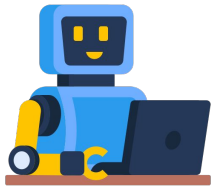
PagedAttention & vLLM for Efficient LLM Inference

Woosuk Kwon



Contents

- PagedAttention
- vLLM: A real-world open-source LLM serving system
  - Models
  - Parallelism
  - Inference optimizations
- Q&A

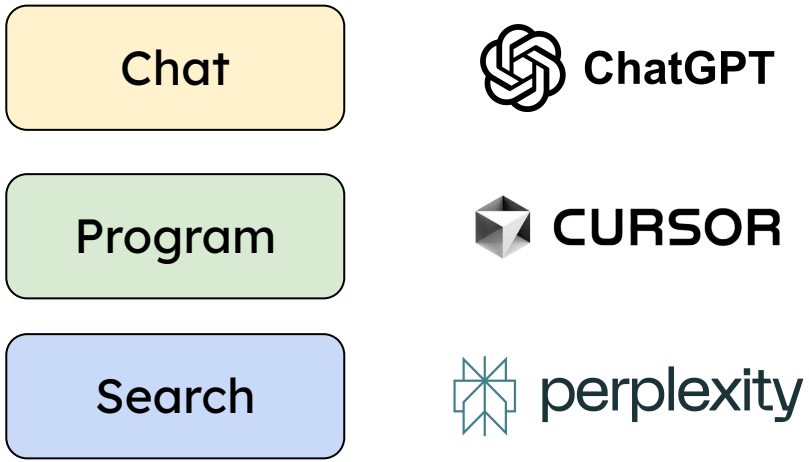
Contents

- PagedAttention

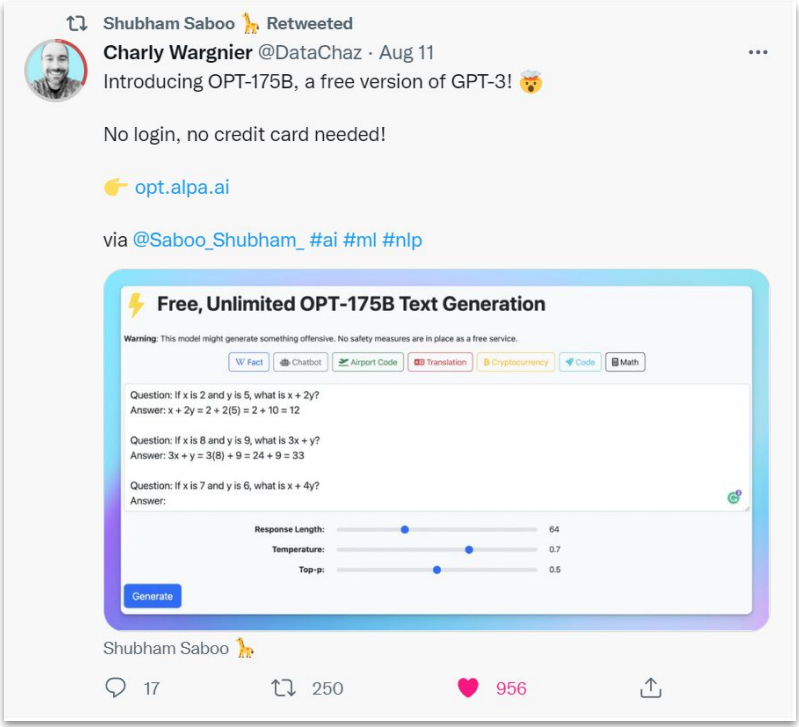
- vLLM: A real-world open-source LLM serving system
  - Models
  - Parallelism
  - Inference optimizations

- Q&A

The era of Large Language Models (LLMs)



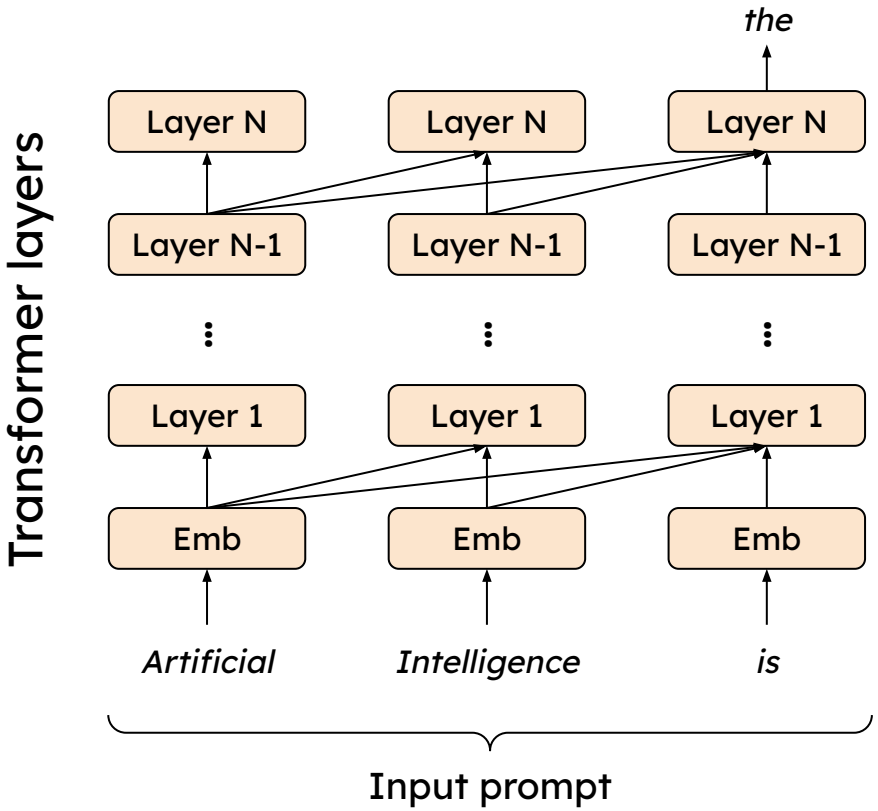
OPT-175B serving demo (Fall 2022)



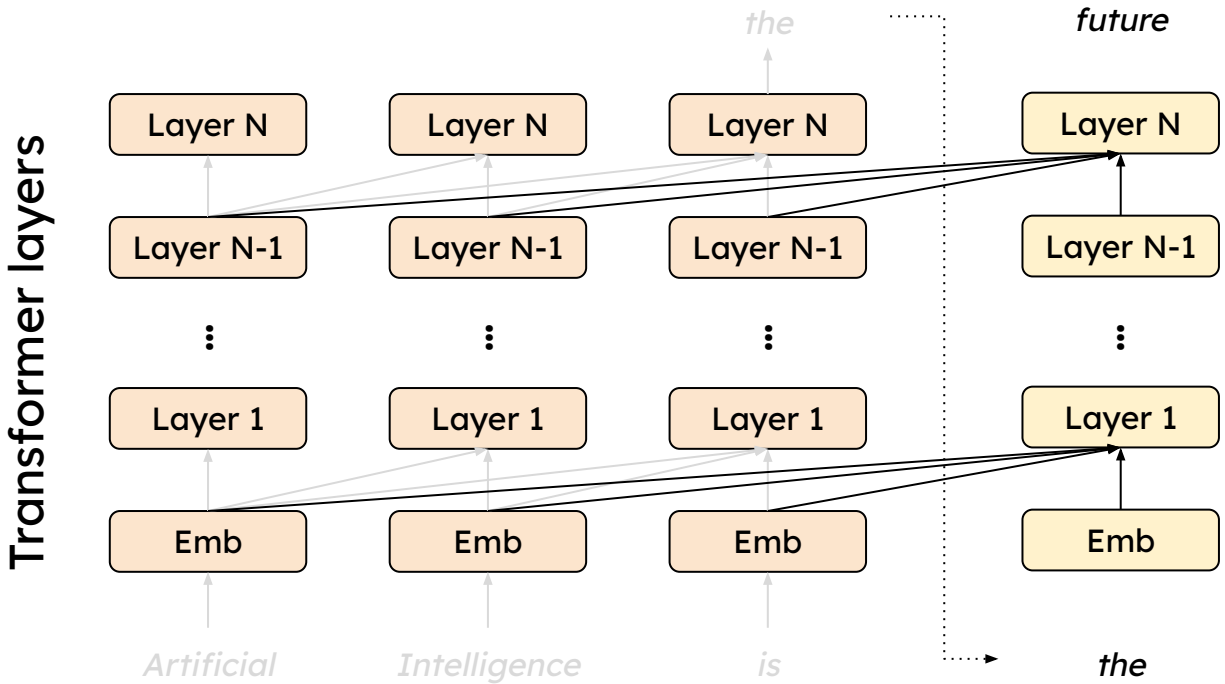
Serving LLMs is (surprisingly) slow and expensive

- A single A100 GPU can only serve < 1 request per second
  - Moderate size of model (13B parameters) and input
- A ton of GPUs are required for production-scale LLM services

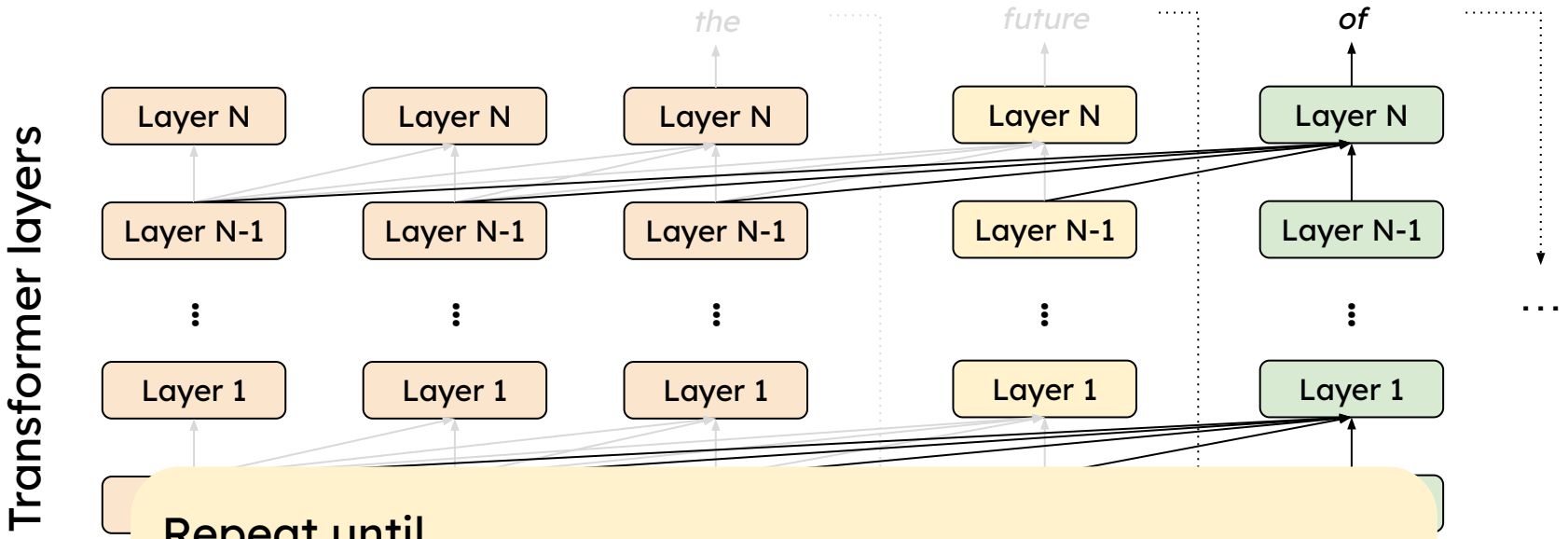
LLM inference process



LLM inference process



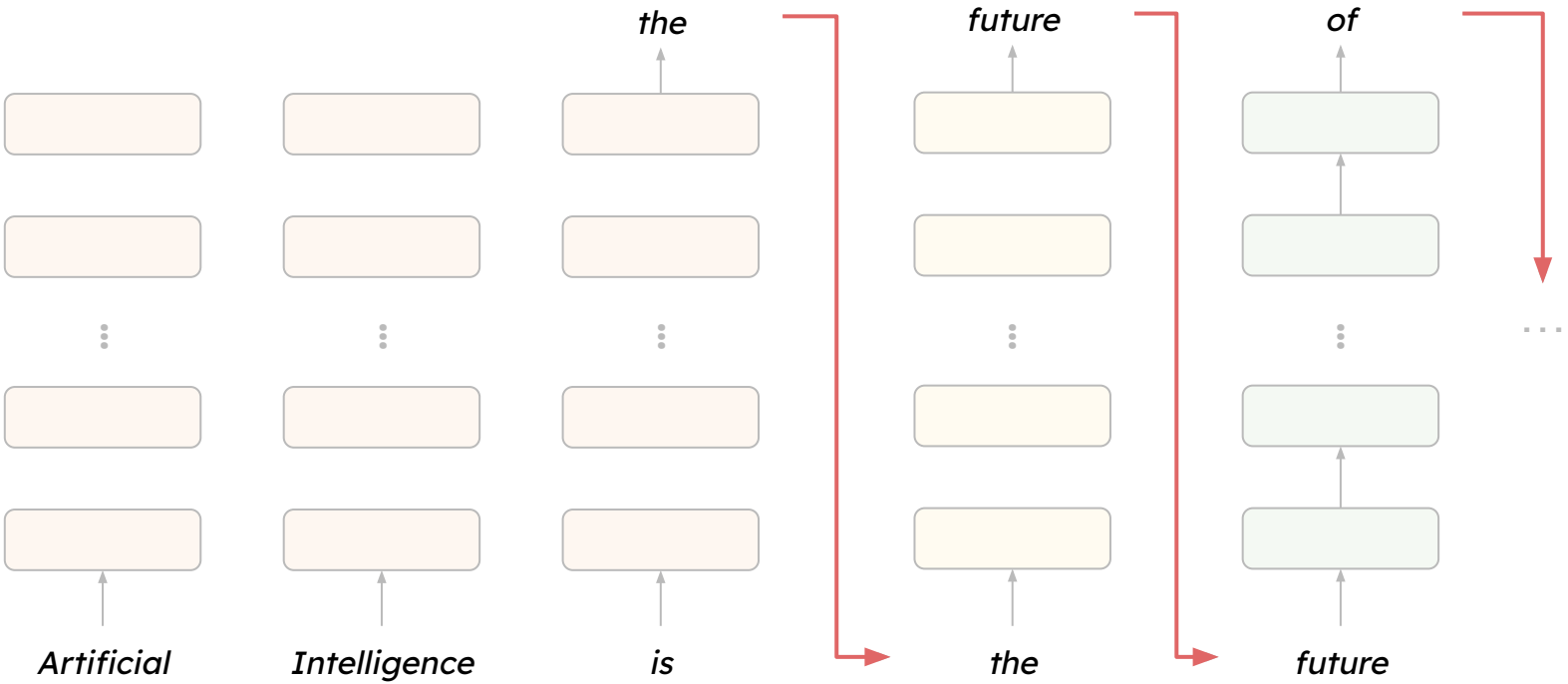
LLM inference process



Repeat until

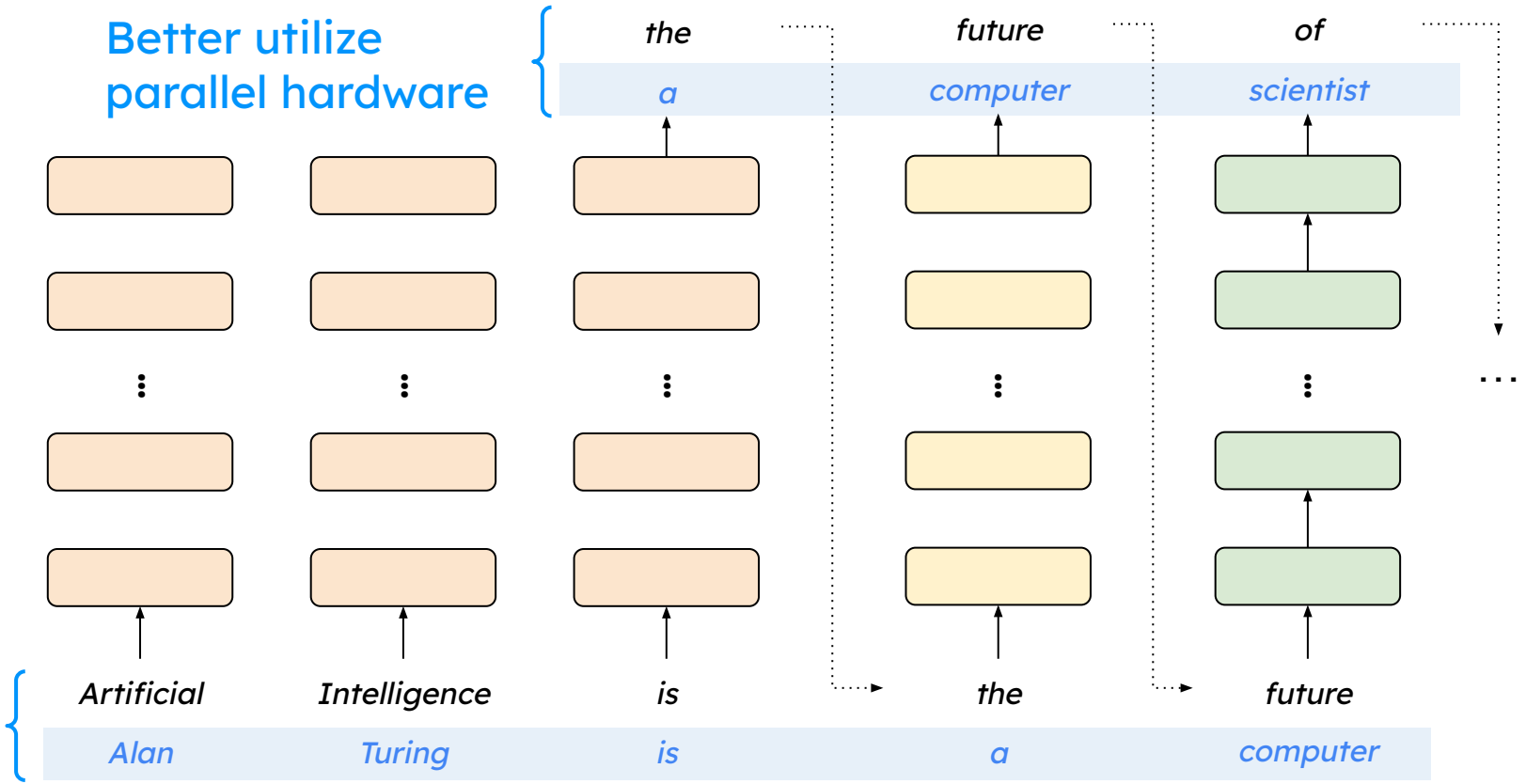
- Reaches maximum length (e.g., 2048 tokens)
- Generate certain tokens (e.g., "*<|end of sequence|>*")
- …

Why is LLM inference inefficient?

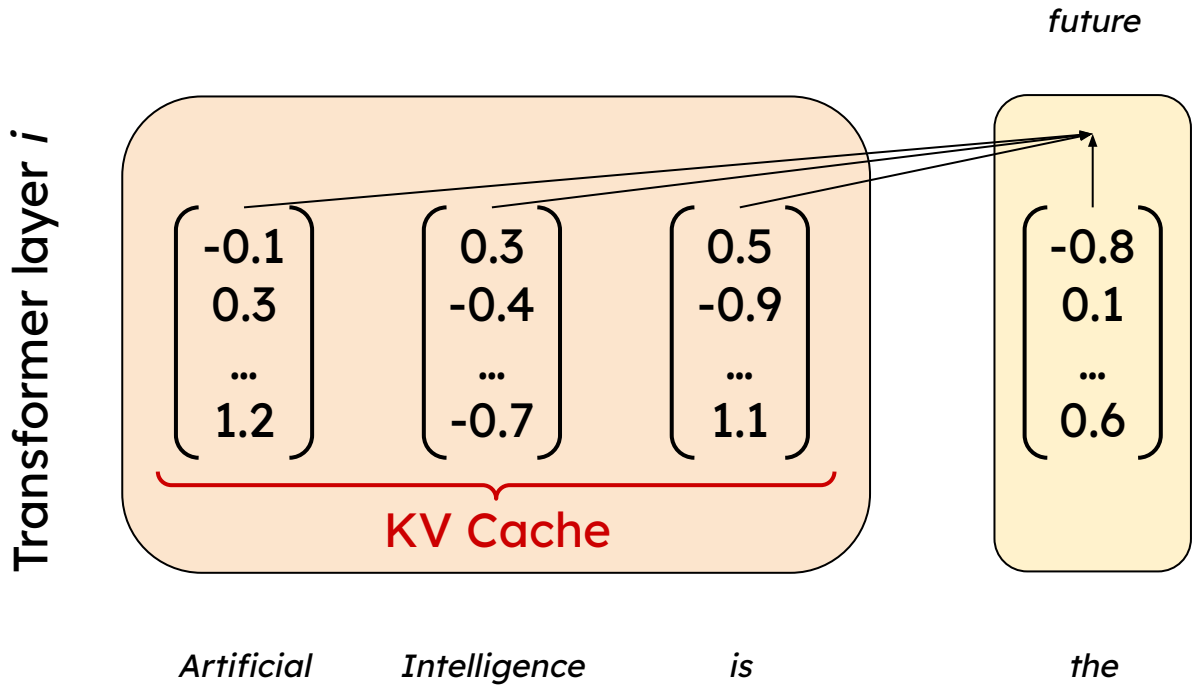


Sequential dependency → Hard for GPU parallelization

Batching?

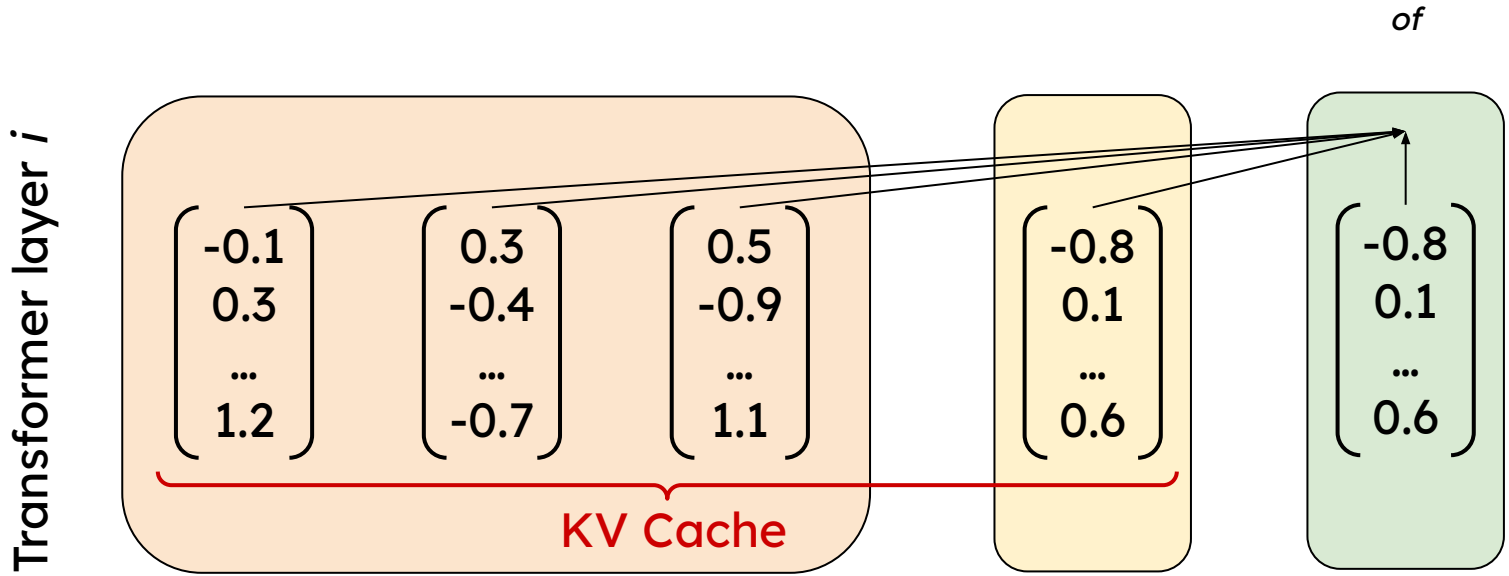

Better utilize parallel hardware
the
future
of
a
computer
scientist
...
Artificial
Intelligence
is
the
future
Alan
Turing
is
a
computer

Attention KV cache



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

Attention KV cache size

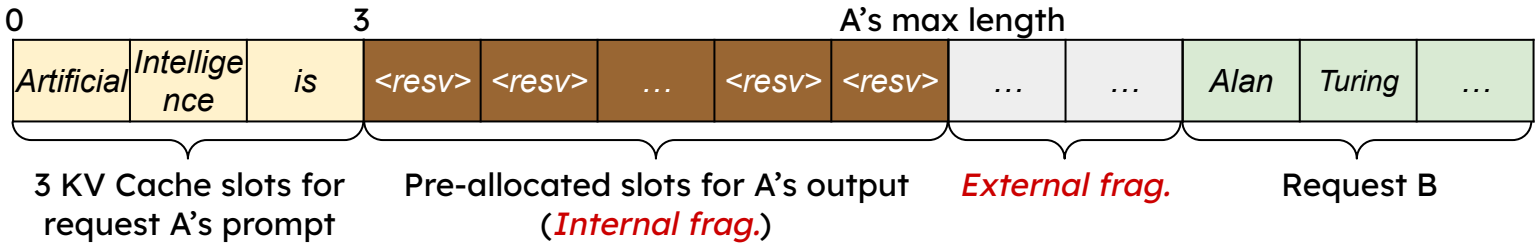


Artific *ure*

KV Cache is **huge**:

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

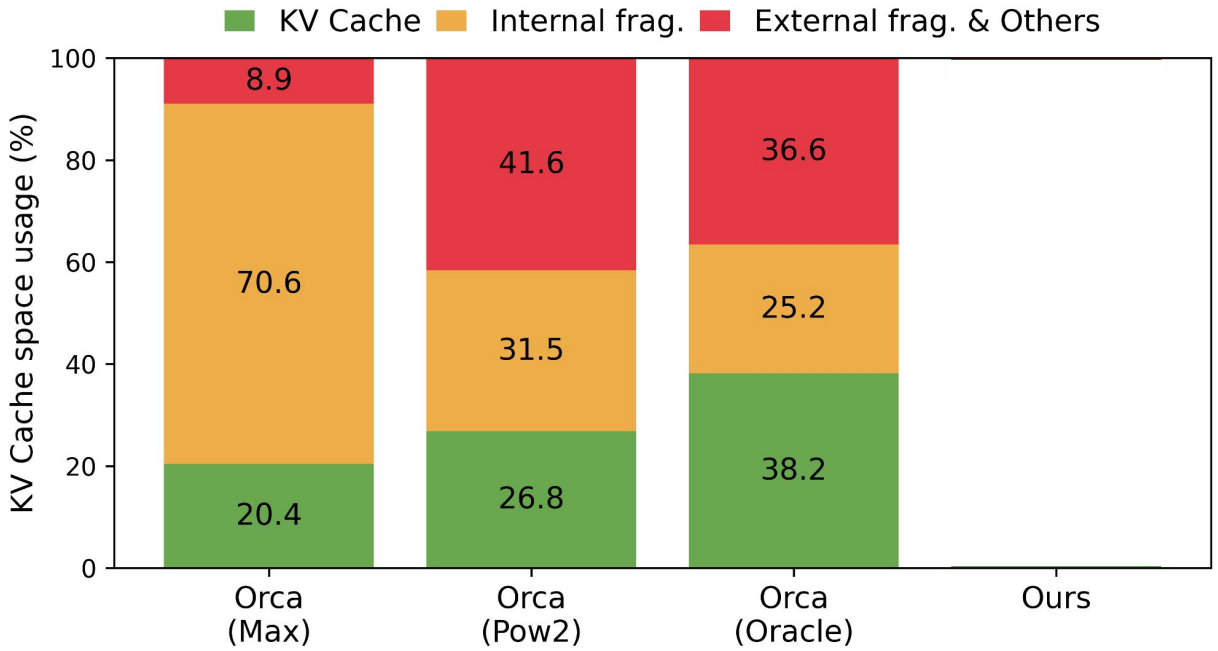
KV cache management in previous systems



- **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

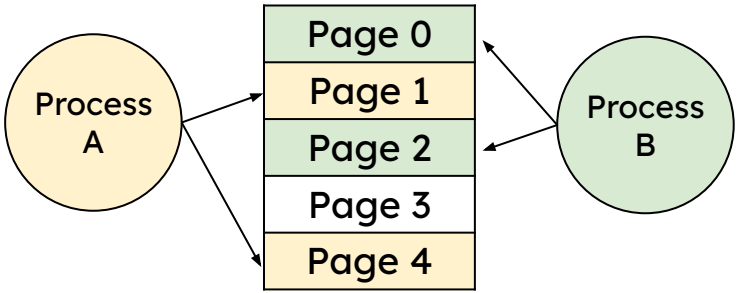


Yu et al. “Orca: A Distributed Serving System for Transformer-Based Generative Models” (OSDI 22).

PagedAttention

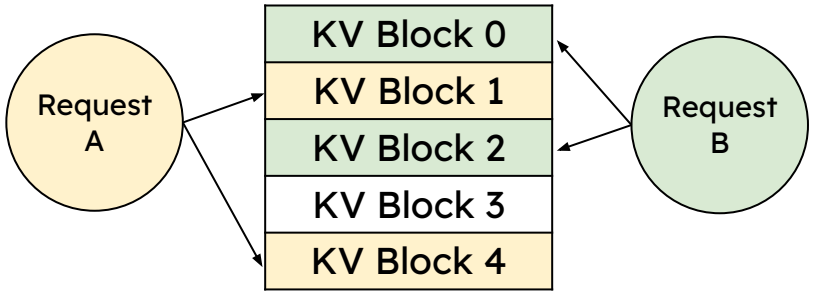
Application-level memory **paging** and **virtualization**
for attention KV Cache

Memory management in OS



Physical Memory

PagedAttention

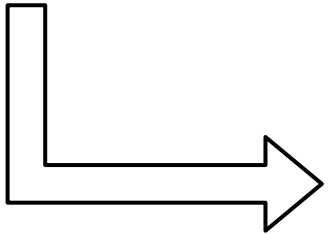


KV Cache

Paging KV cache space into *KV blocks*

Contiguous KV Cache

	Artificial	Intelli-gence	is	the							



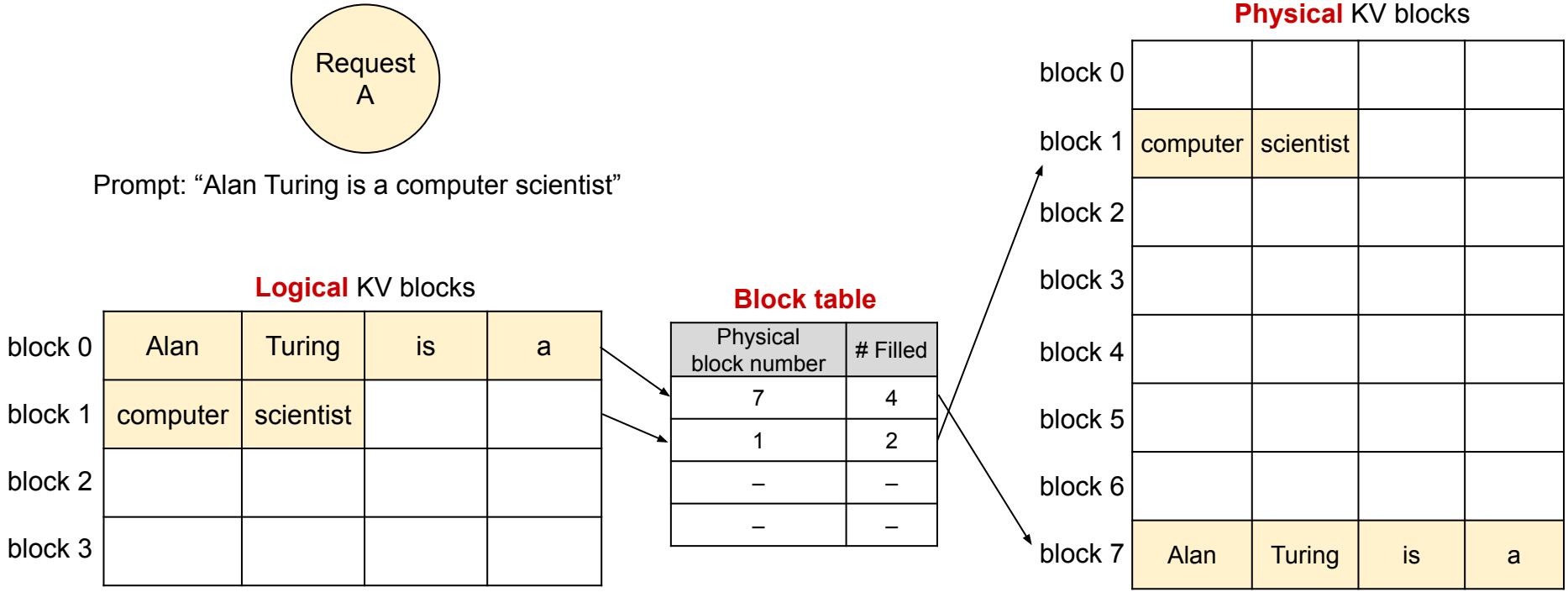
KV Blocks

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

Block size = 4

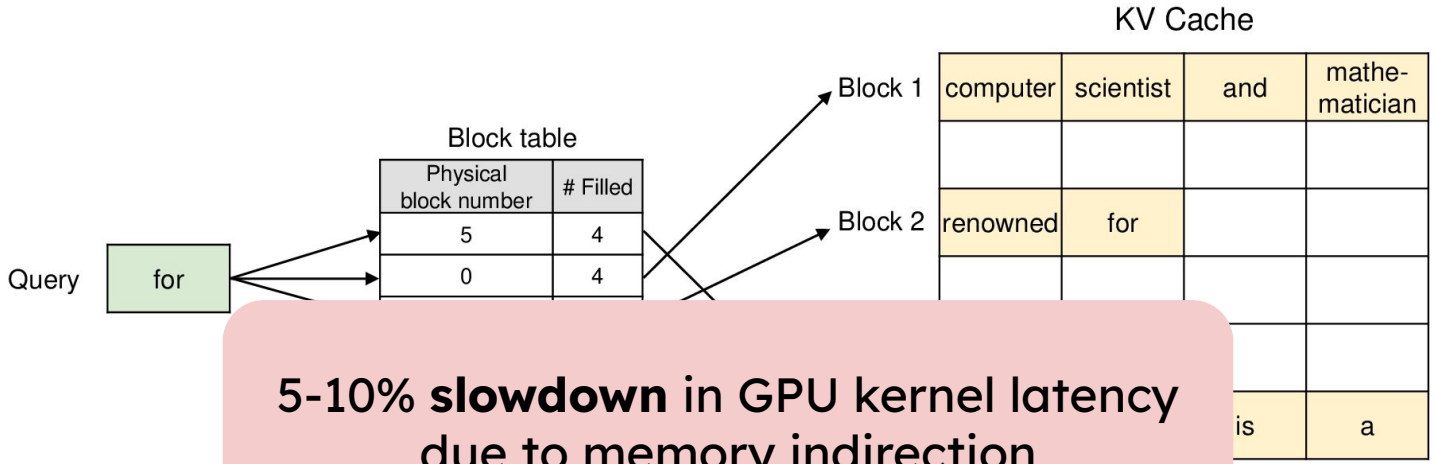
KV block: a **fixed-size** block of memory that stores KV cache **from left to right**

Virtualizing KV Cache



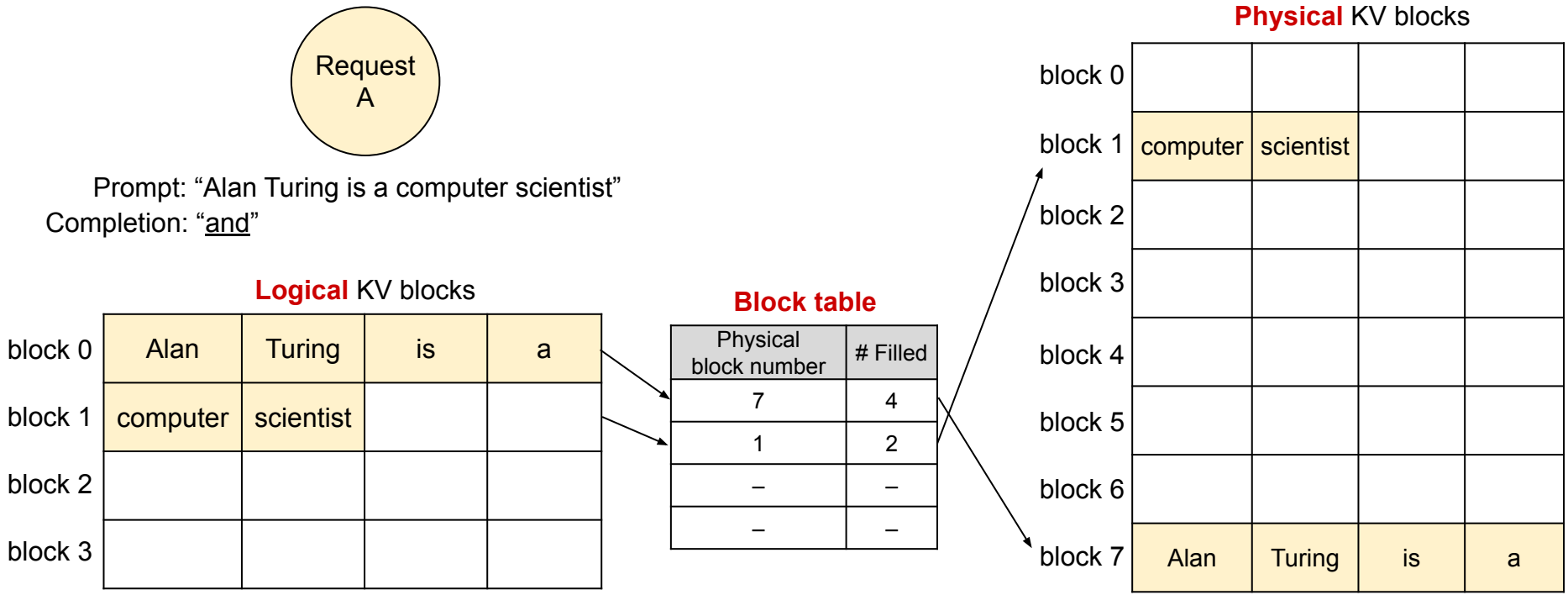
Attention mechanism with virtualized KV cache

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

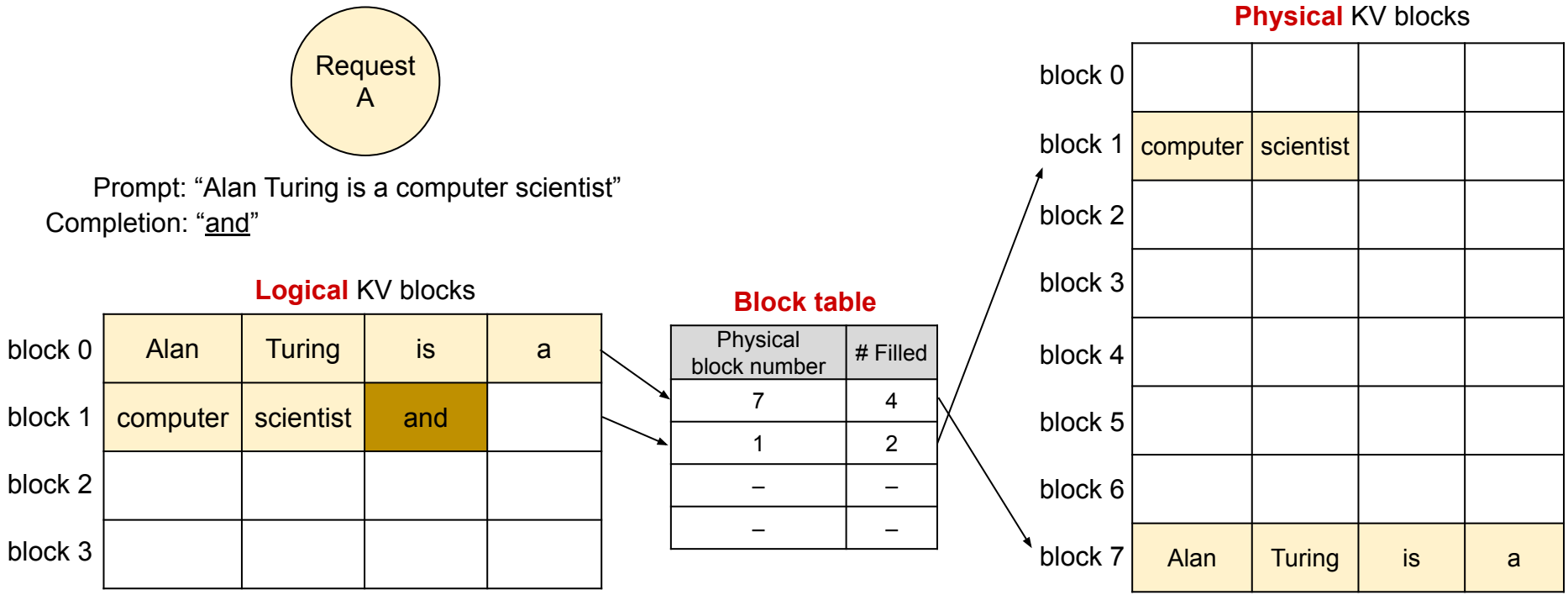


5-10% **slowdown** in GPU kernel latency due to memory indirection

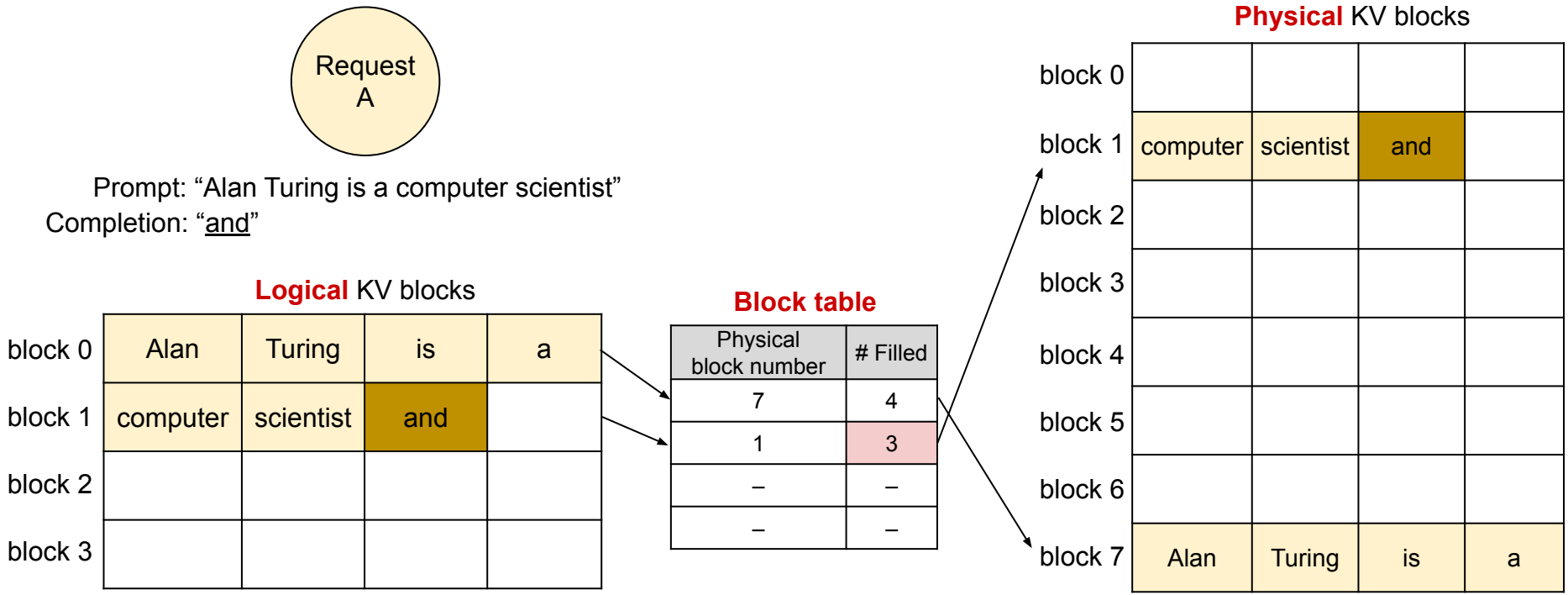
Memory management with PagedAttention



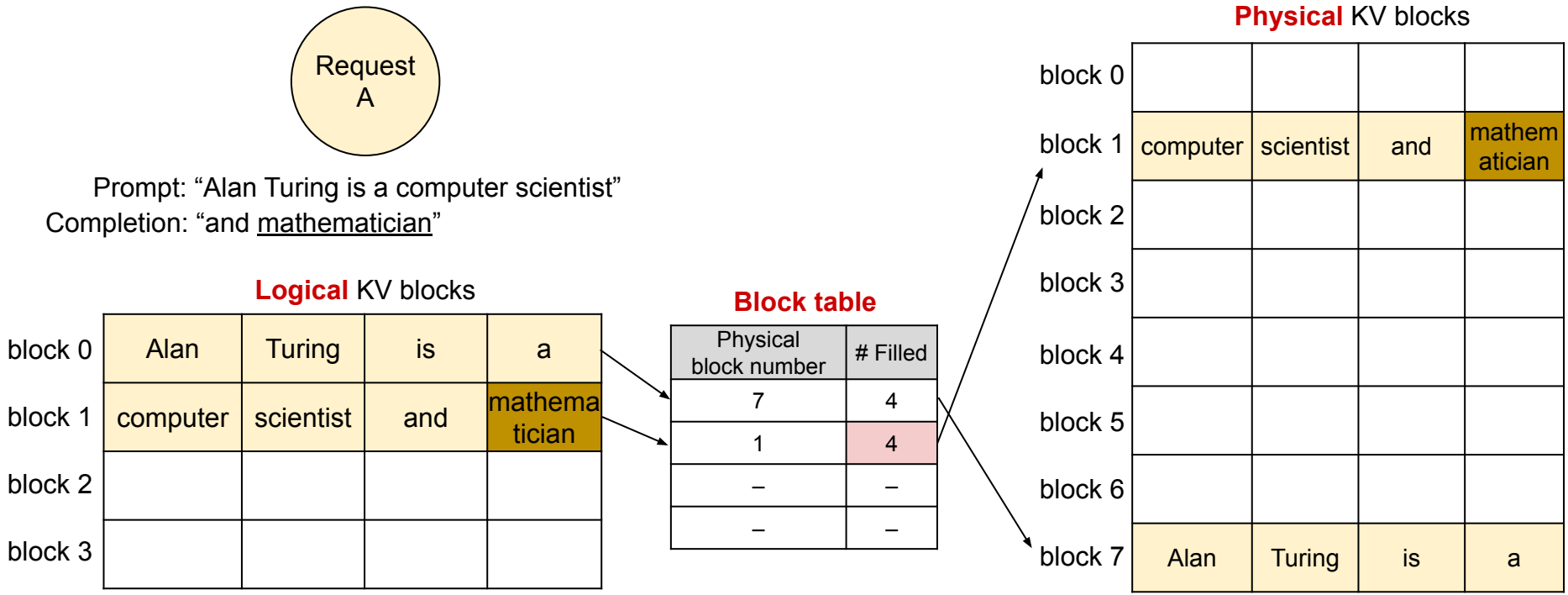
Memory management with PagedAttention



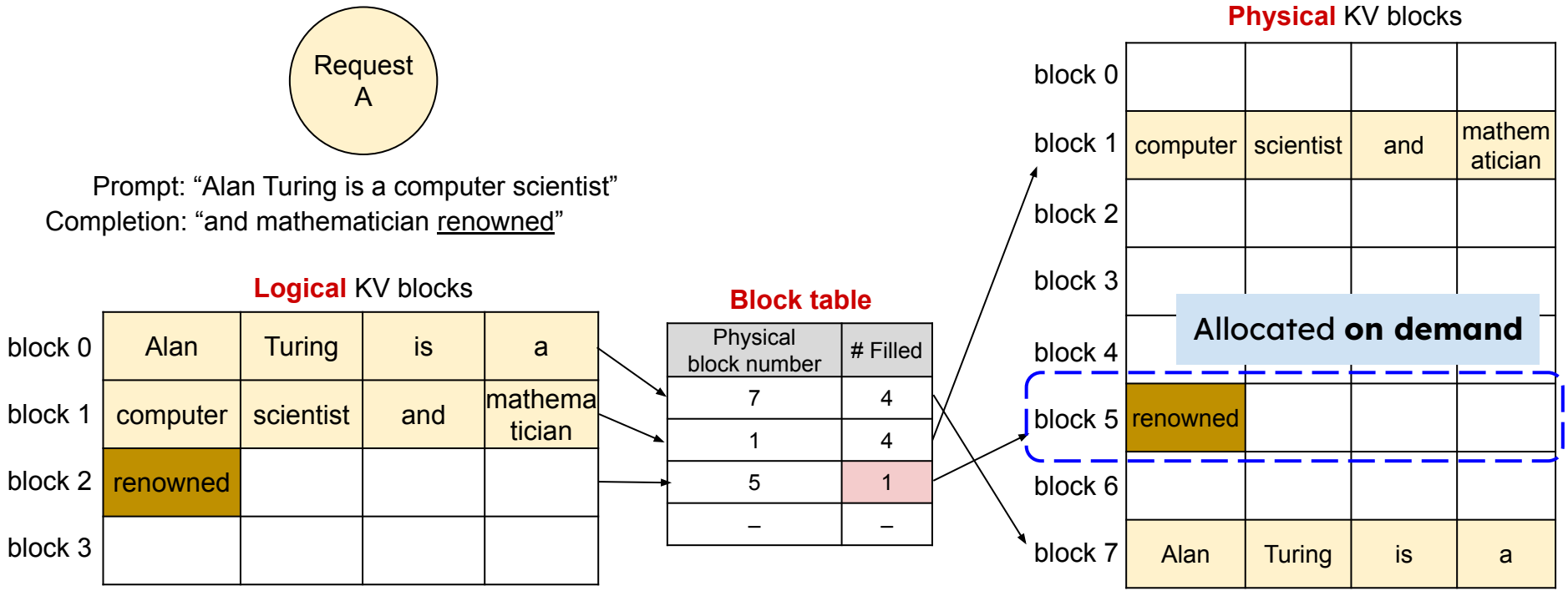
Memory management with PagedAttention



Memory management with PagedAttention



Memory management with PagedAttention

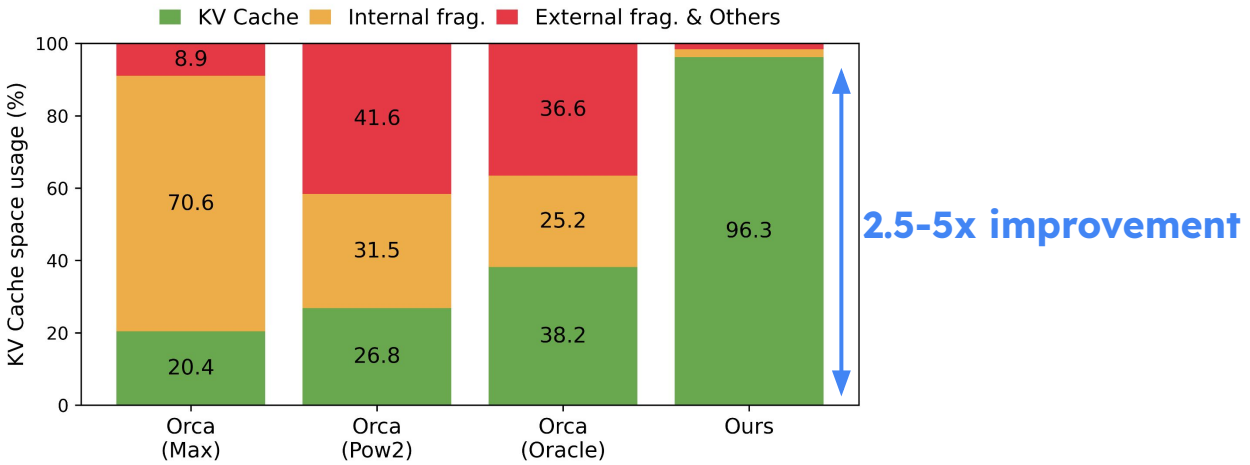


Memory efficiency of PagedAttention

- 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

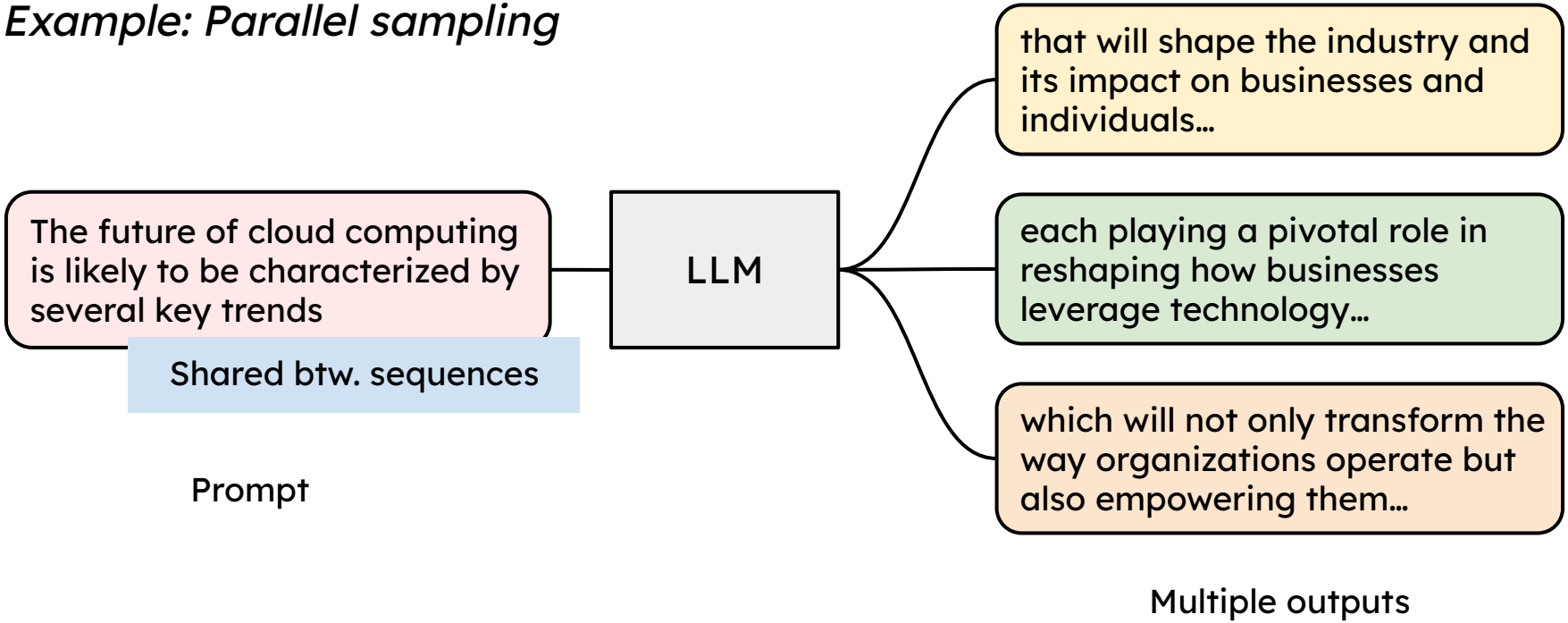
Alan	Turing	is	a
computer	scientist	and	mathematician
renowned			

Internal fragmentation

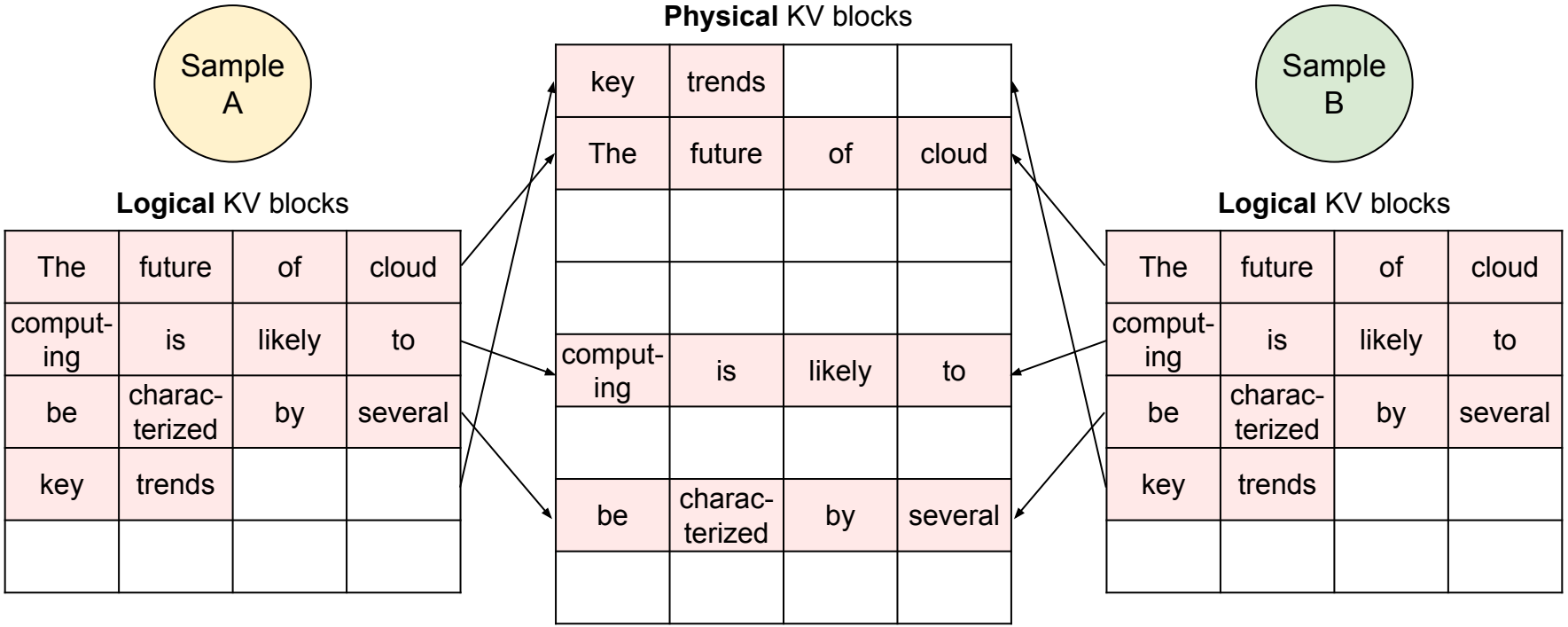


Paging enables sharing

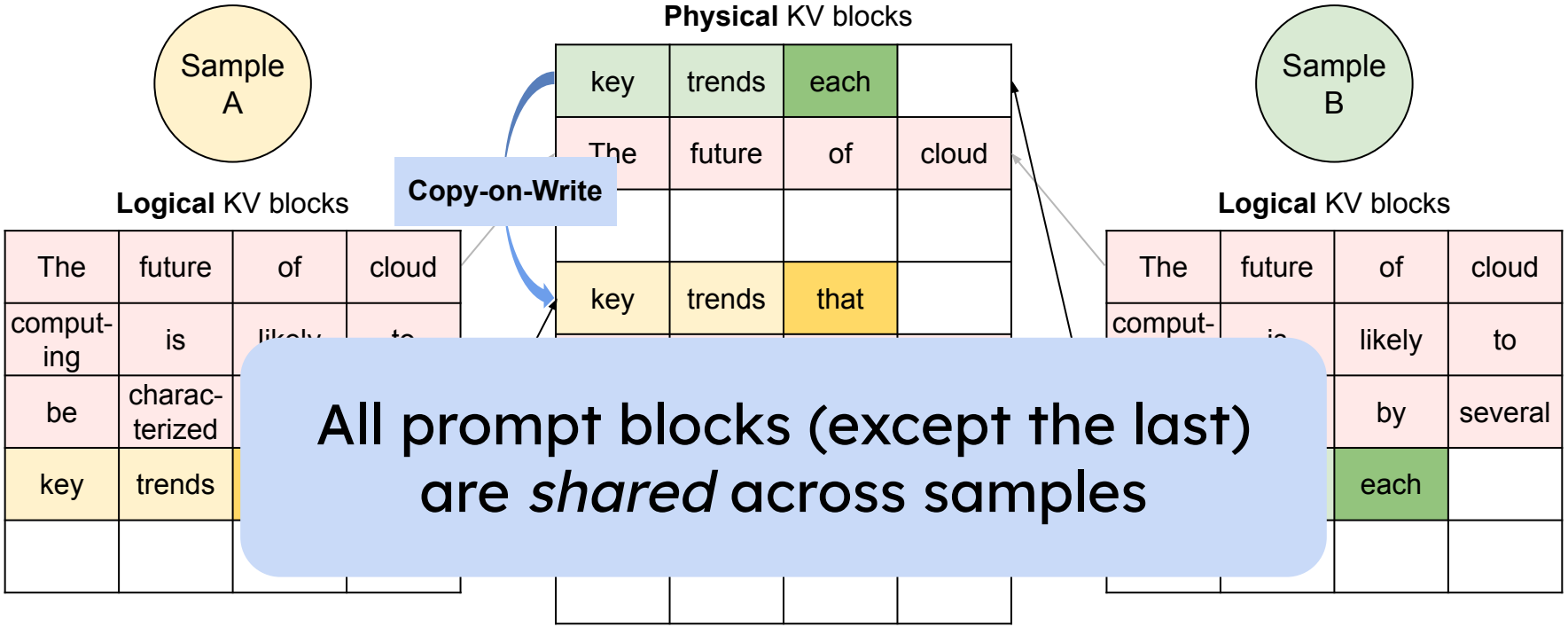
Example: Parallel sampling



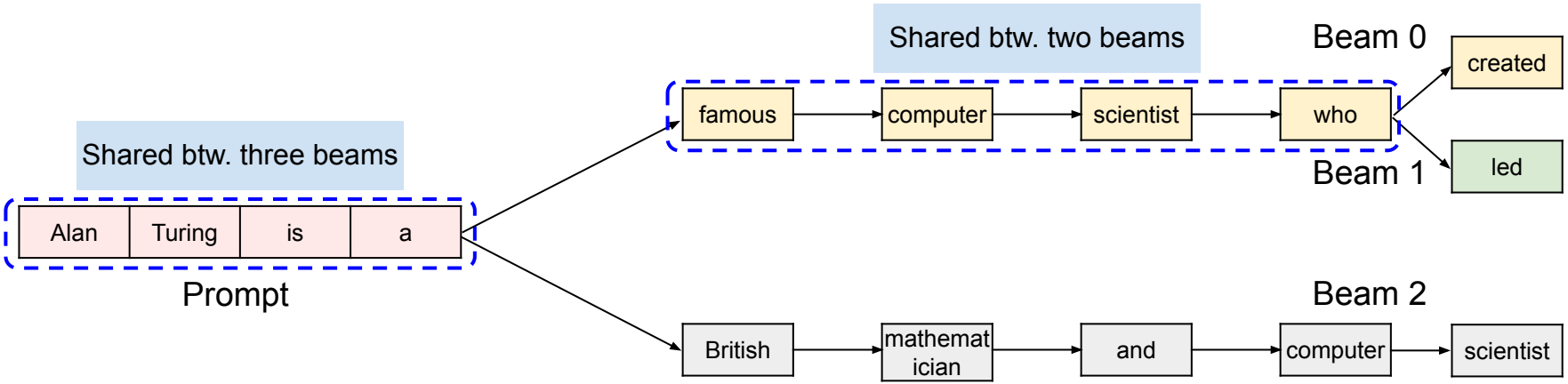
Sharing KV blocks



Sharing KV blocks

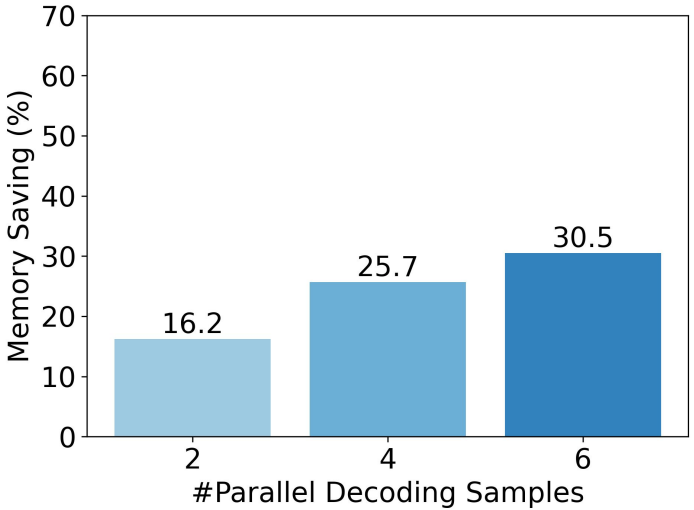


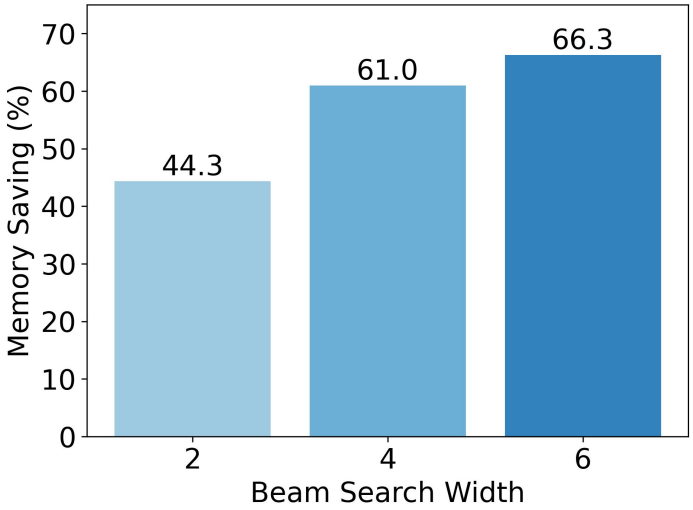
More complex sharing: beam search



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

Memory saving via sharing





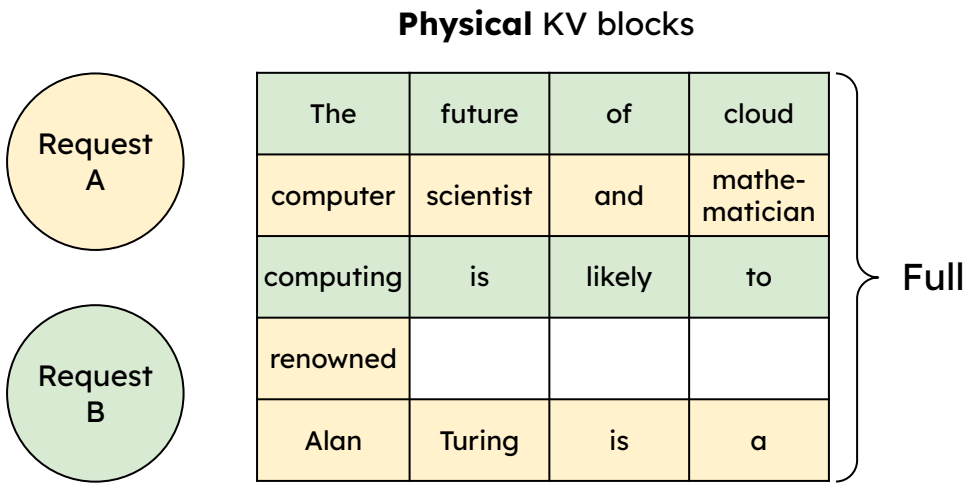
Percentage = (#blocks saved by sharing) / (#total blocks without sharing)
OPT-13B on 1x A100-40G with ShareGPT dataset

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

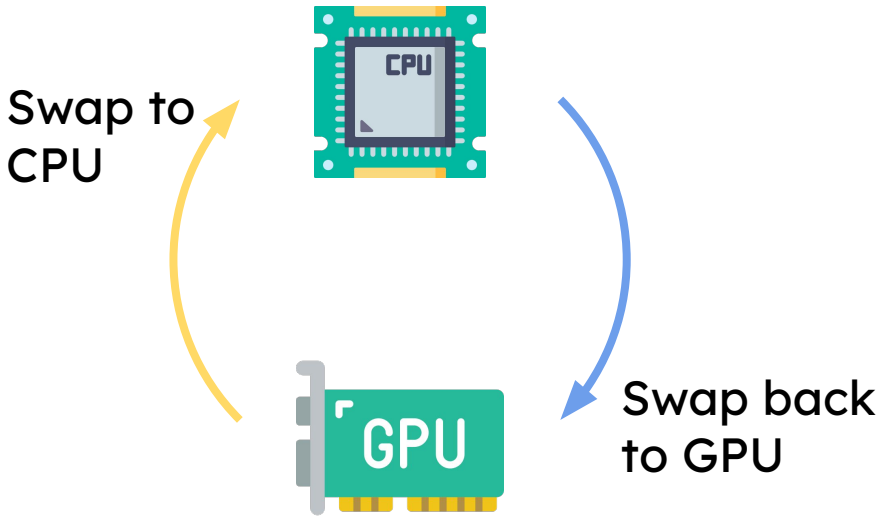
The	future	of	cloud
computer	scientist	and	mathe-matician
computing	is	likely	to
renowned	for		
Alan	Turing	is	a

Cannot allocate a new physical block for Request B

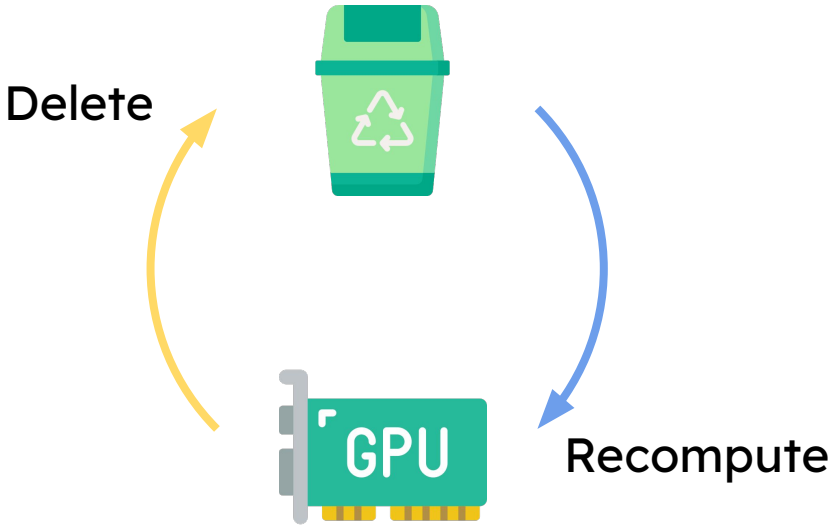
Request preemption & recovery

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.

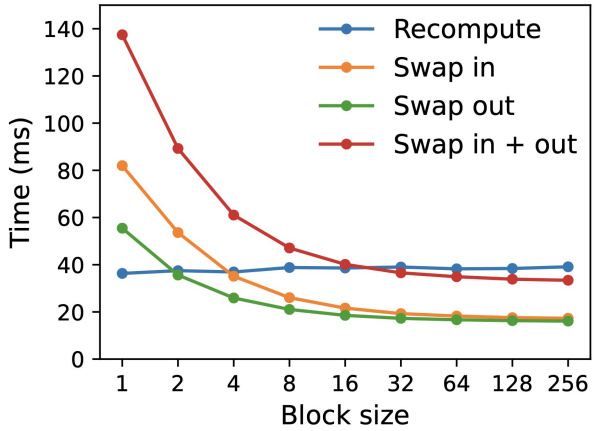


Figure: *Swap/Recomputation latency of 256 tokens.*

Our strategy: Use recomputation when possible 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

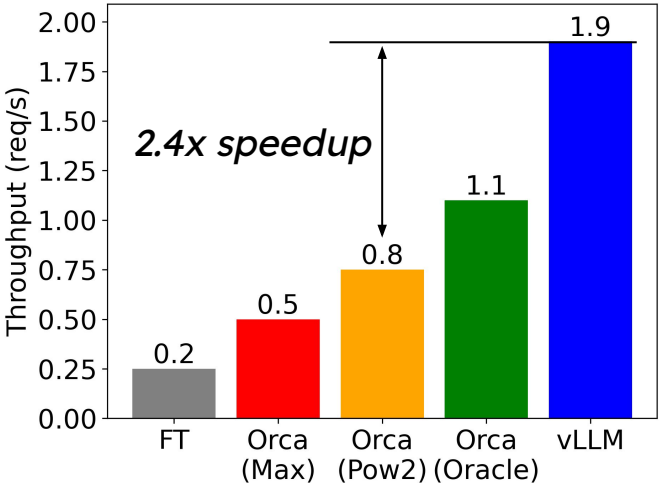
Single-level block table

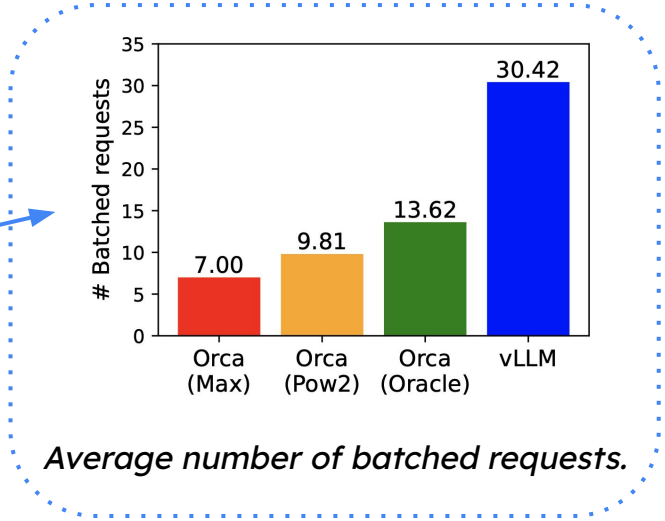
- Block table is tiny compared to the actual data.

Preemption & Recovery

- Request-level preemption
- Recomputation-based recovery

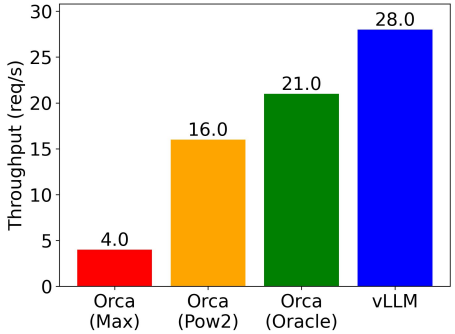
Throughput – greedy decoding



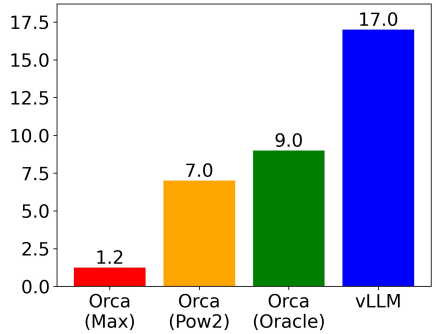


OPT-13B on 1xA100 40G with ShareGPT trace.

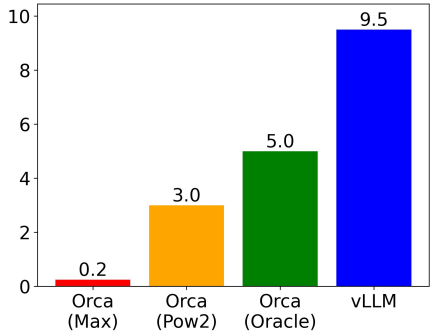
Throughput – beam search



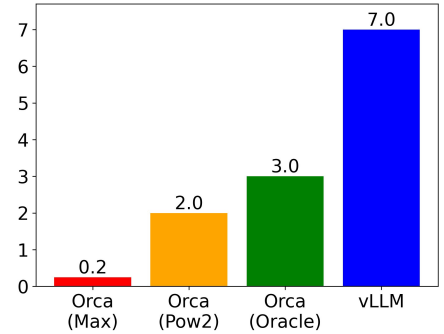
No beam search
***1.8x** speedup*



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



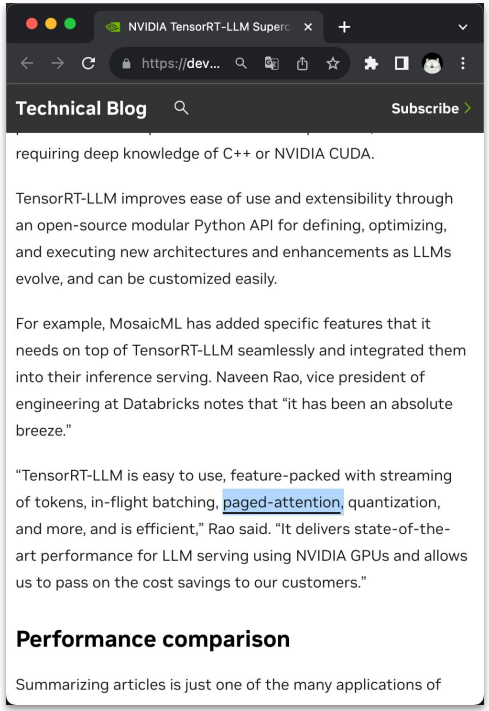
Beam width = 4
***3.2x** speedup*



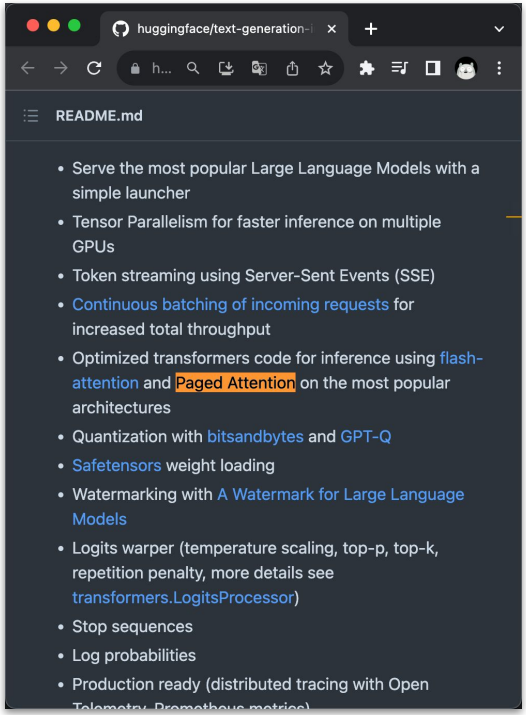
Beam width = 6
***3.5x** speedup*

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

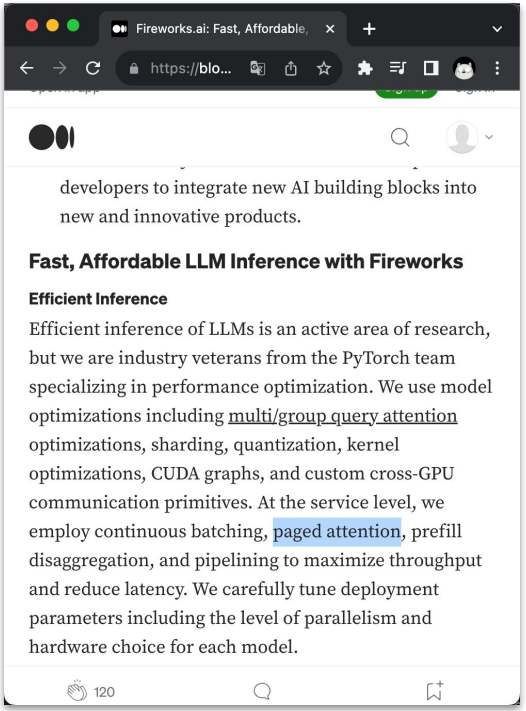
PagedAttention has become the industrial standard



TensorRT-LLM



HuggingFace TGI



Fireworks AI

. . .

Contents

- PagedAttention

- vLLM: A real-world open-source LLM serving system
  - Models
  - Parallelism
  - Inference optimizations

- Q&A

vLLM's Goal

Build the fastest and

easiest-to-use open-source

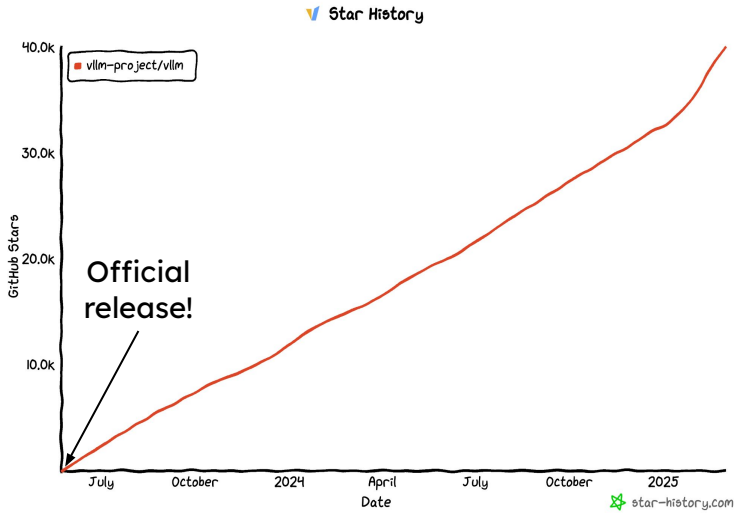
LLM inference & serving engine

vLLM Today

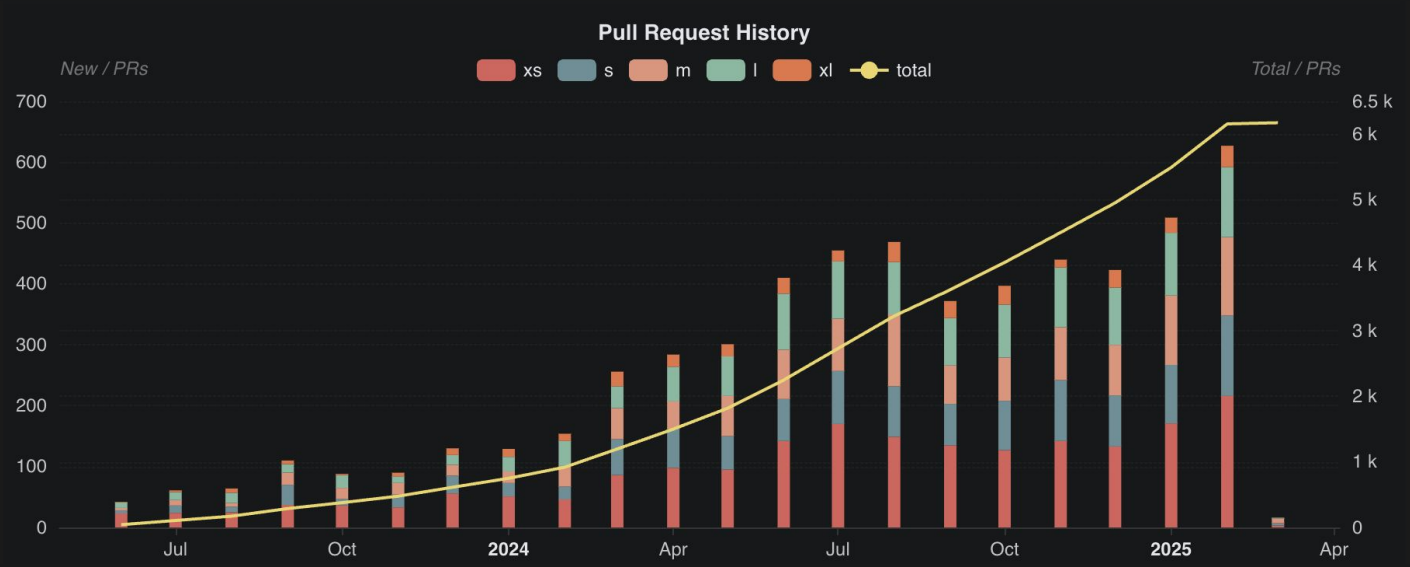
https://github.com/vllm-project/vllm

```
$ pip install vllm
```

43K Stars



vLLM Community



- 600+ PRs a month
- 890+ contributors
- 30+ major contributors from UCB, RedHat, Anyscale, Roblox, Meta, etc.

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="meta-llama/Meta-Llama-3.1-8B")
# 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 meta-llama/Meta-Llama-3.1-8B
```

Client

```
$ curl http://localhost:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "meta-llama/Meta-Llama-3.1-8B",
        "prompt": "San Francisco is a",
        "max_tokens": 7,
        "temperature": 0
    }'
```

Key Focuses in vLLM

Models

Parallelism

Inference optimizations

Performance engineering

Key Focuses in vLLM

Models

Parallelism

Inference optimizations

Performance engineering

Broad Model Support

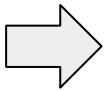
- Transformer-like LLMs (e.g., Llama)
- Mixture-of-Expert LLMs (e.g., DeepSeek)
- Multi-modal LLMs (e.g., Qwen-VL)
- State-Space Models (e.g., Mamba)
- Embedding Models (e.g. E5-Mistral)

Easy Model Integration

- vLLM is based on PyTorch and HuggingFace Transformers
- vLLM requires minimal changes to the model code

```
class LlamaAttention(nn.Module):

def __init__(self, ...):
    self.qkv_proj = nn.Linear(...)
    self.attn = Attention(...)
    ...

def forward(self, x):
    qkv = self.qkv_proj(x)
    q, k, v = qkv.split(...)
    attn_output = self.attn(q, k, v)
    ...
```

Original model (HuggingFace)



```
class LlamaAttention(nn.Module):

def __init__(self, ...):
    self.qkv_proj = ColParallelLinear(...)
    self.attn = PagedAttention(...)
    ...

def forward(self, x):
    qkv = self.qkv_proj(x)
    q, k, v = qkv.split(...)
    attn_output = self.attn(q, k, v)
    ...
```

vLLM

First-class Support for Multi-modal Models

vLLM provides a structured way to register models with various modalities (e.g., image, video, and audio)

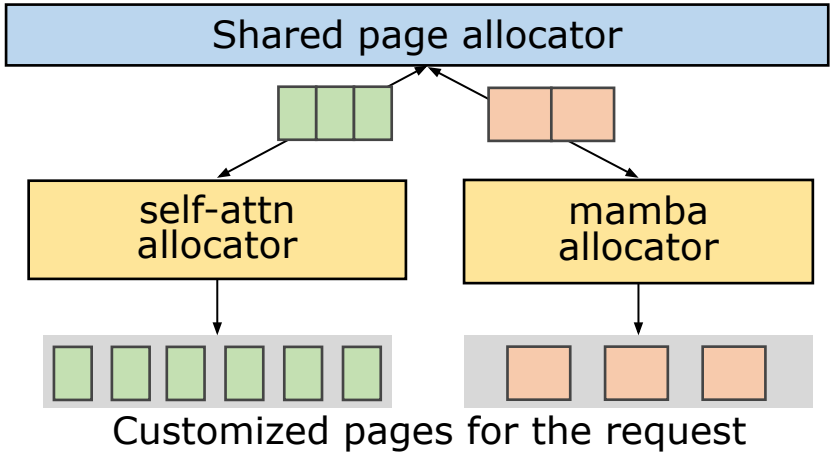
```
@MULTIMODAL_REGISTRY.register_image_input_mapper(image_input_mapper)
@MULTIMODAL_REGISTRY.register_input_mapper("video", video_input_mapper)
@MULTIMODAL_REGISTRY.register_max_image_tokens(get_max_image_tokens)
@MULTIMODAL_REGISTRY.register_max_multimodal_tokens("video",
                                                    get_max_video_tokens)
@INPUT_REGISTRY.register_dummy_data(dummy_data_for_qwen2_vl)
@INPUT_REGISTRY.register_input_processor(input_processor_for_qwen2_vl)
class Qwen2VL(nn.Module, SupportsMultiModal):

    def __init__(self,
                 config: Qwen2VLConfig,
                 multimodal_config: MultiModalConfig):
        super().__init__()
        ...
```

Hybrid Attention Models

- Various attention variants to efficiently support long context:
  - Multi-head Latent Attention (DeepSeek)
  - Sliding window attention
  - Local chunked attention
  - State-space models (Mamba)
- Importantly, all of the above can be mixed in a single model
  - Gemma 3: Global attention + sliding window attention
  - Llama 4: Global attention + local chunked attention
  - Jamba: Global attention + Mamba

Hybrid memory allocator

- We are building a two-layer memory allocator/evictor to manage the heterogeneous memory:



- Jenga: Effective Memory Management for Serving LLM with Heterogeneity

Manual CUDA kernels v.s. torch.compile

- vLLM once maintained a large set of CUDA kernels, with different kernel fusion and quantization.
- The modifications to the model code because of these kernels make the model code less readable and harder to maintain.

Key Focuses in vLLM

Models

Parallelism

Inference optimizations

Performance engineering

Why do we need parallelism?

- Models are getting much bigger than a single GPU
  - Deepseek V3: 671B parameters → ~700GB in FP8
  - NVIDIA H100: 80GB HBM
  - Memory is not only used for parameters, activations and KV cache also takes memory
- Use more GPUs to multiple the compute power
  - 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
  - Limit the number of bytes transferred
  - Hide the overhead via overlapping with computation
  - Network is also heterogeneous
    - Latest NVLink: 1.4TB/s
    - Infiniband: 50GB/s
- Goal 2: Minimize waiting from data dependency and stragglers' effect
  - A system is often bottlenecked by the slowest component

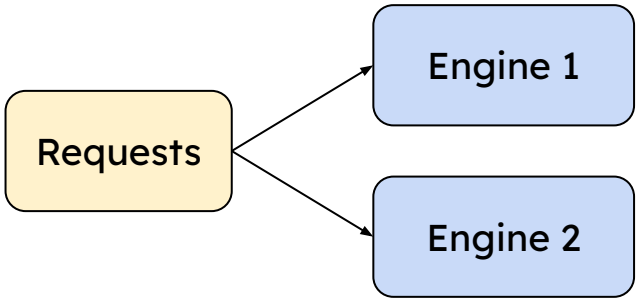
Data parallelism (mostly outside of vLLM)

- 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, which can limit the KV cache size, and thus serving throughput

👎 No reduction in latency



Tensor parallelism (in vLLM)

- Partition the weights in the linear layers

👍 Reduce the serving latency if the network is fast

👍 KV Cache can also be partitioned

👎 Heavy communication across TP shards. Often require NVLink to work effectively.

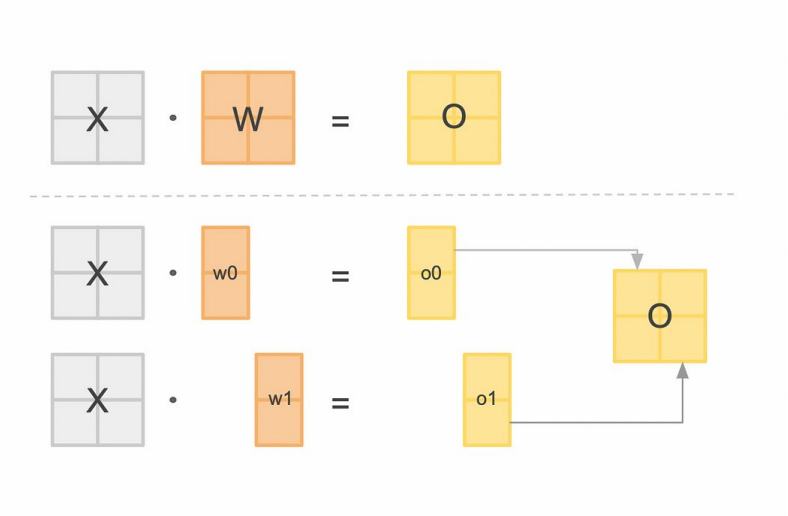


Image source: https://www.titanml.co/resources/harmonizing-multi-gpus-efficient-scaling-of-llm-inference

Expert parallelism (in vLLM)

- Mixture of Expert: A set of linear layers (experts) where each token only pick a subset to run on
- Distribute different experts to different nodes

👍 Friendly for GPU kernels

👍 Smaller communication than TP

👎 Only apply for MoE layers. Attention needs DP.

👎 Different experts have different loads, which needs to be balanced

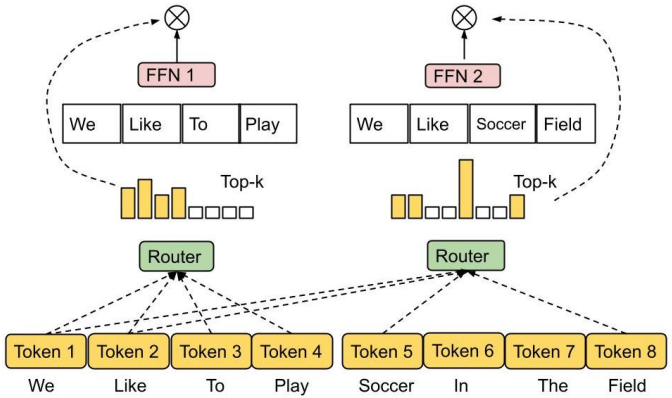


Image source: https://research.google/blog/mixture-of-experts-with-expert-choice-routing/

Sequence and context parallelism (not in vLLM yet)

- 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.

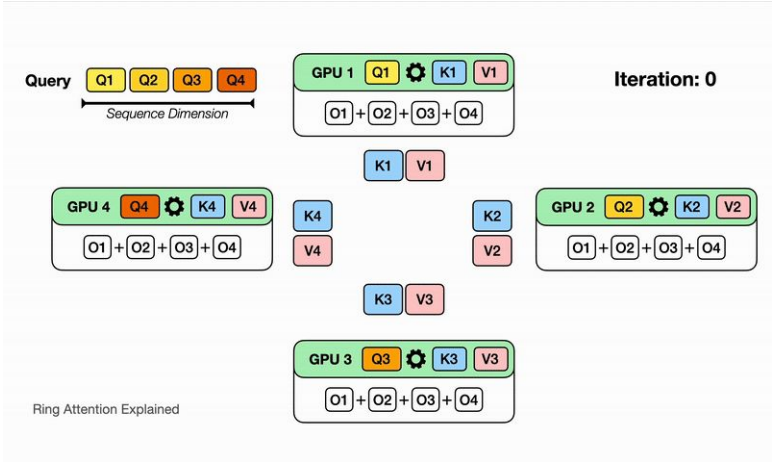


Image source: https://coconut-mode.com/posts/ring-attention/

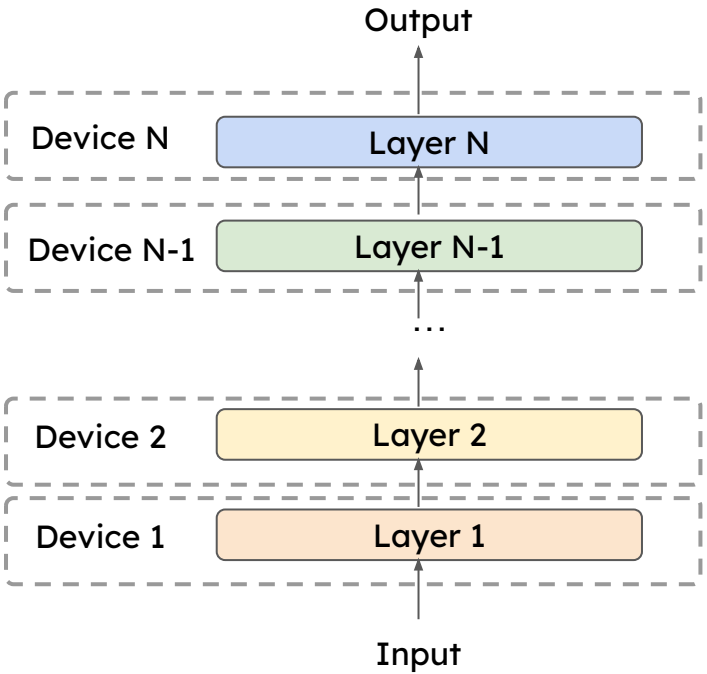
Pipeline parallelism (in vLLM)

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

👍 Lowest communication overhead

👎 Increased latency

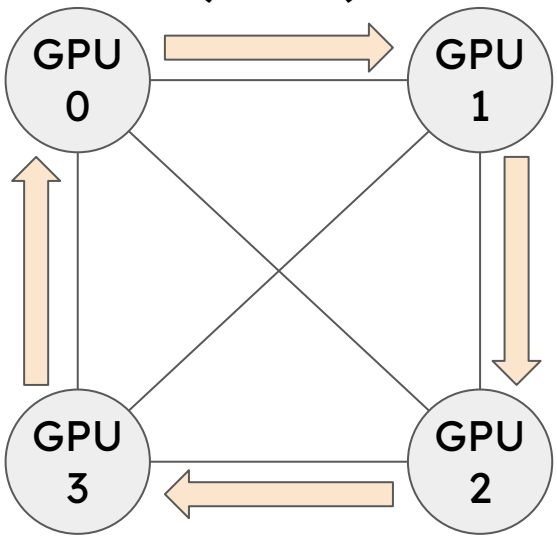
👎 Throughput bottlenecked by the slowest stage



Communication kernel

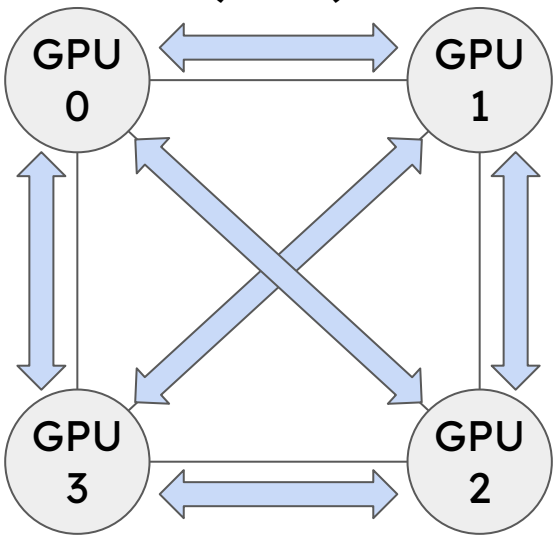
- Communication kernels have different requirements in inference:

Ring All Reduce
(NCCL)



Optimized for large tensors

One-shot All Reduce
(Ours)



Optimized for small tensors

Prefill-Decode Disaggregation

- Different parallelization strategies have different characteristics in computation/memory/communication requirements
- Different stages of LLM inference have different characteristics as well:
  - Prefill: typically compute bound
  - Decode: typically memory bound and require large KV cache memory
- **Disaggregation:** Have different setups for prefill and decode. Transfer the KV cache between them.
- *Side benefit:* Separate concern on time-to-first-token/time-per-output-token

Key Focuses in vLLM

Models

Parallelism

Inference optimizations

Performance engineering

Inference Optimizations

- Optimizing the the model:
  - Quantization

- Optimizing “prefill”:
  - Prefix caching, CPU KV cache offloading, etc.

- Optimizing “decode”:
  - Speculative decoding, Jump decoding, etc.

Quantization

- Use reduced precisions to store & compute the model
  - BFloat 16 & Float 16 are the standard for “unquantized” models
  - Quantization typically means using 8 or lower bits (e.g., FP8, INT8, FP4)

- Native hardware support makes quantization increasingly effective (e.g., FP8 in Hopper & FP4 in Blackwell)

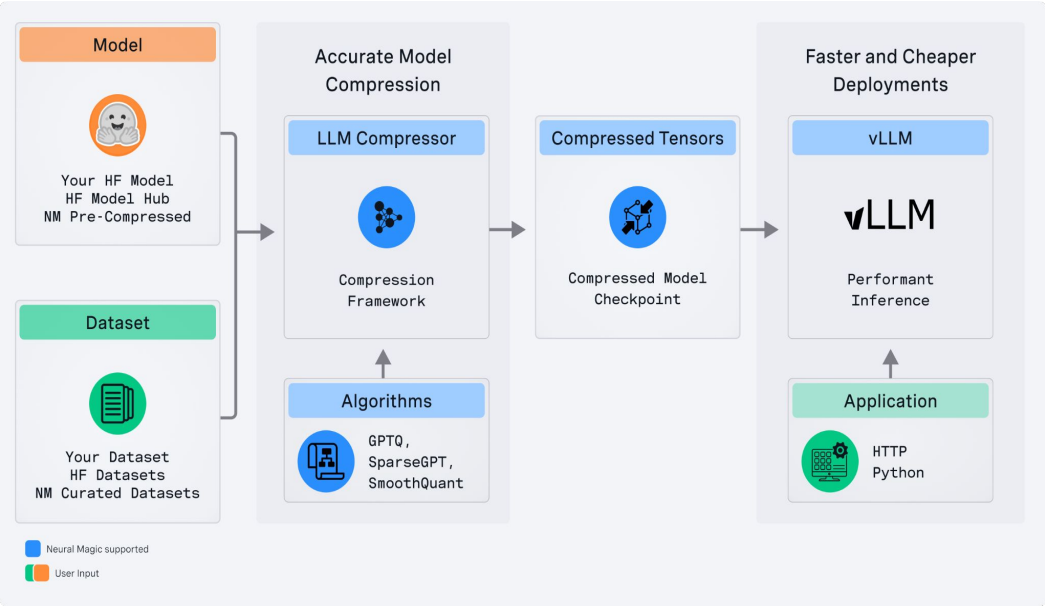
LLM Quantization

- LLMs have 3 axes to quantize
  - Size: Weights >= KV cache >>> Activation

1. Weight quantization (e.g., FP8, INT8, GPTQ, AWQ)
   - Main benefits: Reduced storage & memory footprints
2. KV cache quantization (e.g., FP8)
   - Main benefits: Reduced KV cache storage & Faster attention
3. Activation quantization (e.g., FP8, INT8)
   - Main benefits: Faster GeMM & communication (in distributed inference)

Quantization support in vLLM

LLM Compressor



- LLM Compressor library for quantizing the model weights

- vLLM provides highly-tuned GPU kernels for quantized ops

Automatic Prefix Caching

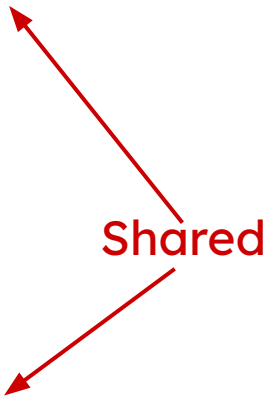
Example 1: Shared system prompt

Request A

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions.
User: Hello!

Request B

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions.
User: How are you?



Automatic Prefix Caching

Example 2: Multi-round conversation

Prompt (round 1)

Human: *What's AI?*

LLM Result (round 1)

LLM: *AI is technology that simulates human intelligence, like Siri or Google Maps.*

Shared

Prompt (round 2)

Human: *What's AI?*

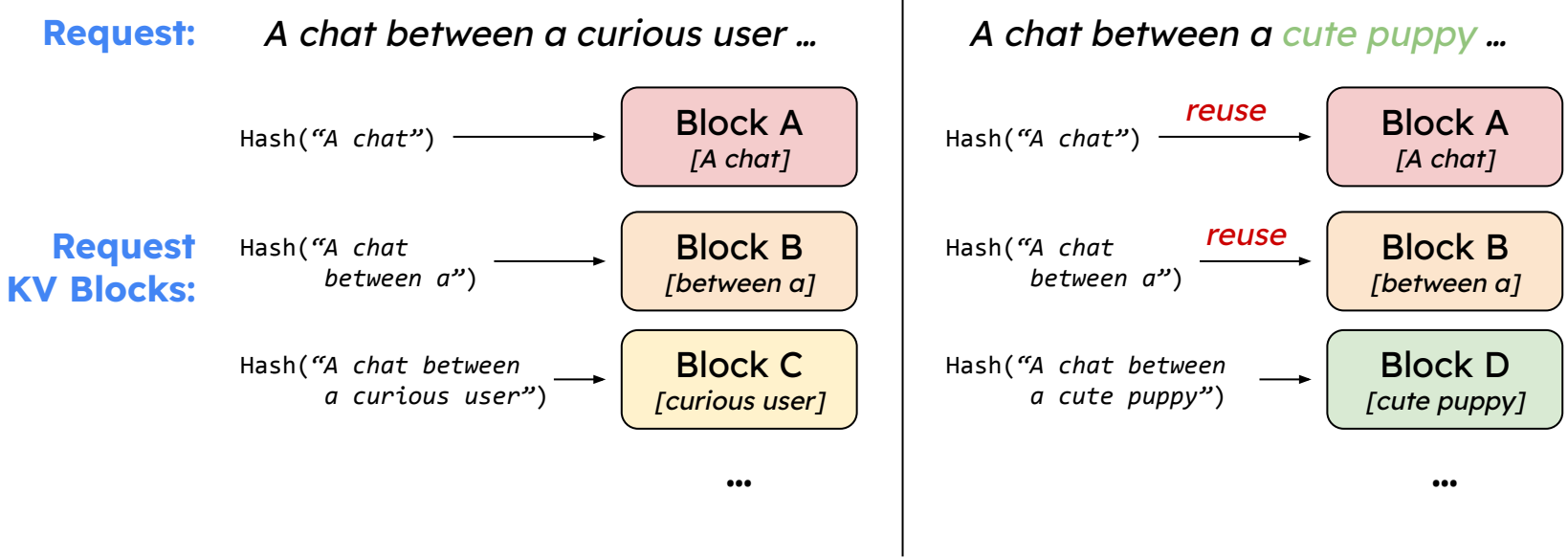
LLM: *AI is technology that simulates human intelligence, like Siri or Google Maps.*

Human: *Cool, thanks!*

LLM Result (round 2)

LLM: *No problem!*

Hash-Based Automatic Prefix Caching



Speculative Decoding

Small model writes a draft → Large model verifies it

Much faster: verifying 5 tokens takes similar time as generating 1 token

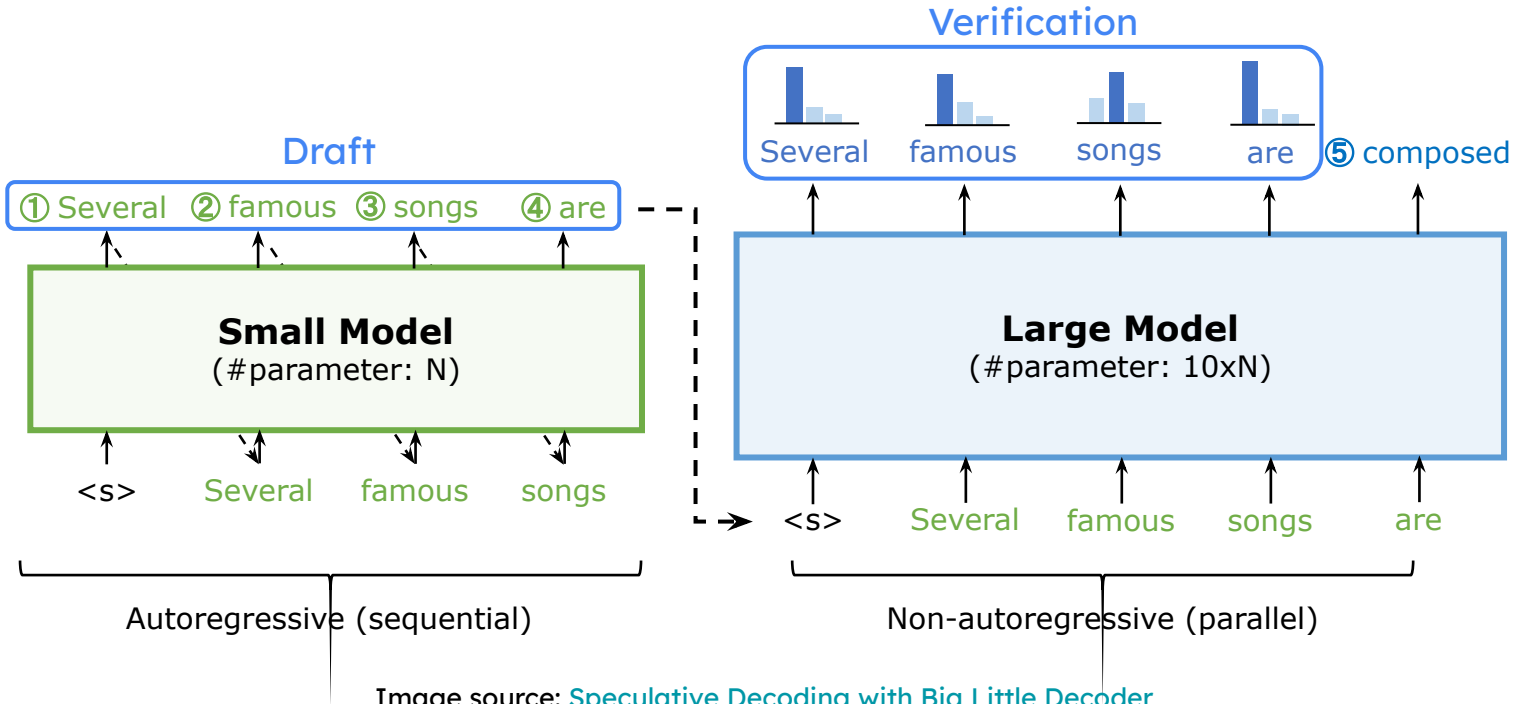
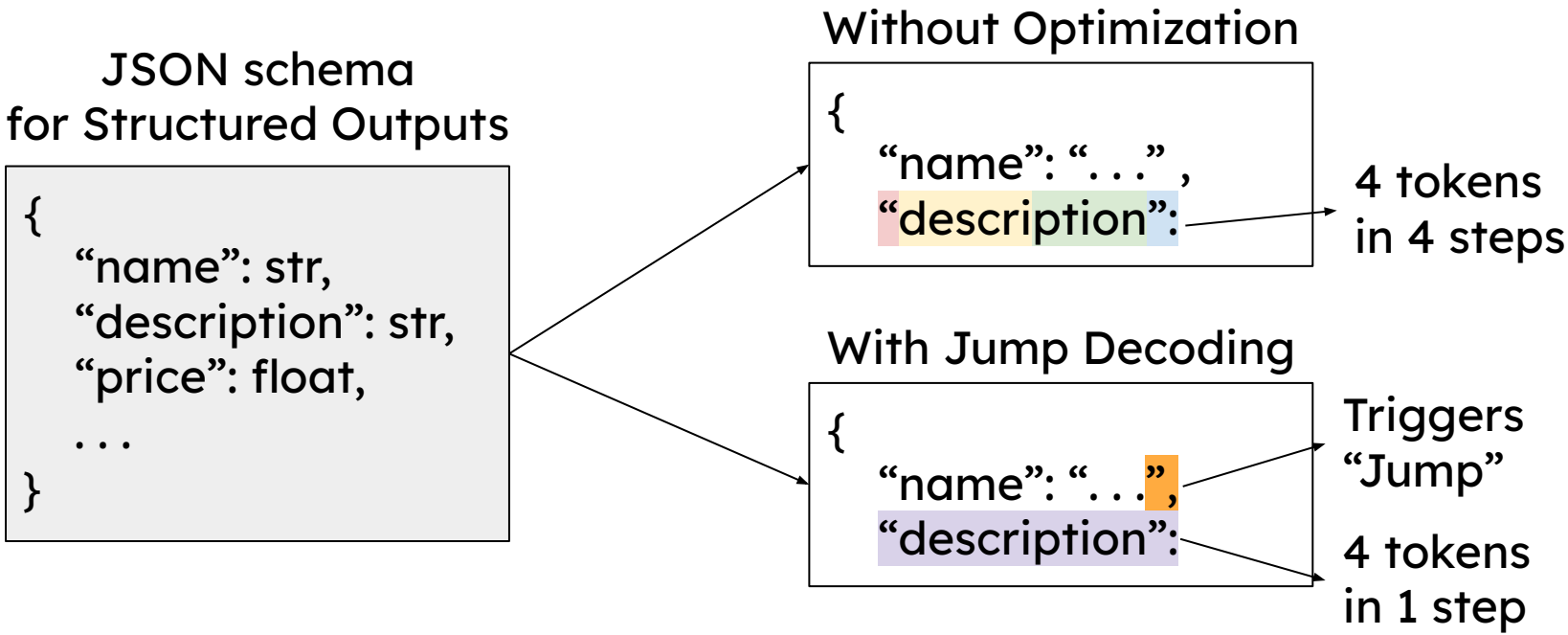


Image source: Speculative Decoding with Big Little Decoder

Jump Decoding

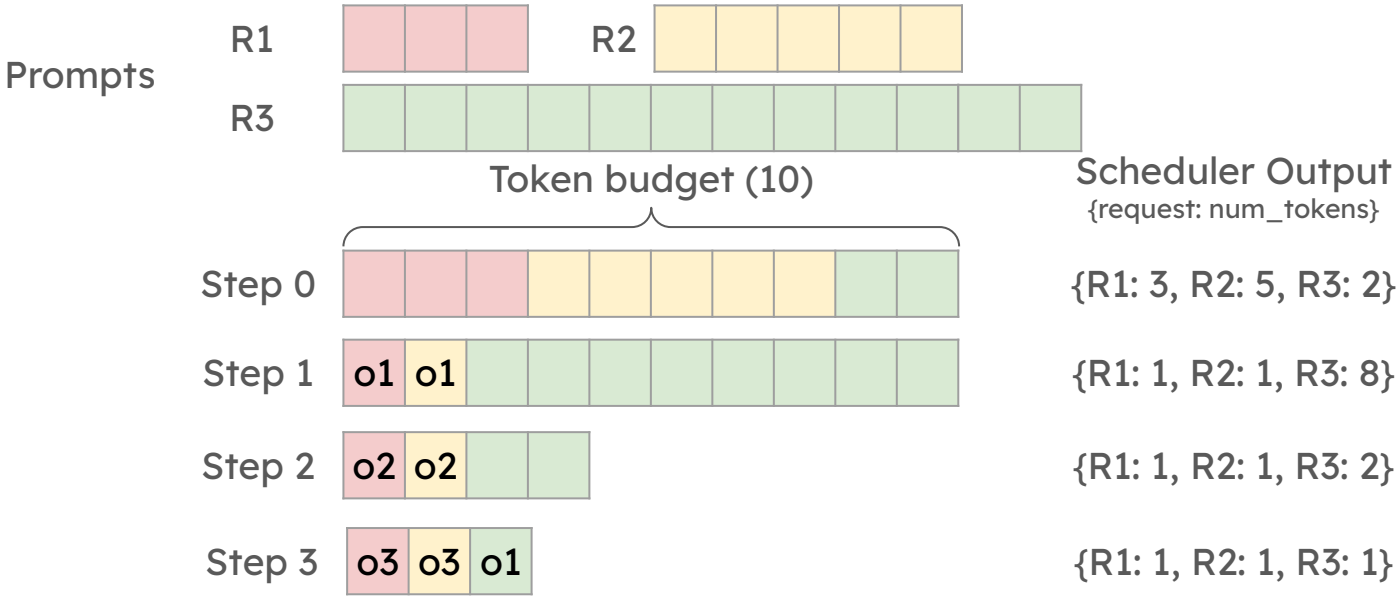
"Jump" the token generation using the predefined JSON schema



Reference: Fast JSON Decoding for Local LLMs with Compressed Finite State Machine

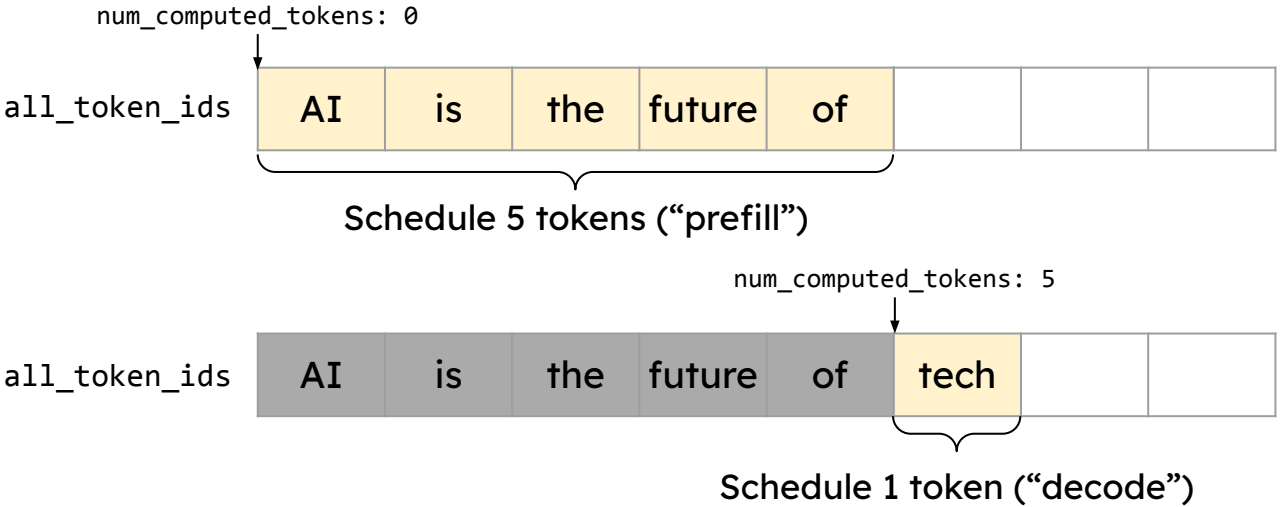
Unified representation for scheduling

- The scheduling decision is simply represented as a dictionary of `{request_id: num_tokens}`
- “token budget” to control the per-step execution time



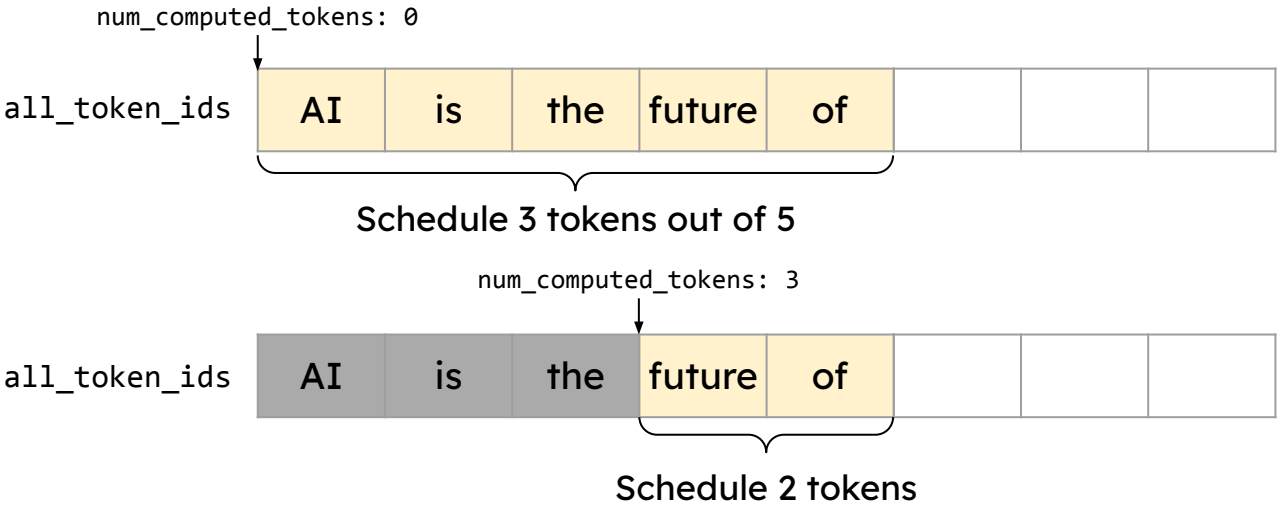
Unified representation for scheduling (cont'd)

- Unification of "prefill" and "decode"
  - There's no concept of prefill and decode
  - Schedule based on the difference between `num_compute_tokens` and `len(all_token_ids)`
- Ex1) "Prefill" & "Decode"



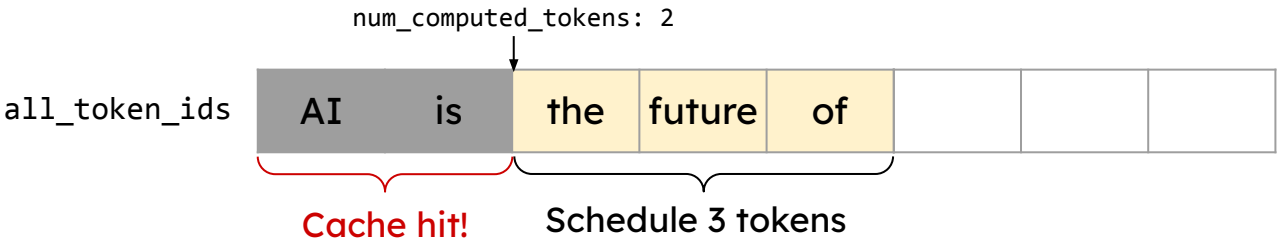
Unified representation for scheduling (cont'd)

- Unification of "prefill" and "decode"
  - There's no concept of prefill and decode
  - Schedule based on the difference between `num_compute_tokens` and `len(all_token_ids)`
- Ex2) Chunked prefills



Unified representation for scheduling (cont'd)

- Unification of "prefill" and "decode"
  - There's no concept of prefill and decode
  - Schedule based on the difference between `num_compute_tokens` and `len(all_token_ids)`
- Ex3) Prefix caching



Let’s combine them all together!

Without optimizations

Prompt

<SYSTEM> You are a helpful, respectful and honest assistant. …
Structure your response strictly in a given JSON format.
<USER> Generate a description for this item: …

Output

Optimizations Combined

Prompt

<SYSTEM> You are a helpful, respectful and honest assistant. …
Structure your response strictly in a given JSON format.
<USER> Generate a description for this item: …

Output

Key Focuses in vLLM

Models

Parallelism

Inference optimizations

Performance engineering

Performance Engineering

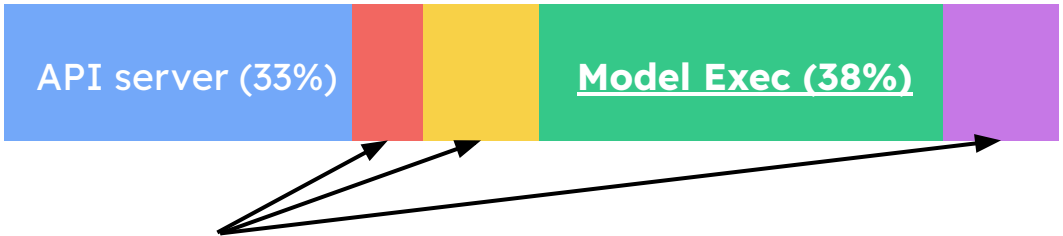
- The biggest lesson we've learned in vLLM

> "To fully utilize the GPU, we need to pay close attention to everything happening on the CPU (i.e., CPU overheads)"

- Ex) If you build an inference engine in PyTorch without caring much about CPU overheads, you will likely get 10-20% GPU utilization

CPU Overheads

- CPU overheads in an old version of vLLM



- Why so much overheads?
  - Python is slow
  - We didn't utilize multiple threads/processes efficiently in Python
  - PyTorch has performance pitfalls
  - Continuous batching makes input preparation complicated

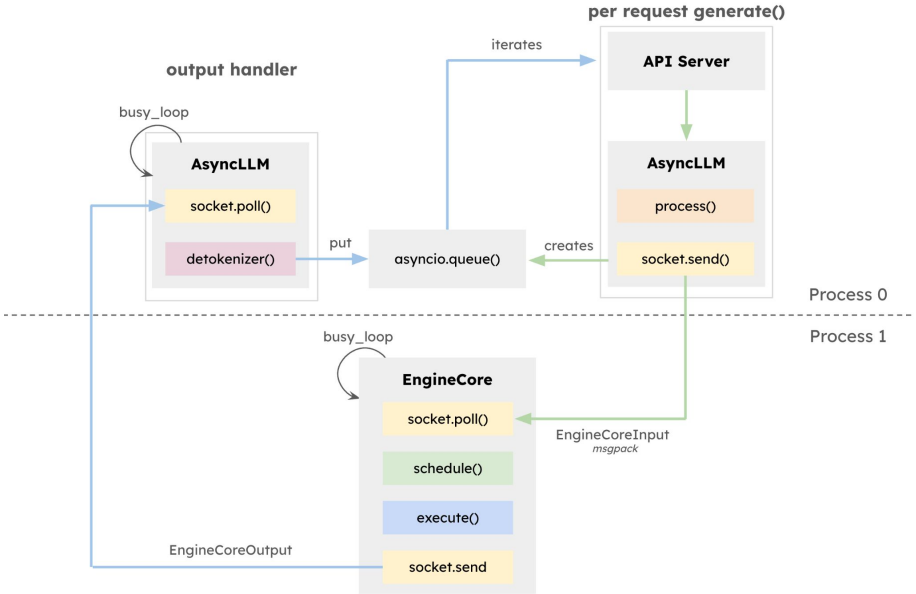
Optimized Engine Loop & API Server

Goal: Make sure GPU is not stalled

- By pre-processing
  - E.g., converting JPEG images into input tensors (resizing, cropping, …)
- By post-processing
  - E.g., de-tokenizing output token IDs into output strings
- By HTTP request handling
  - E.g., streaming outputs to 100s of concurrent users

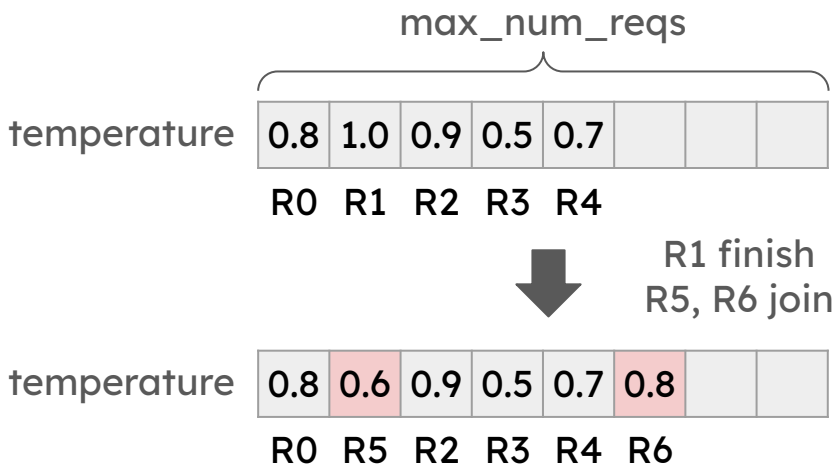
Optimized Engine Loop & API Server (cont'd)

- Two-process approach
  - Process 0 (Frontend): Pre-/post-processing & API Server
    - Importantly, de-tokenization happens in Process 0
  - Process 1 (EngineCore): Schedule & execute the model every step
    - A busy loop that is NOT blocked by Process 0

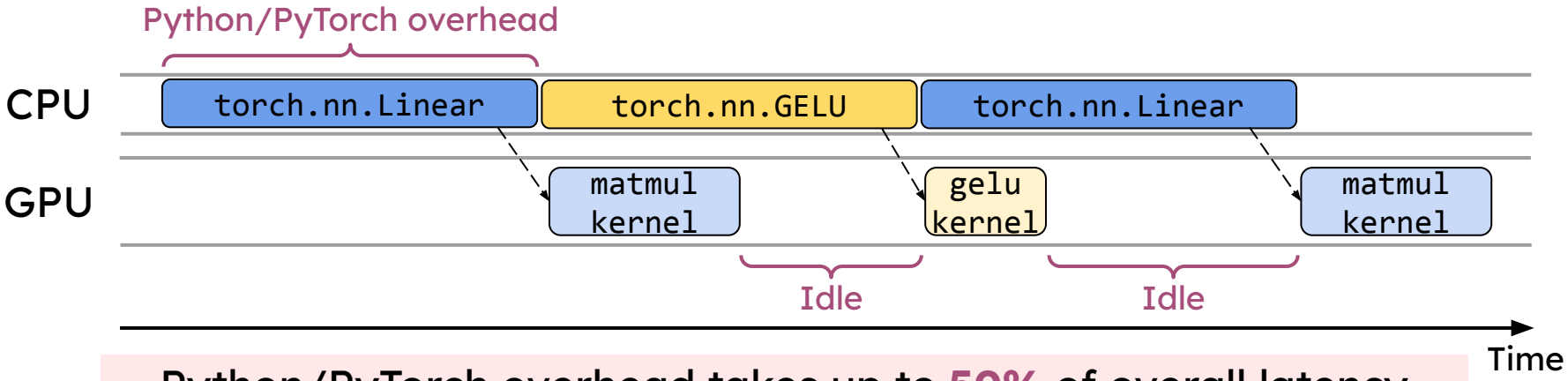


Incremental Input Preparation (Persistent Batch)

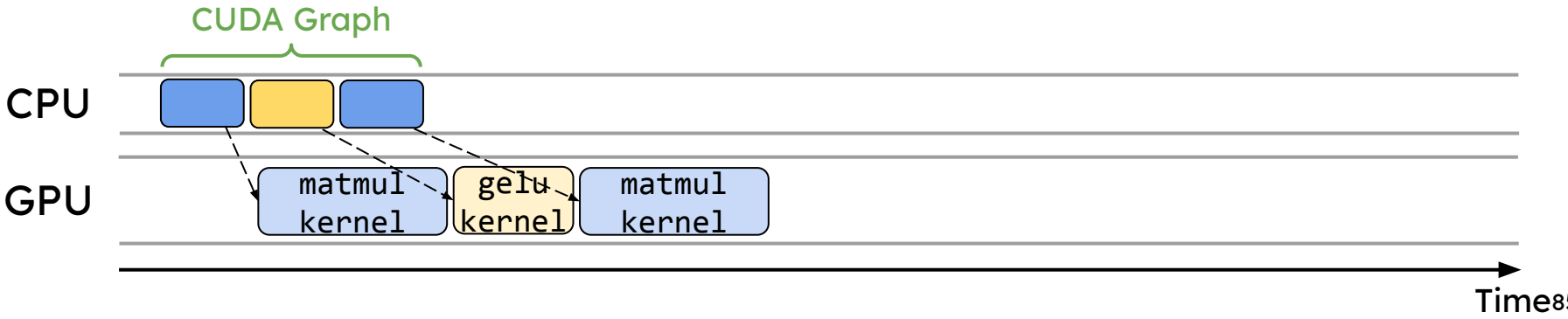
- In V0, input tensors are re-created from scratch at every step
- In V1, input tensors are cached, and we only apply the diff at each step
  - Typically, the diff is minimal because only a few requests join or finish at each step
  - The gain is larger for larger objects like block table



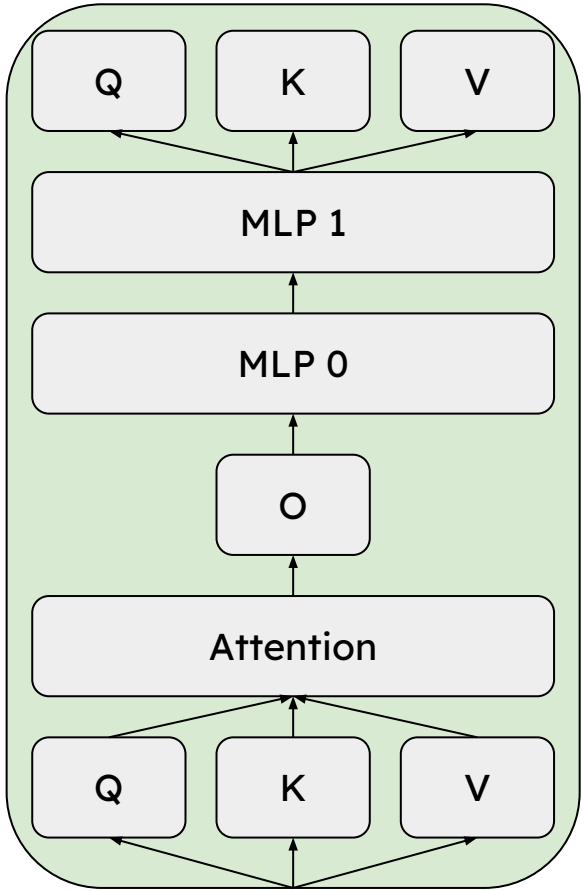
CUDA Graph for Low Latency



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



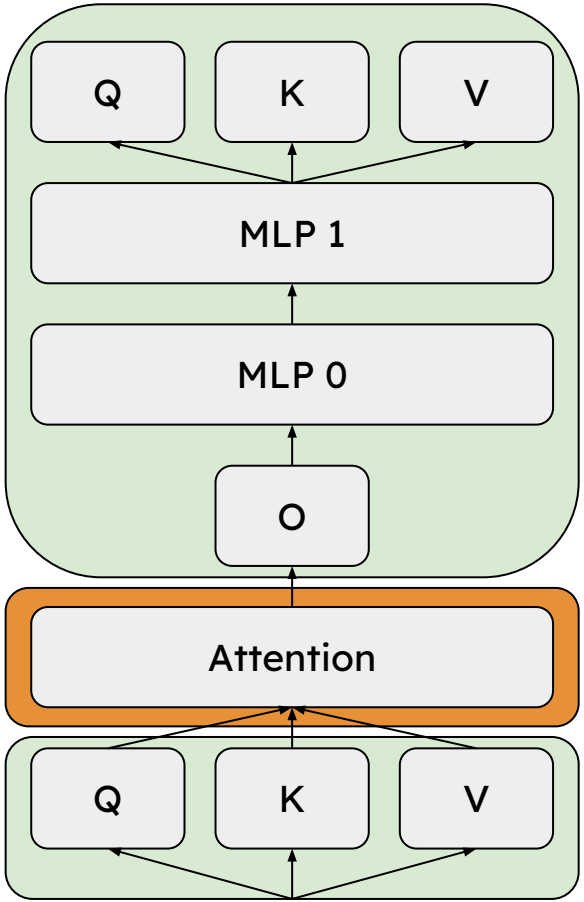
Piecewise CUDA Graphs



- V0: Single CUDA graph for the entire model
- Pros: Minimal CPU overheads in model execution
- Cons: Limited flexibility
  - Static shapes are required
  - No CPU operations are allowed

→ Increased development burden

Piecewise CUDA Graphs (cont'd)



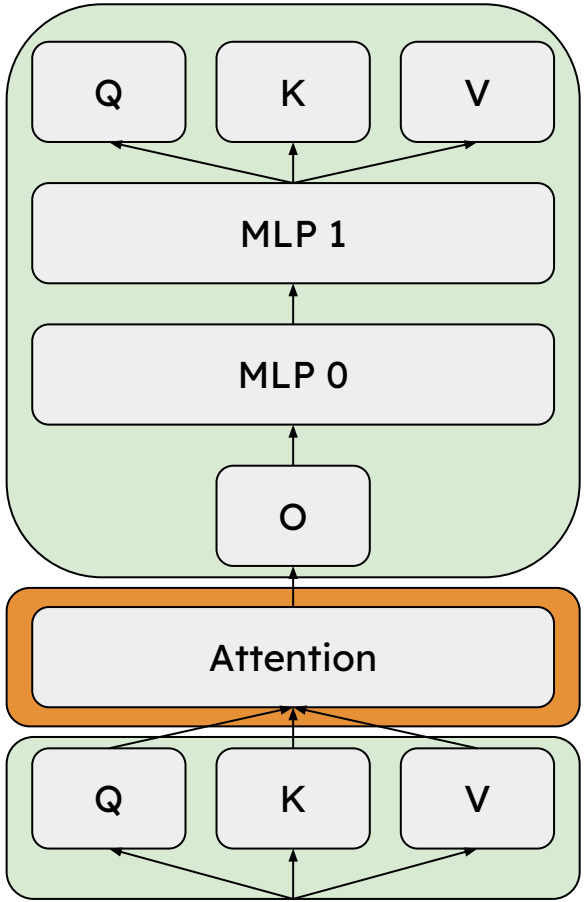
CUDA graph N

PyTorch Eager

CUDA graph N-1

Graph split using
torch.compile

Piecewise CUDA Graphs (cont'd)



- V1: Splits the model into pieces
  - Runs the attention op in **eager-mode** PyTorch
  - Runs other ops with **CUDA graphs**
    - Easy to capture, since the ops are token-wise
- Pros: Maximum freedom in implementing the attention op
  - No restriction on shapes
  - Any CPU operations are allowed
- Cons: CPU overheads unhidden by CUDA graphs could slow down the model execution
  - Negligible for 8B+ models on H100

Q & A