Paged Attention & vLLM for Efficient LLM Inference Engine

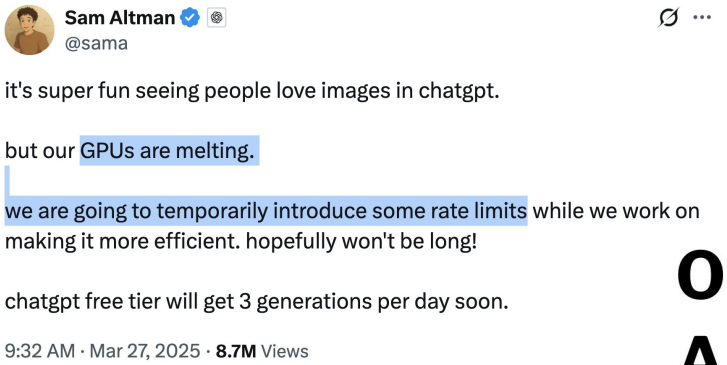
Woosuk Kwon, Inferact

Topics

- **Paged Attention: Efficient memory management for KV cache**
- vLLM: A real-world open-source inference engine
- Q&A

LLM inference is slow and expensive

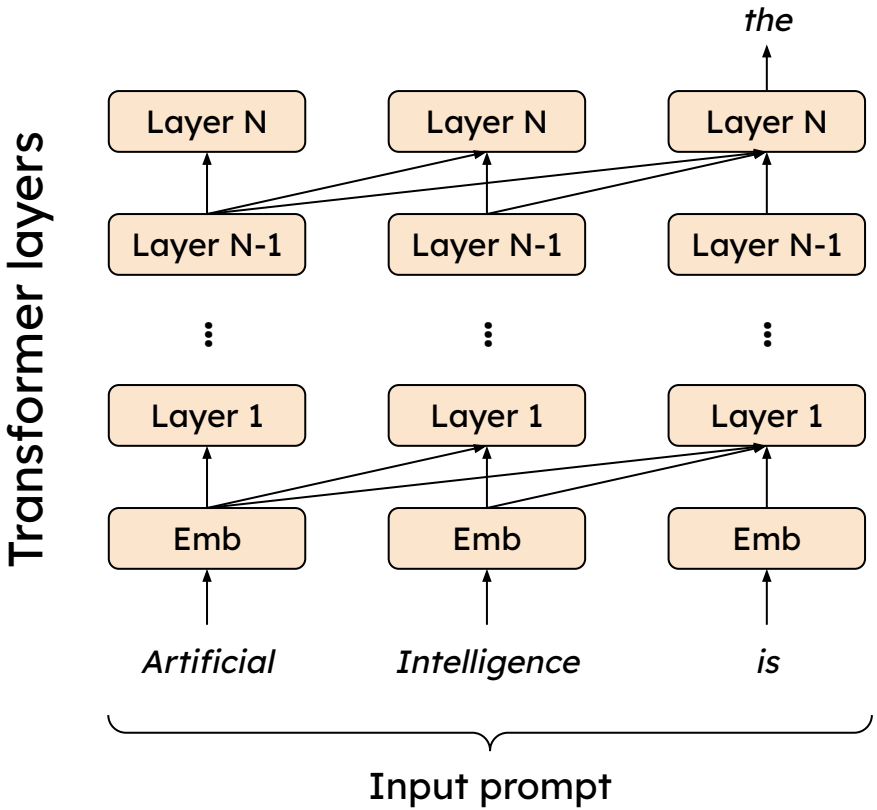
- LLMs typically run on high-end GPUs (e.g., NVIDIA H100)
- Yet, each GPU can only serve a handful of QPS; even simple queries (with reasoning) can take minutes



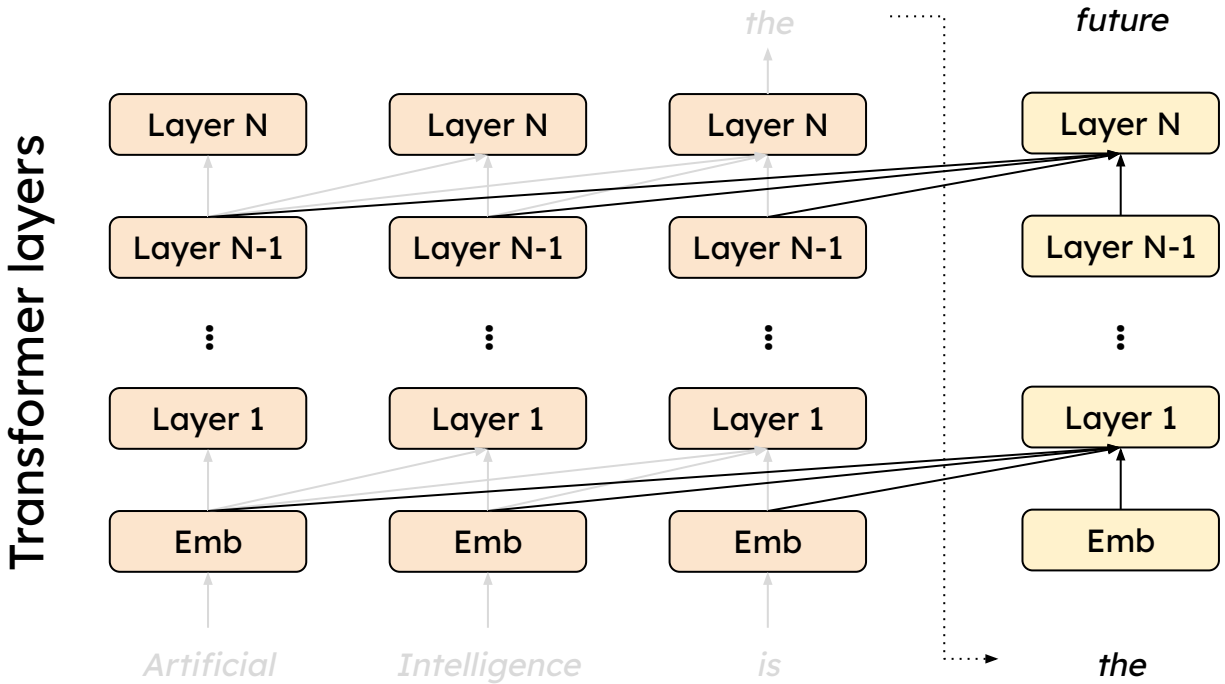
LLM inference is slow and expensive

- 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
  - A major obstacle to broader adoption of AI
- The problem is getting harder due to the increasing scale:
  - Larger model sizes
  - Growing context lengths
  - More tokens generated per query

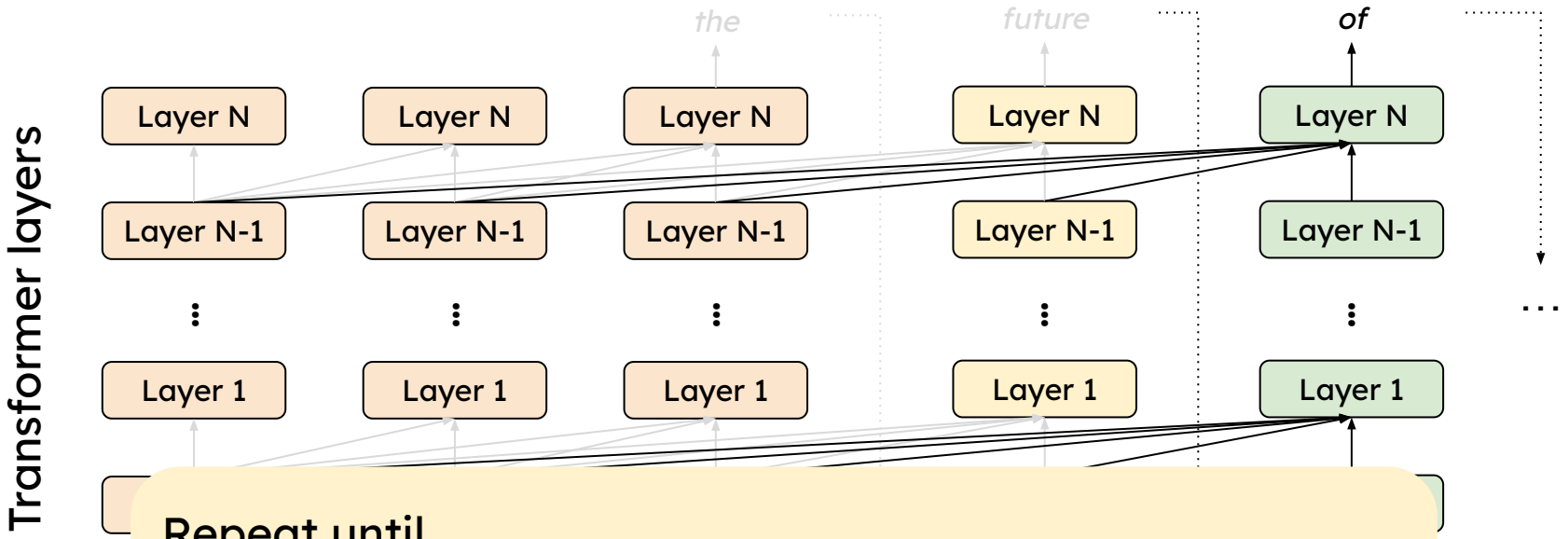
LLM inference process



LLM inference process



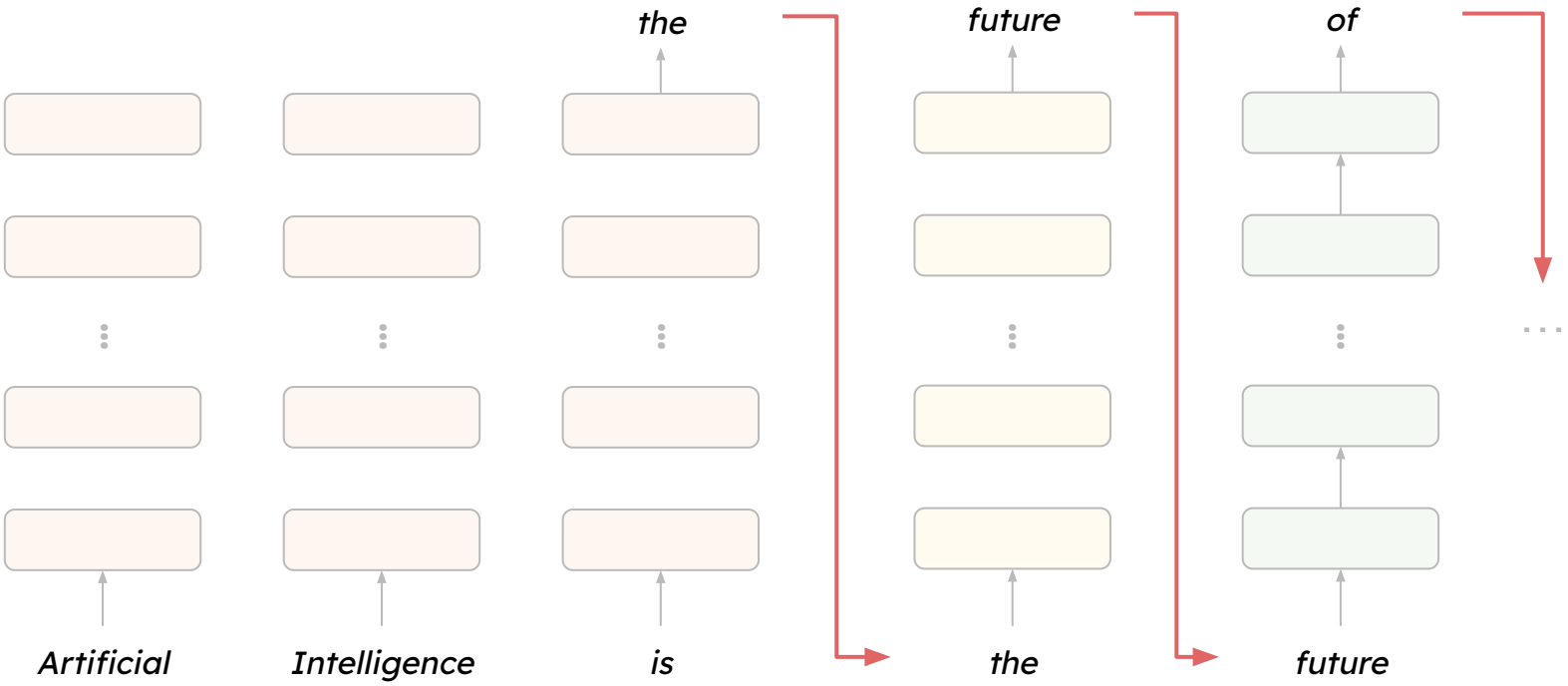
LLM inference process



Repeat until

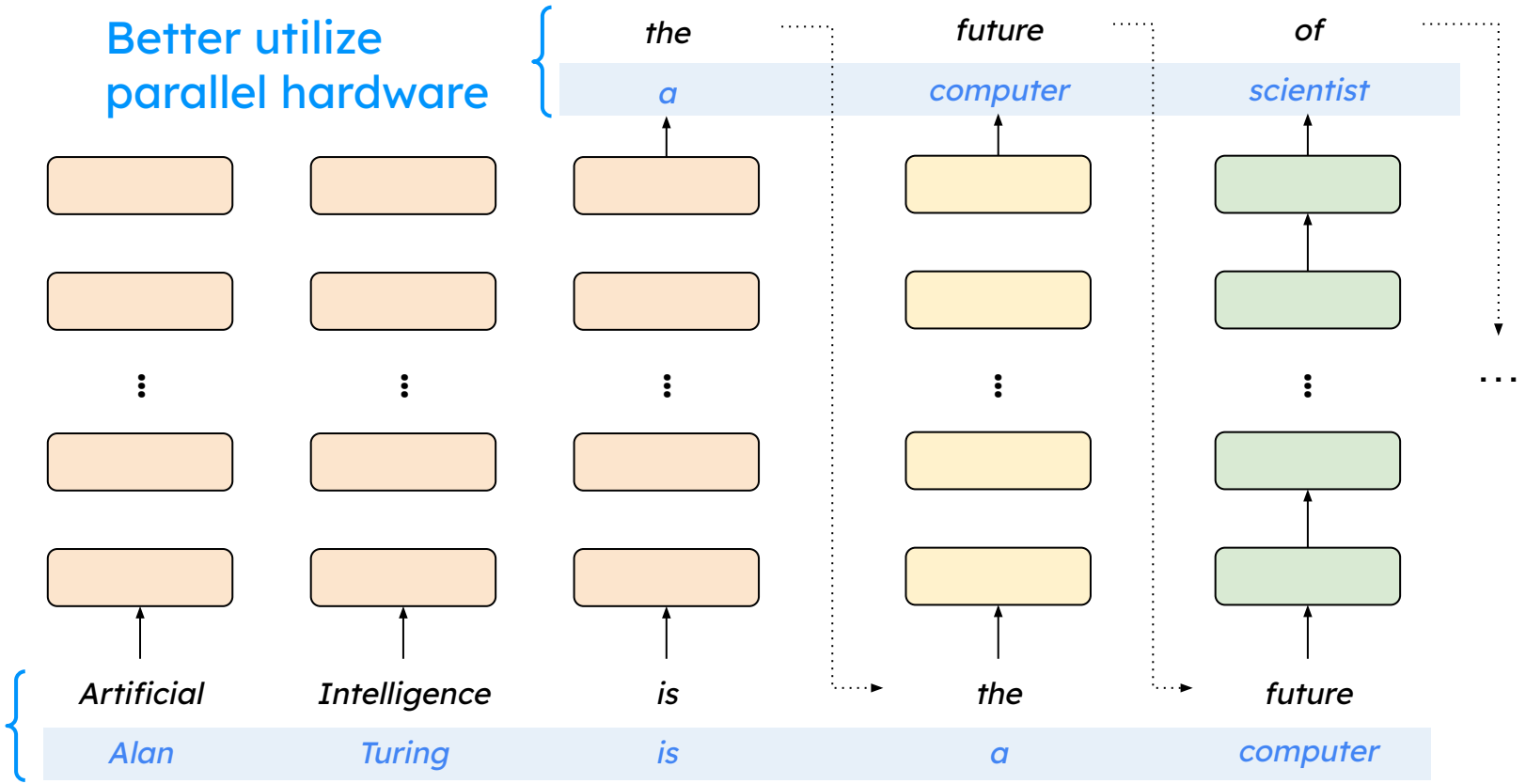
- Reaches maximum length (e.g., 1M tokens)
- Generate certain tokens (e.g., “*<|end of sequence|>*”)
- …

Why is LLM inference inefficient?

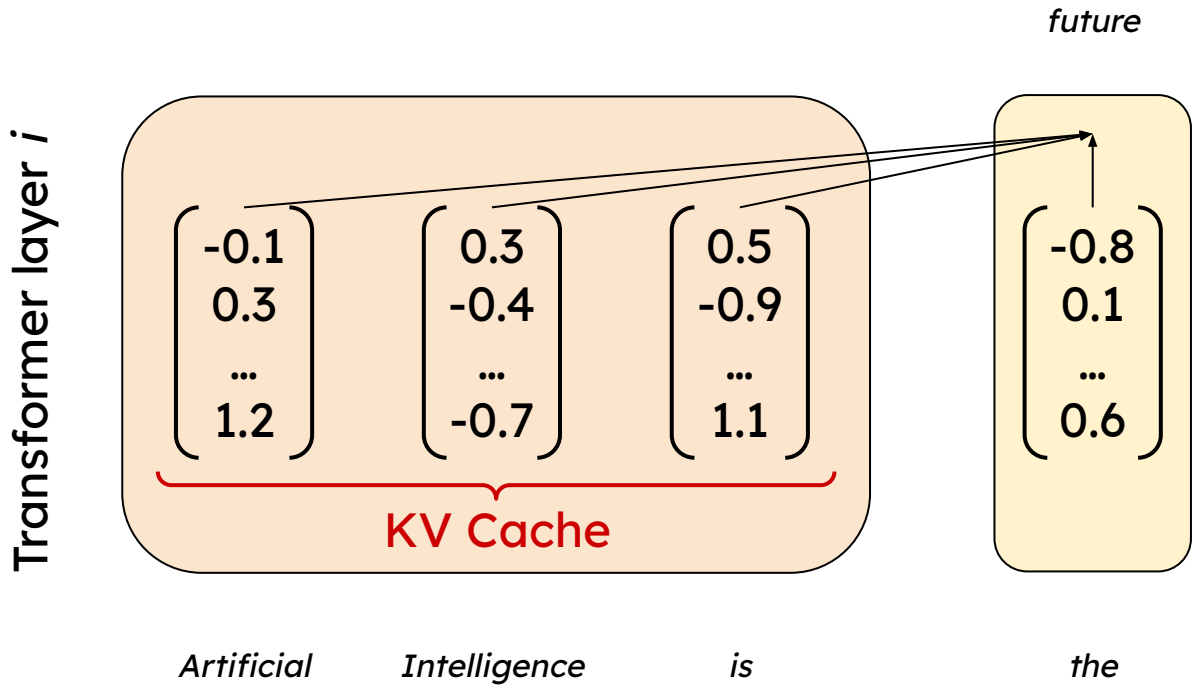


Sequential dependency → Hard for GPU parallelization

Batching?

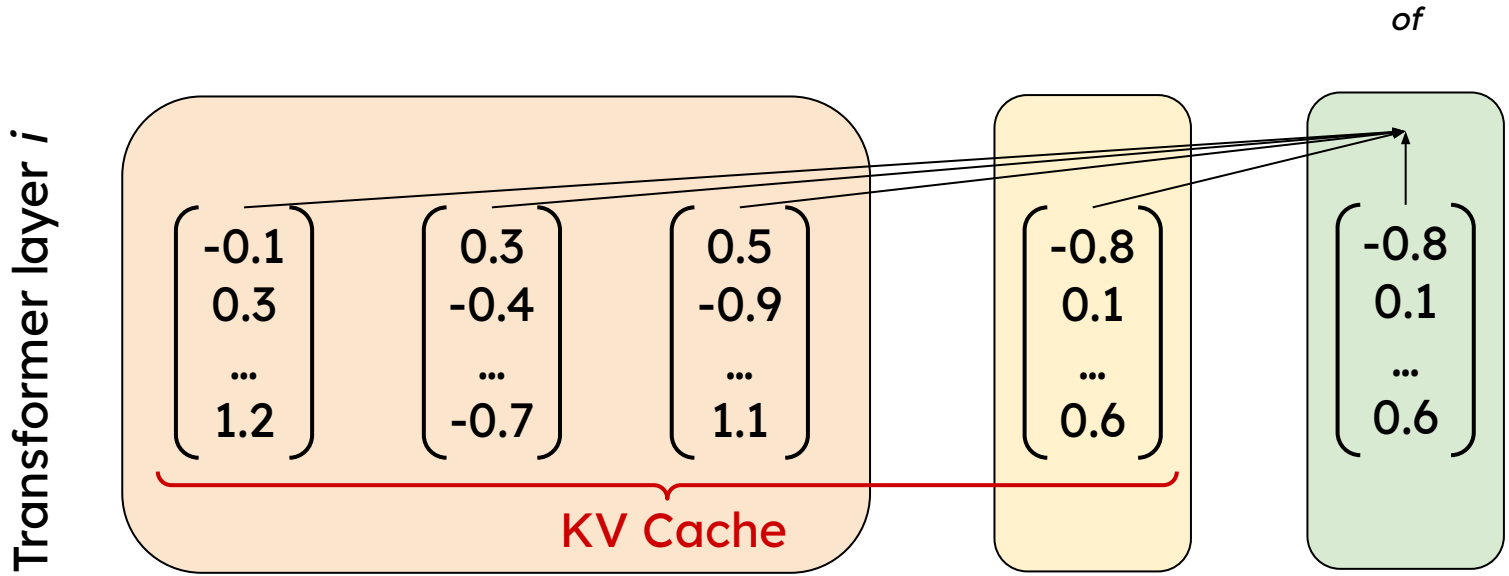

Better utilize parallel hardware
the
future
of
a
computer
scientist
Artificial
Intelligence
is
the
future
Alan
Turing
is
a
computer

Attention KV cache



Intermediate **vector** repr.
("Attention key & value")

Attention KV cache size



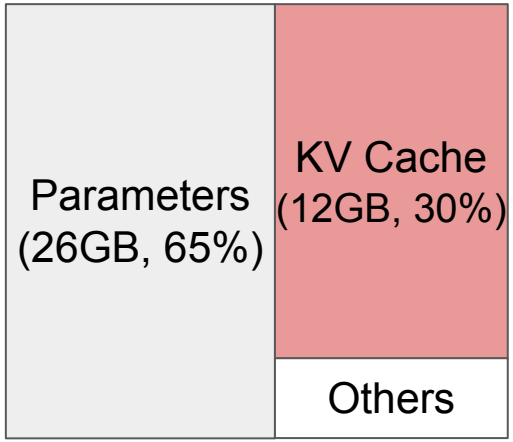
Artific *ure*

KV Cache is **huge**:

- Each token: ~1 MB.
- One full request: ~several GBs.

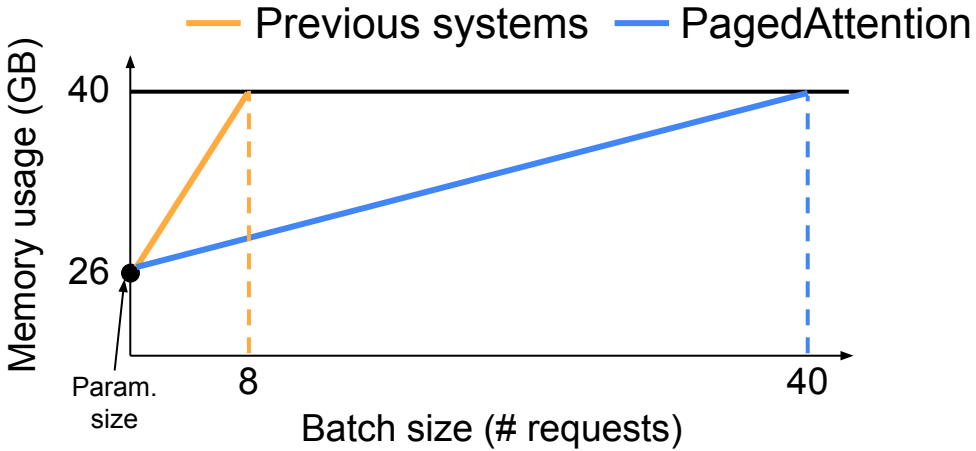
Key insight

Efficient management of KV cache is crucial for high-throughput LLM serving



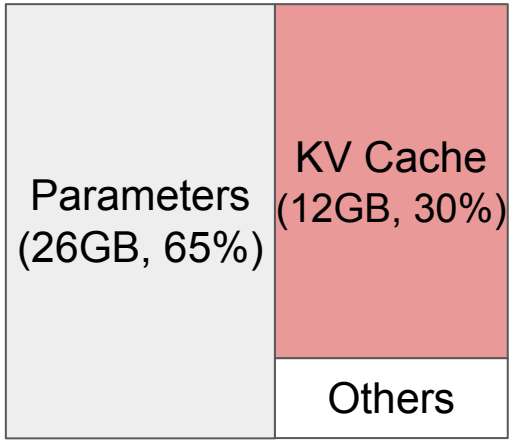
13B LLM on A100-40GB*

*Common setup in 2023



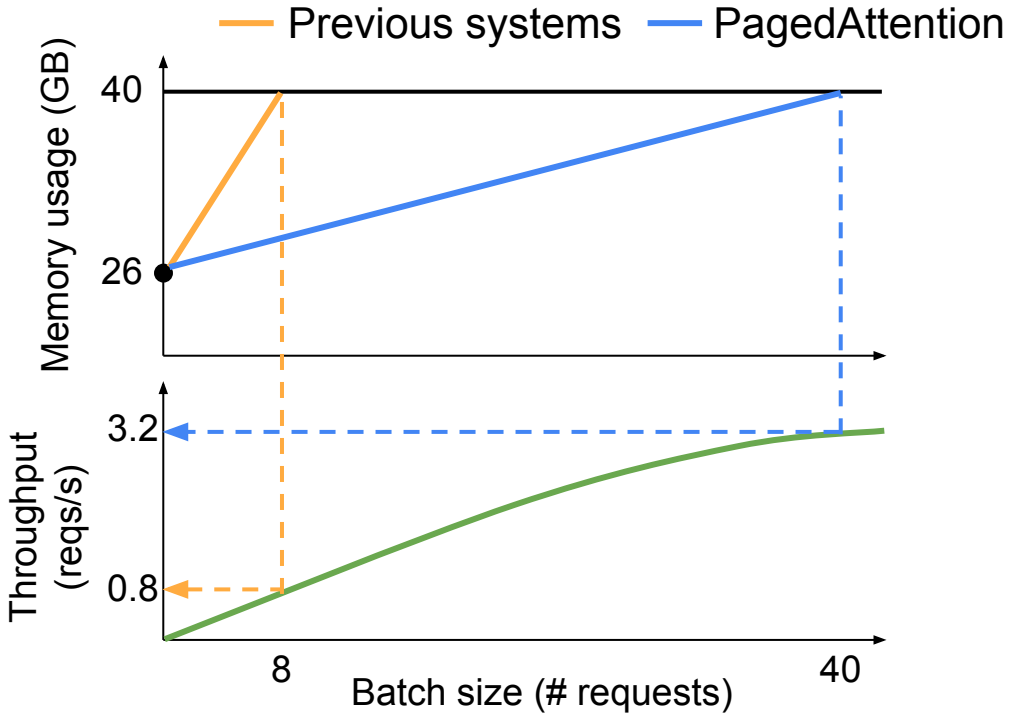
Key insight

Efficient management of KV cache is crucial for high-throughput LLM serving



13B LLM on A100-40GB*

*Common setup in 2023



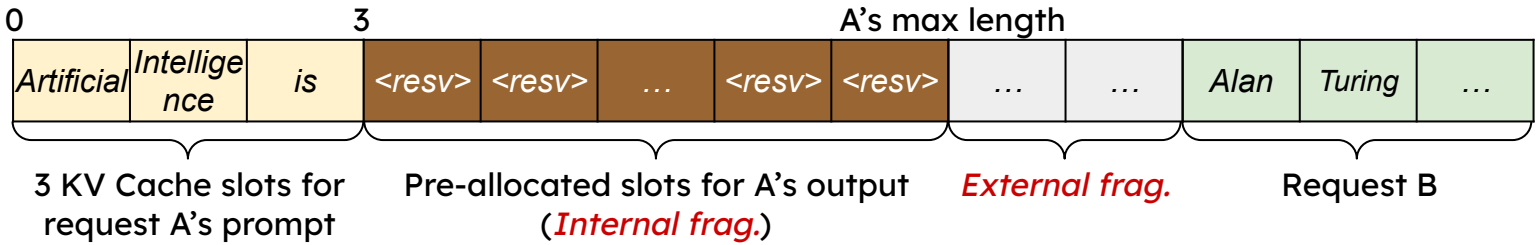
Case for recent MoE models

- 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

- In other words, 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

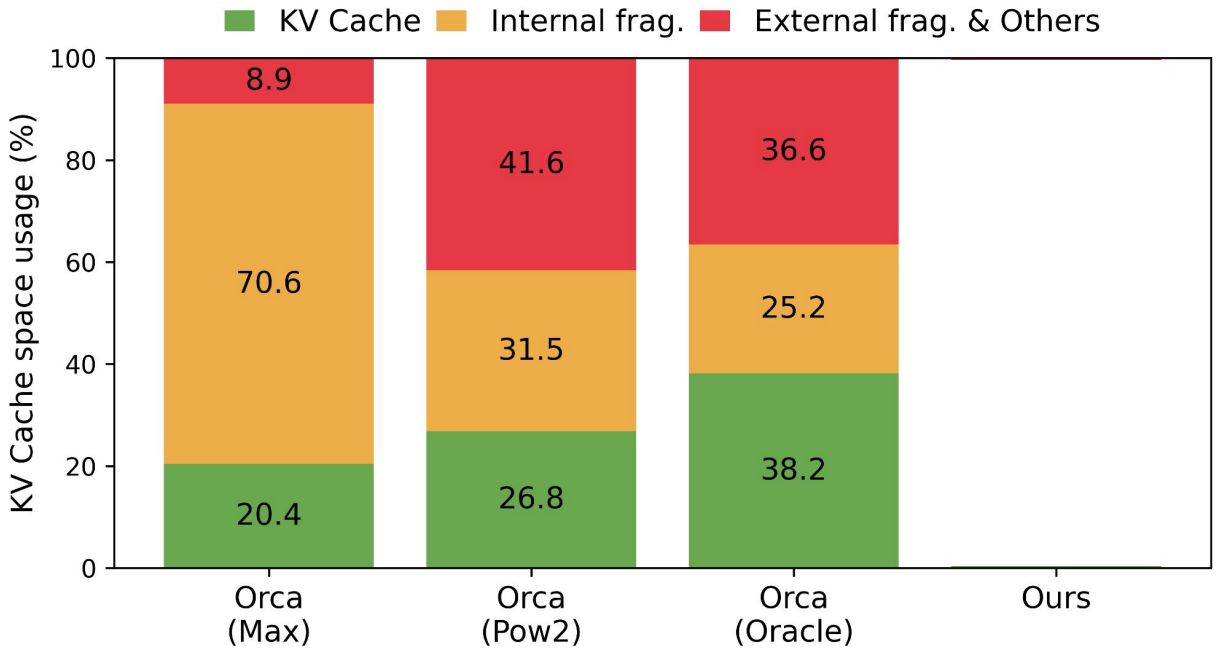
KV cache management in previous systems



- **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

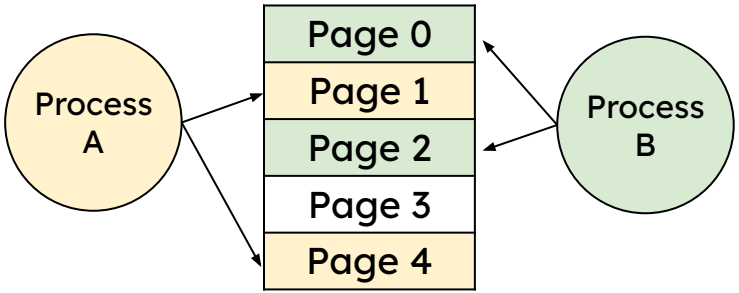


Yu et al. “Orca: A Distributed Serving System for Transformer-Based Generative Models” (OSDI 22).

PagedAttention

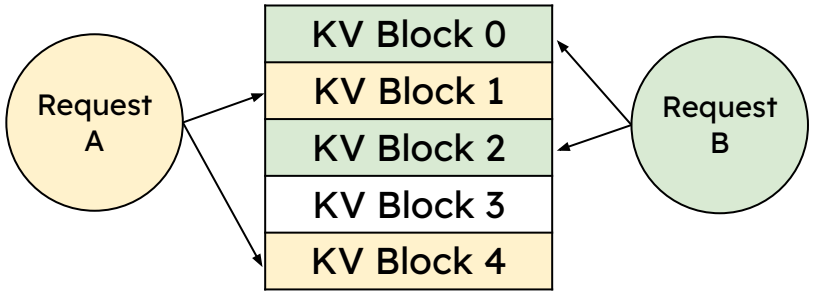
Application-level memory **paging** and **virtualization** for attention KV Cache

Memory management in OS



Physical Memory

PagedAttention

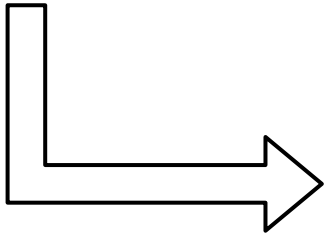


KV Cache

Paging KV cache space into *KV blocks*

Contiguous KV Cache

	Artificial	Intelli-gence	is	the							



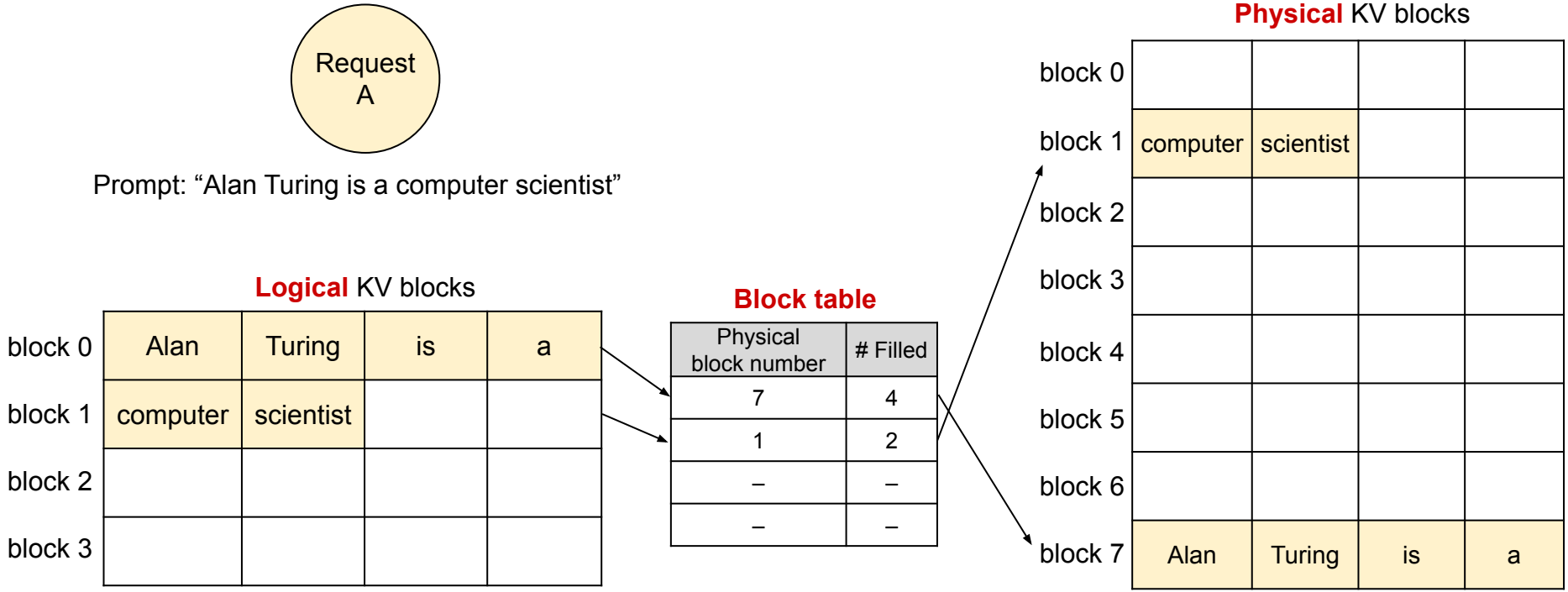
KV block: a **fixed-size** block of memory that stores KV cache **from left to right**

KV Blocks

block 0				
block 1				
block 2				
block 3				
block 4	Artificial	Intelli-gence	is	the
block 5				

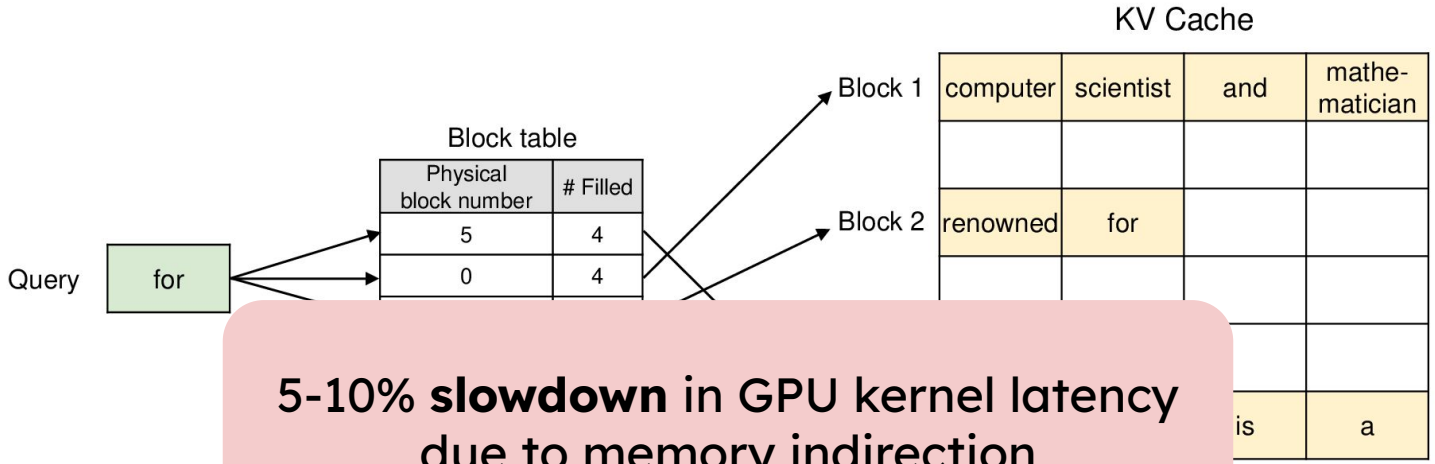
Block size = 4

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

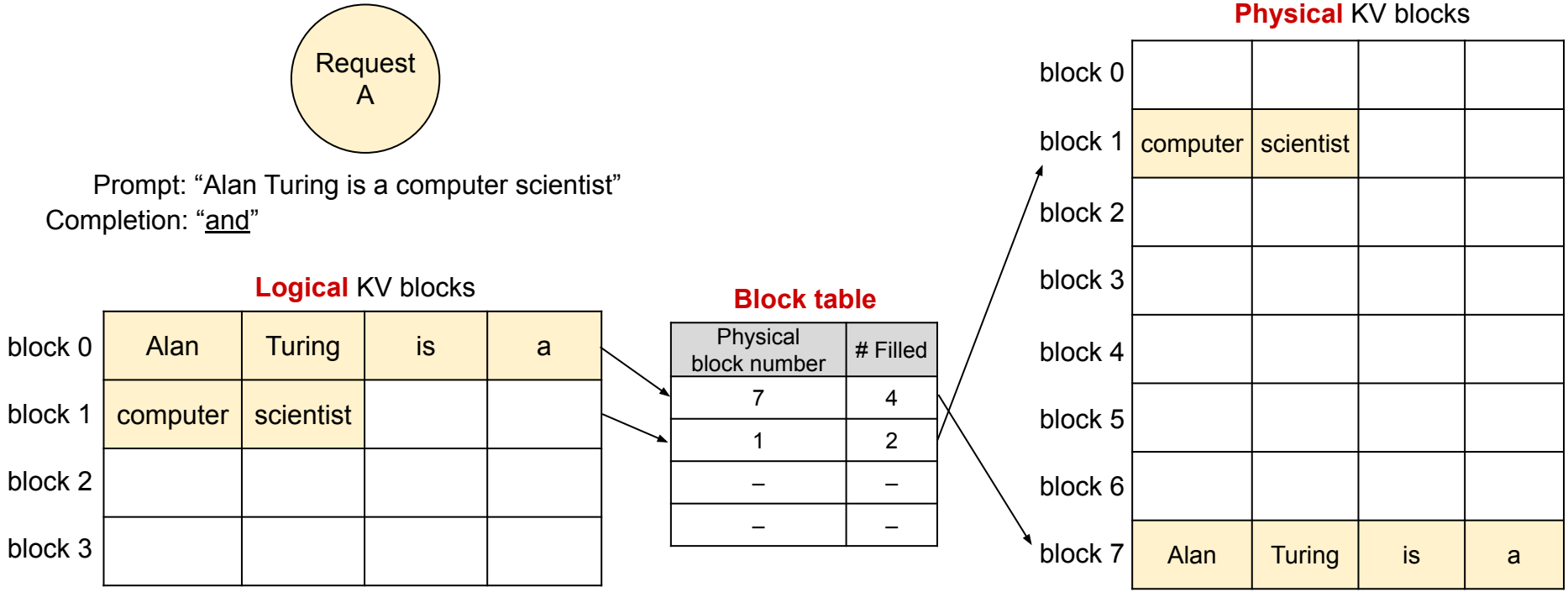
Paged Attention Pseudocode

```python
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
```

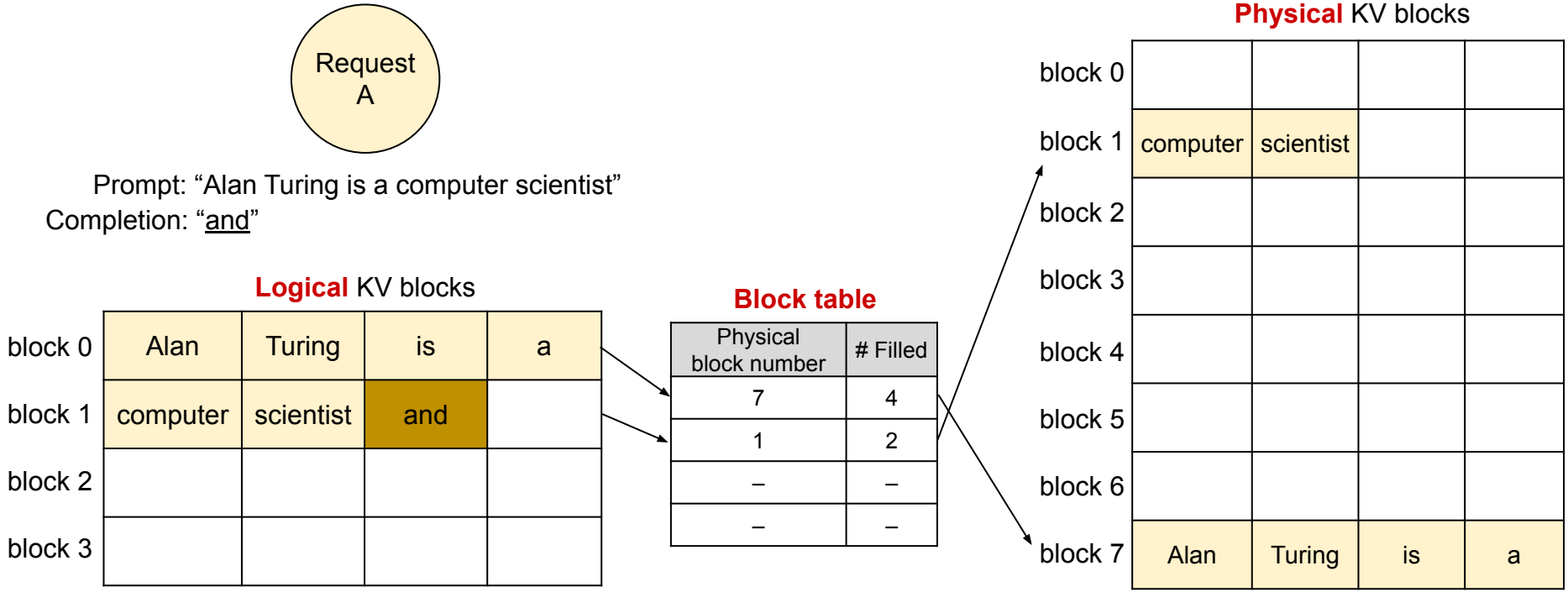
```python
def paged_attn(
    query: torch.Tensor,        # [batch, 1, head_size]
    key_cache: torch.Tensor,    # [total_num_blocks, block_size, head_size]
    value_cache: torch.Tensor,  # [total_num_blocks, block_size, head_size]
    kv_len: int,
    block_table: torch.Tensor,  # [batch, num_blocks]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

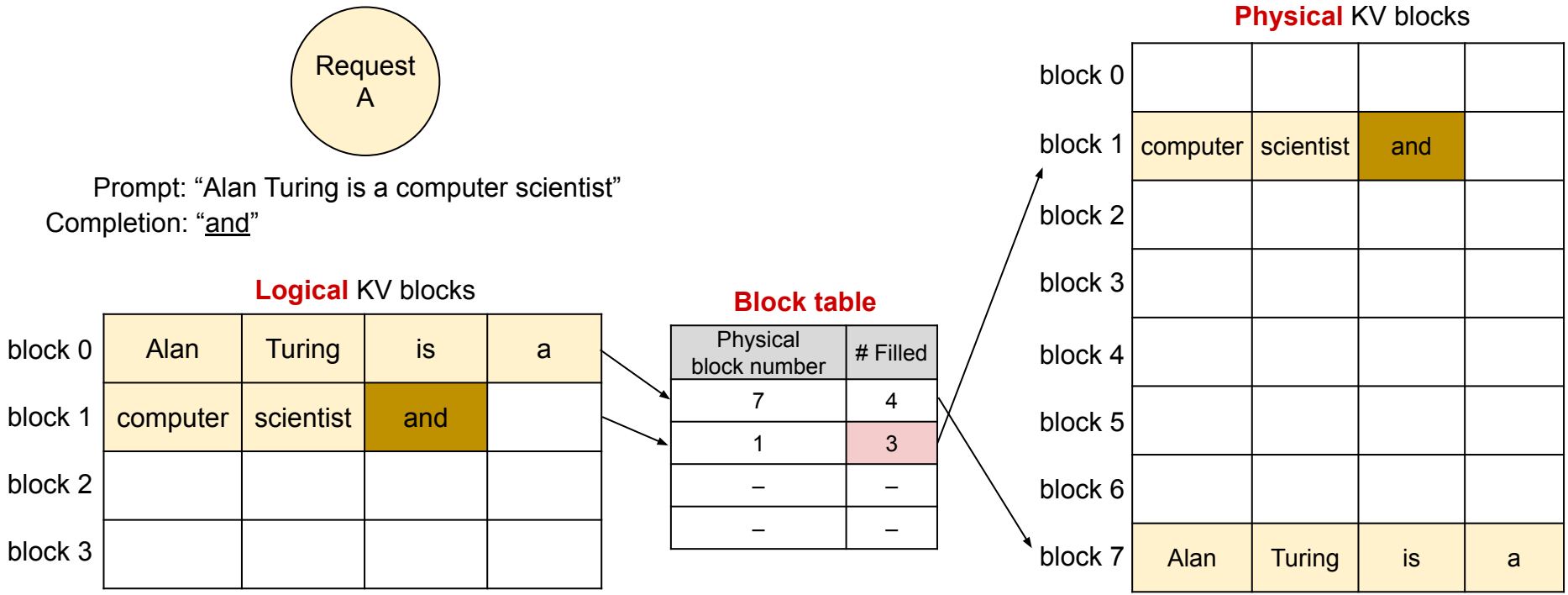
Memory management with PagedAttention



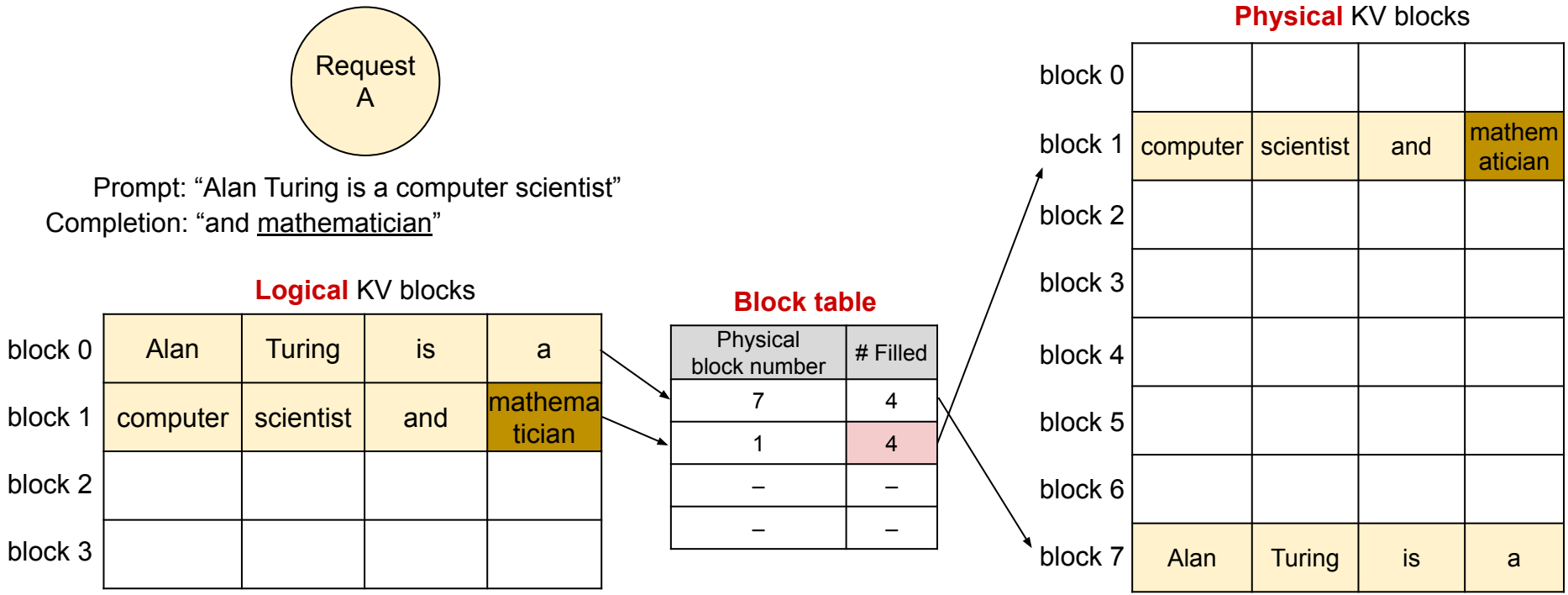
Memory management with PagedAttention



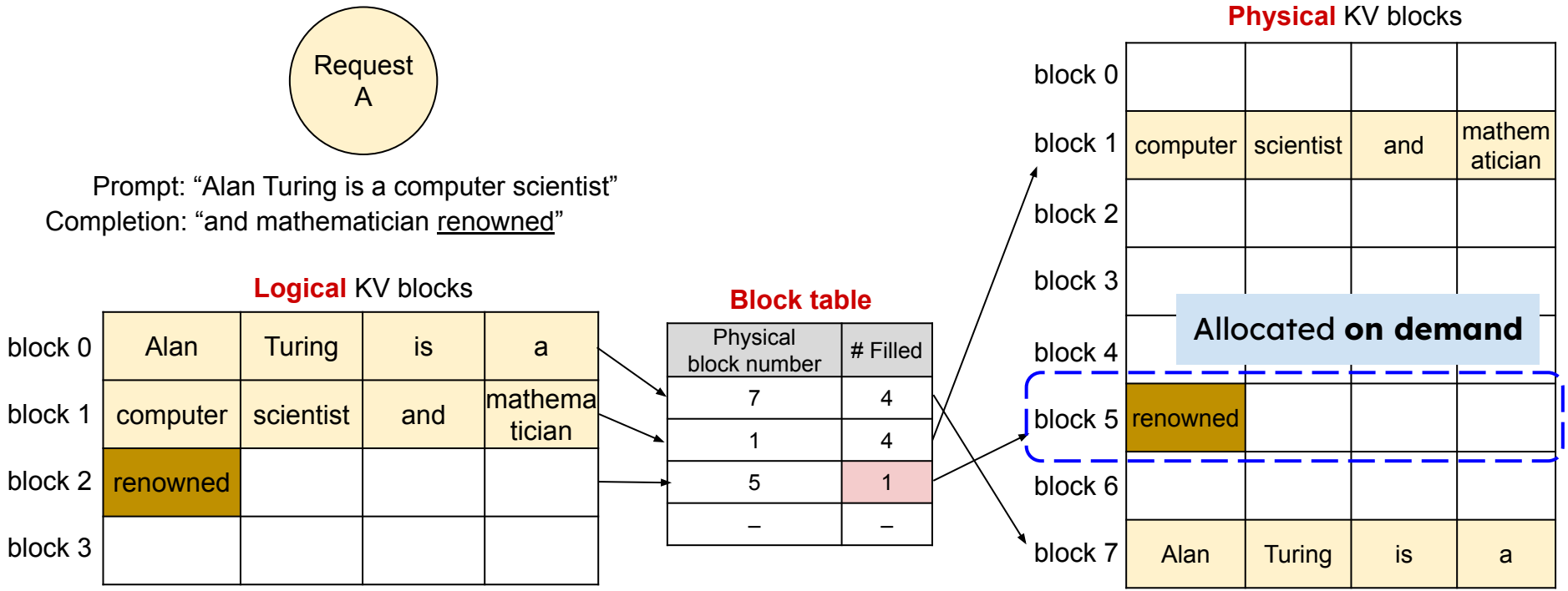
Memory management with PagedAttention



Memory management with PagedAttention



Memory management with PagedAttention

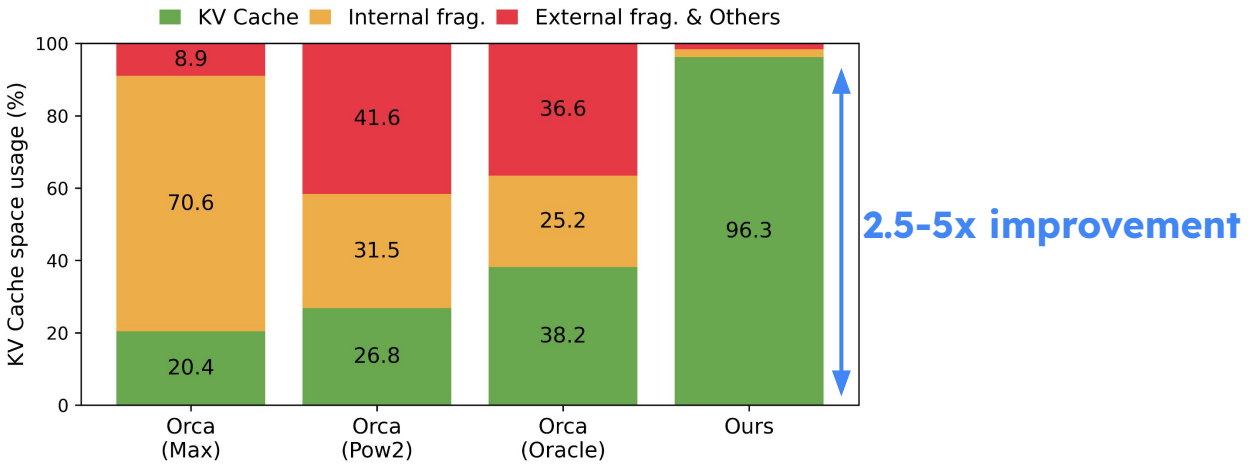


Memory efficiency of PagedAttention

- 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

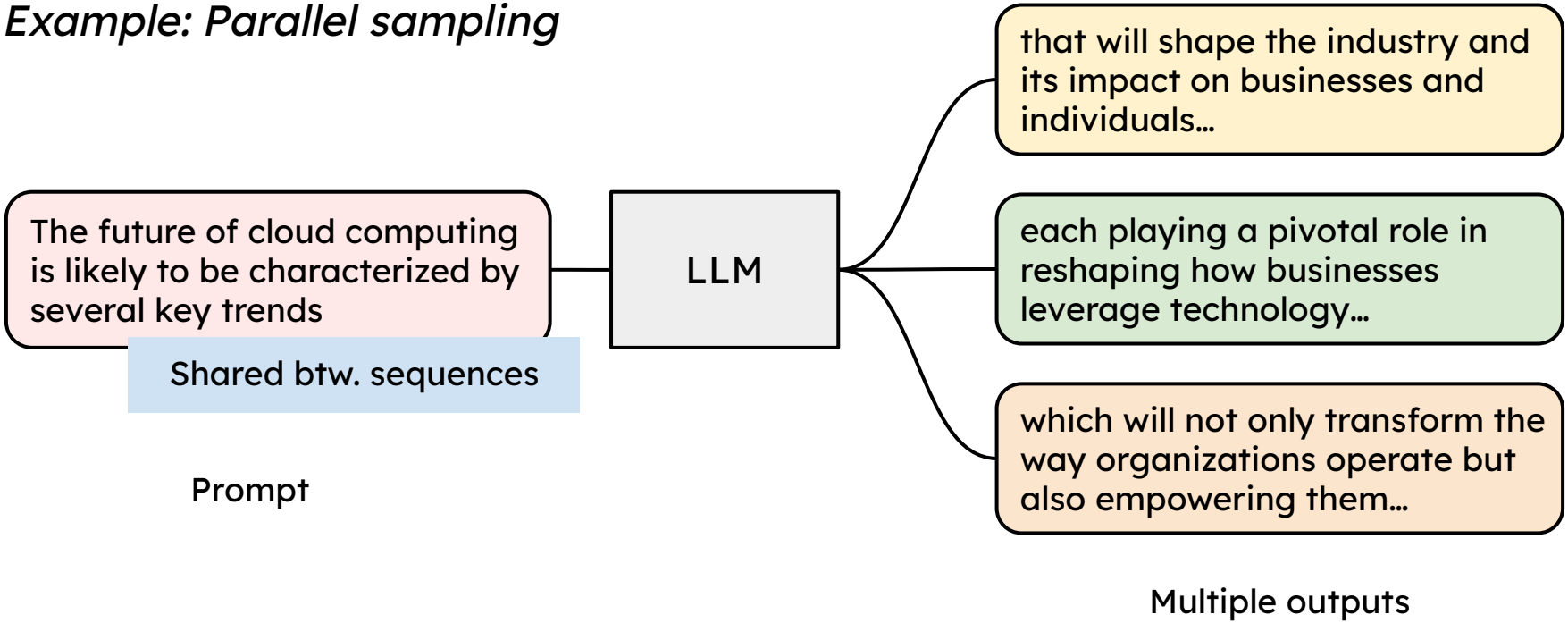
Alan	Turing	is	a
computer	scientist	and	mathematician
renowned			

Internal fragmentation

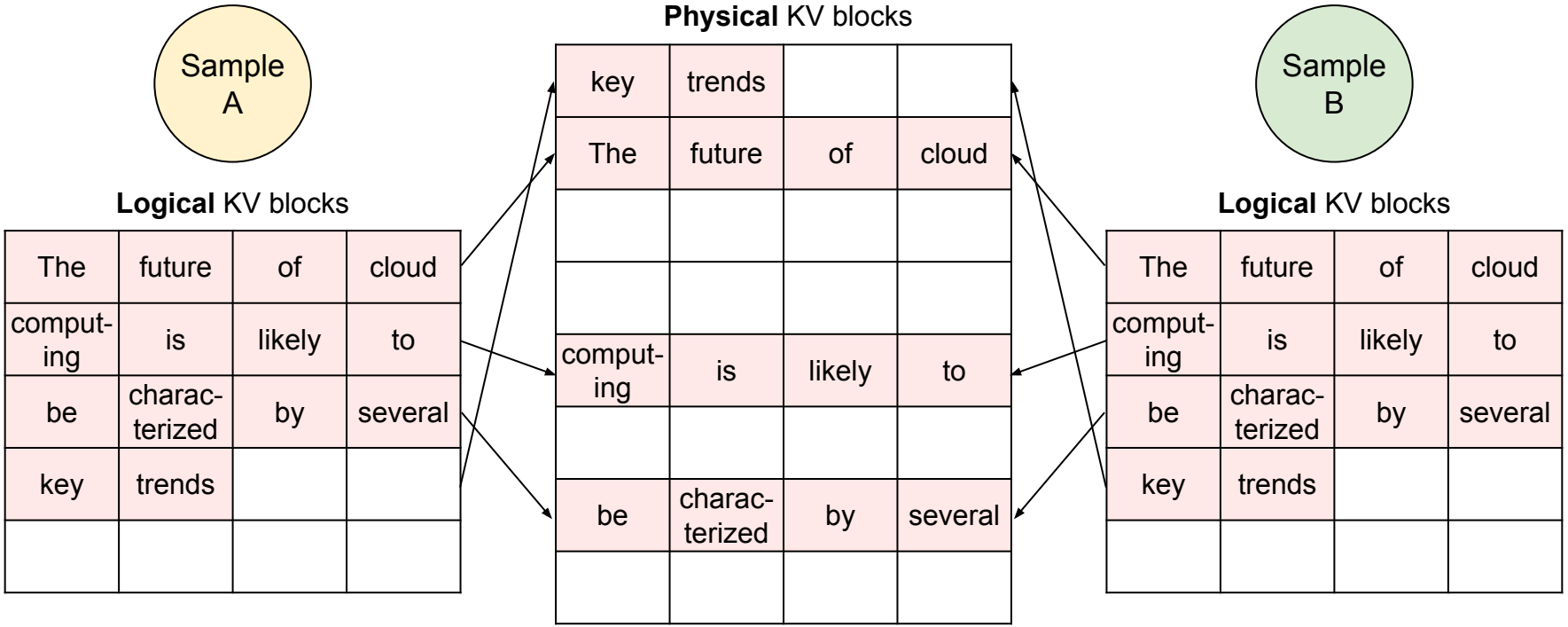


Paging enables sharing

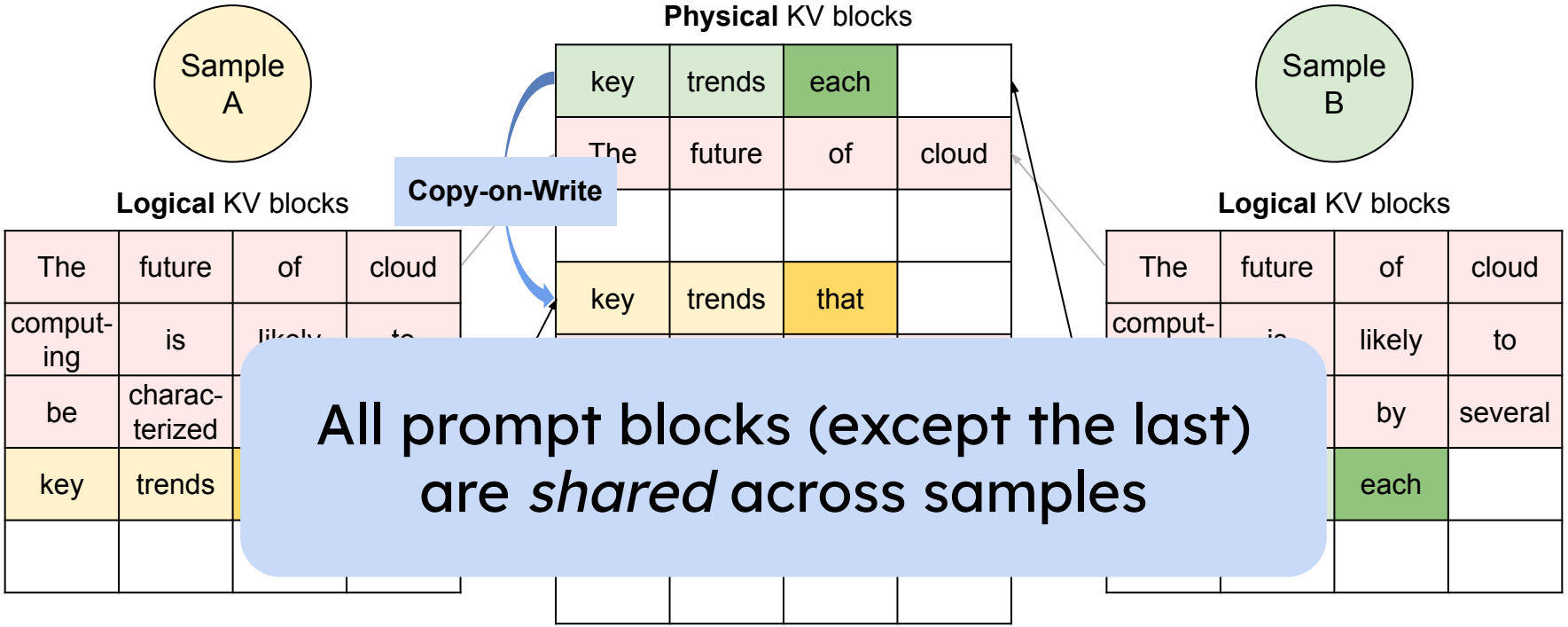
Example: Parallel sampling



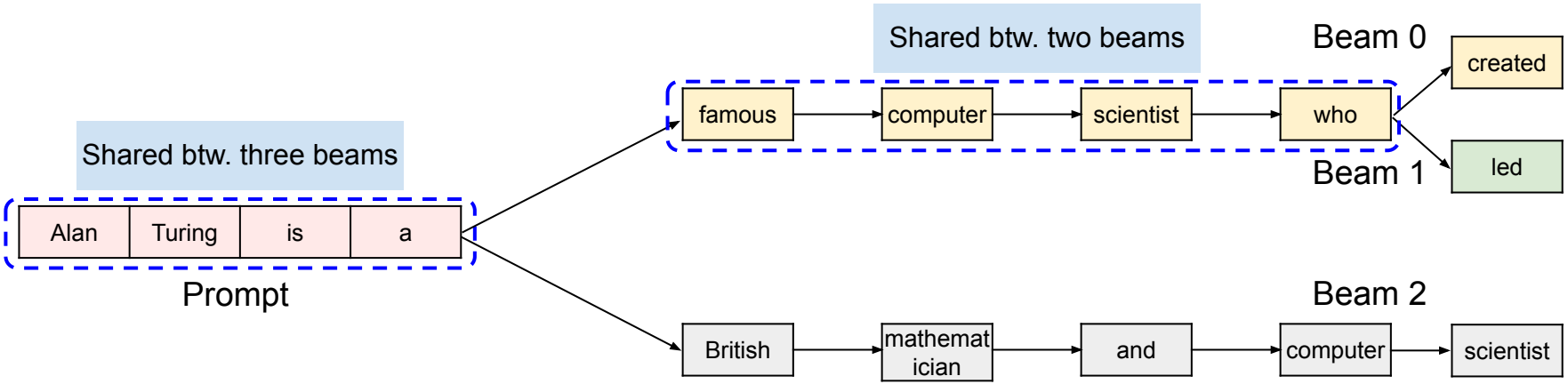
Sharing KV blocks



Sharing KV blocks

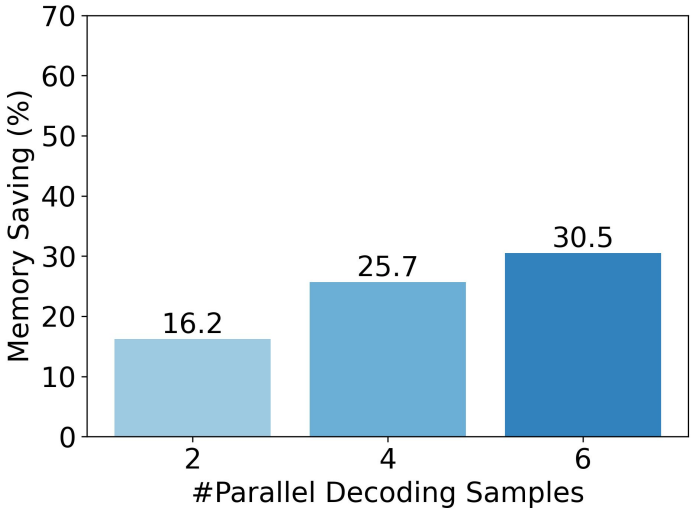


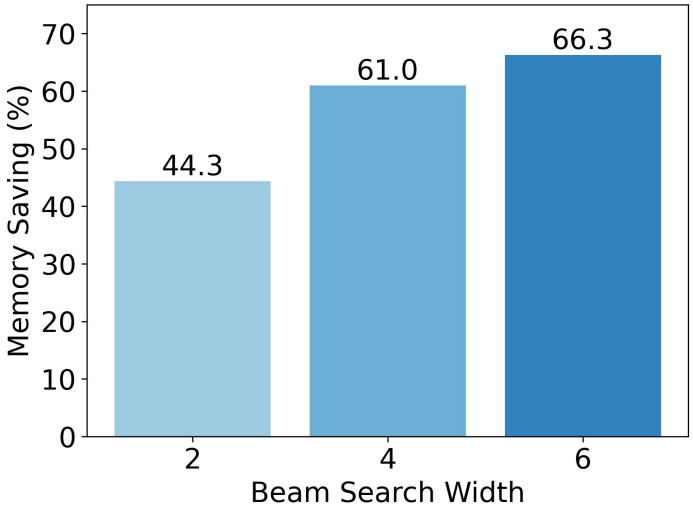
More complex sharing: beam search



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

Memory saving via sharing





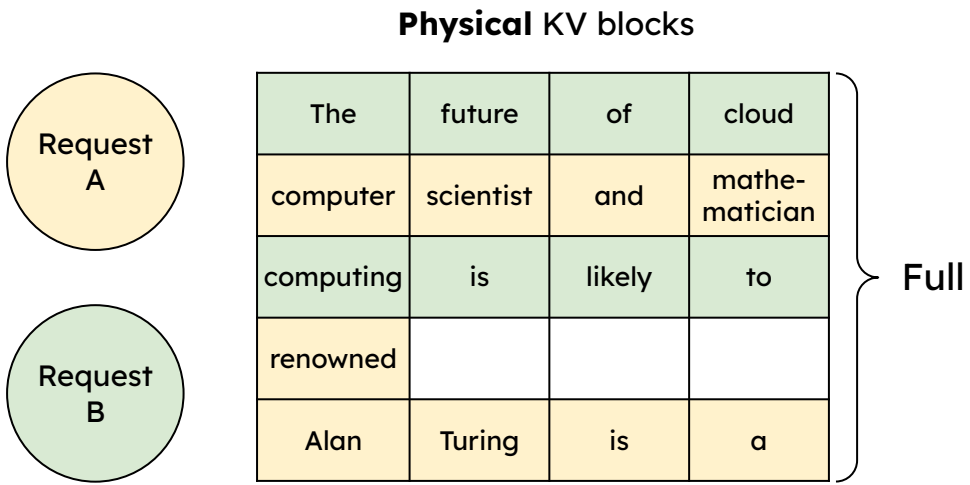
Percentage = (#blocks saved by sharing) / (#total blocks without sharing)
OPT-13B on 1x A100-40G with ShareGPT dataset

How does PagedAttention benefit LLM Serving?

Reduce memory fragmentation with paging

Further reduce memory usage with sharing

Out of KV block memory



Out of KV block memory

Request A

Request B

Physical KV blocks

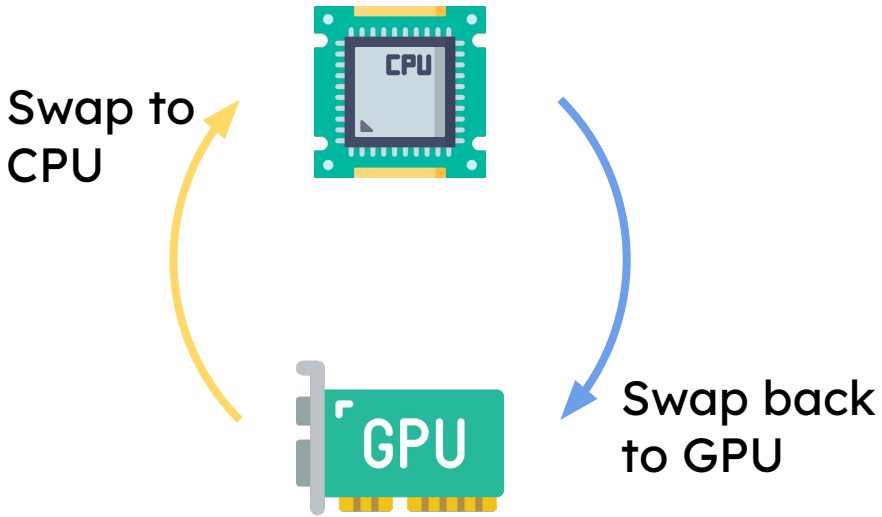
The	future	of	cloud
computer	scientist	and	mathe-matician
computing	is	likely	to
renowned	for		
Alan	Turing	is	a

Cannot allocate a new physical block for Request B

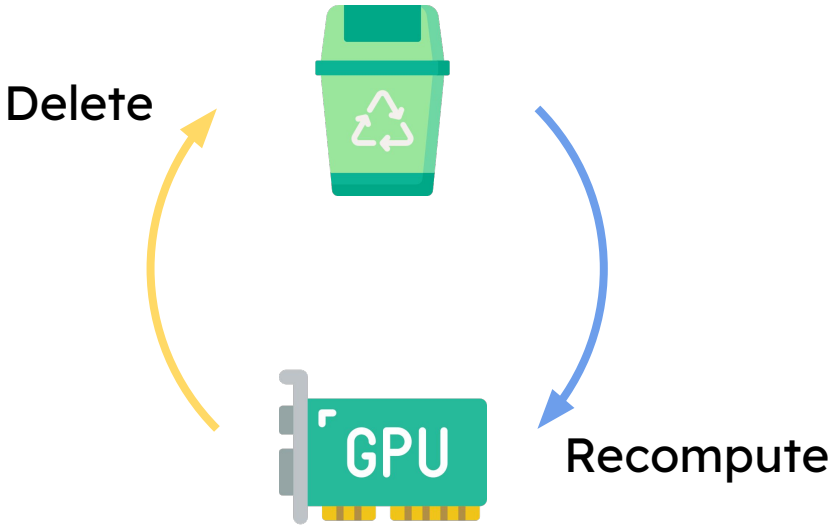
Request preemption & recovery

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

Option 1: Swapping



Option 2: Recomputation



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.

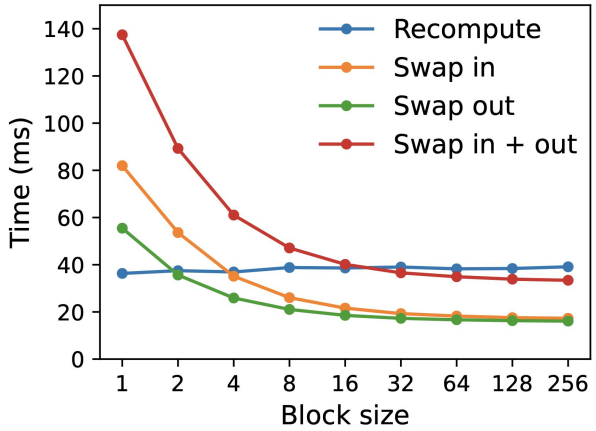


Figure: *Swap/Recomputation latency of 256 tokens.*

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

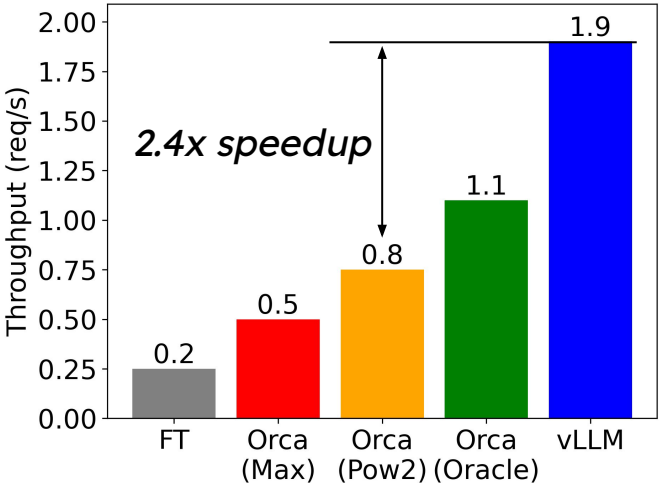
Single-level block table

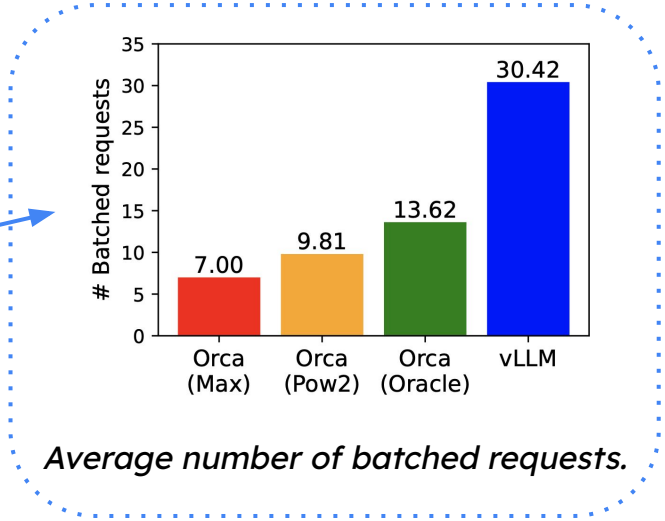
- Block table is tiny compared to the actual data.

Preemption & Recovery

- Request-level preemption
- Recomputation-based recovery

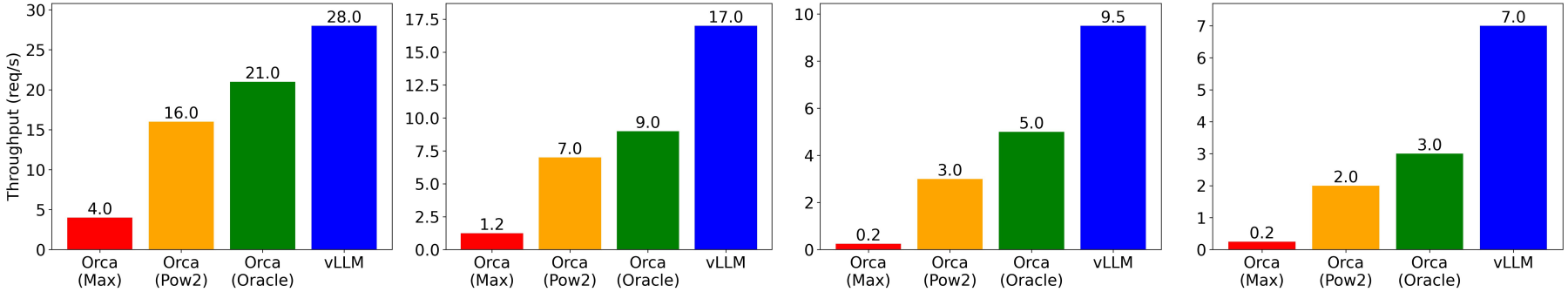
Throughput – greedy decoding





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

Throughput – beam search



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

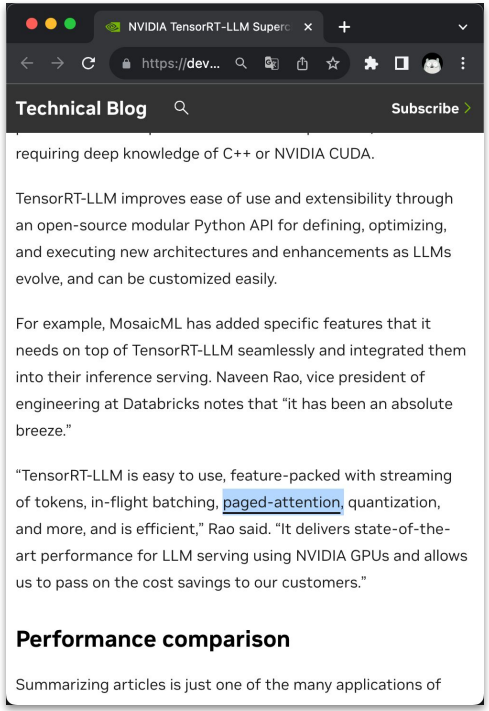
Beam width = 2
***2.4x** speedup*

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

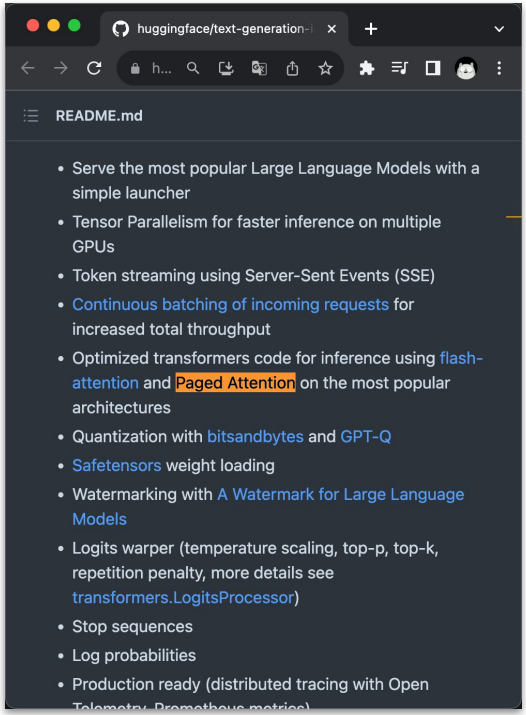
Beam width = 6
***3.5x** speedup*

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

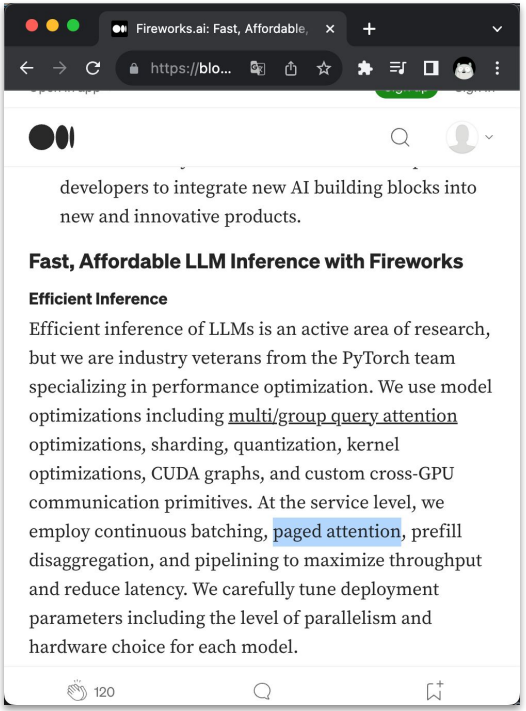
PagedAttention has become the industrial standard



TensorRT-LLM



HuggingFace TGI



Fireworks AI

. . .

Topics

- Paged Attention: Efficient memory management for KV cache
- **vLLM: A real-world open-source inference engine**
- Q&A

What is vLLM?

- vLLM is a popular open-source LLM inference engine to make inference *efficient* and *effortless*
- vLLM is *collaboratively* developed by many organizations

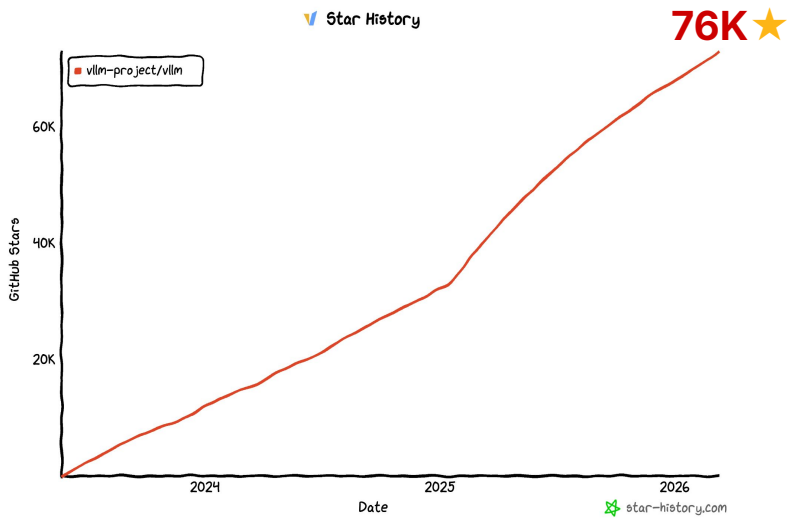
Berkeley UNIVERSITY OF CALIFORNIA

NVIDIA

Red Hat

Meta

inferact



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

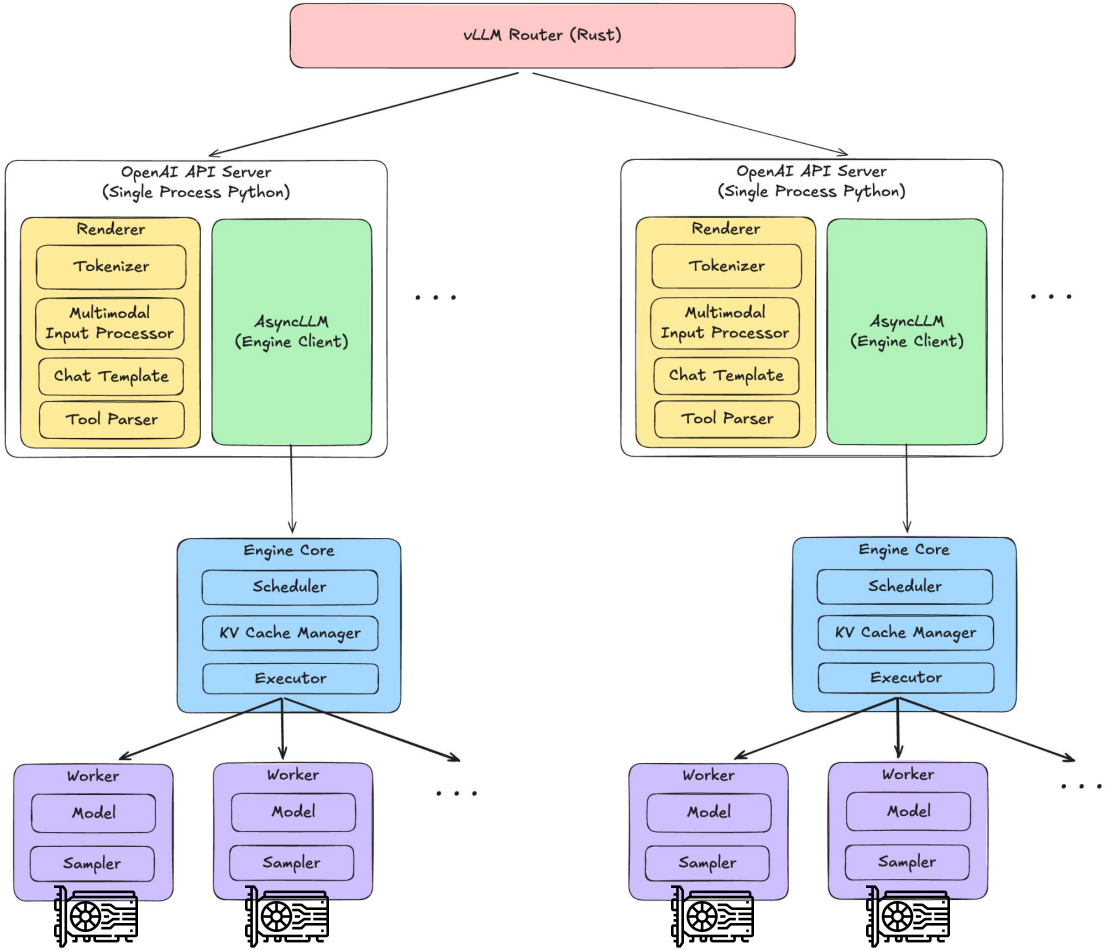
Server

```
$ 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

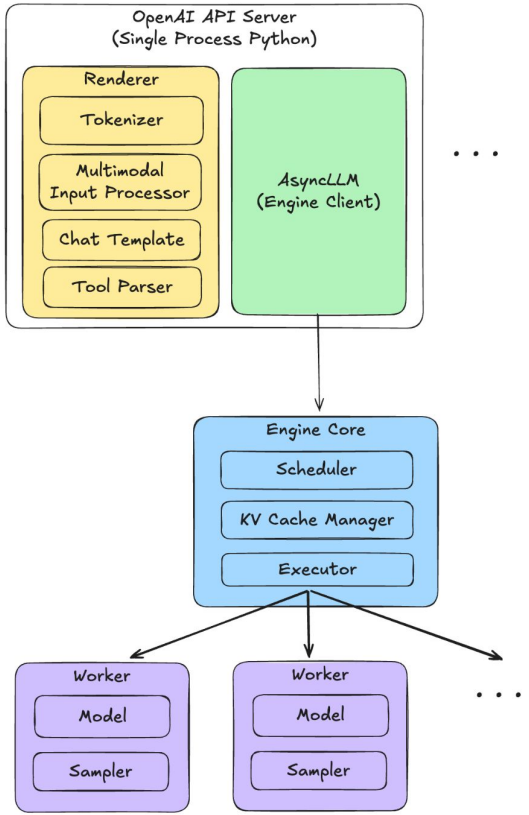
Why does CPU Overhead matter in LLM inference?

- 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%

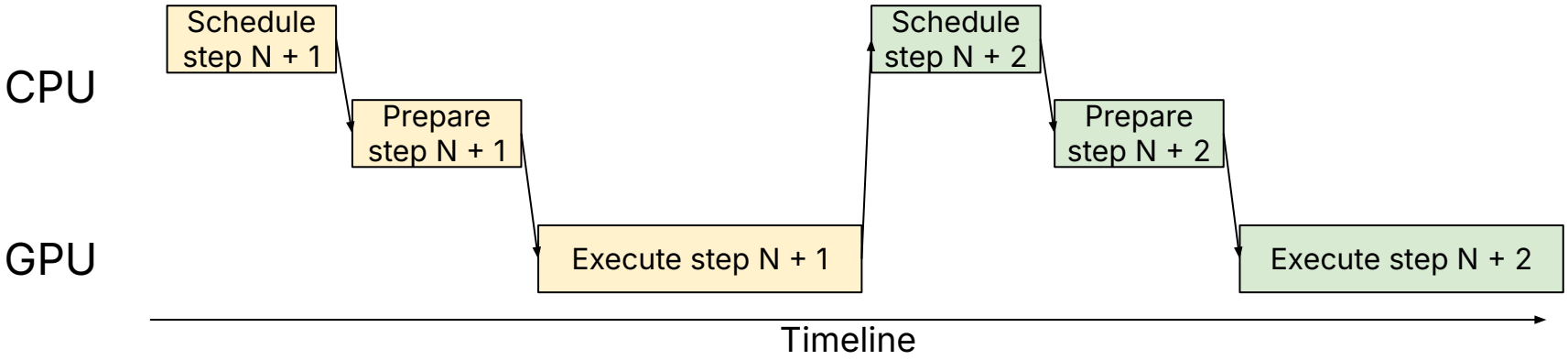
- Unfortunately, **1 ms overhead is pretty easy to hit in Python**

Minimizing CPU Overheads



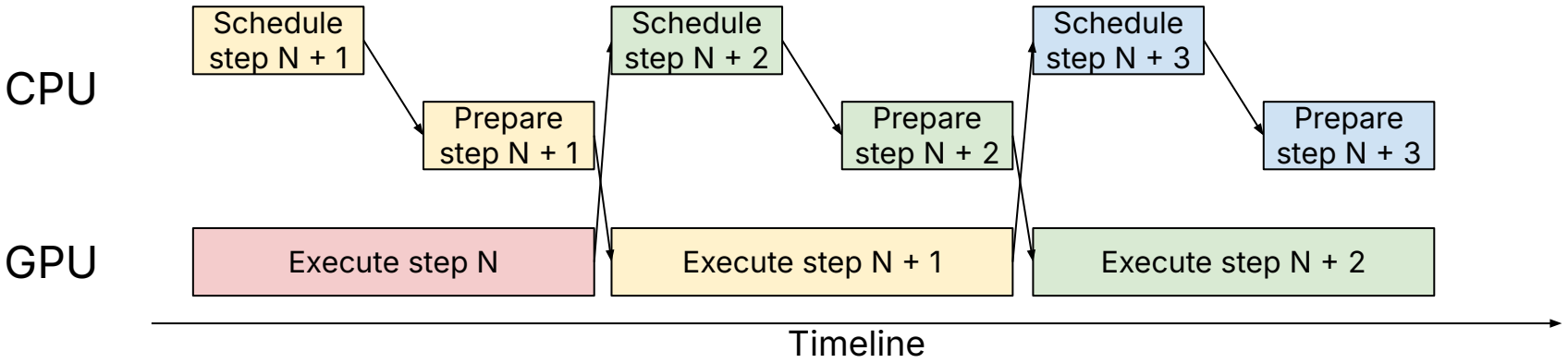
- API Server: Rewrite Python to Rust
- Scheduler & KV Cache Manager
  - **Async Scheduling**
  - Async De-tokenization
- Worker
  - **GPU-native Input Preparation**
  - **CUDA Graphs**

Synchronous Scheduling



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

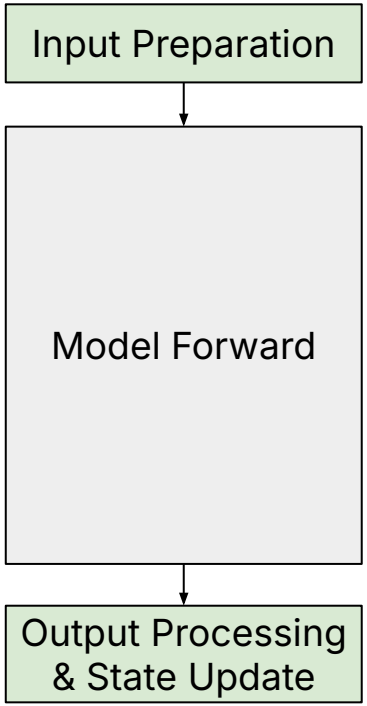
Asynchronous Scheduling



- 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*

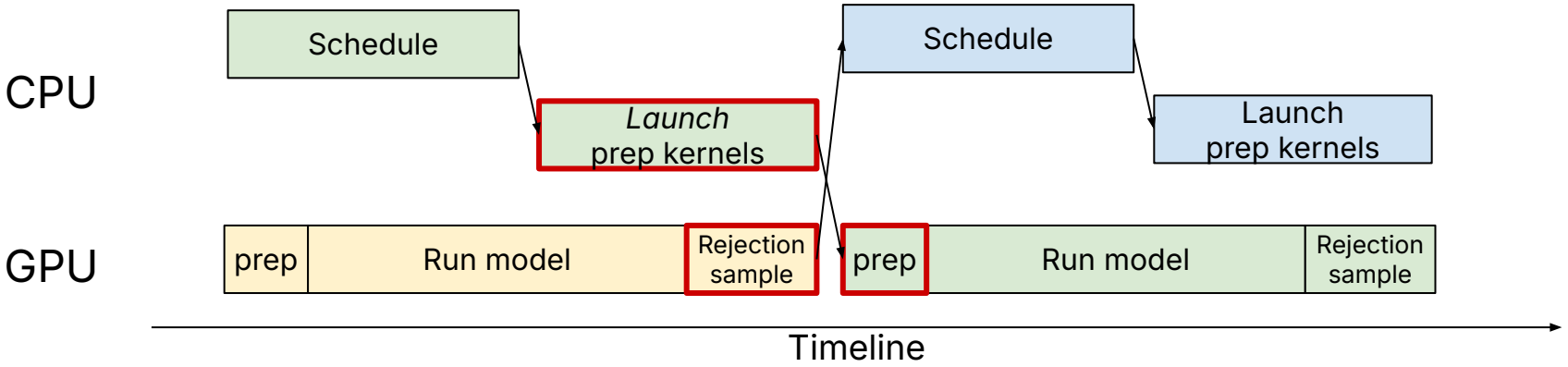
Zhu et al, “NanoFlow: Towards Optimal Large Language Model Serving Throughput“

GPU-Native Input Preparation



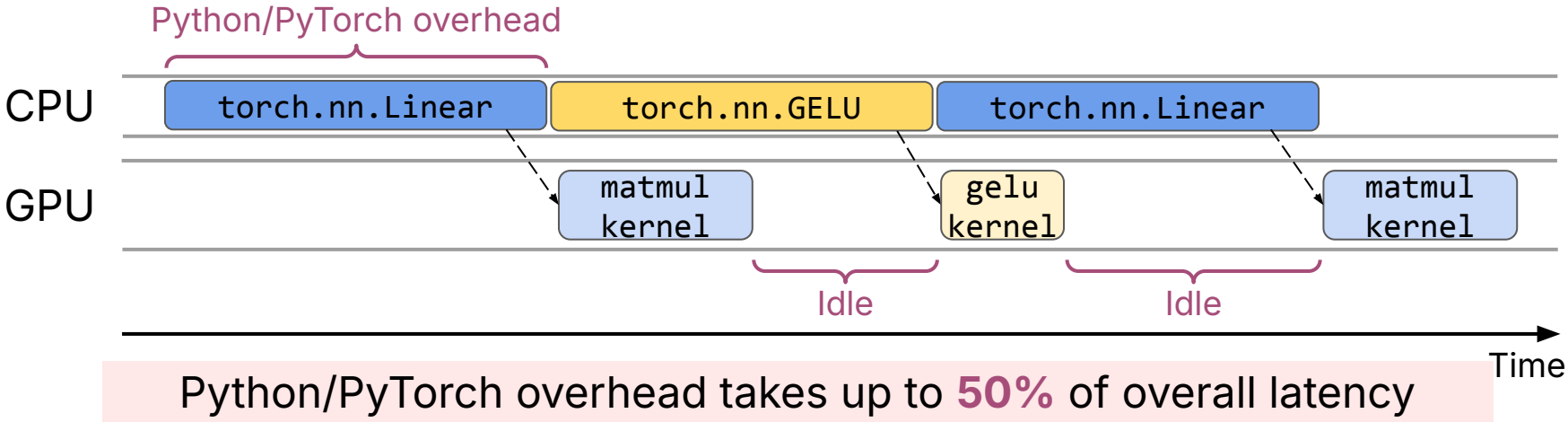
- vLLM involves complex bookkeeping for batching, paged attention, and sampling parameters
- Previously, this logic was implemented with many small PyTorch ops, mostly executed on CPU, which is slow and complex
- MRV2 replaces this with custom Triton kernels, which is fast and clean

Async Scheduling + Spec Decoding

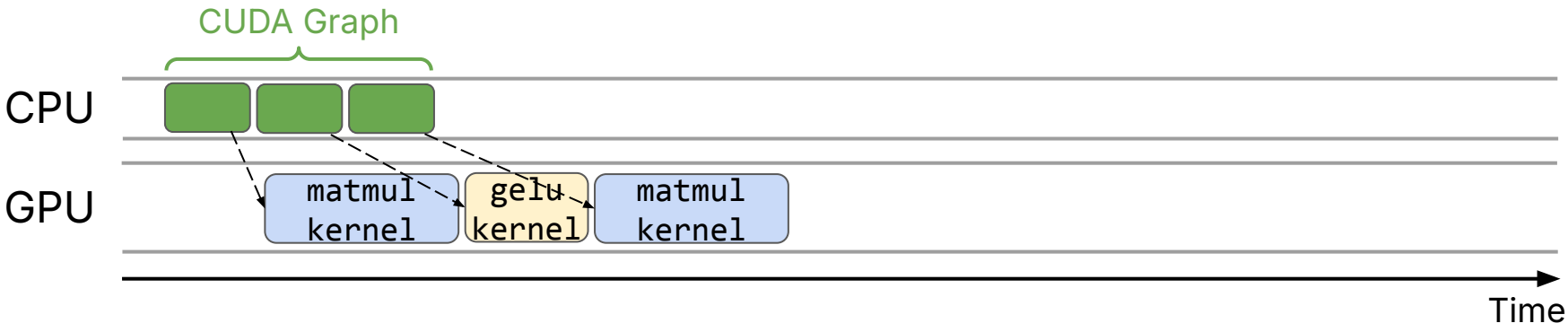


- GPU-native input preparation naturally makes async scheduling and spec decoding compatible
  - The input preparation kernels can see the results of rejection sampling

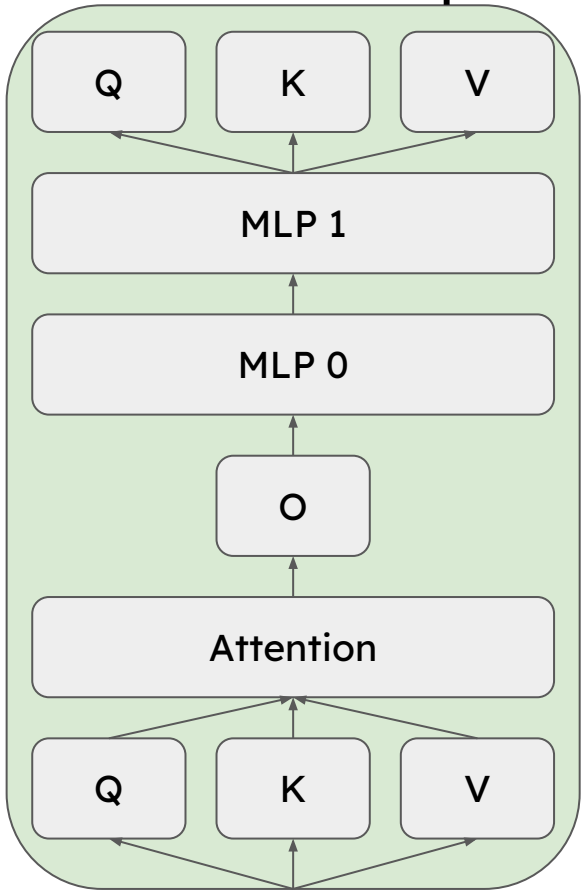
CUDA Graph for Minimizing CPU Overheads



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

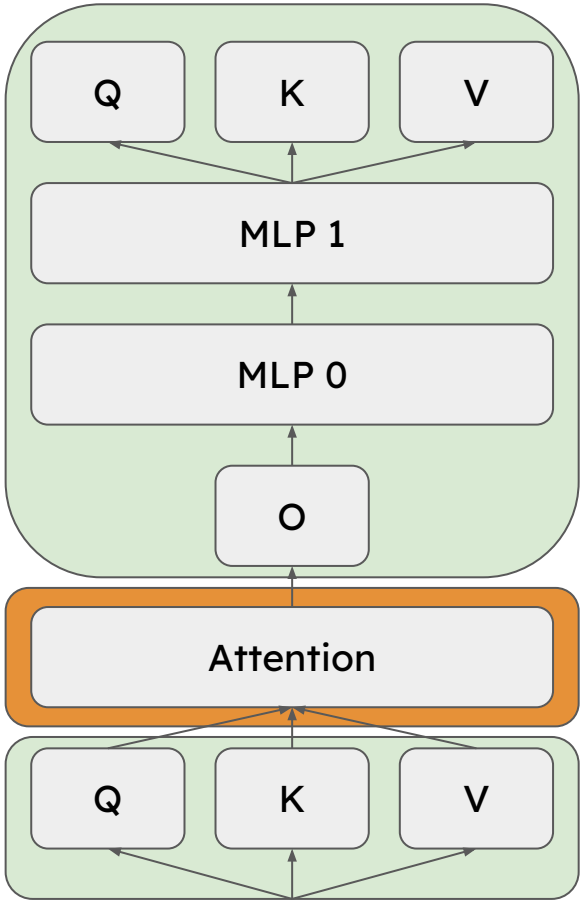
Dynamisms in LLM Inference

- 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

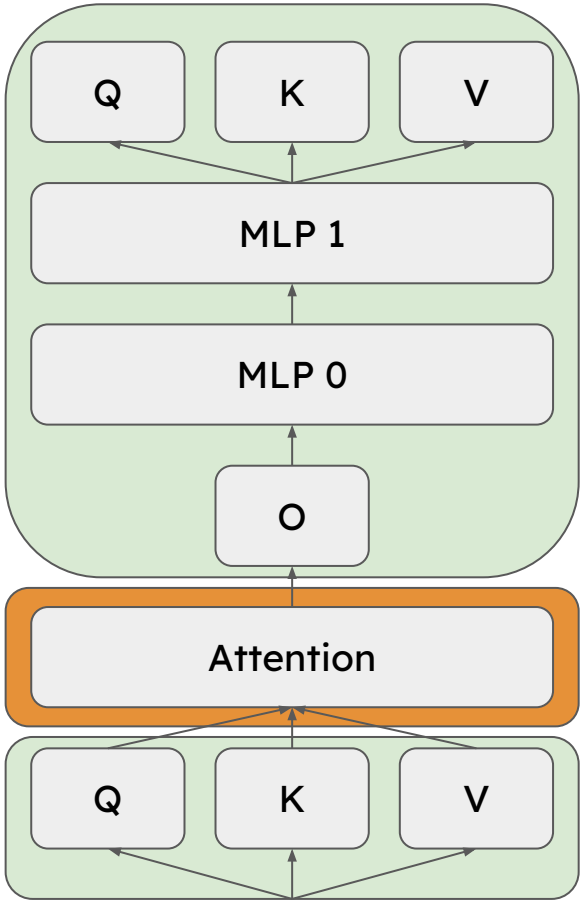


Token-wise Operations
(Stateless & Easy to deal with)

Dynamic & Stateful & Complex

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

Our solution: Piecewise CUDA Graphs



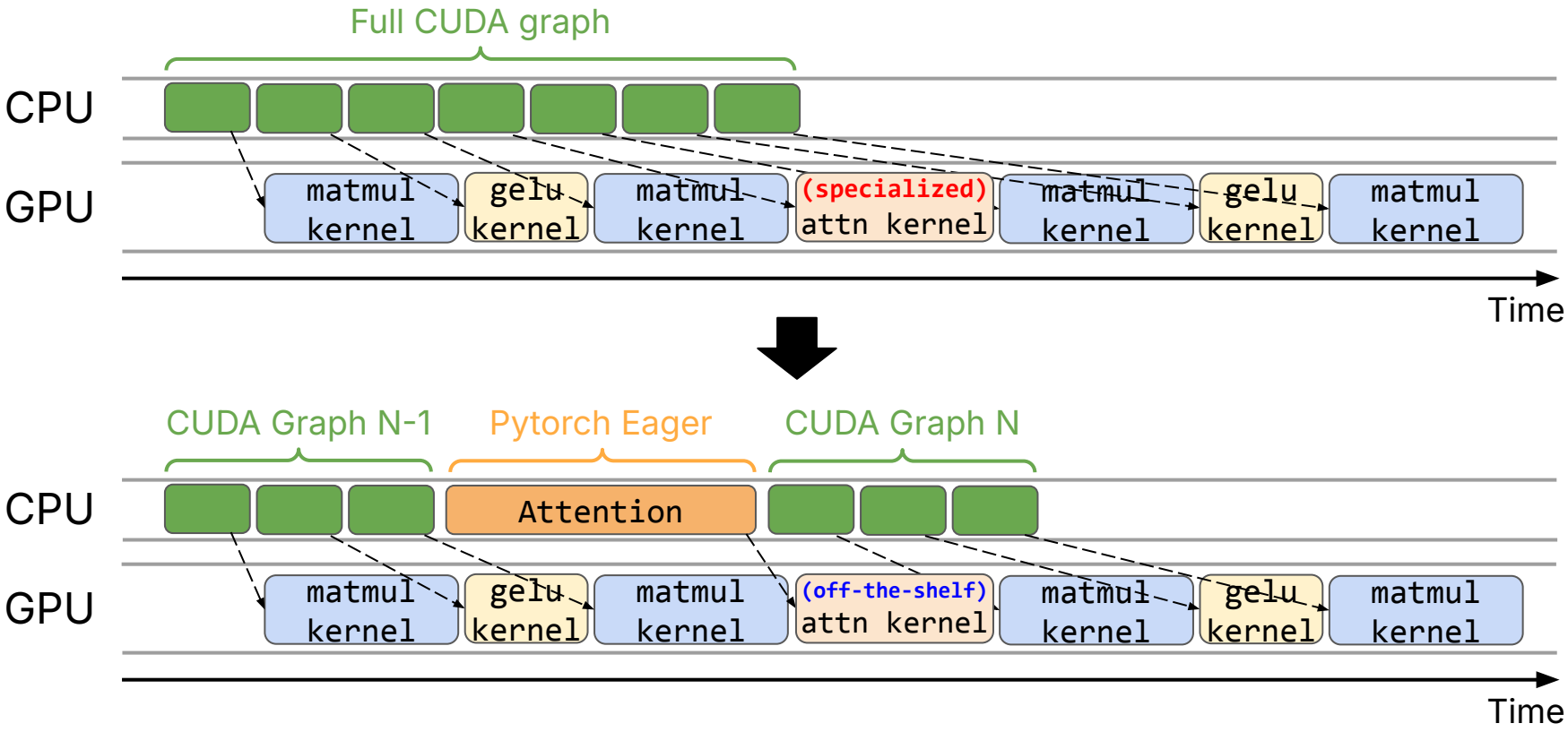
CUDA graph N

PyTorch Eager

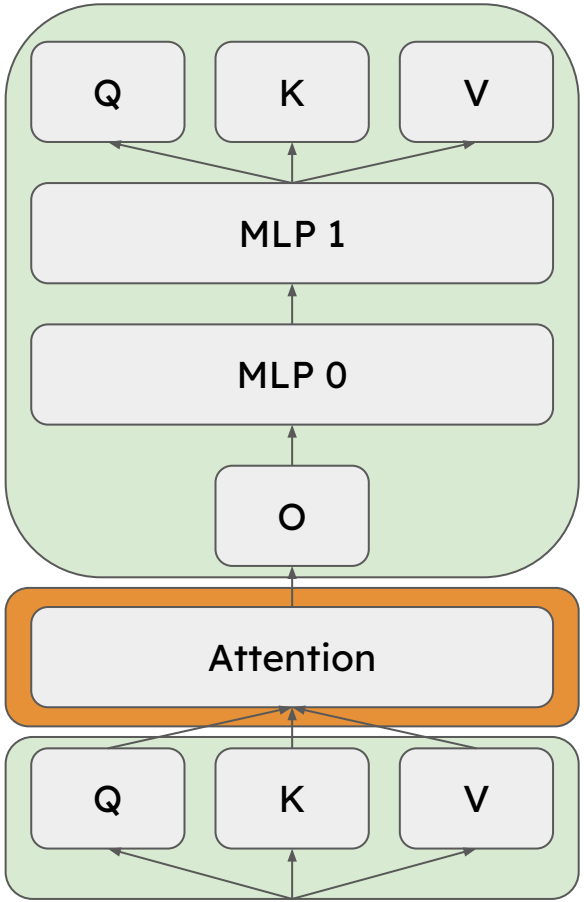
CUDA graph N-1

Graph split using
torch.compile

Full vs. Piecewise CUDA graphs

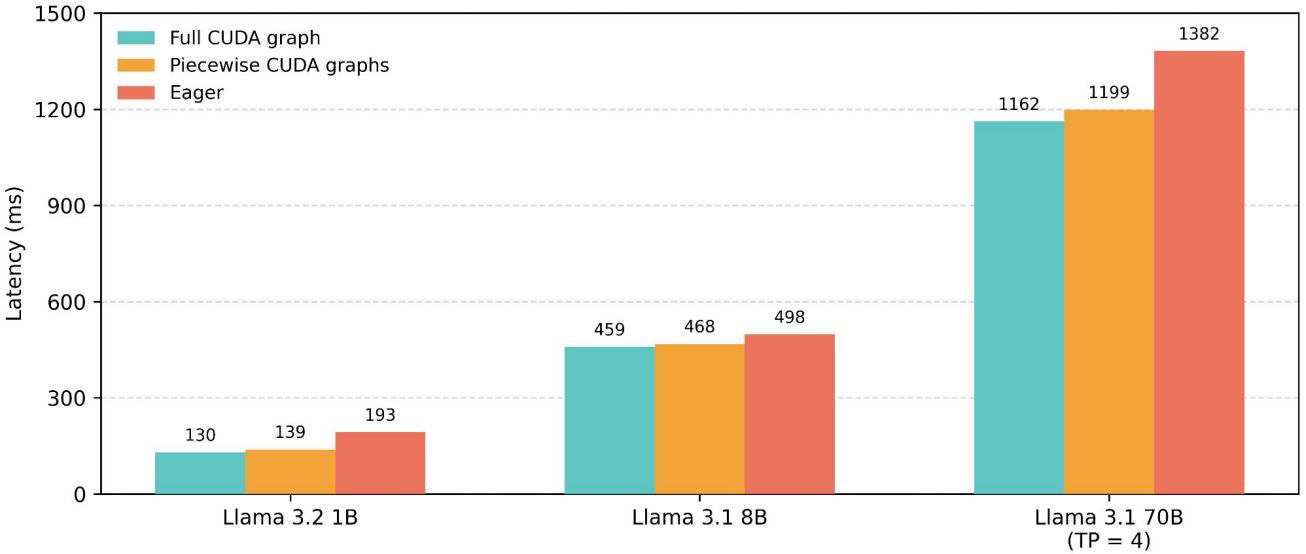


Piecewise CUDA Graphs



- 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?

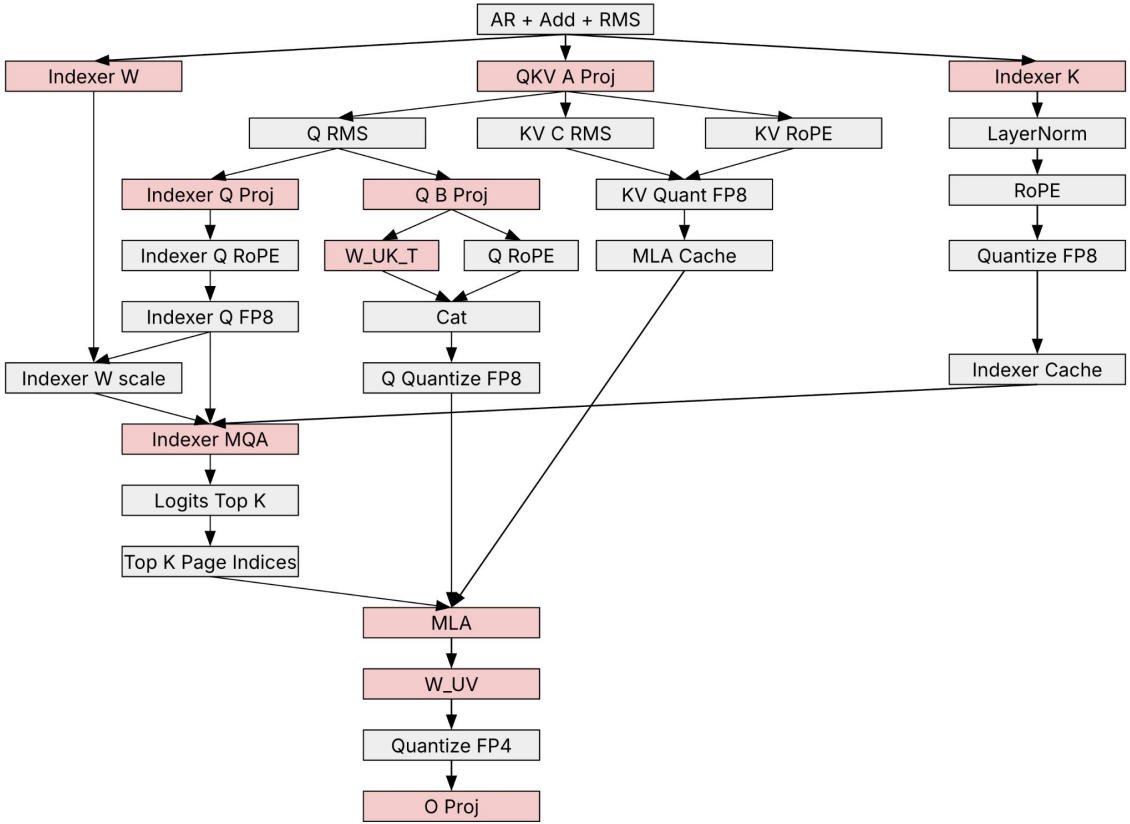
1. Minimizing CPU overheads
2. **Efficient GPU kernels**
3. Model parallelisms
4. Efficient memory management & caching

GPU Kernels in vLLM

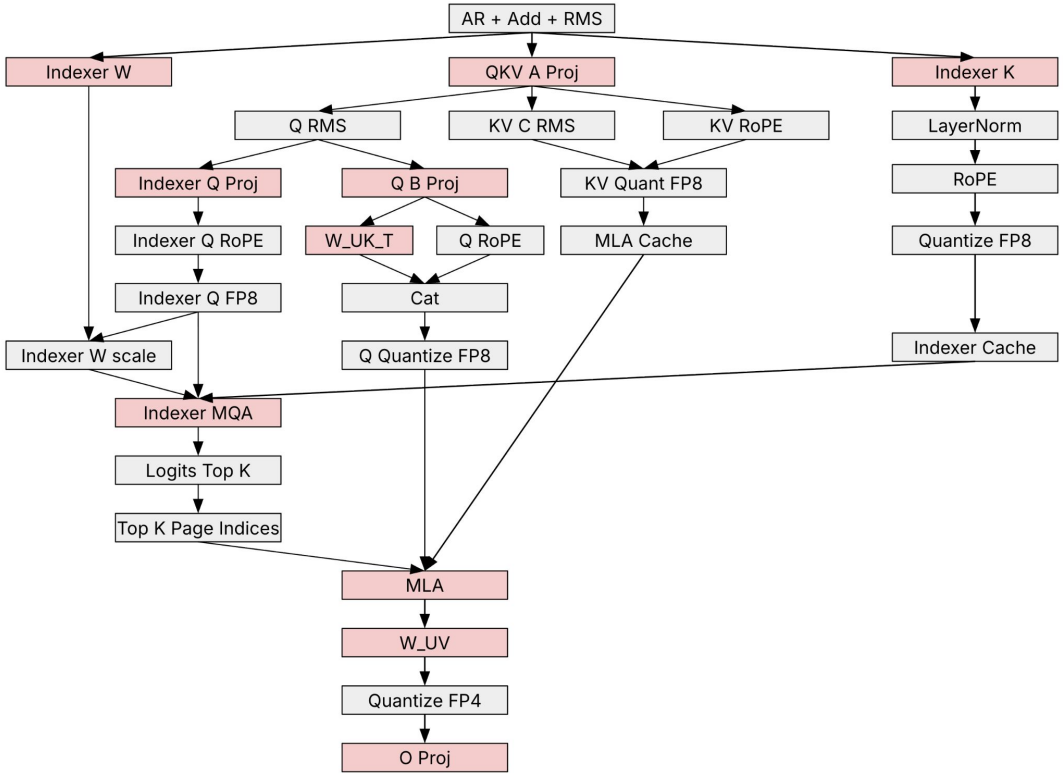
- For complex or performance-critical kernels, vLLM leverages FlashInfer or other libraries
  - vLLM provides `AttentionBackend` and `FusedMoE` abstractions to plug in different kernel implementations

- 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

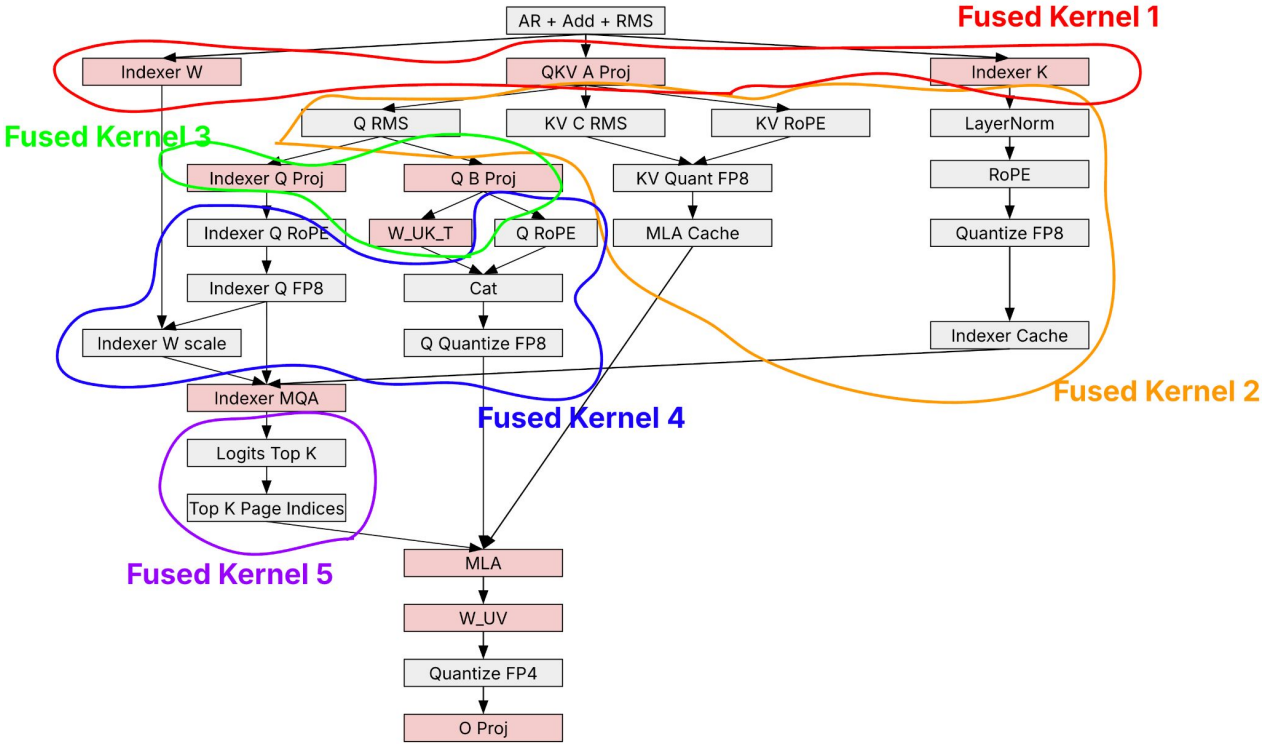
Example: DeepSeek V3.2

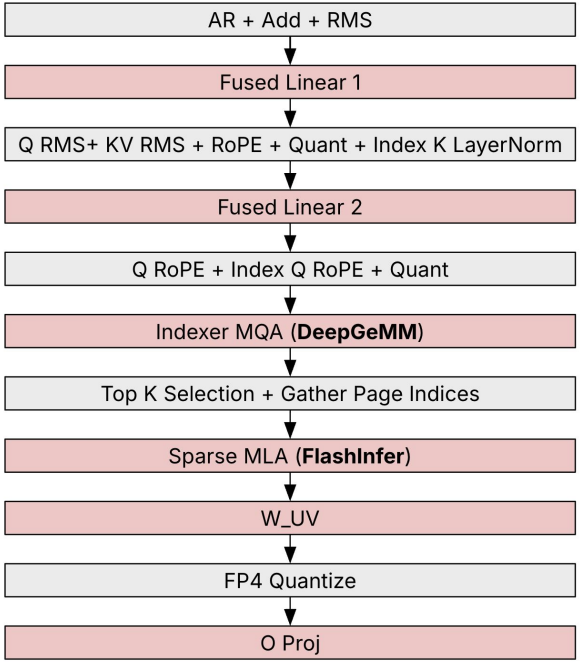


Example: DeepSeek V3.2



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

Things to consider for parallelism

- Theoretically, all tensors can be arbitrarily distributed and all dimensions of all tensors can be sharded.
- Goal 1: Minimize communication overhead
  - Limit the number of bytes transferred
  - Hide the overhead via overlapping with computation
  - Network is also heterogeneous
    - Latest NVLink: 1.8TB/s
    - Infiniband: 50GB/s
- Goal 2: Minimize load imbalance
  - A system is often bottlenecked by the slowest component

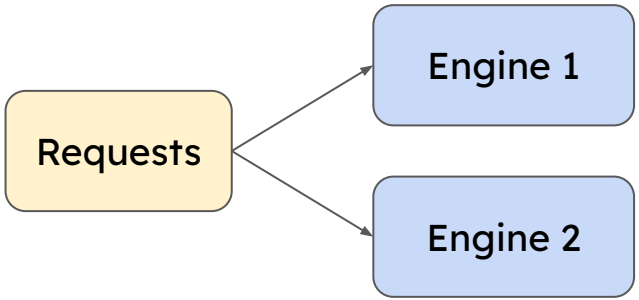
Data parallelism

- Replicated engines
- A load-balancer routes different requests to different engines based on
  - Engine load
  - KV cache memory usage
  - (Prefix) Caching

👍 No communication between engines

👎 No parameter memory saving

👎 No reduction in latency



Tensor parallelism

- Partition the weights in the linear layers

👍 Reduce the serving latency if the network is fast

👍 KV Cache can also be partitioned

👎 Heavy communication. Often require NVLink to work effectively.

👎 Limited by the number of KV heads

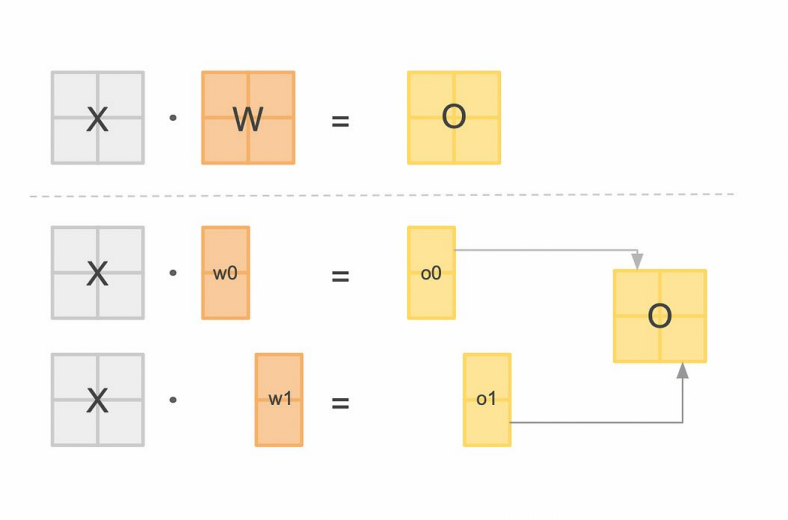


Image source: https://www.titanml.co/resources/harmonizing-multi-gpus-efficient-scaling-of-llm-inference

Expert parallelism

- Distribute different experts to different nodes

👍 Friendly for GPU kernels

👍 Smaller communication than TP

👎 Only apply for MoE layers. Attention needs other parallelism (e.g., DP).

👎 Load imbalance

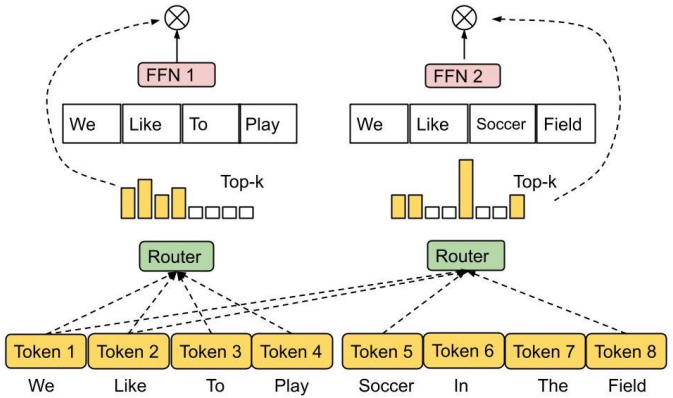


Image source: https://research.google/blog/mixture-of-experts-with-expert-choice-routing/

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.

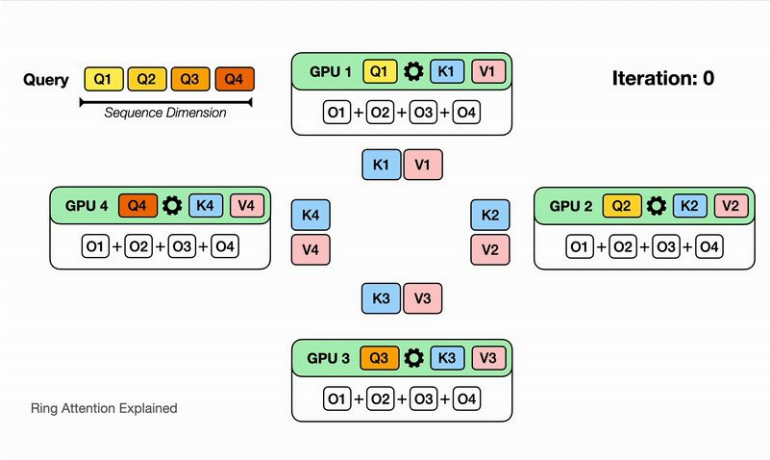


Image source: https://coconut-mode.com/posts/ring-attention/

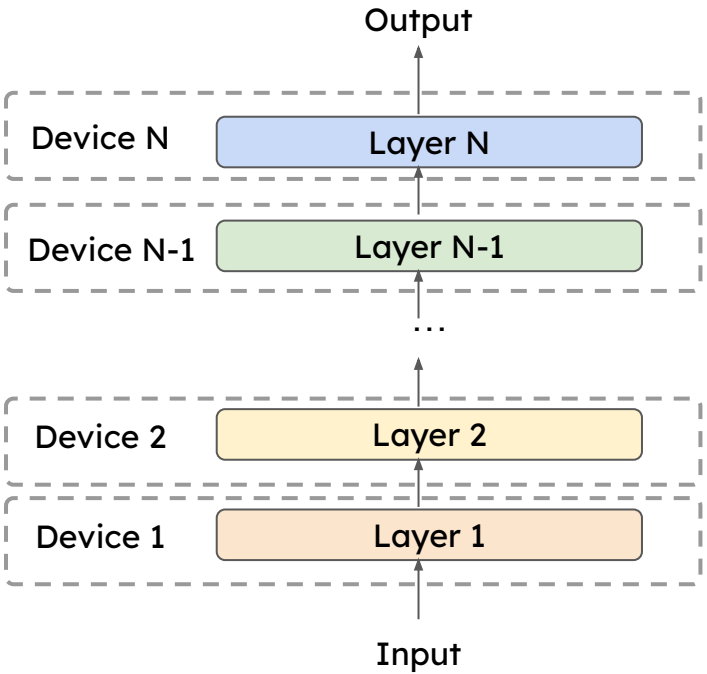
Pipeline parallelism

- Distribute the layers to different GPUs and pipeline the execution across devices

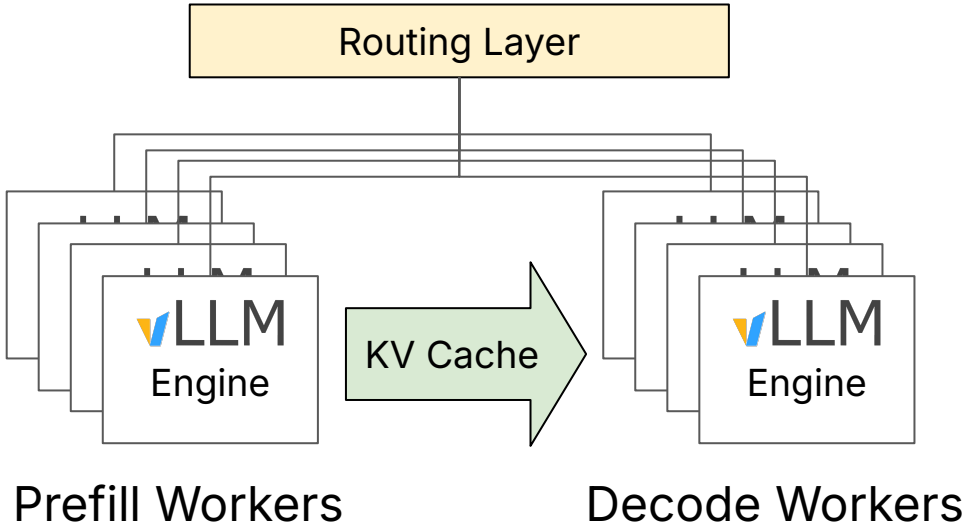
👍 Lowest communication overhead

👎 Increased latency

👎 Load imbalance between stages

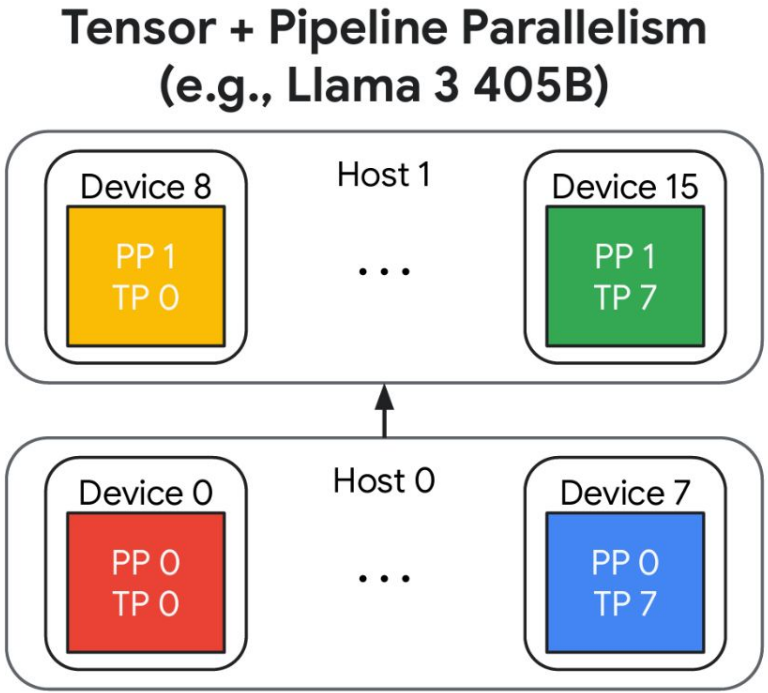


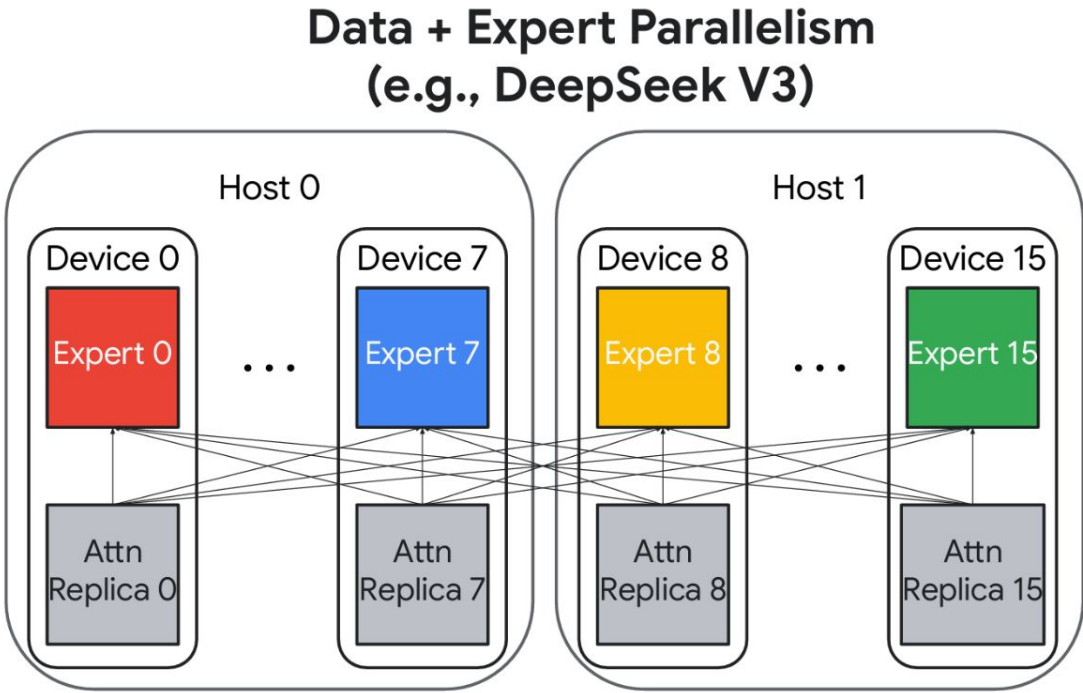
Prefill Disaggregation (PD)



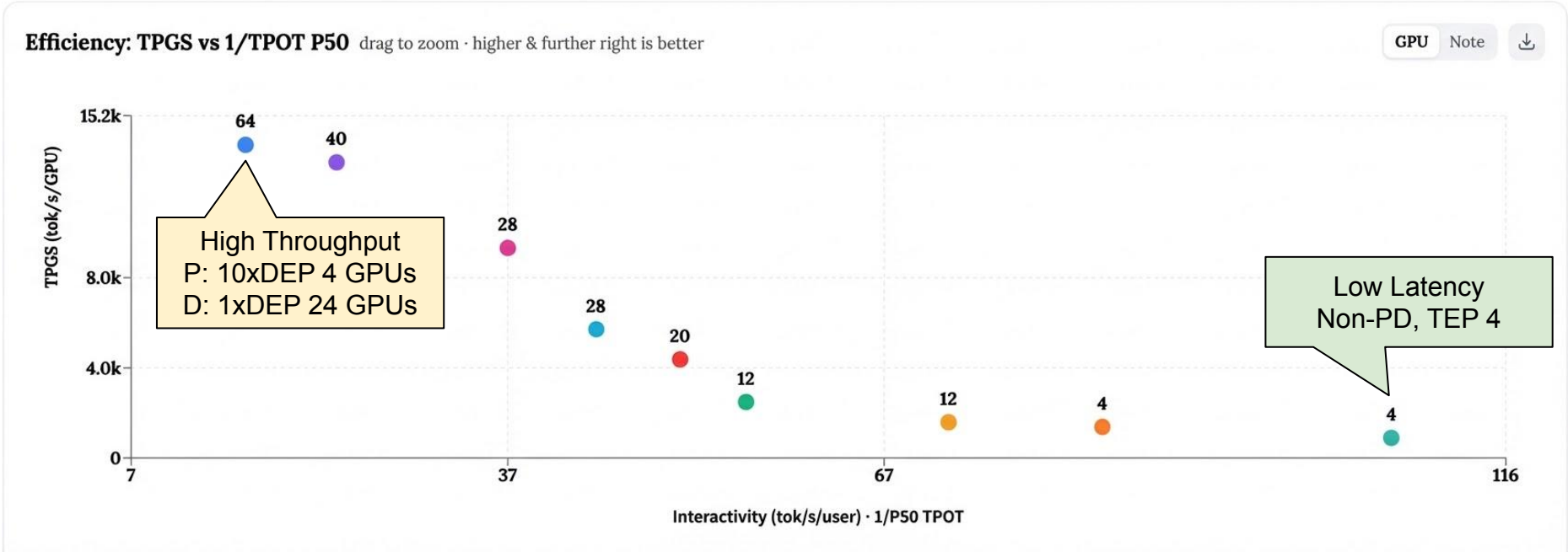
- Why PD?
  - More controllable TTFT & TPOT
  - Optimize prefill & decode independently
- Routing backends
  - vLLM Router, NVIDIA Dynamo, llm-d, Ray Serve LLM
- NIXL for KV cache transfer

Mixed Parallelism





Case Study: Kimi K2.5 (NVFP4)

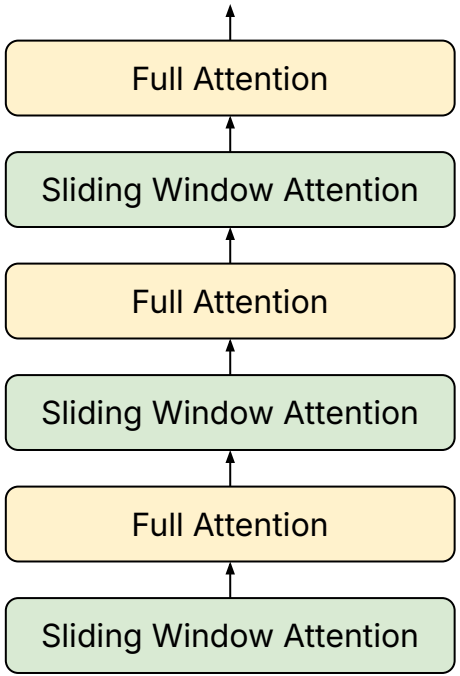


8K input & 1K output, GB200

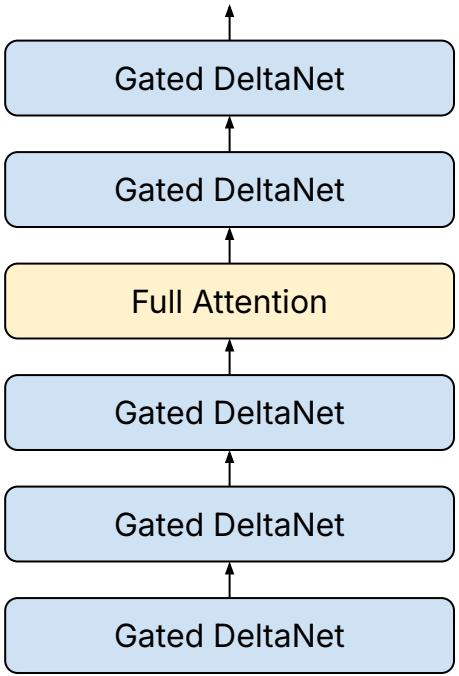
How does vLLM optimize LLM inference?

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

Hybrid Memory Allocator



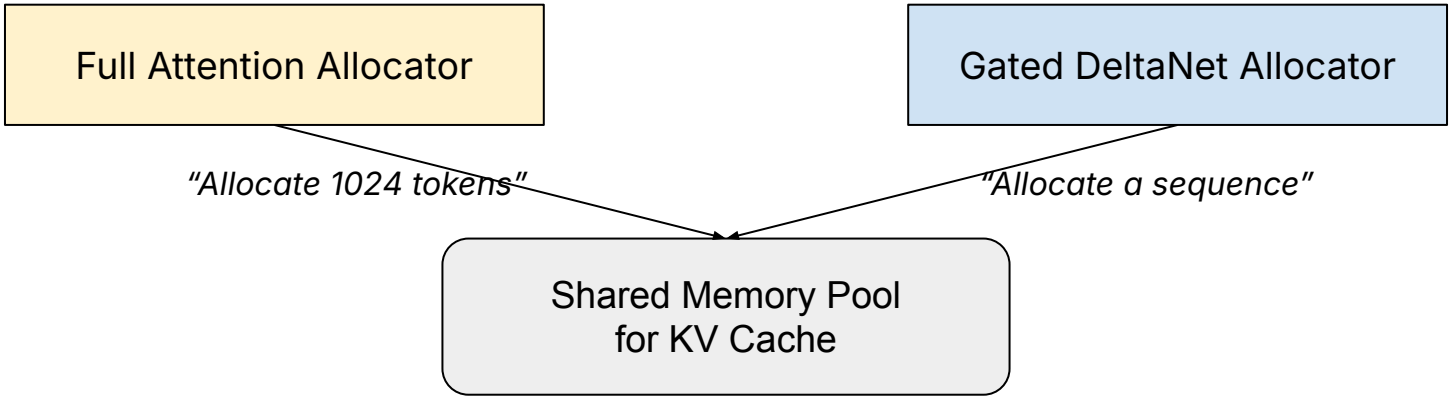
GPT-OSS



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

- vLLM provides a *clean core* that modularizes models and enables various features with zero CPU overhead
- vLLM provides *full-stack performance optimizations* for state-of-the-art models from kernels to model parallelisms


vllm.ai


slack.vllm.ai


x.com/vllm_project


inferact.ai