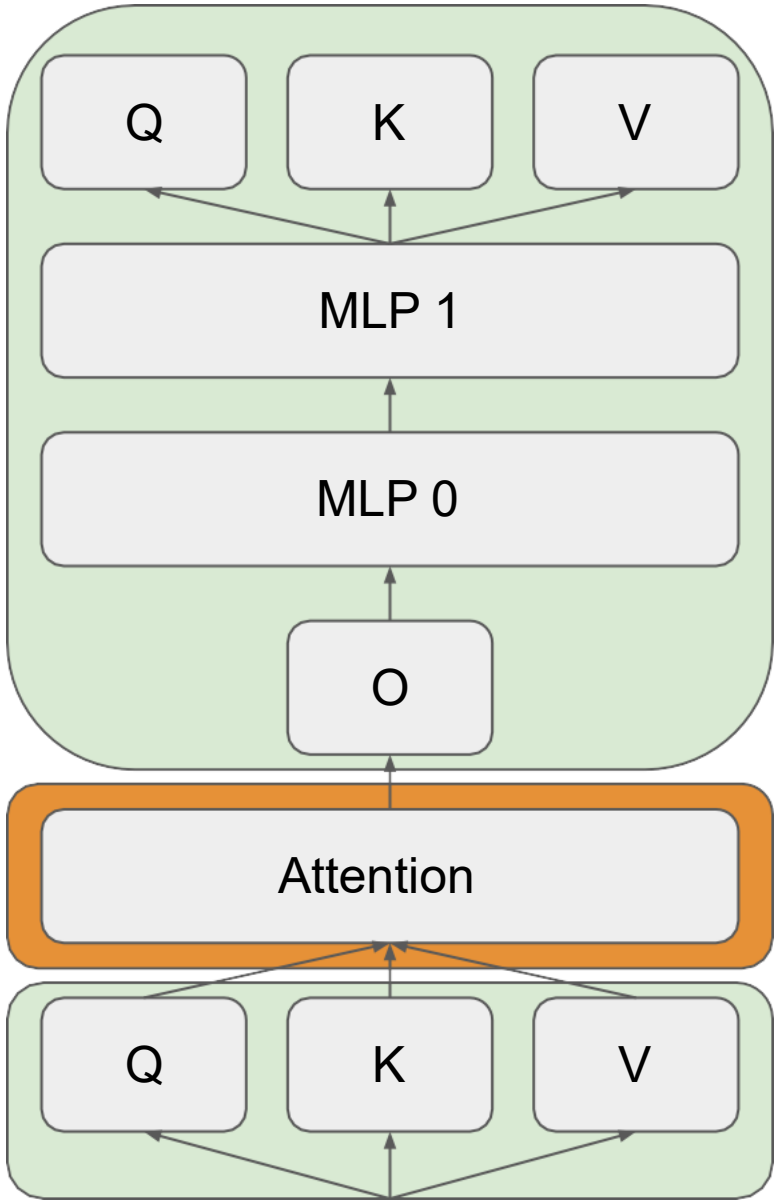
CUDA graph N-1

Graph split using
torch.compile

Full vs. Piecewise CUDA graphs

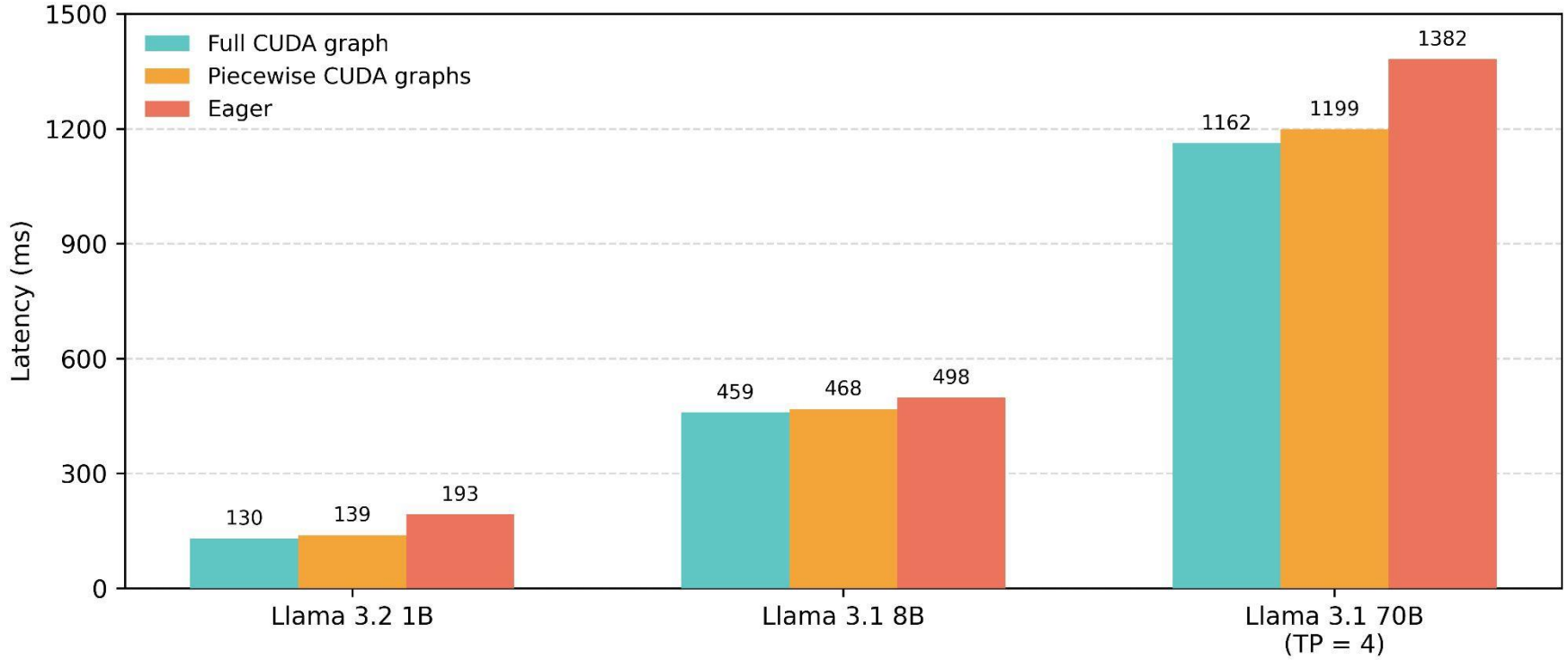


Piecewise CUDA Graphs



- Split the model into pieces
  - Runs the attention op in **eager-mode** PyTorch
  - Runs other ops with **CUDA graphs**
- Pros: High flexibility in implementing the attention op
  - No restriction on shapes
  - Any CPU operations are allowed at runtime
- Cons: CPU overheads in the attention op could potentially be a performance bottleneck

Performance: Full vs. Piecewise CUDA Graphs



- **6-39%** faster than PyTorch eager
- **2-7%** slower than full CUDA graph (in the worst case)
  - **0-2%** slower than full CUDA graph for batch size >= 8

*Input len: 4K, Output len: 50, Batch size: 1, H100 GPU

How does vLLM optimize LLM inference?

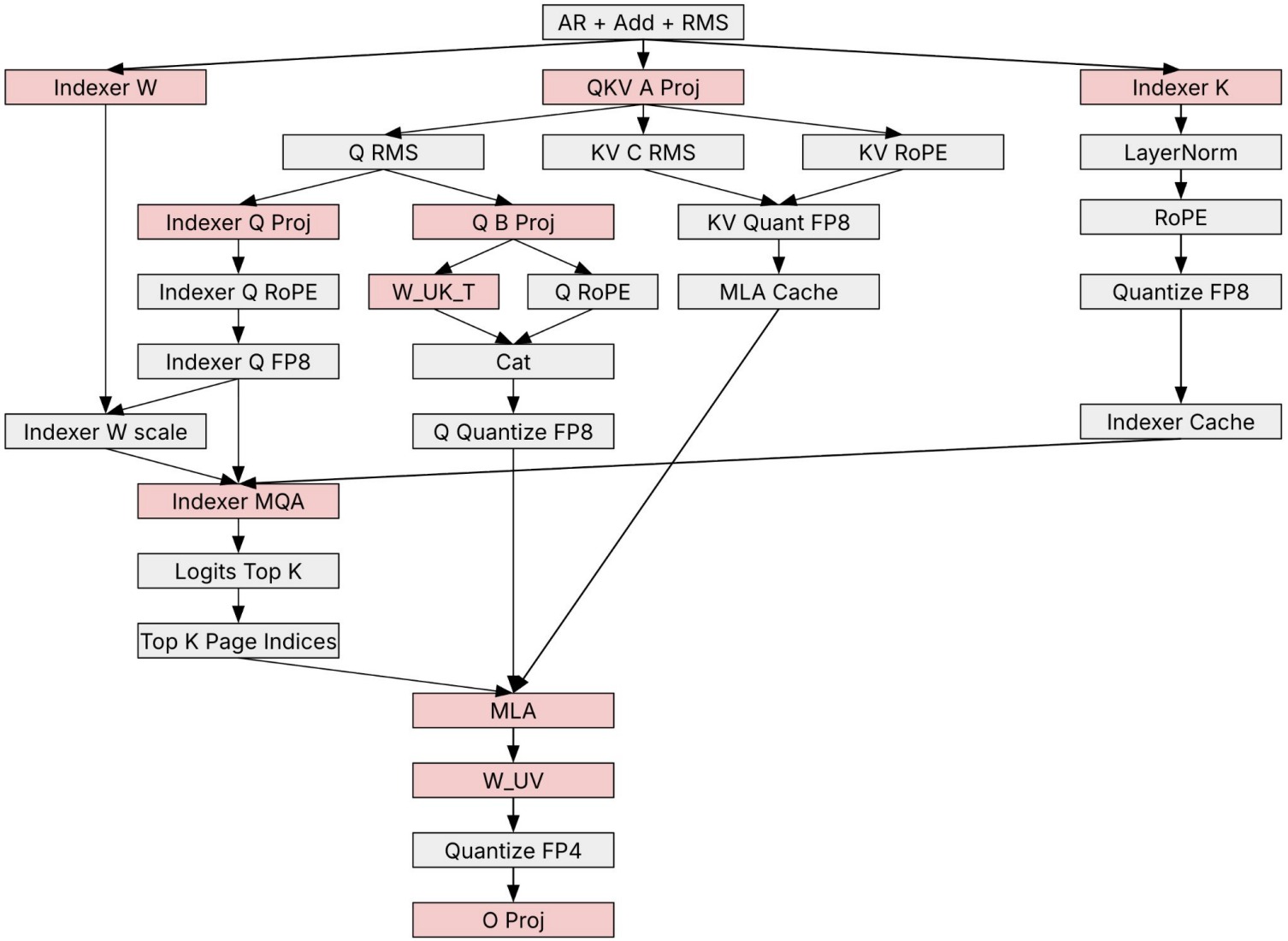
1. Minimizing CPU overheads
2. **Efficient GPU kernels**
3. Model parallelisms
4. Efficient memory management & caching

GPU Kernels in vLLM

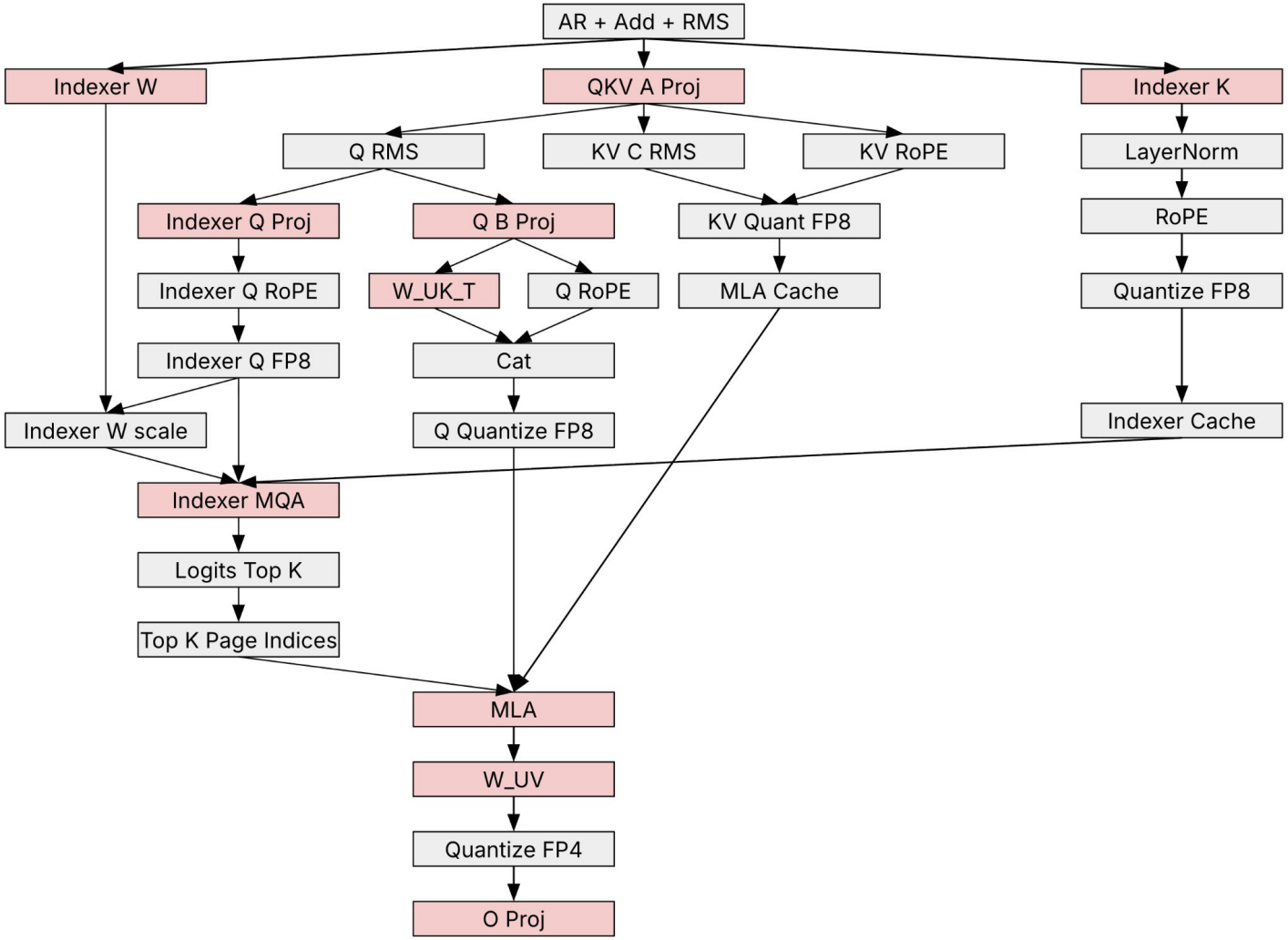
- For complex or performance-critical kernels, vLLM leverages FlashInfer or other libraries
  - vLLM provides `AttentionBackend` and `FusedMoE` abstractions to plug in different kernel implementations

- For memory-bound kernels (e.g., RMS norm, RoPE), vLLM either
  - Uses `torch.compile` to automatically fuse and generate kernels
  - Or, manually implements fused kernels

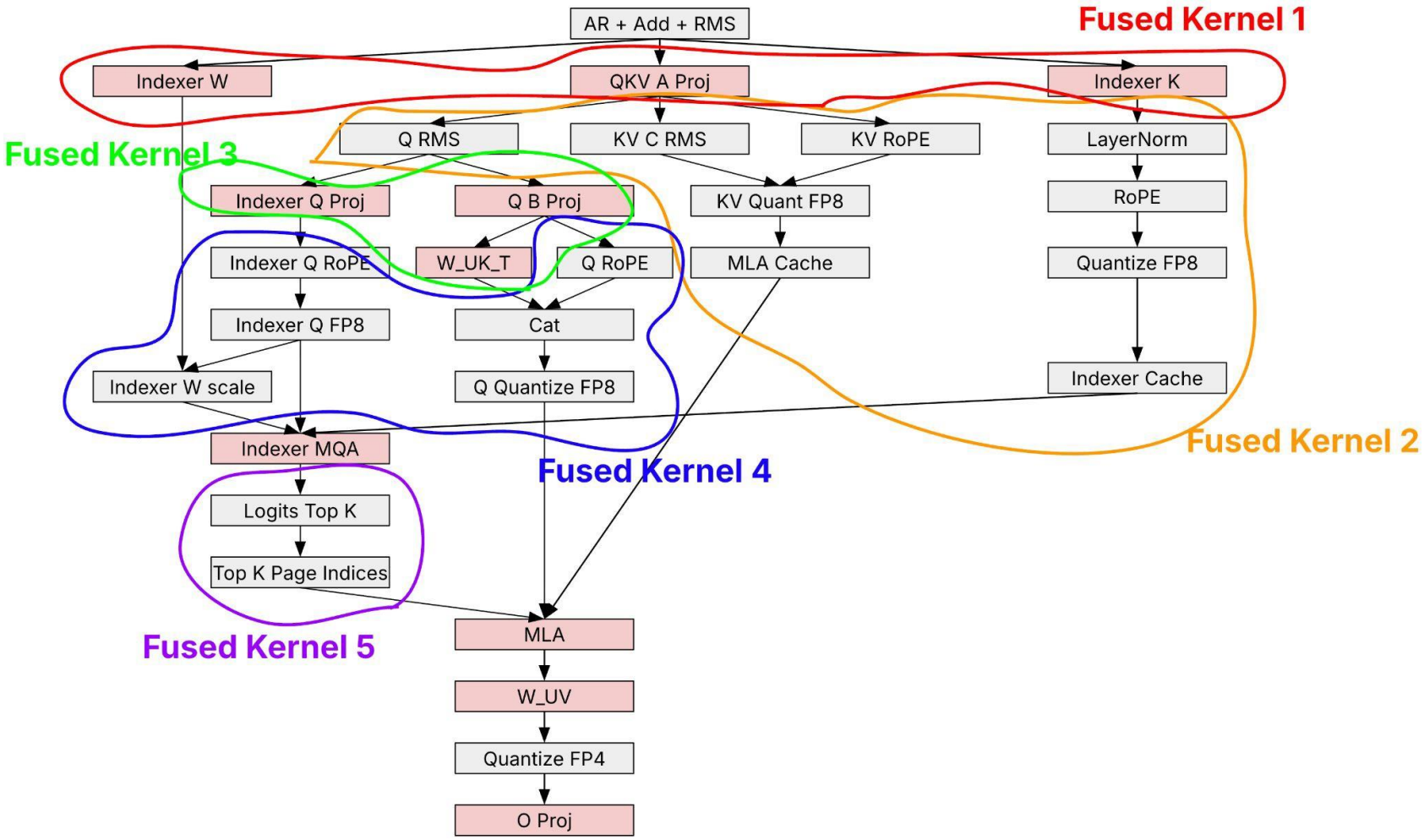
Example: DeepSeek V3.2

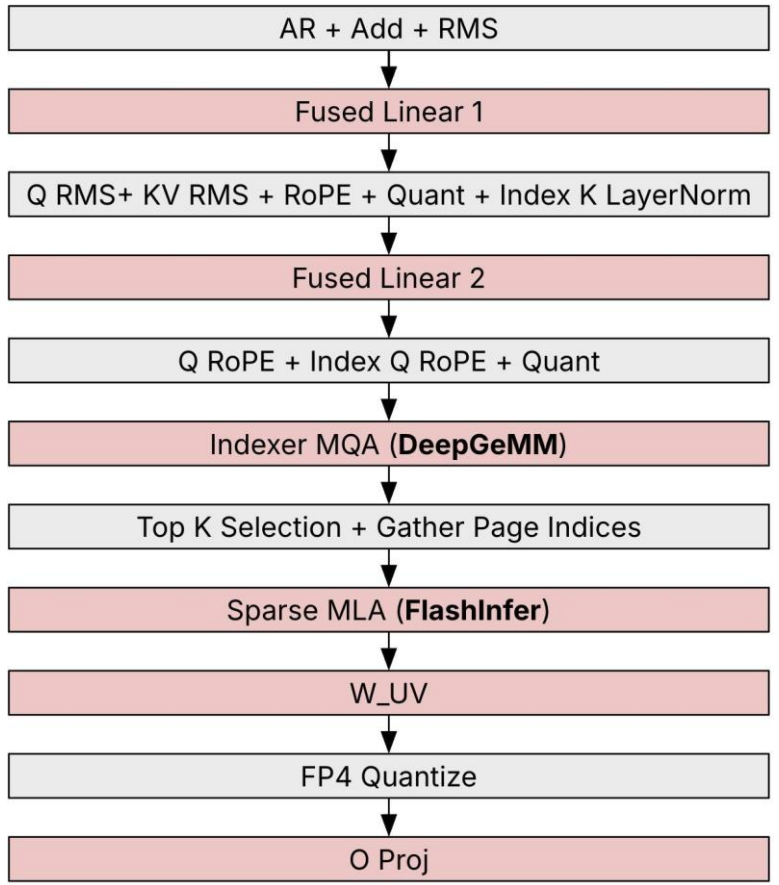


Example: DeepSeek V3.2



Example: DeepSeek V3.2 (fused)





How does vLLM optimize LLM inference?

1. Minimizing CPU overheads
2. Efficient GPU kernels
3. **Model parallelisms**
4. Efficient memory management & caching

Why do we need model parallelism?

- Models are **getting bigger**
  - Kimi K 2.5 has 1+ T parameters (600+ GB after quantization)
  - B200 GPU: 285 GB HBM
  - Memory is not only used for parameters. Activations and KV cache also takes memory.

- Use more GPUs to **multiple the FLOPS and bandwidth**
  - Reduce serving latency
  - Increase serving throughput

Parallelism for Inference: Recap

Goals: minimize **communication** (NVLink 1.8 TB/s vs. InfiniBand 50 GB/s) and **load imbalance**

	What is split	+ for serving	− for serving	Lecture
DP	Whole engine replicated; a load balancer routes requests by load, KV-cache usage, prefix-cache hits	No communication between engines	No parameter-memory saving; no latency reduction	L14–15
TP	Weights of the linear layers (KV cache split by heads)	Lower latency if the network is fast	Heavy communication, needs NVLink; limited by the number of KV heads	L16
EP	Experts of the MoE layers across GPUs (attention needs another scheme, e.g., DP)	Kernel-friendly; less communication than TP	Only for MoE layers; load imbalance across experts	L17
CP	Sequence dimension of a long request; GPUs exchange Qs or KVs	Very long contexts; balances the KV cache across GPUs	No parameter-memory saving; requests must be load-balanced	next slide
PP	Layers across GPUs, executed as a pipeline	Lowest communication	Higher latency; load imbalance between stages	L16

Real deployments combine them (see "Mixed Parallelism"): e.g., TP + PP for dense models, DP attention + EP for MoE models

Context parallelism

- Extension of DP to subsequences of a single long sequence
- Communicate Qs or KVs across different shards

👍 Parallelize the sequence dimension for very long sequences. Balance the KV Cache across shards

👎 No parameter memory saving

👎 For sampling one request, only one GPU will be running. Require balancing requests loads.

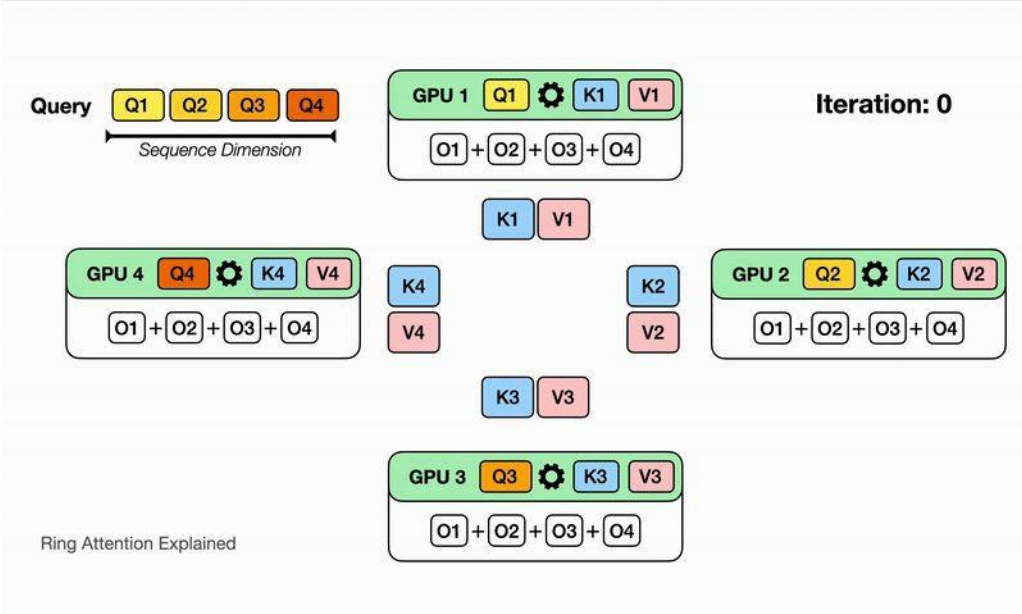
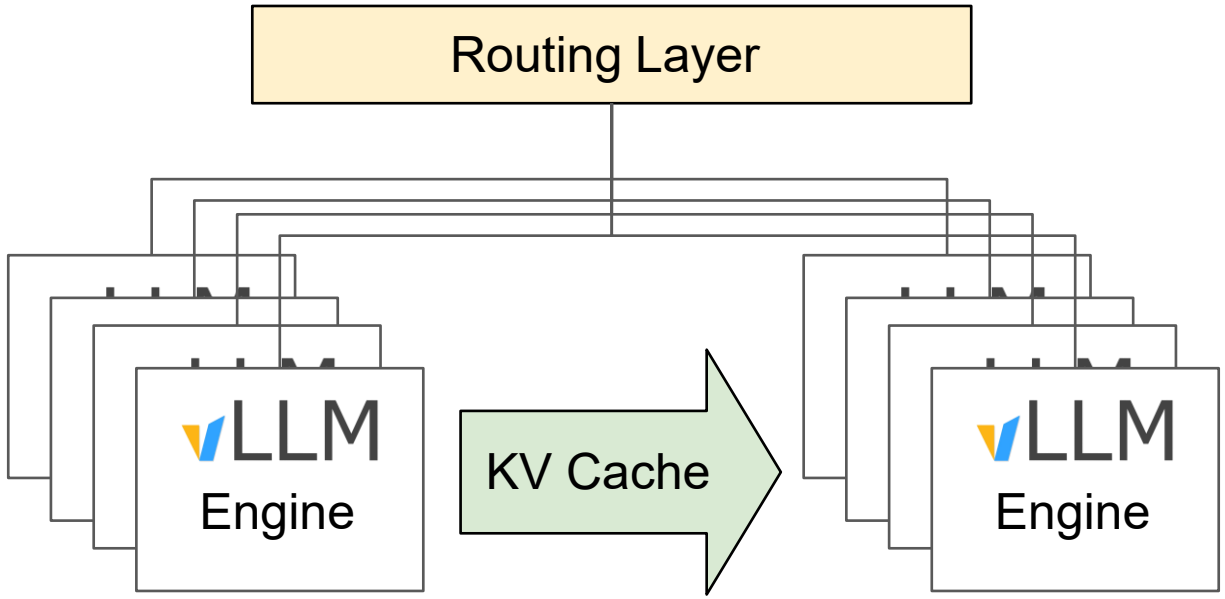


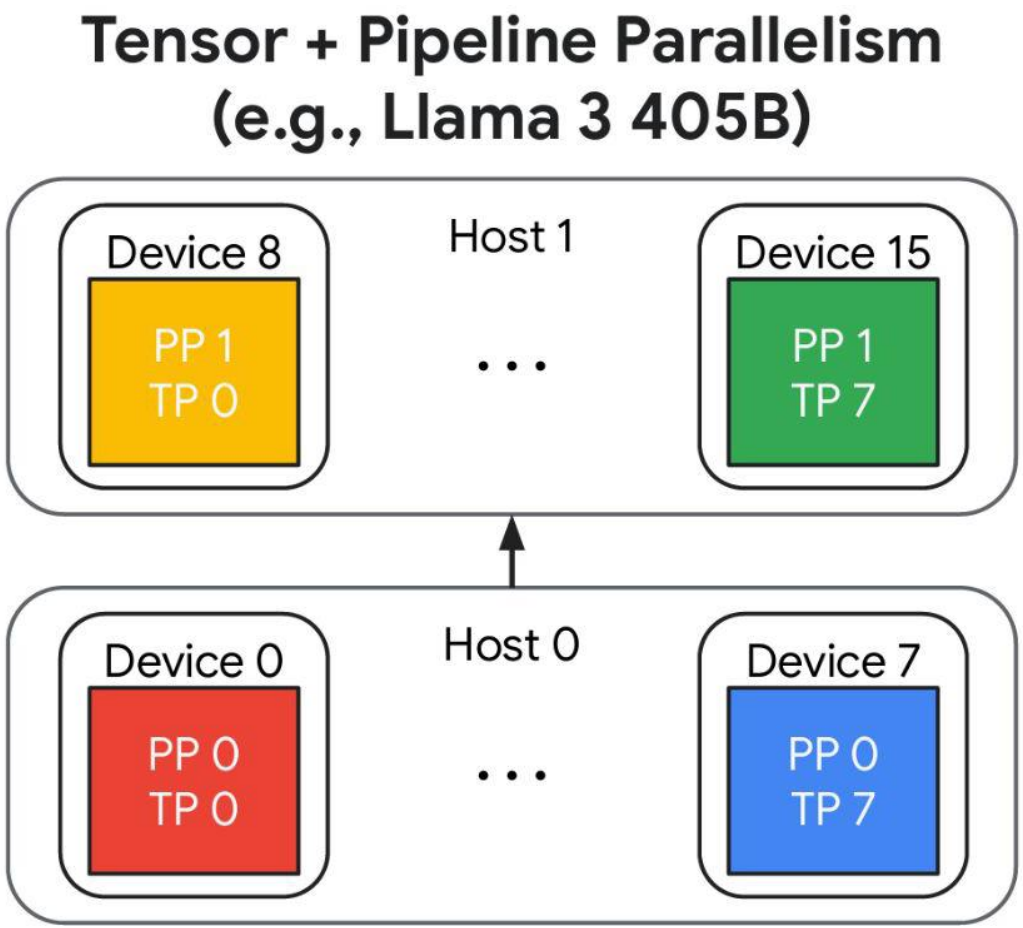
Image source: https://coconut-mode.com/posts/ring-attention/

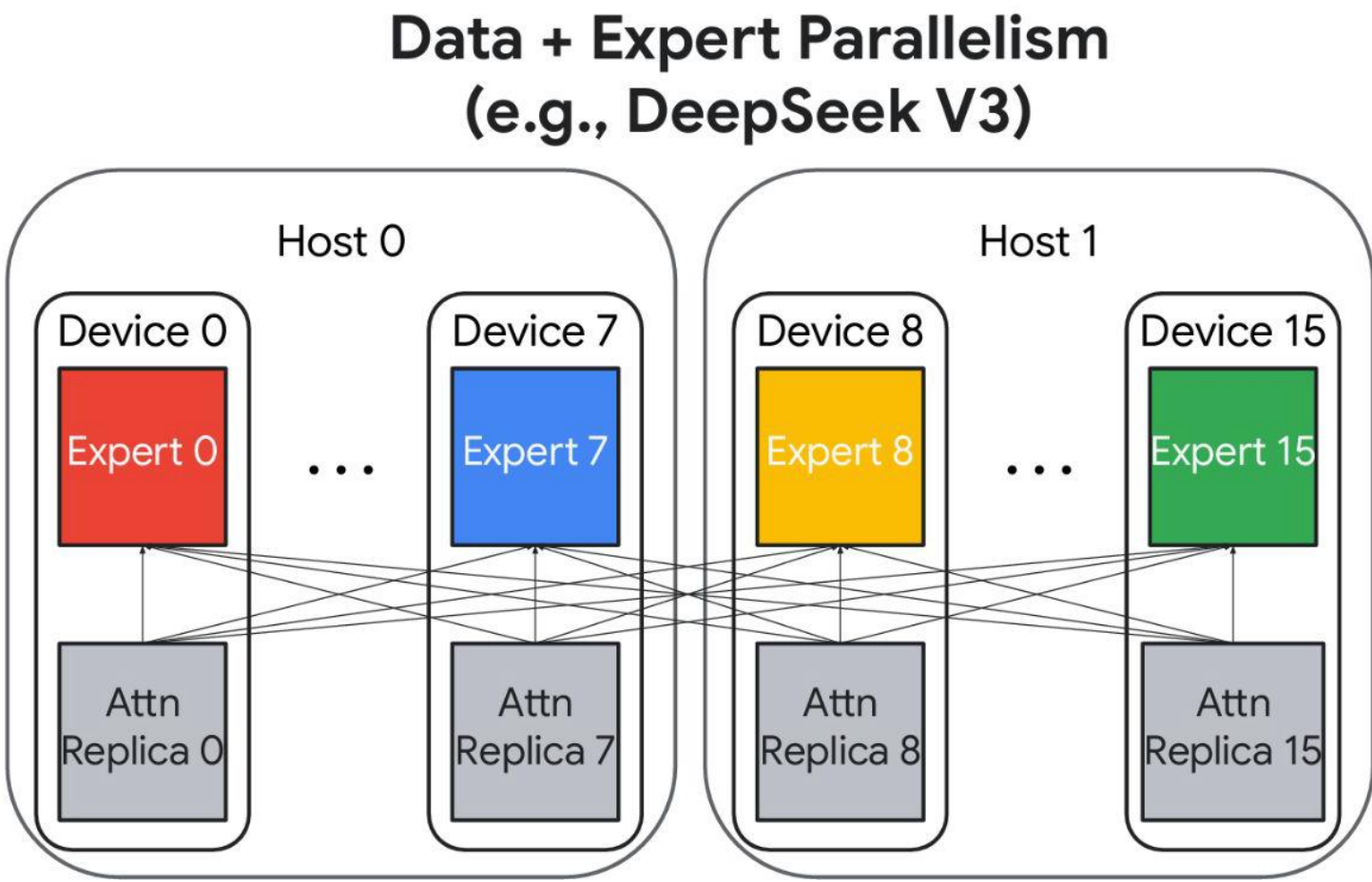
Prefill Disaggregation (PD)



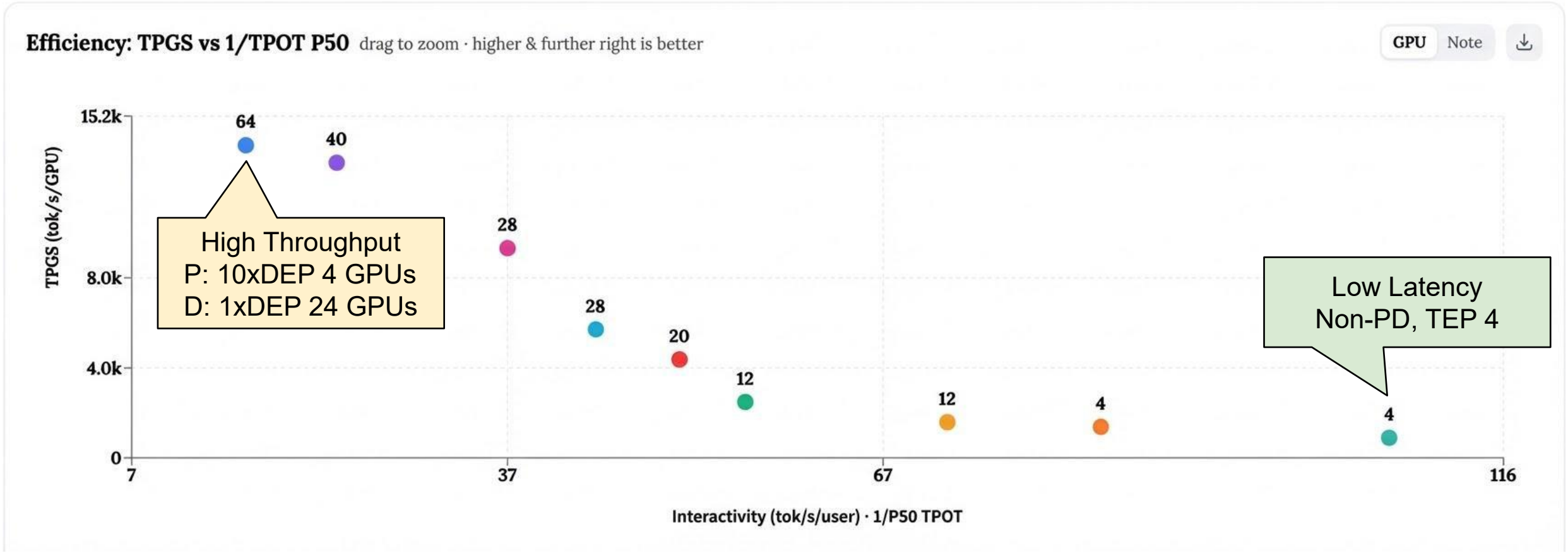
- Why PD?
  - More controllable TTFT & TPOT
  - Optimize prefill & decode independently
- Routing backends
  - vLLM Router, NVIDIA Dynamo, llm-d, Ray Serve LLM
- NIXL for KV cache transfer

Mixed Parallelism





Case Study: Kimi K2.5 (NVFP4)

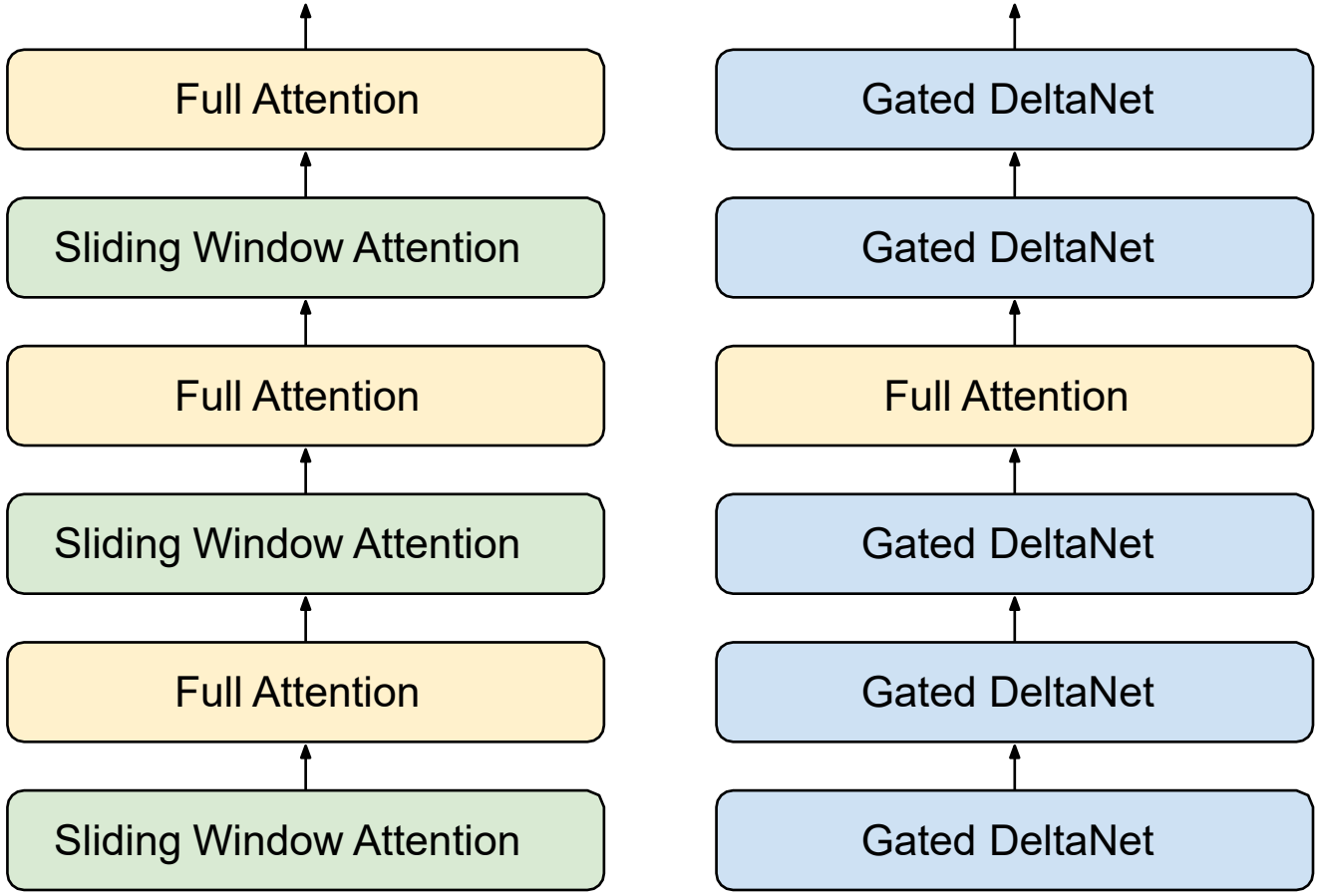


8K input & 1K output, GB200

How does vLLM optimize LLM inference?

1. Minimizing CPU overheads
2. Efficient GPU kernels
3. Model parallelisms
4. **Efficient memory management & caching**

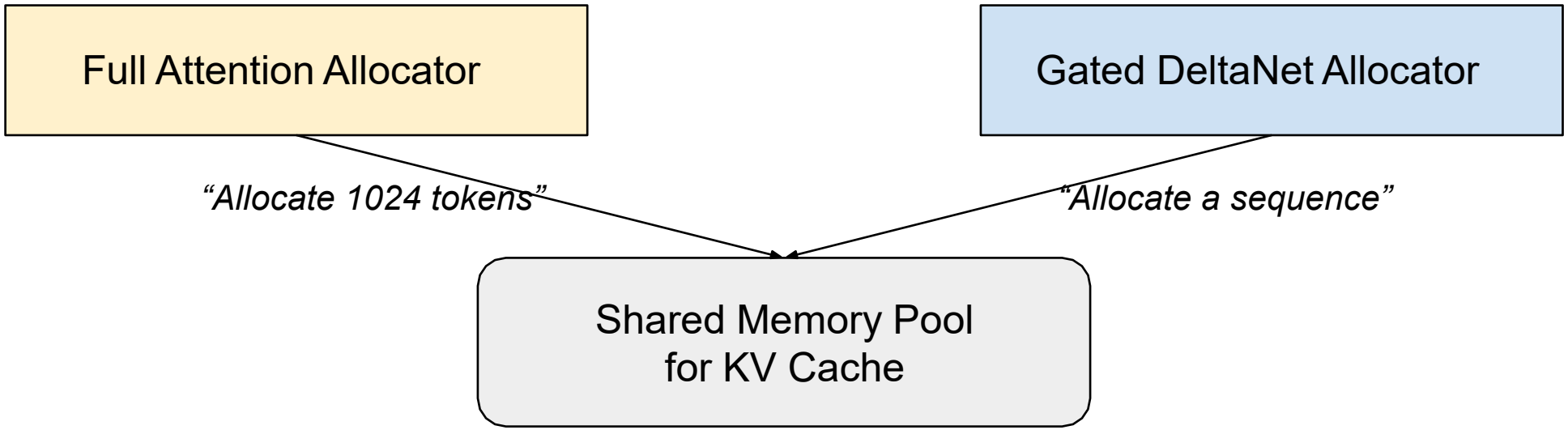
Hybrid Memory Allocator



GPT-OSS Qwen 3.5

- Modern LLMs use *hybrid* architecture (full attention + X) for efficient long context support
- How to manage the KV cache for the hybrid models?
  - A straightforward solution is *static partitioning*
  - However, it could cause severe memory fragmentation

Hybrid Memory Allocator (cont'd)



- vLLM uses *dynamic partitioning* to flexibly utilize all available memory
- vLLM adjusts the allocation granularity ("*block sizes*") to minimize memory fragmentation
- 0-12% memory waste for all OSS models

Summary

- **PagedAttention**: store the KV cache in fixed-size blocks, mapped by a per-request block table (like OS paging)
  - Waste only in each request's last block → larger batches → **2–4× higher throughput**
  - Ref counts + **copy-on-write** share KV blocks across samples (parallel sampling, beam search)
  - Out of KV blocks → **preempt** a request, then swap it back in or recompute it
- **Continuous batching**: requests join and leave every iteration; their KV cache grows block by block
- **Shared prefixes** across requests (system prompts, chat history) can reuse KV → next lecture: SGLang RadixAttention
- **vLLM engine**: hides CPU overhead (async scheduling, CUDA graphs), fast kernels, model parallelism