Better KV Cache for LLM Serving

Yuhan Liu

04/09/2025

The Trends: LLM **Inference** will be **HUGE**

Only ~10 companies are dedicated to **training** new LLMs.

But **1,000,000s** of apps and orgs run **LLM inference**

> “ The significant computational requirements for INFERENCE scaling now surpass the pre-training compute demands

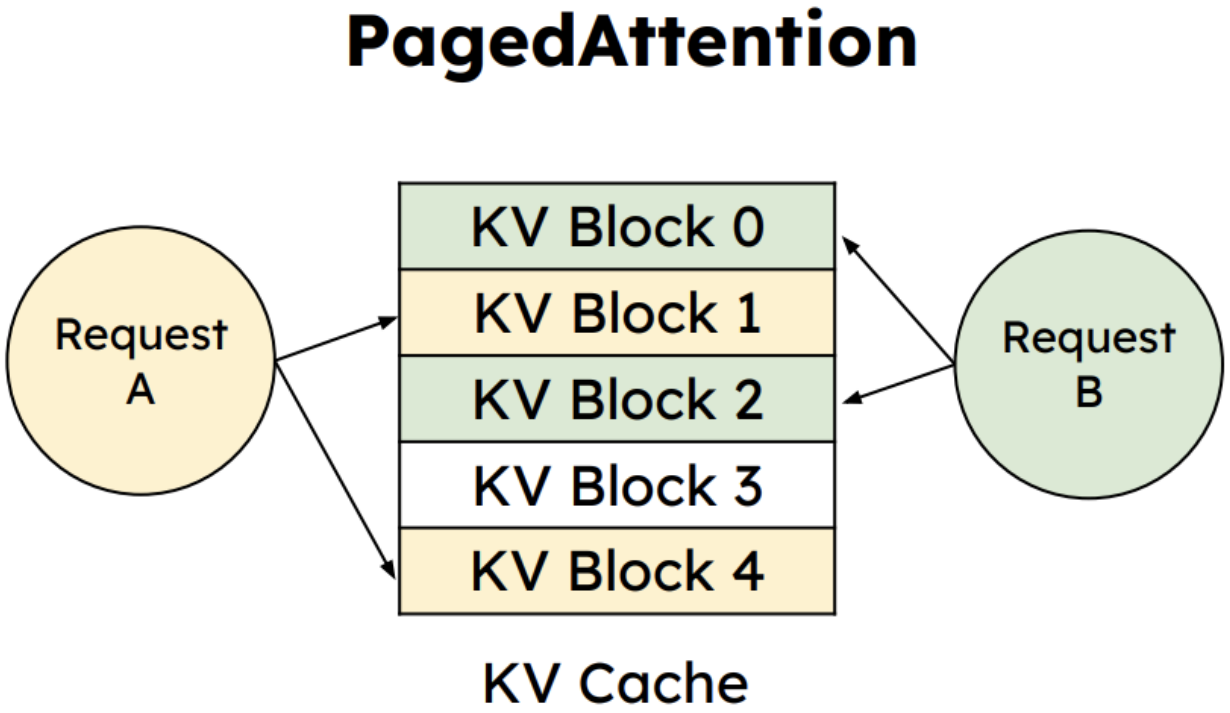
Jensen Huang, NVIDIA

Example: Inference Delay of LLMs is HIGH

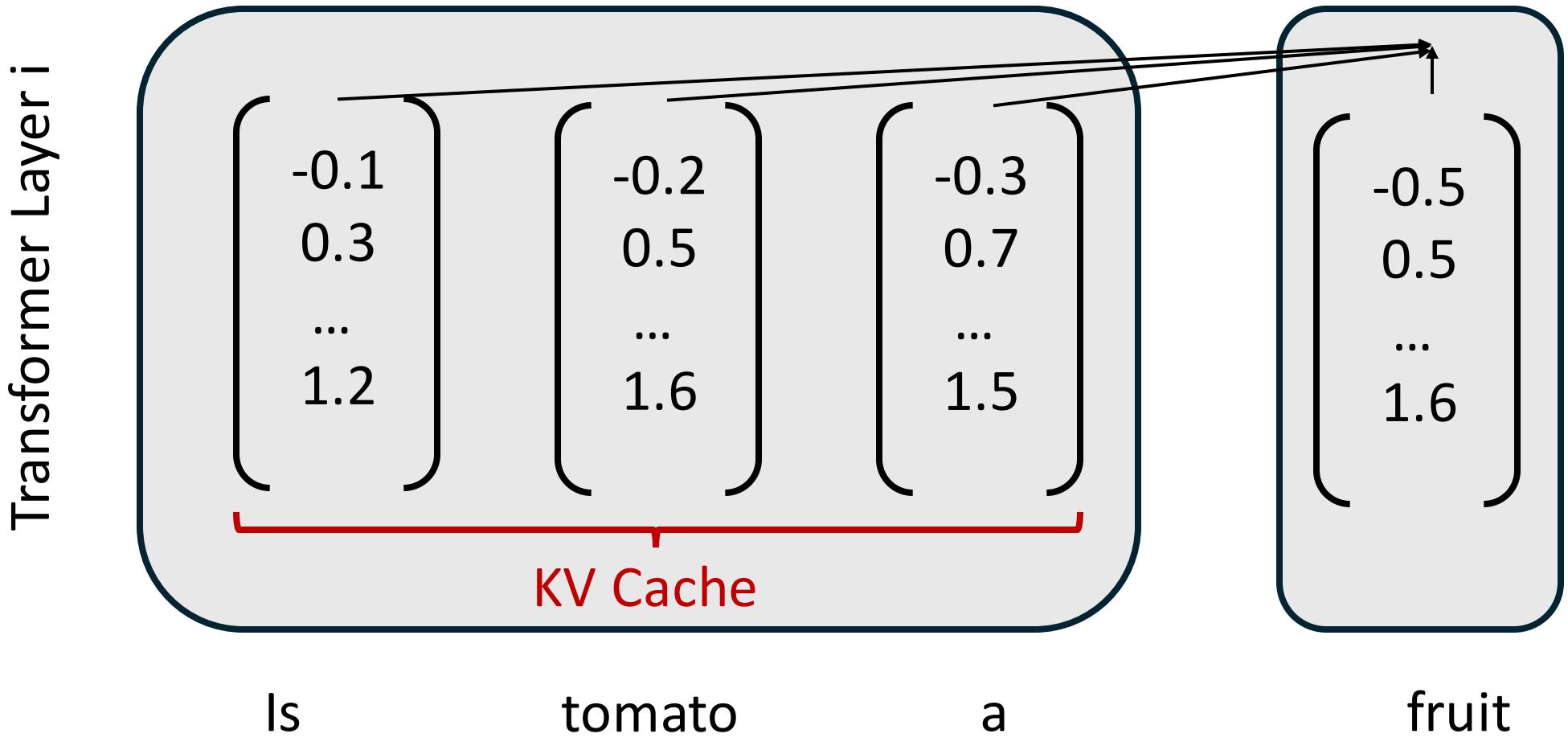
Running a 20K-Token input with Llama-3.1-70B on four A100 GPUs takes 4.8 SECONDS, with 4.2K tokens/sec throughput

Previous Lecture: PagedAttention

Application-level memory paging and virtualization for KV Cache



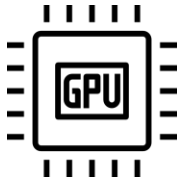
KV Cache in LLM



Better KV Cache Management for LLMs

Serving Engines

KV Cache Management



Paged Attention

And a better one!

LMCache: KV Cache Software Library for Production Use

3-10x delay savings in a lot of use cases (e.g., RAG and multi-round QA)

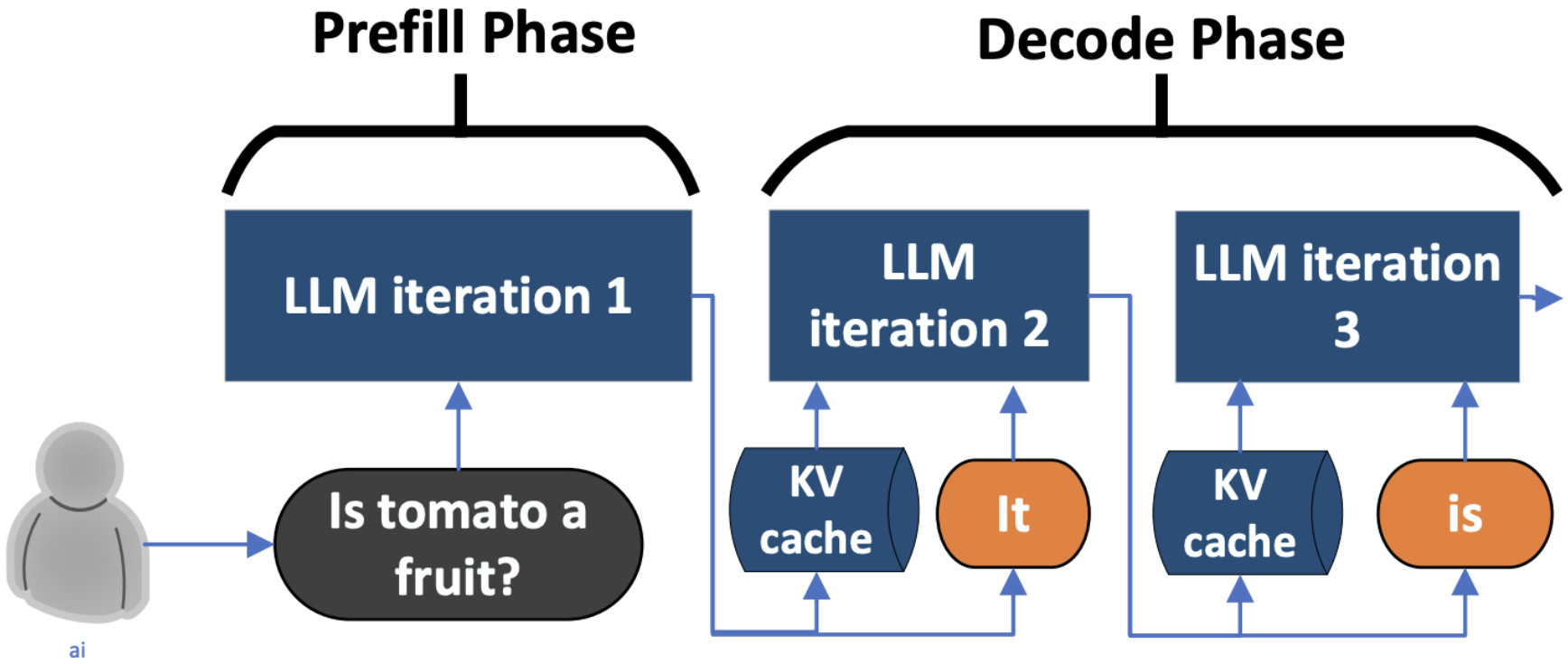
Store KV cache of reusable texts across various locations including GPU, CPU DRAM, local Disk , remote Disk

Adopted by NINE industry companies

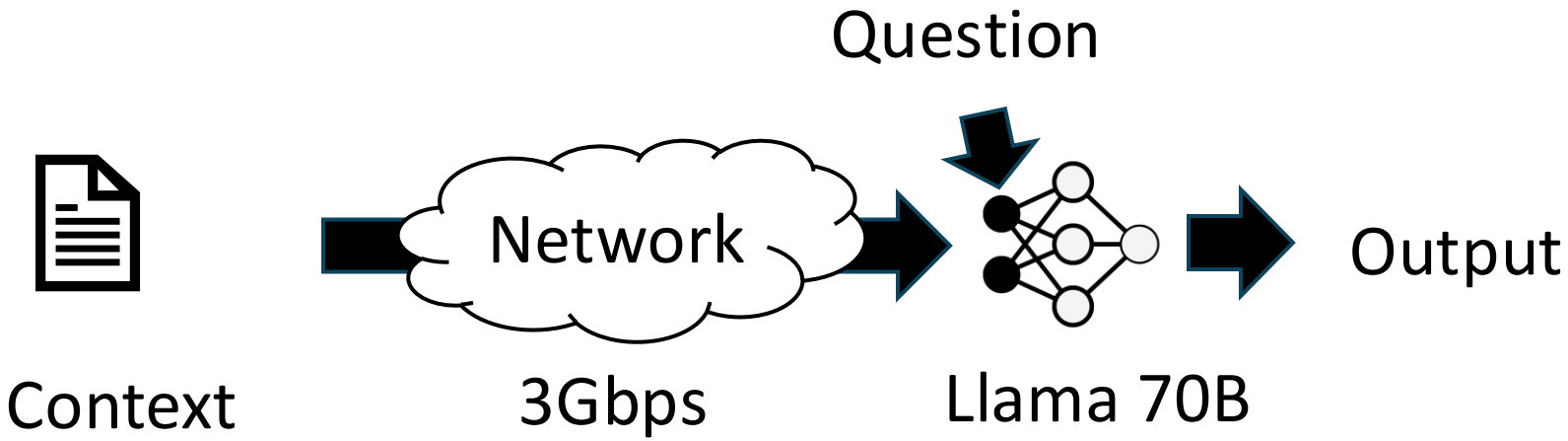
This Lecture: Prefill Optimization

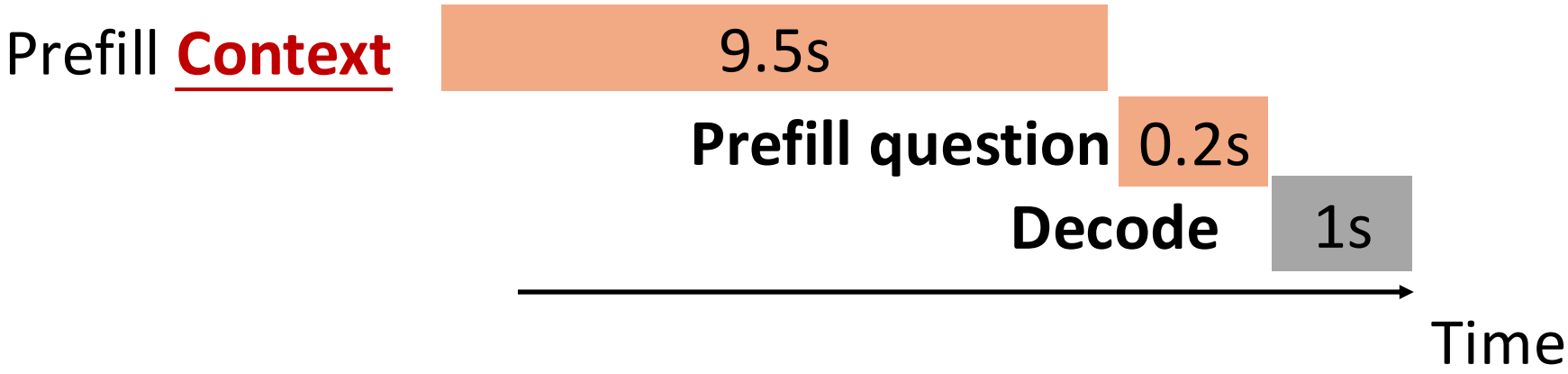
- In this lecture, we will focus on prefill optimizations within LMCache in detail
  - ***CacheGen (SIGCOMM'24):*** Compressing KV cache into compact bitstreams for faster transferring
    - Compressing deltas
    - Layer-wise quantization
    - Smart Arithmetic Coding
  - ***CacheBlend (EuroSys'25 Best Paper):*** Blending different KV cache for different chunks in RAG for reducing inference latency
    - Selective recompute highly deviated tokens
    - Loading controller to pipeline recompute with loading

What’s Prefill?

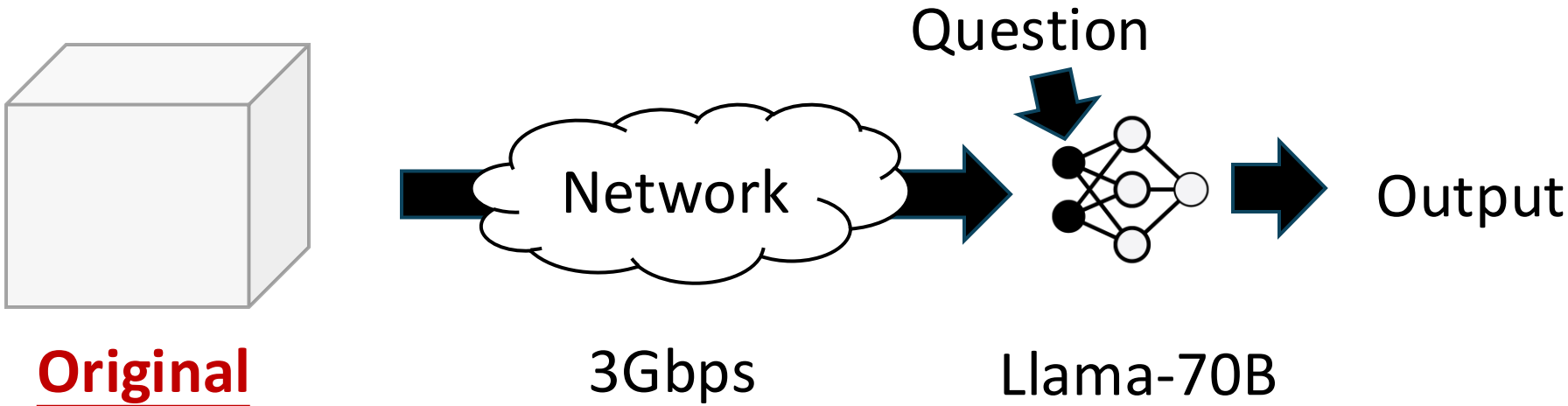


Challenge in KV Cache Management



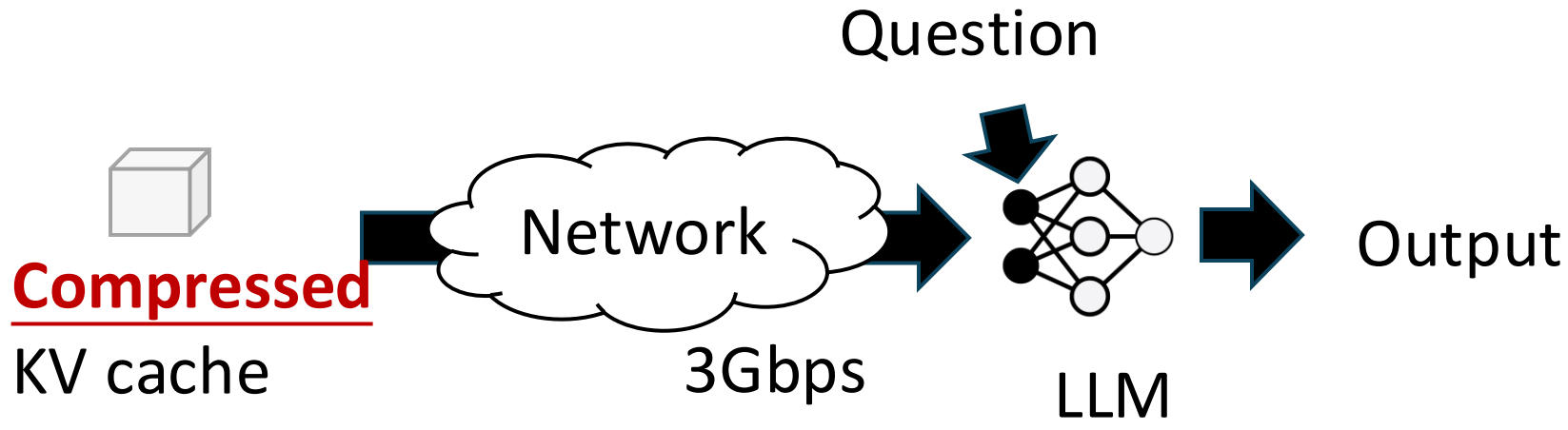


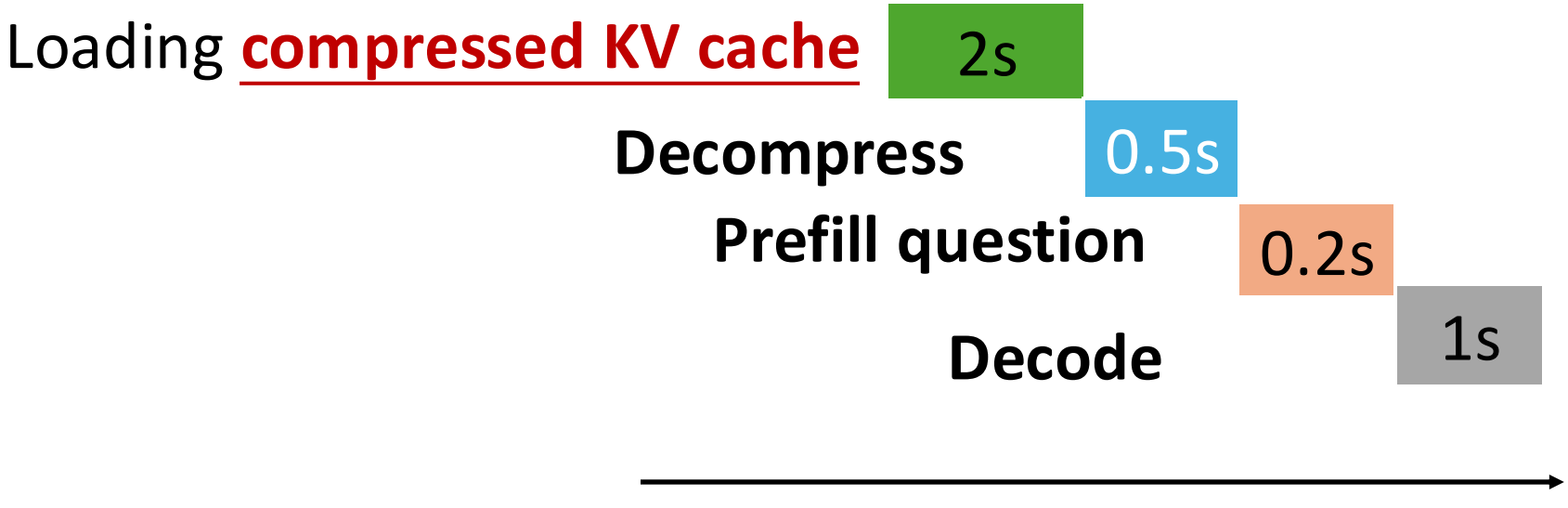
Challenge in KV Cache Management



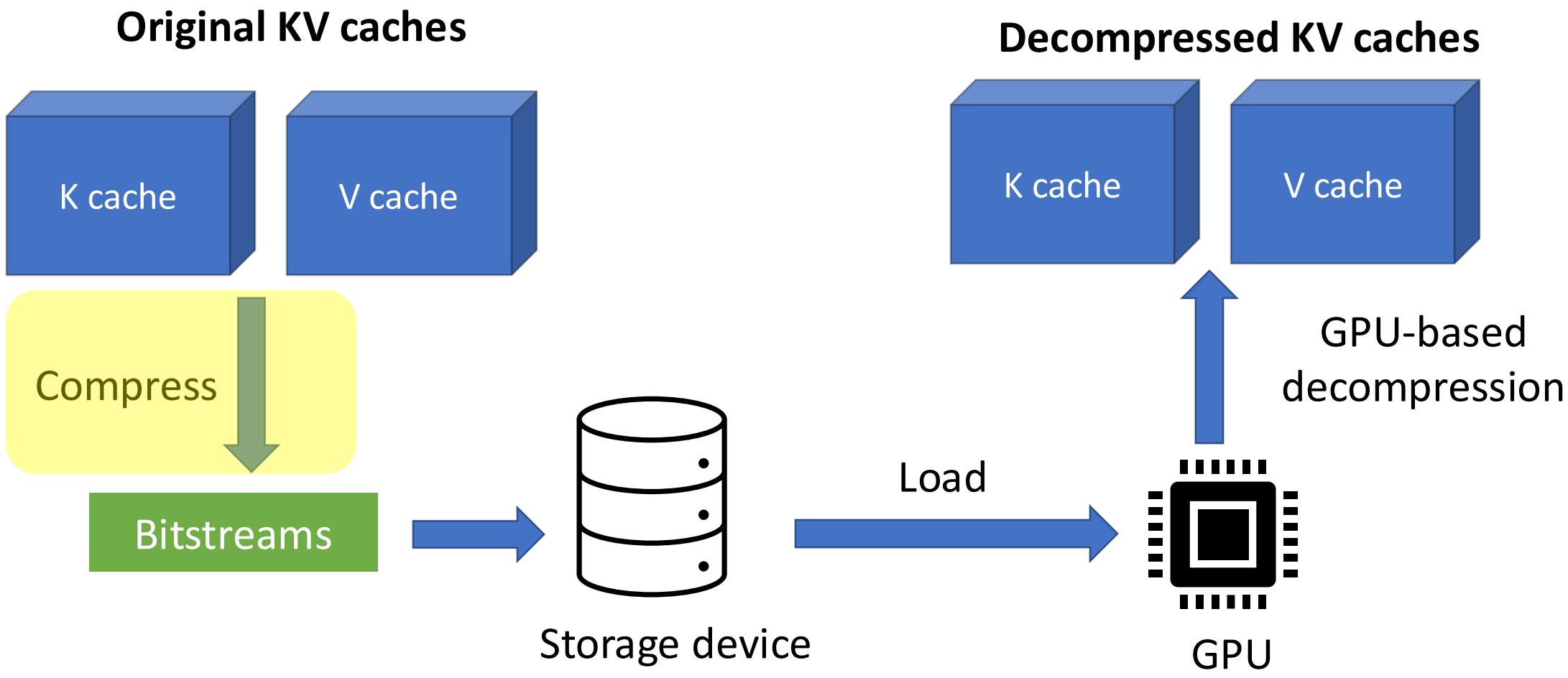
We want to reduce the size of the KV Cache to reduce prefill computation and loading time

Compressing KV Cache for Faster Transferring

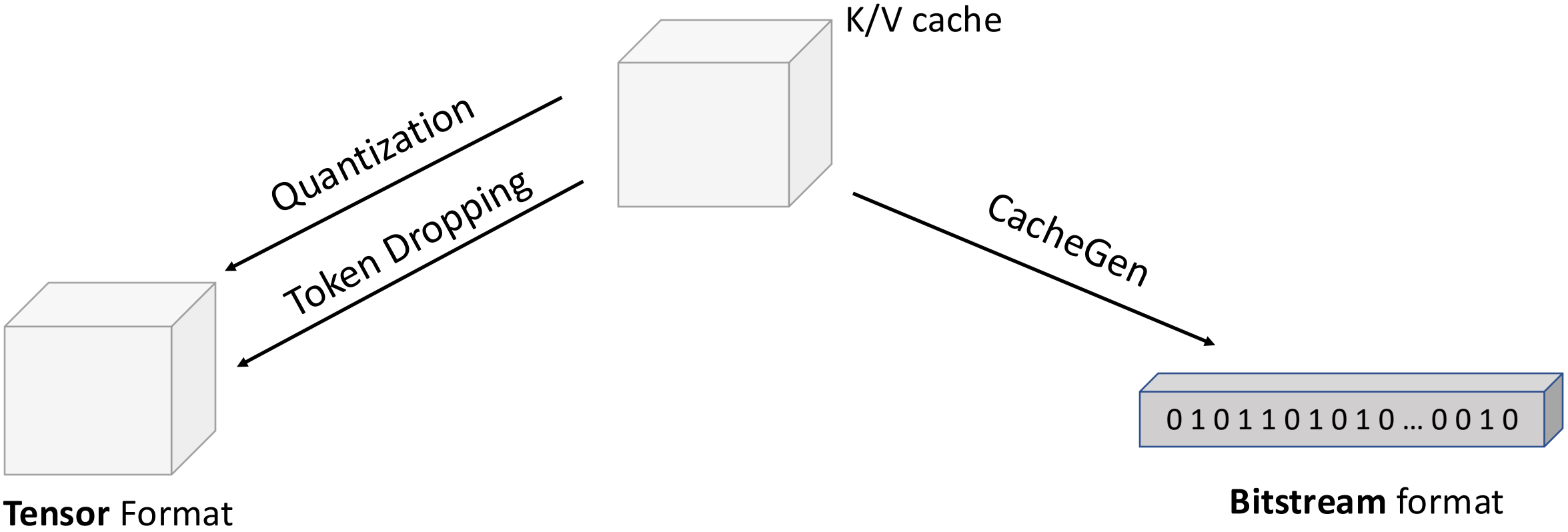




Prefill with Compressed KV Cache

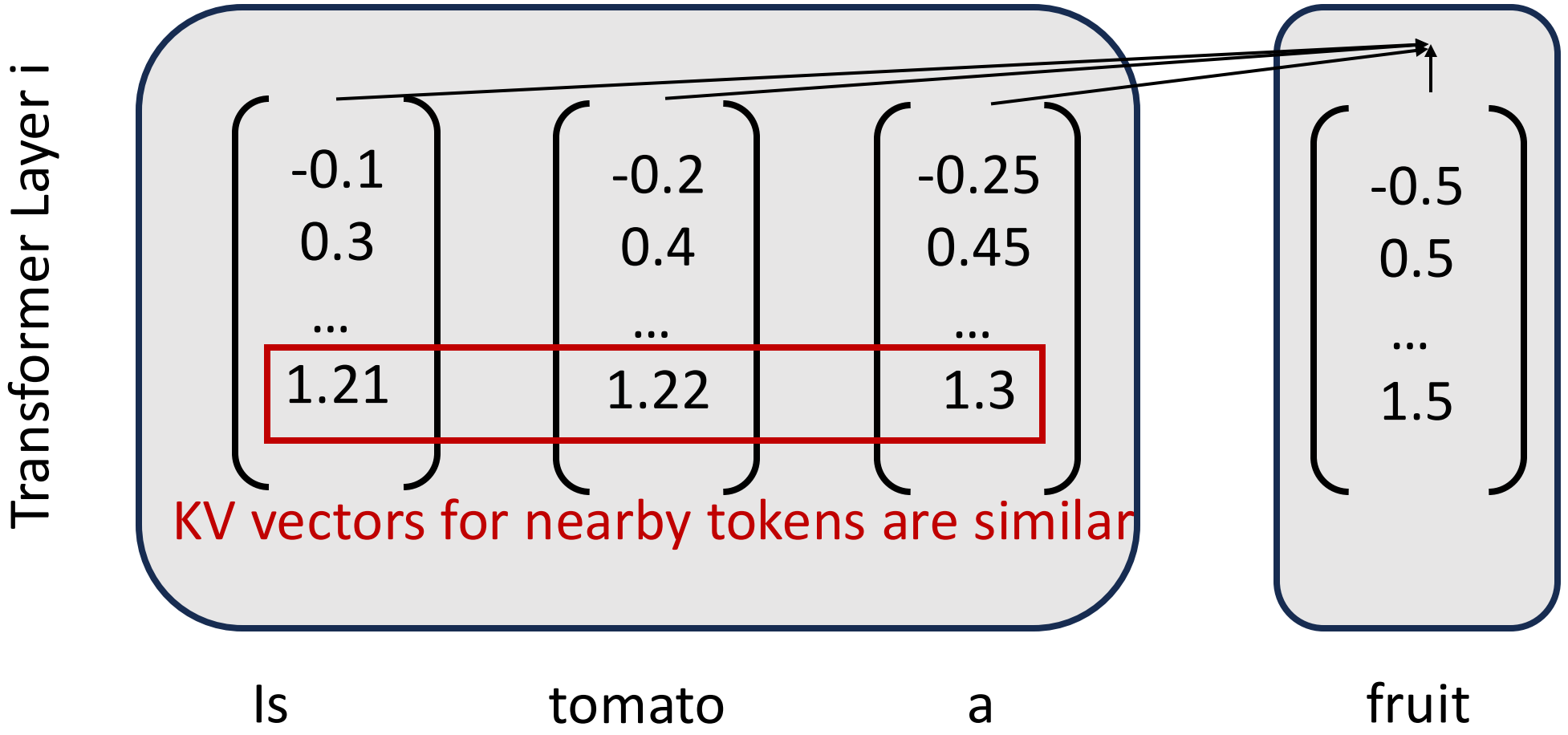


CacheGen Moves **Bits**, not Tensors



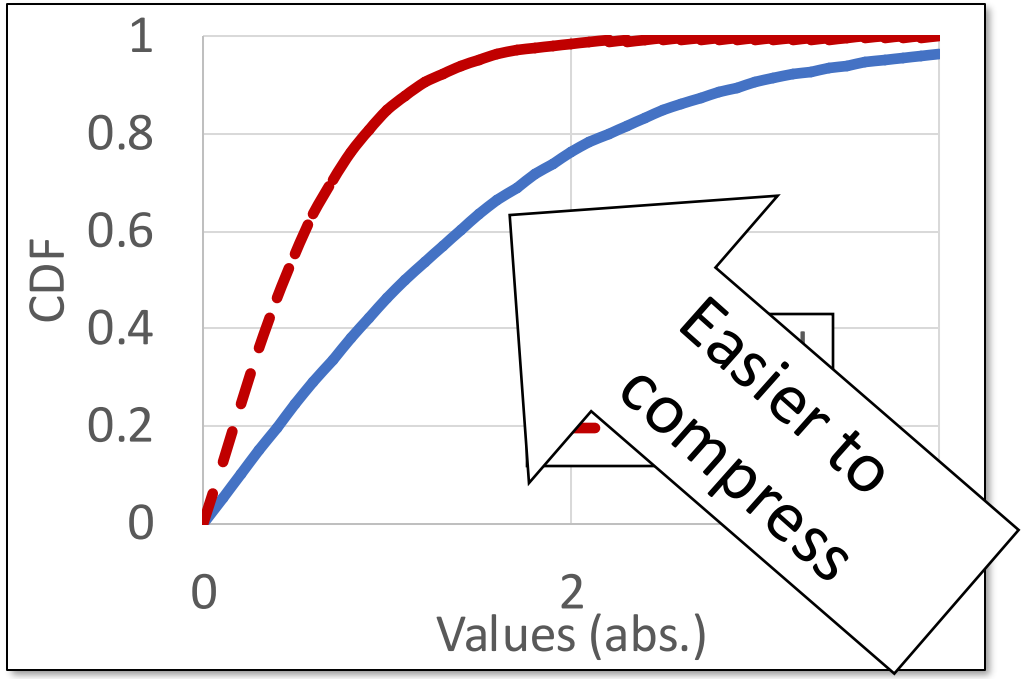
CacheGen compresses tensor into **BITSTREAMS** instead of Tensors!

Opportunity #1: Nearby tokens have similar KV vectors



Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

Technique #1: Compressing similar values with deltas

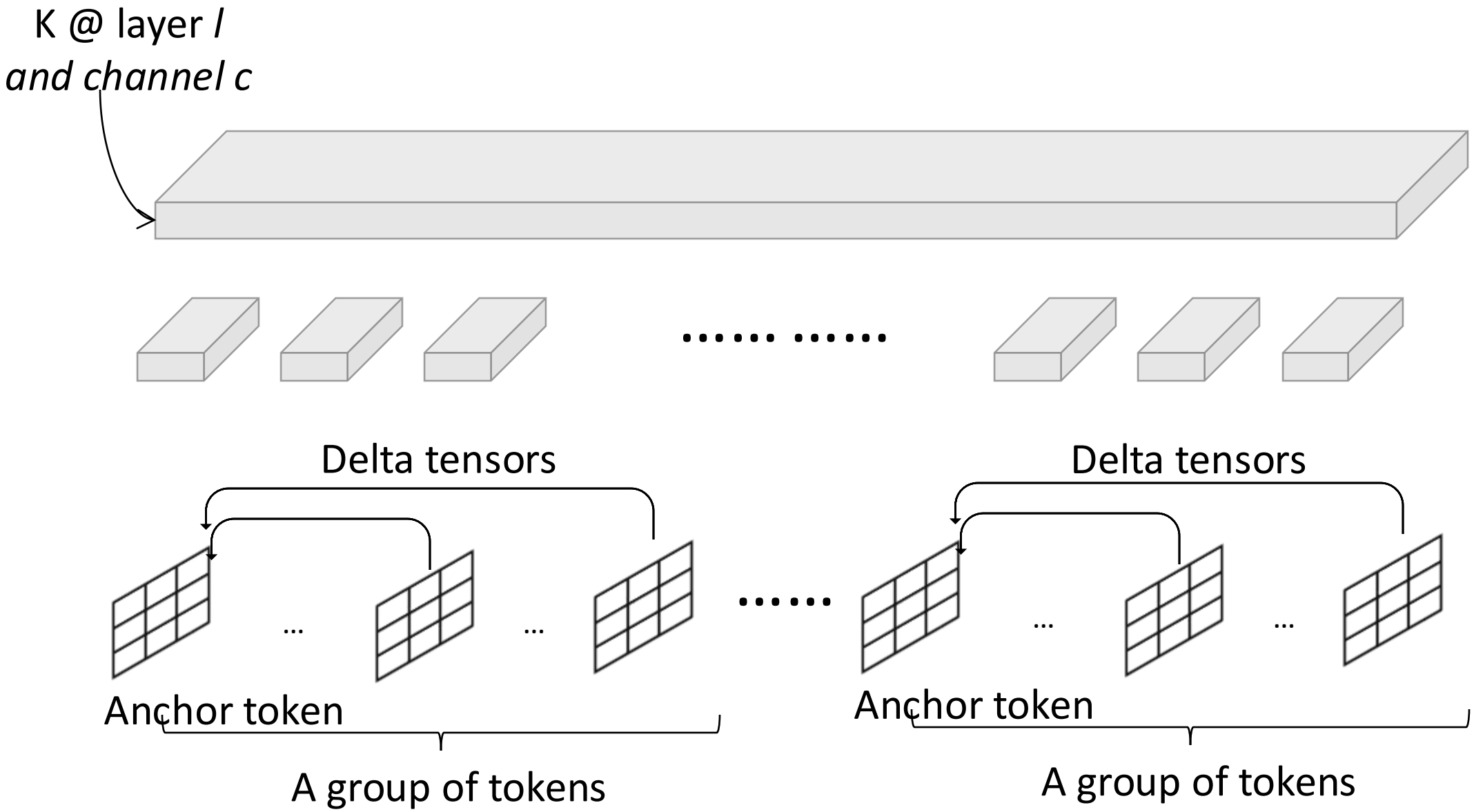


For any token i at a channel and a layer

Original: $|K_i|, |V_i|$

Delta: $|K_i - k_{i-1}|, |V_i - V_{i-1}|$

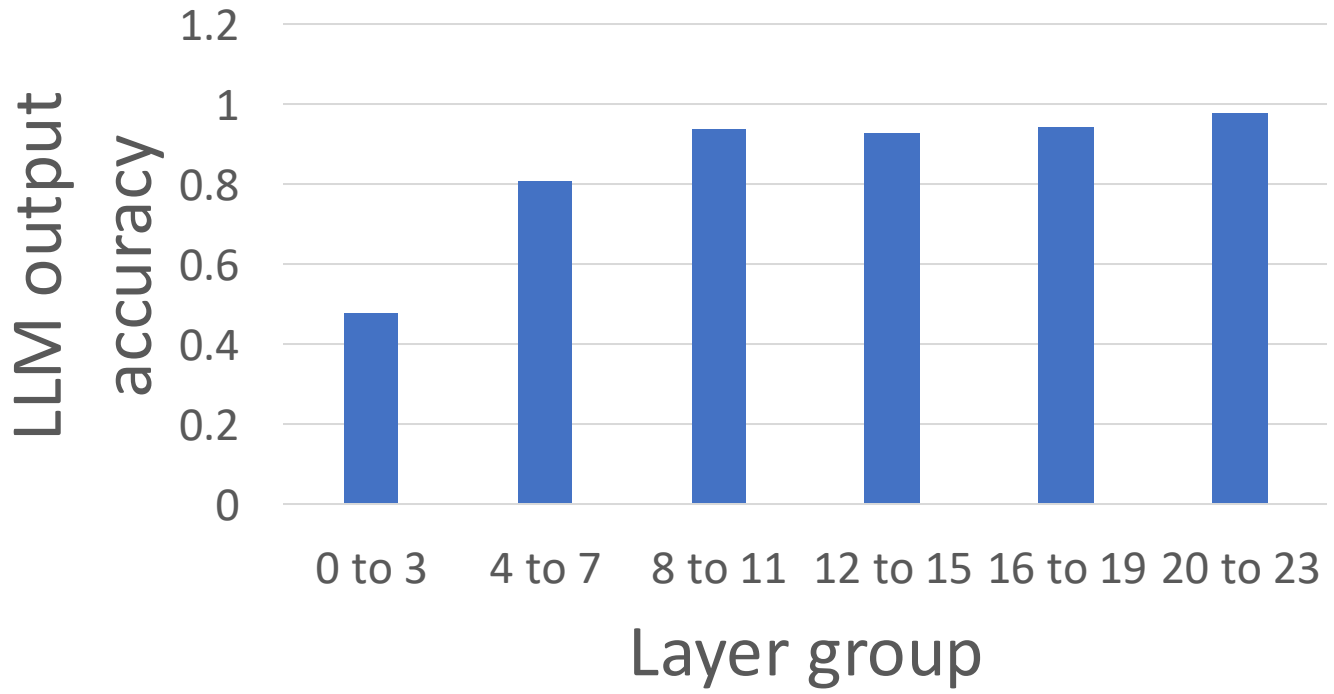
Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.



Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

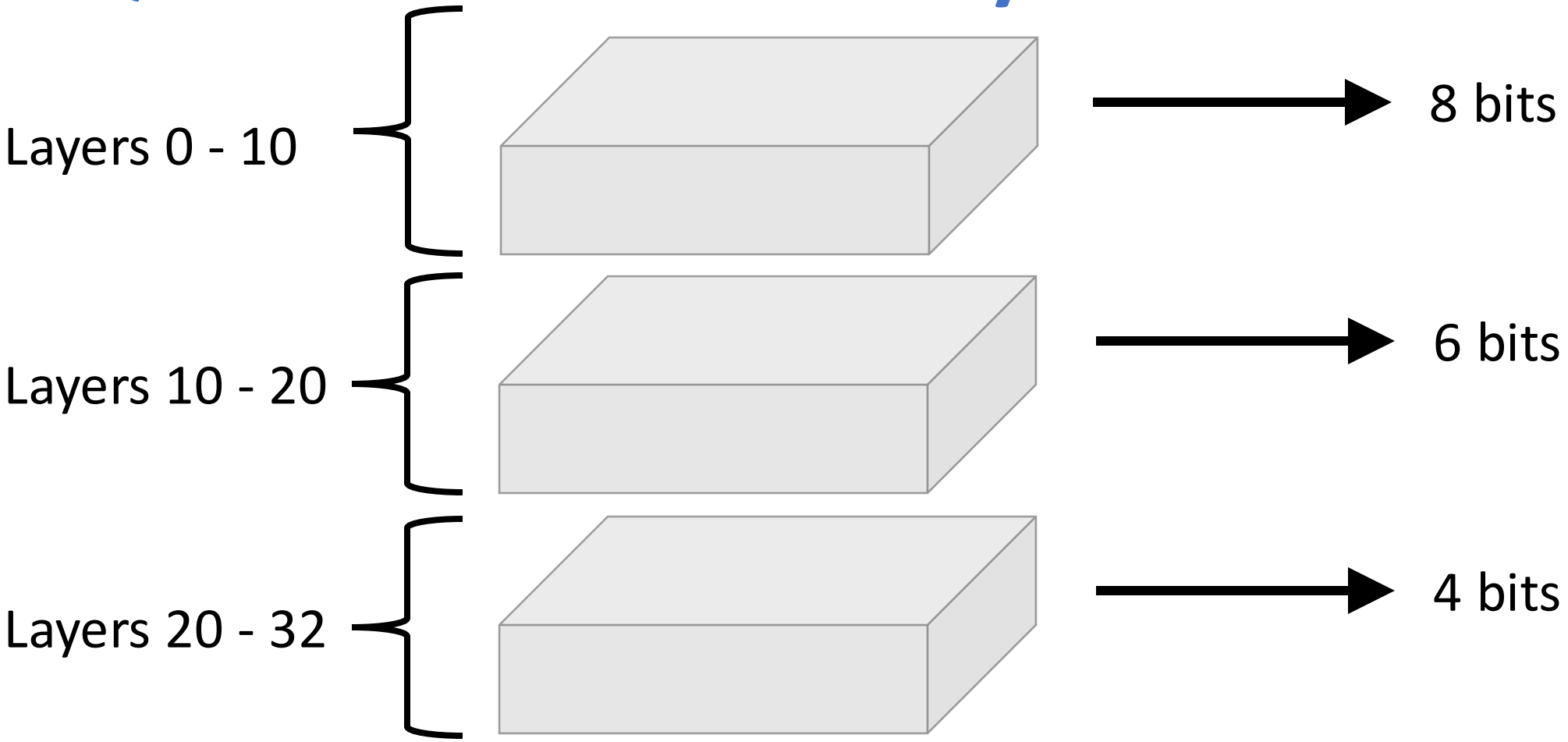
Opportunity #2: Different Layers Have Different Sensitivity to Compression Errors

Apply the same amount of rounding error to different layer groups



Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

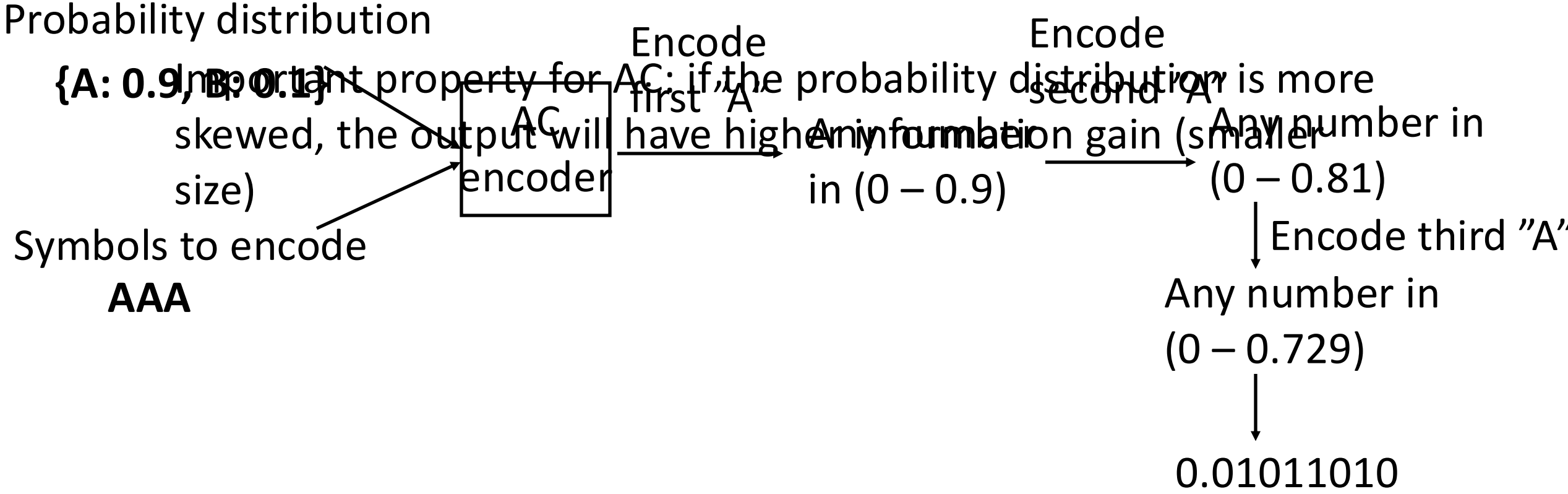
Technique #2: Applying More Aggressive Quantization to Later Layers



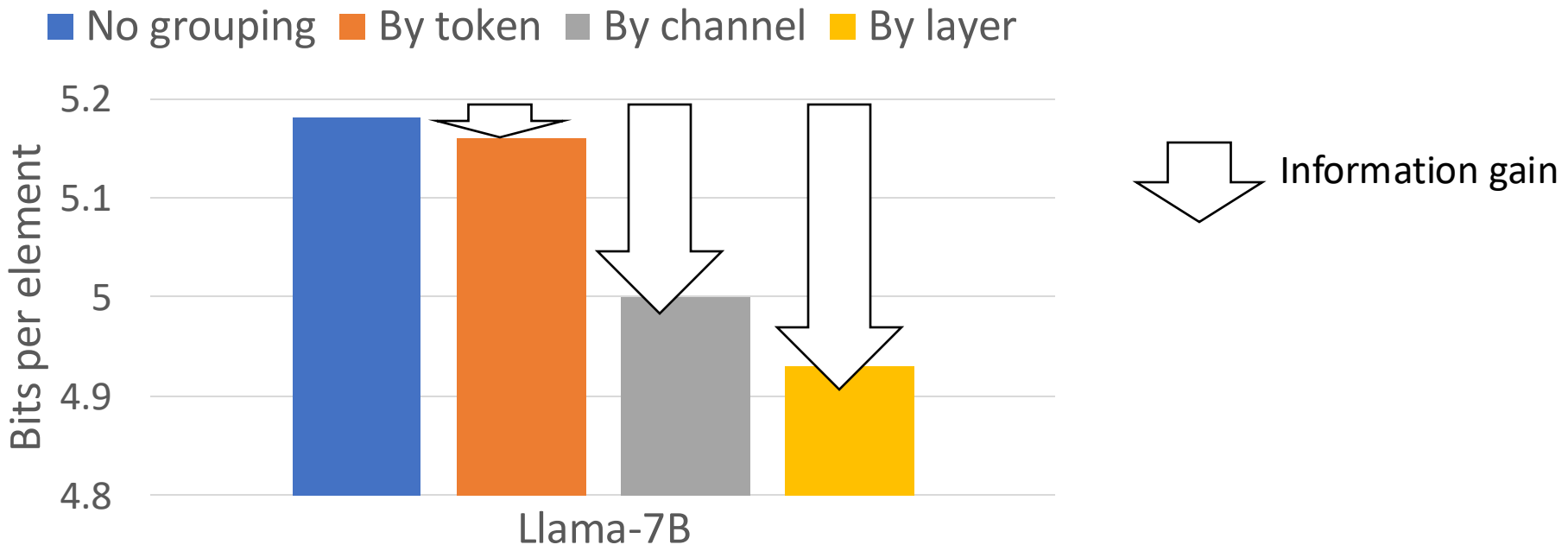
Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

Introduction to Arithmetic Coding

Arithmetic Coding (AC) is a lossless compression which encodes the input data into bitstreams



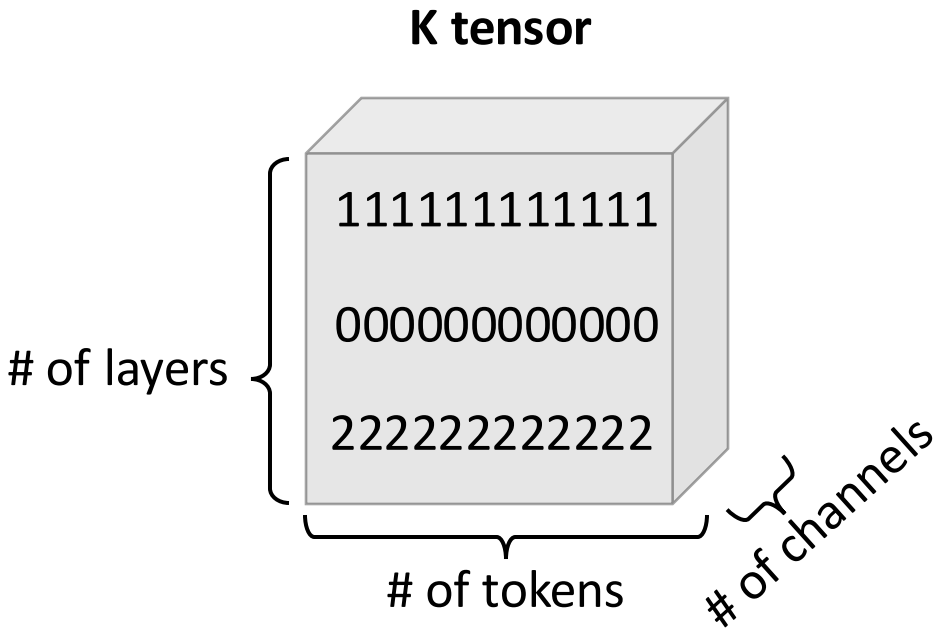
Opportunity #3: Grouping KV by channel and layer is better than other groupings



The information gain of grouping by their channel or layer > the information gain of grouping by tokens.

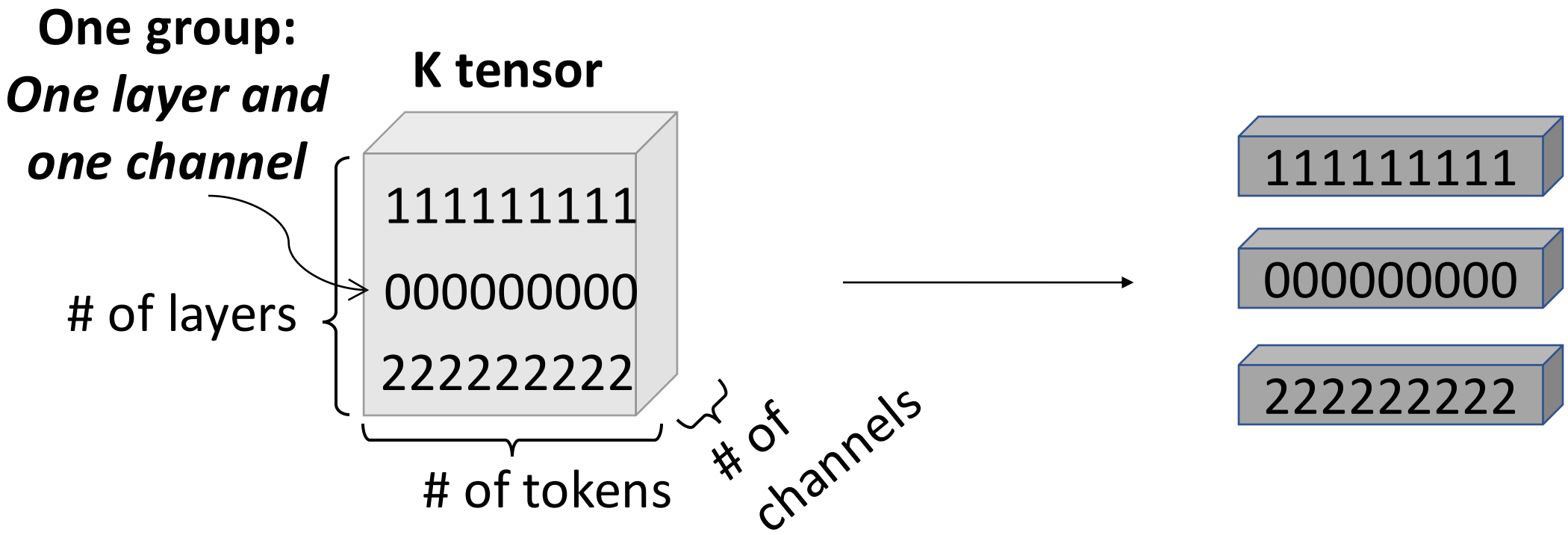
serving." *SIGCOMM 2024 Conference*.

Technique #3: Smart AC by grouping KV by channel and layer



A strawman: put every element together and do arithmetic coding

Technique #3: Smart AC by grouping KV by channel and layer

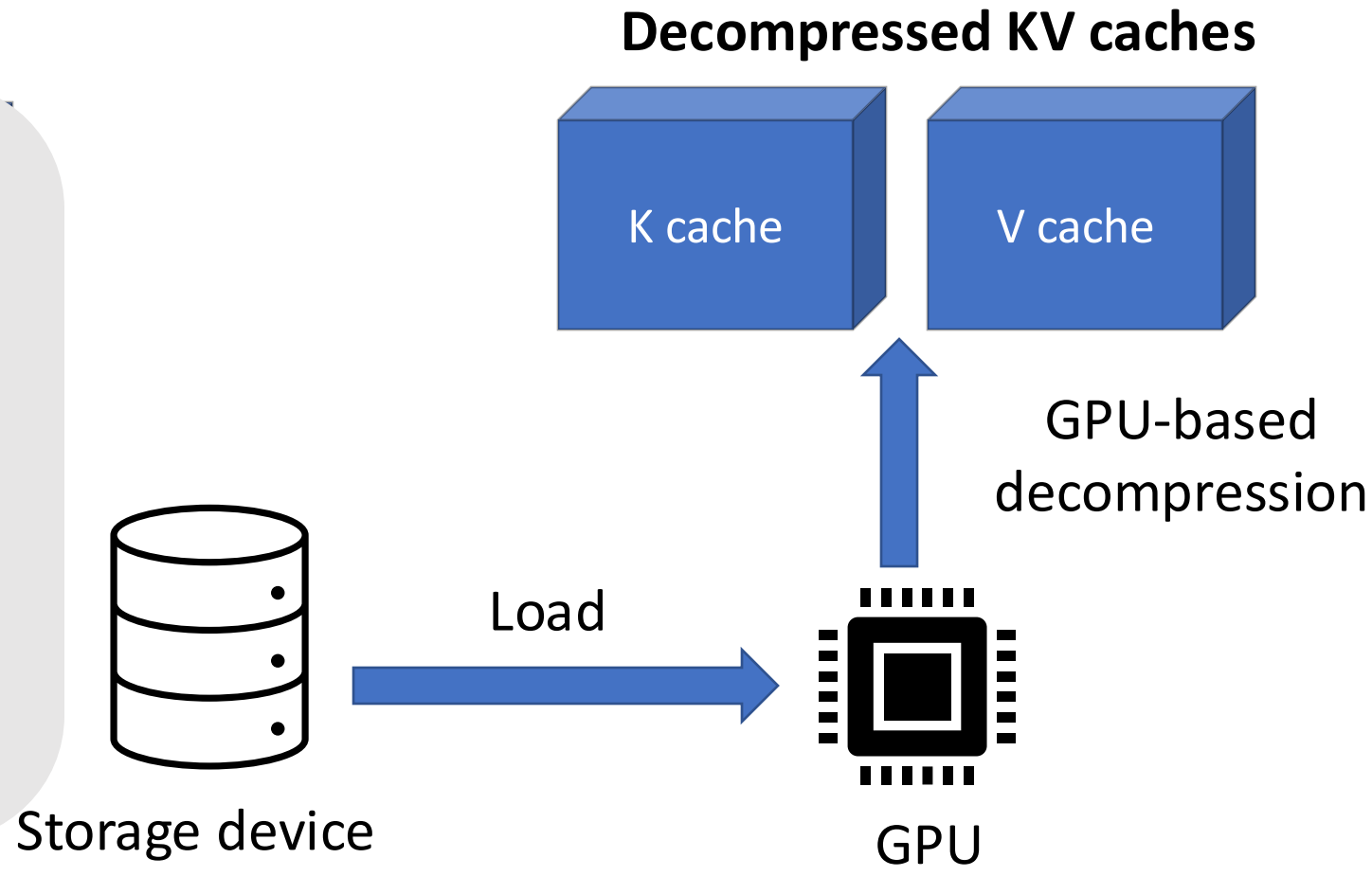


Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

Prefill with Compressed KV Cache

Original KV caches

- Compressing similar values with deltas
- Layer-wise quantization
- Smart Arithmetic Coding



Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

Minimizing decompression overhead

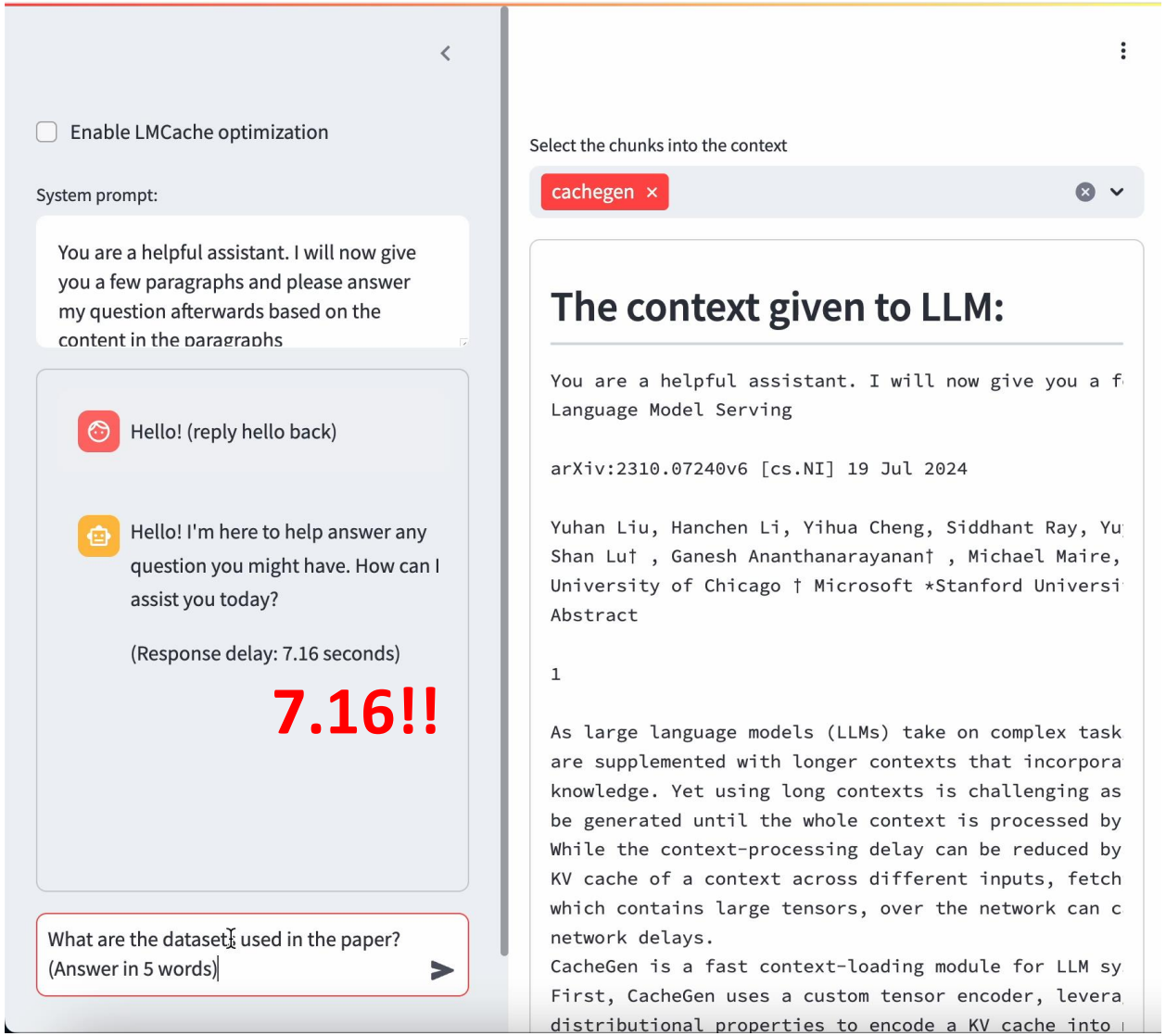
It is slow to decompress the bitstreams to tensor format with Arithmetic Coding

We implement GPU-based decompression with CUDA kernels

We pipeline loading KV caches with decompression

Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

Naïve Prefill



CacheGen

2.7 ☺

Implementation: Compression function

```
def serialize(kv_cache, config, compressed_bits, chunk_size) ->
CacheGenGPUEncoderOutput:
    # Split KV into separate K and V tensors
    …
    # Quantize the tensors
    ….
    # Calculate CDFs for AC
    …
    # Initialize output buffers
    …
    # Process in chunks
    data_chunks = []
    for i in range(0, chunk_size, CACHEGEN_CHUNK_SIZE):
     ….
```

https://github.com/LMCache/LMCache/blob/dev/lmcache/experimental/storage_backend/naive_serde/cachegen_encoder.py

Implementation: Decompression function

```
def from_bytes(bytestream) -> torch.Tensor:
    # 1. Load encoded data from bytes
    …
    # 2. Move tensors to GPU
    …
    # 3. Get dimensions
    …
    # 4. Decode the data
    key, value = decode_function_gpu(…)

    # 5. Dequantize the tensors
    ...
```

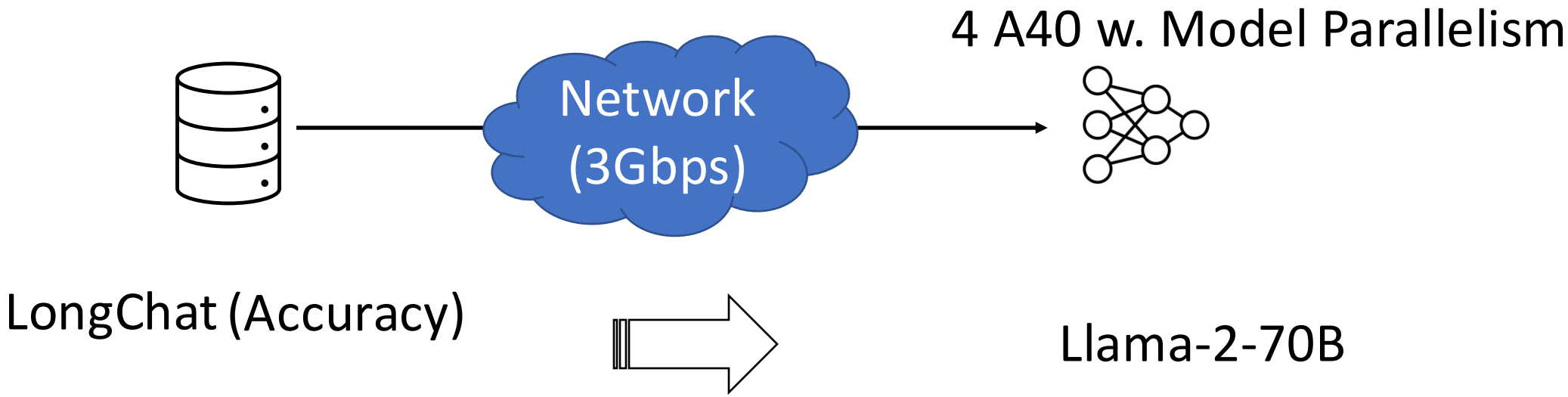
https://github.com/LMCache/LMCache/blob/dev/lmcache/experimental/storage_backend/naive_serde/cachegen_decoder.py

Implementation: Decompression CUDA Kernel

```
// Main Decoding Kernel
template<…>
__global__ void decode_with_accessor_kernel(
…
) {
// Shared memory allocation:
// 1. CDF tensor [MAX_LP, BLOCK_SIZE]
// 2. Bytestream buffer [BLOCK_SIZE, OUTPUT_BUFFER_LENGTH_PER_THREAD]
// 3. Lengths buffer [BLOCK_SIZE]

// Main decoding loop:
// 1. Initialize arithmetic coding state (low, high, value)
// 2. For each token:
// - Calculate count from current state
// - Find symbol using binary search
// - Update arithmetic coding state
// - Handle renormalization
}
```

https://github.com/LMCache/LMCache/blob/dev/csrc/ac_dec.cu

Evaluation



System Metrics:

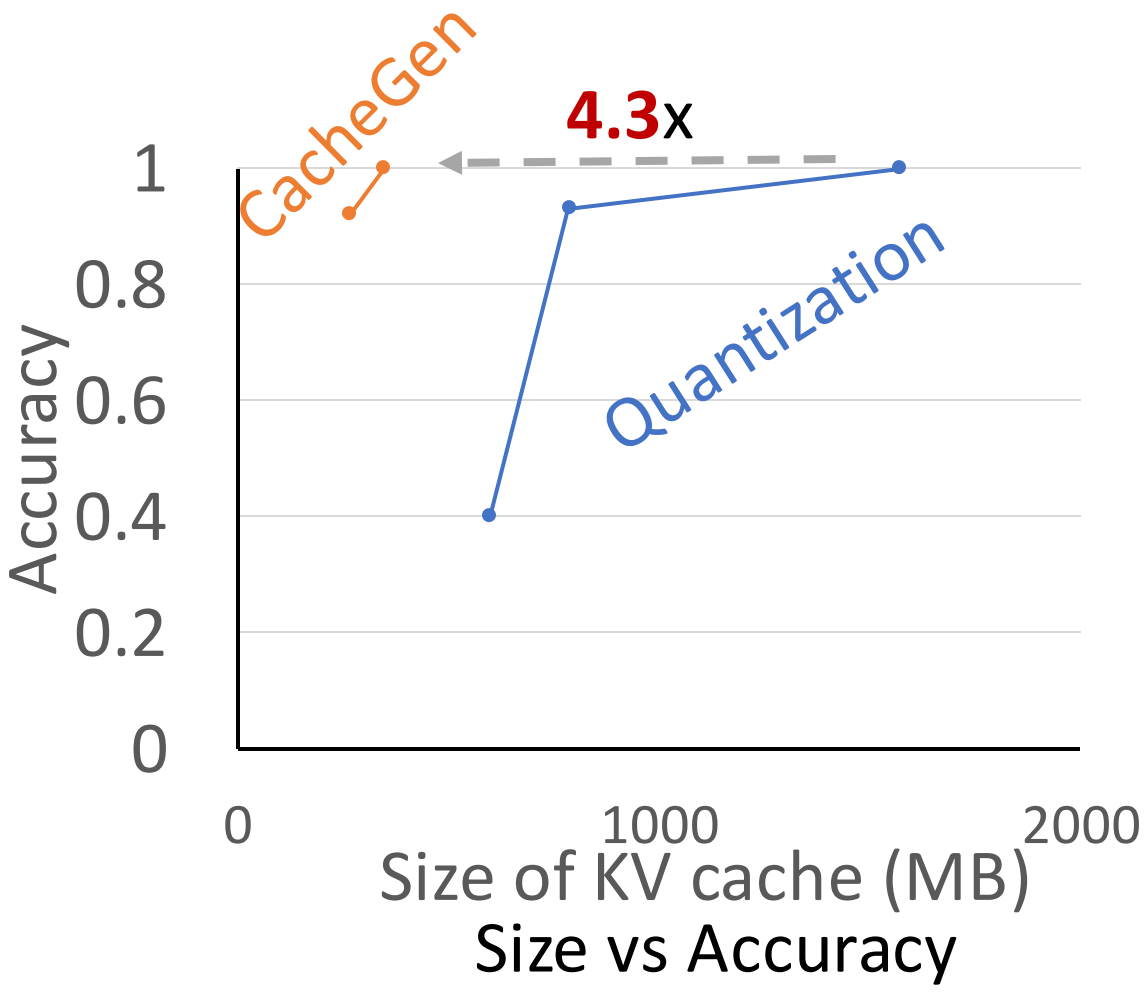
Size of KV caches (MB)

TTFT: Time-to-first-token (Total Prefill Delay)

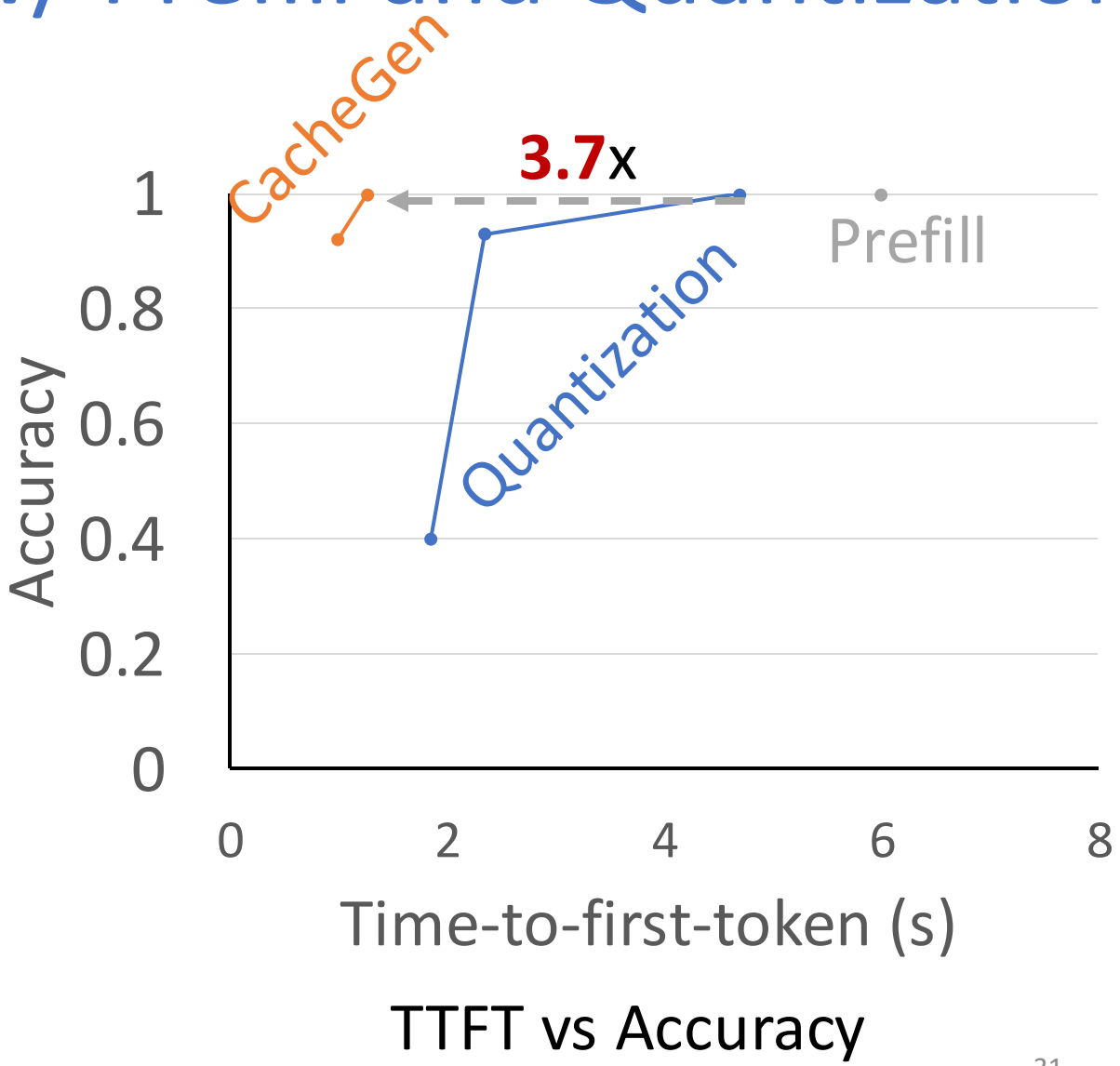
Workload Generator: Send different LongChat requests sequentially

Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

CacheGen Achieves More Efficiency with High Accuracy Compared w/ Prefill and Quantization



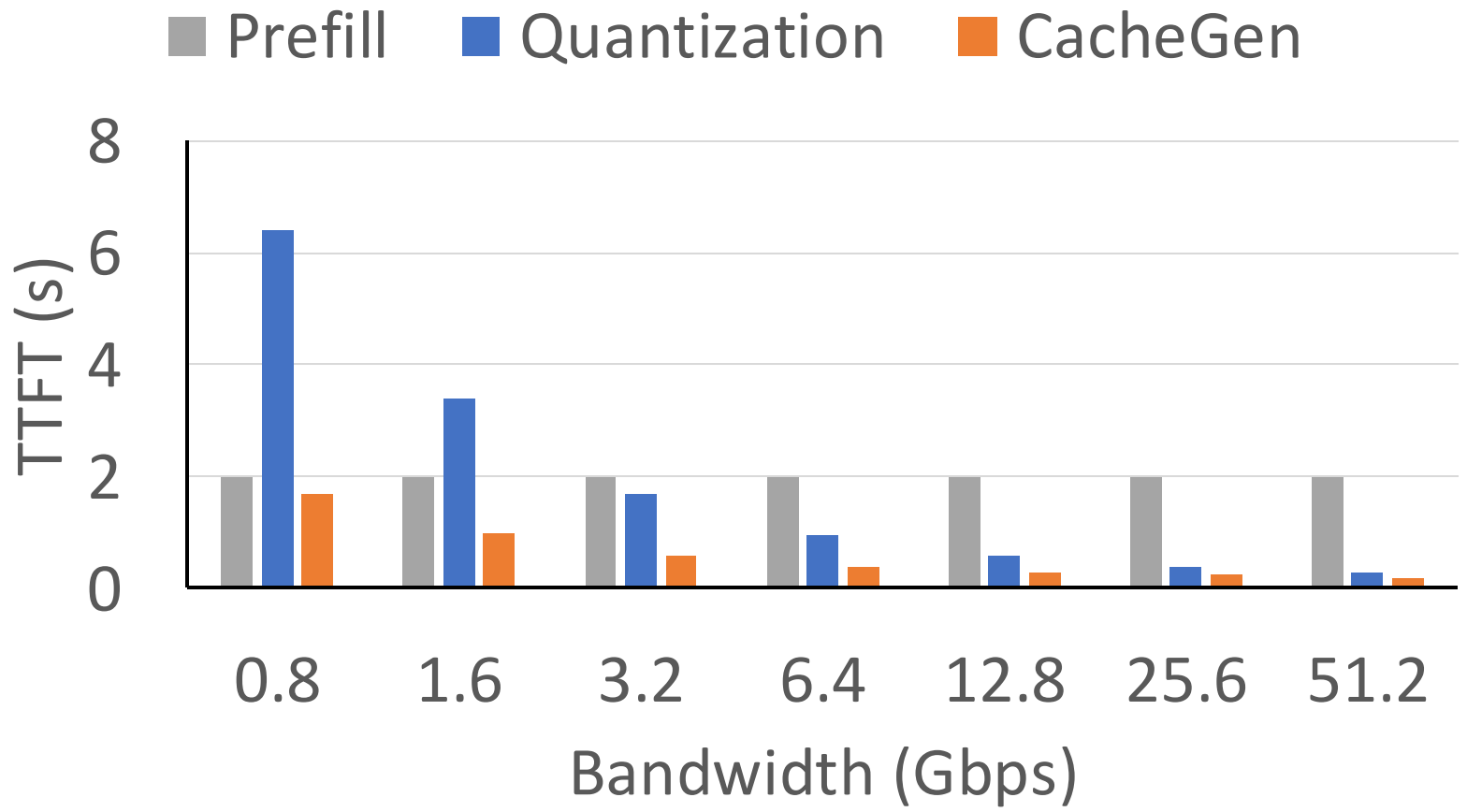
Size vs Accuracy



TTFT vs Accuracy

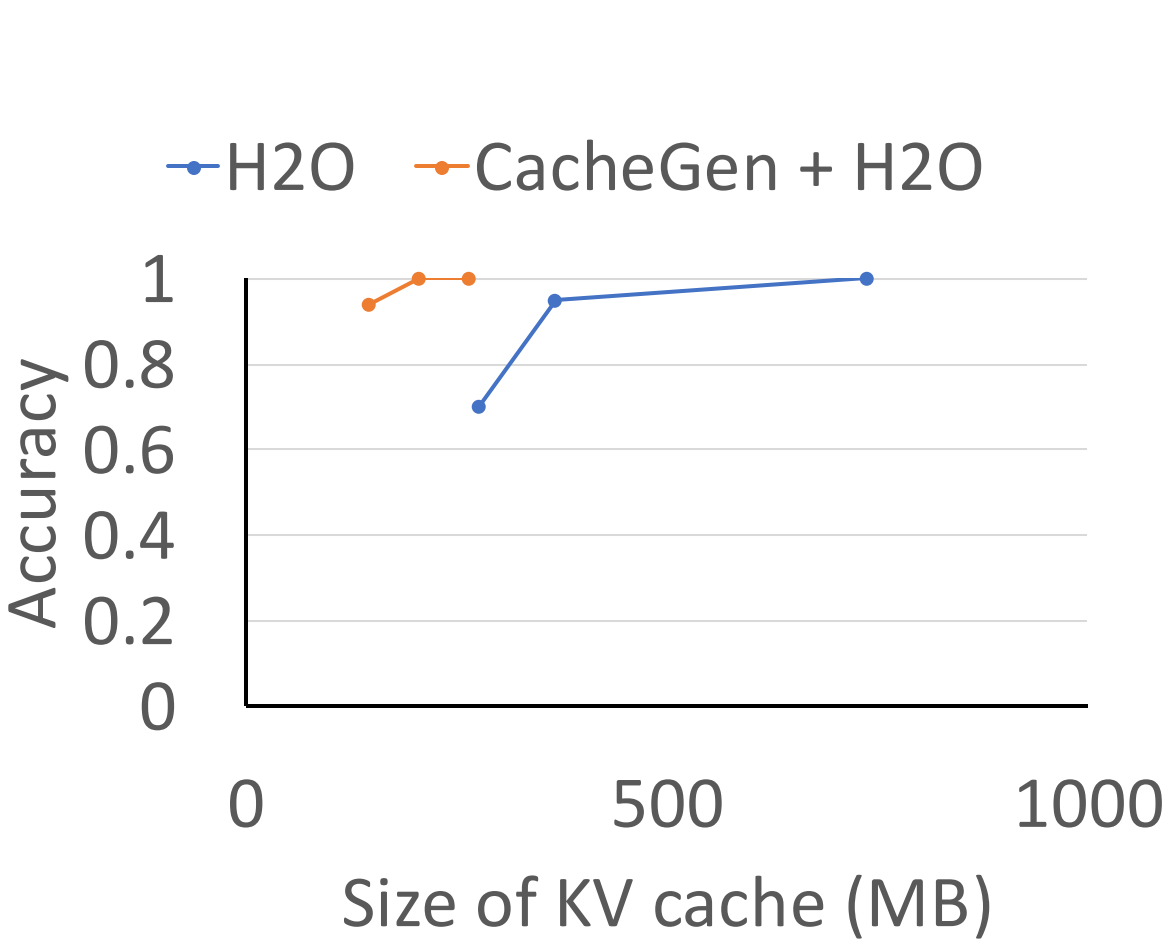
Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

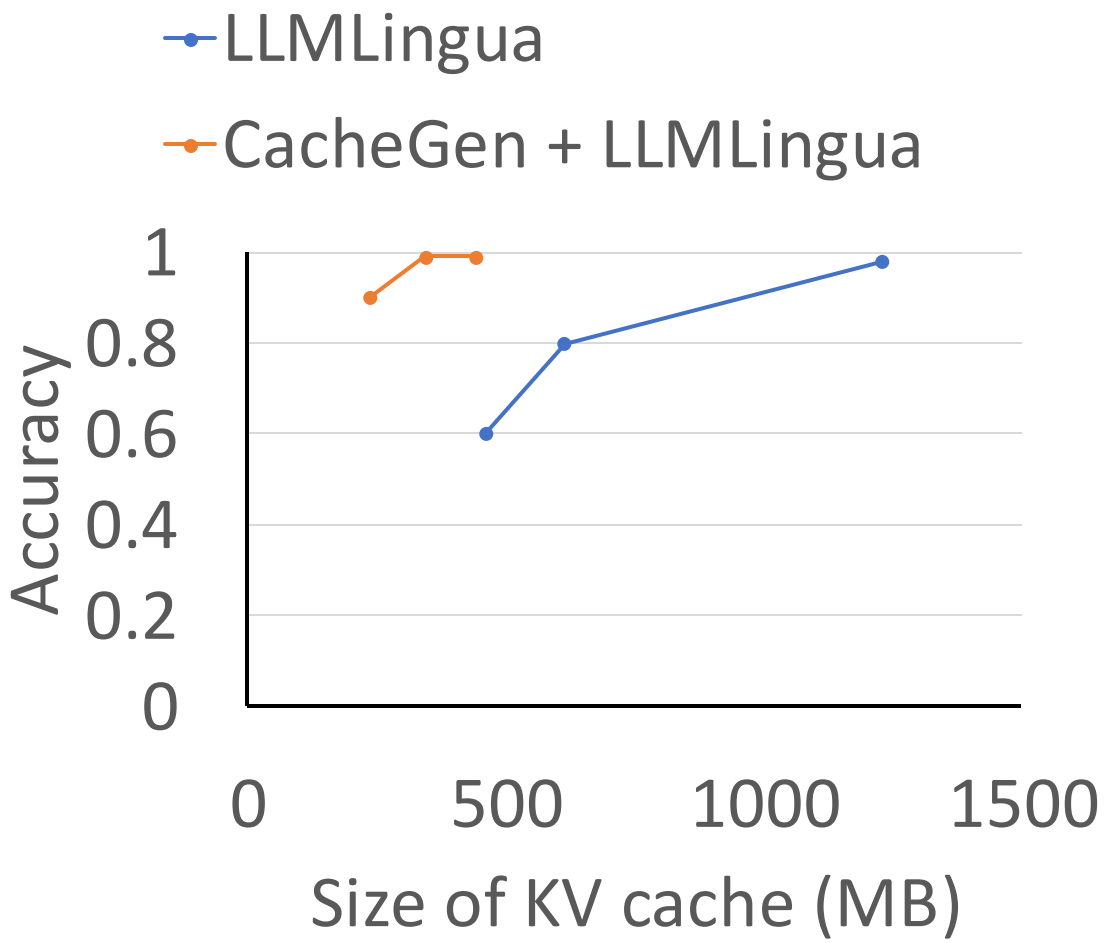
CacheGen Reduces Prefill Delay Under Different Conditions



Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

CacheGen further reduces size of KV caches on top of other methods



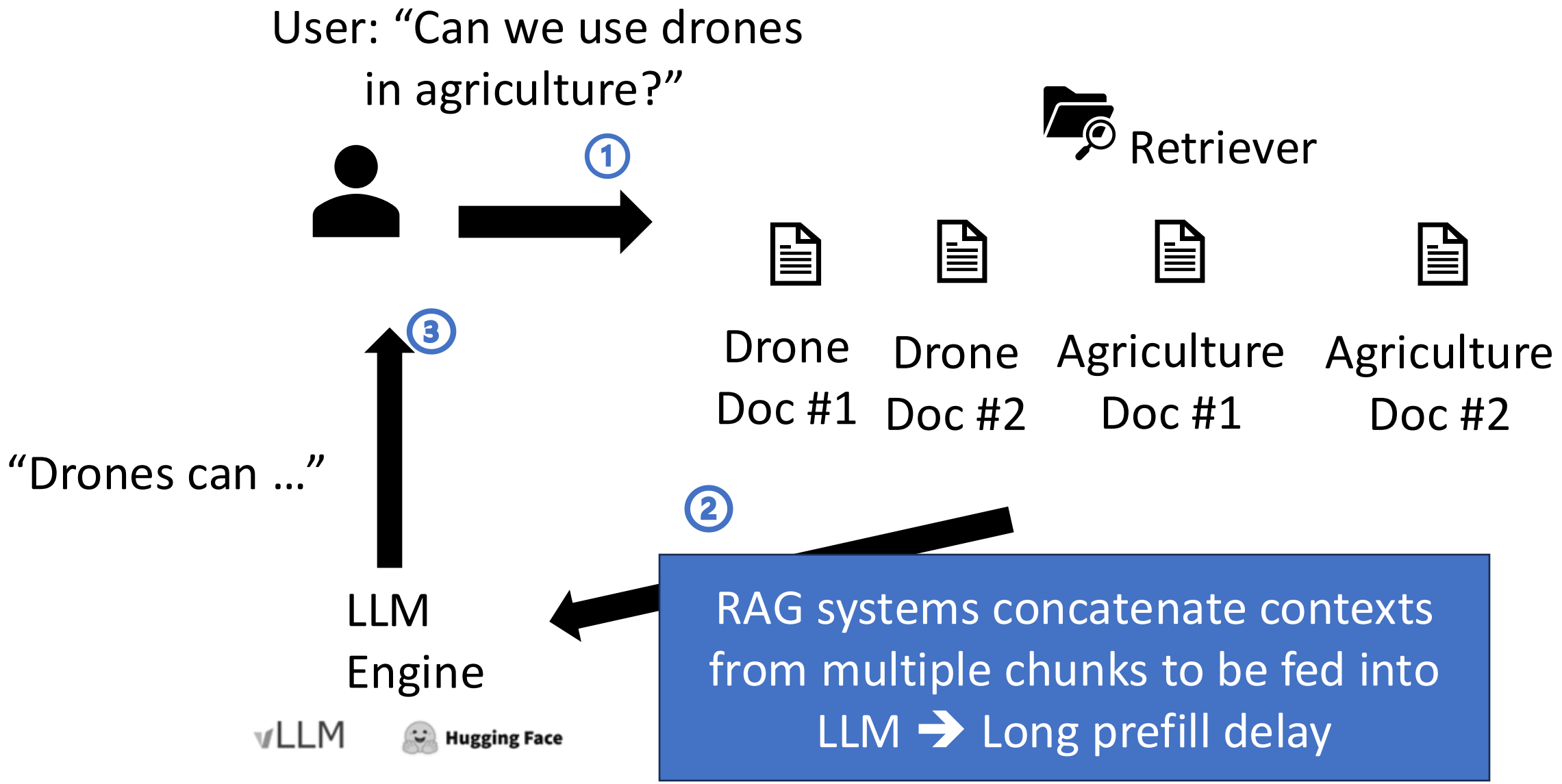


Liu, Yuhan, et al. "Cachegen: Kv cache compression and streaming for fast large language model serving." *SIGCOMM 2024 Conference*.

This Lecture: Prefill Optimization

- ***CacheGen (SIGCOMM'24):*** Compressing KV cache into compact bitstreams for faster transferring
  - Compressing deltas
  - Layer-wise quantization
  - Smart Arithmetic Coding
- ***CacheBlend (EuroSys'25):*** Blending different KV cache for different chunks in RAG for reducing inference latency
  - Selective recompute highly deviated tokens
  - Loading controller to pipeline recompute with loading

Challenge in RAG Systems



Demo: Full Prefill vs CacheBlend

Standard vLLM engine

Retrieved chunks: 5

Context length: 25K tokens

Describe what's ffmpeg in 10 words.

Standard vLLM engine w. CacheBlend

Context length: 25K tokens

Describe what's ffmpeg in 10 words.

Existing solution 1: Prefix caching

Doc 1

"Lionel Messi scored 13 goals at FIFA World Cups."

Doc 2 (Non-prefix)

"Cristiano scored 8 goals at FIFA World Cups."

Query

"Who scored more goals at FIFA World Cups, Messi or Ronaldo?"

Answer

"Lionel Messi scored more goals at FIFA

Prefix caching only achieves 50% hit rate even though 100% of the KV cache are stored in GPU!

[1] Jin, Chao, et al. "Ragcache: Efficient knowledge caching for retrieval-augmented generation." *arXiv preprint arXiv:2404.12457* (2024).
[2] Gao, Bin, et al. "Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving." *ATC 24*.

Existing solution 2: Full KV reuse

Doc 1

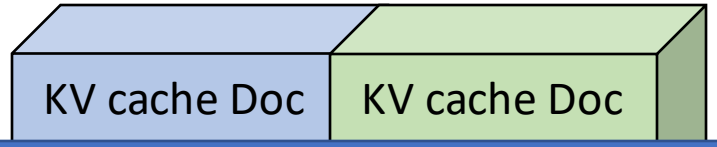
"Lionel Messi scored 13 goals at FIFA World Cups."

Doc 2 (Non-prefix)

"Cristiano scored 8 goals at FIFA World Cups."

Query

"Who scored more goals at FIFA World Cups, Messi or Ronaldo?"



+ [Query]

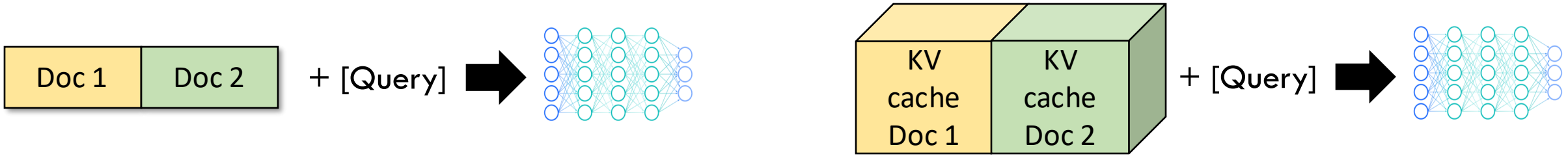
Answer

"The question is asking for information about FIFA World Cups. The names of Messi

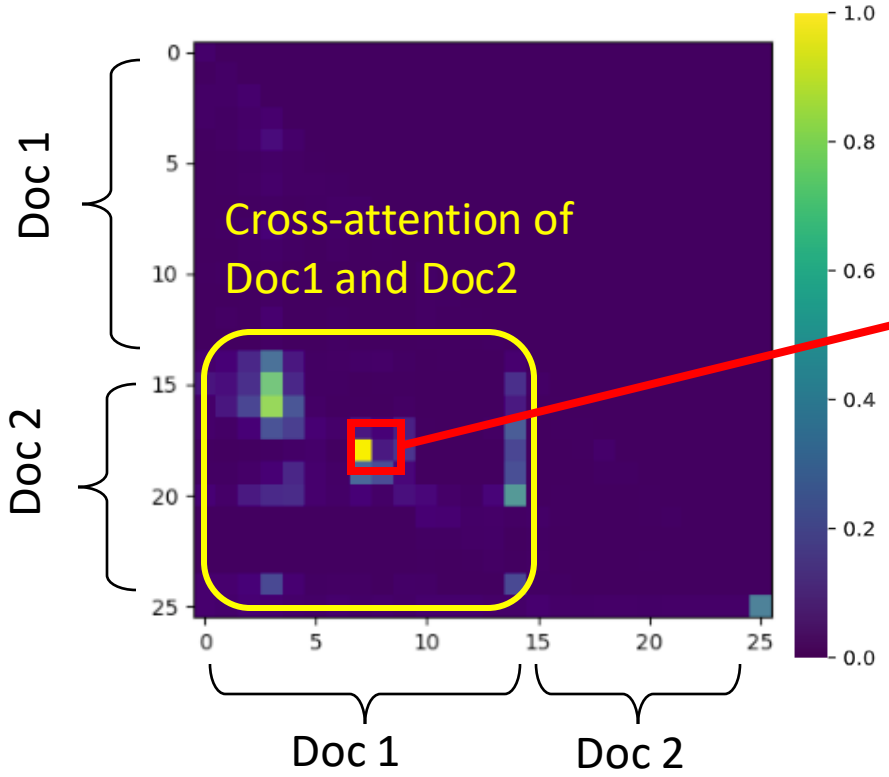
Full KV reuse (naïve concatenation) gives bad generation quality!

Gim, In, et al. "Prompt cache: Modular attention reuse for low-latency inference." *MLSys 2024.*

Why KV reuse leads to bad quality? Missing cross-attention



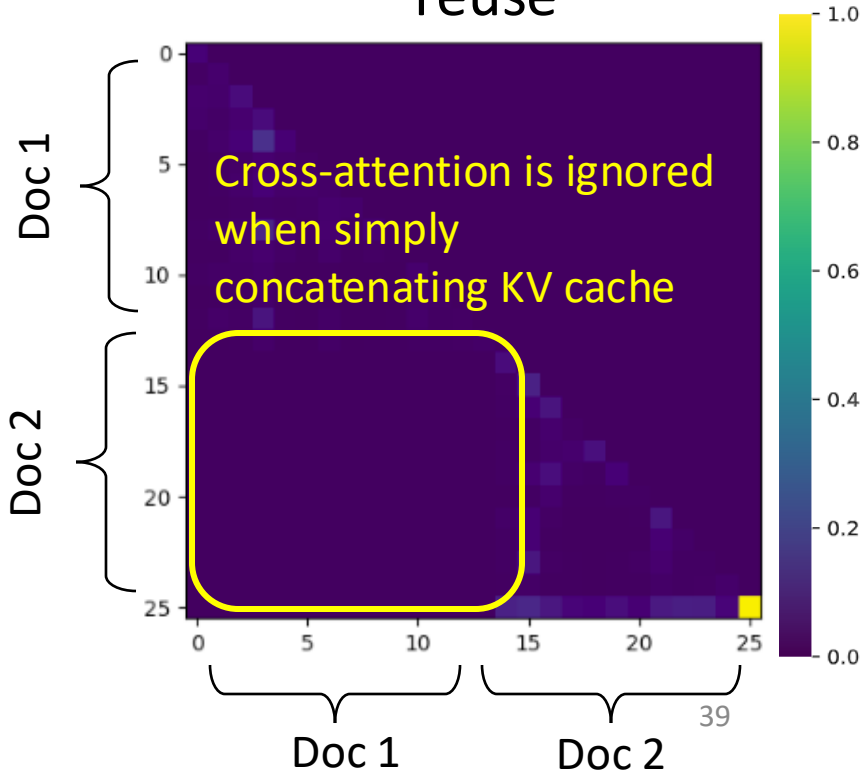
Normal attention matrix



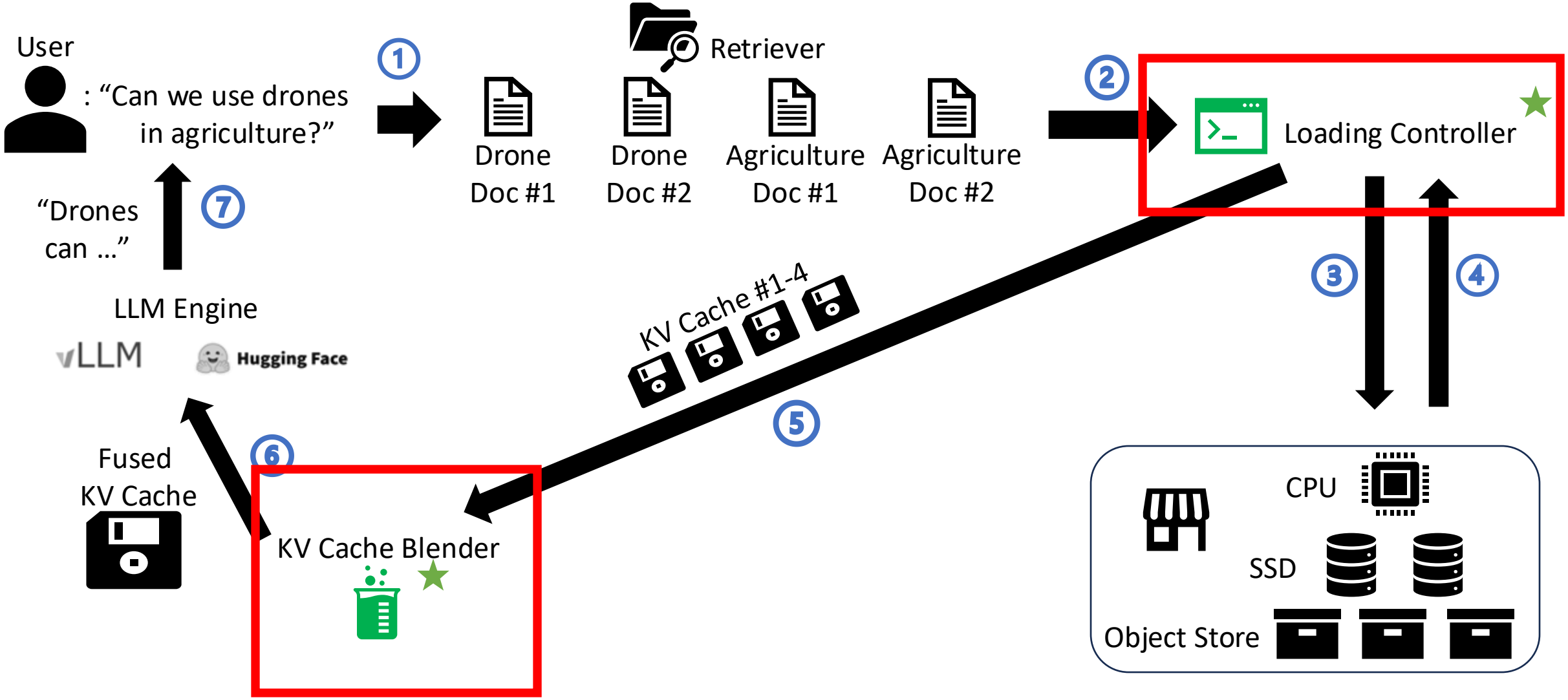
(18, 7):

The **attention** between the **7-th token (in Doc 1)** and the **18-th token (in Doc 2).**

Attention matrix w. full KV reuse



Improved RAG System with CacheBlend



Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

CacheBlend: Selective recomputation

Doc 1

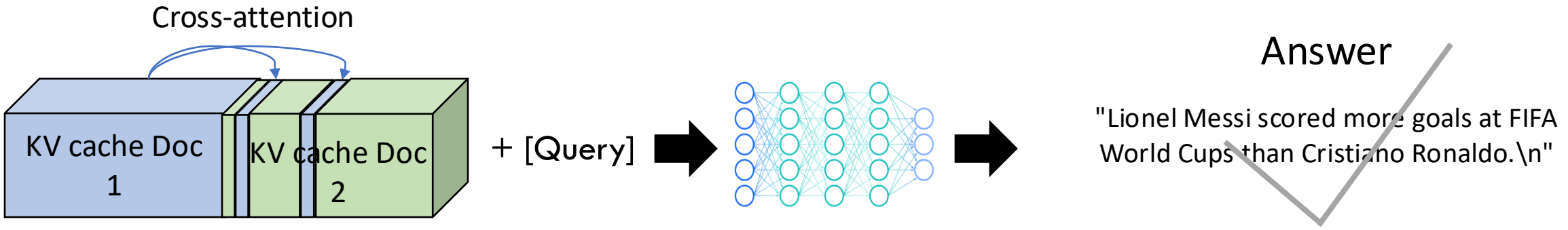
"Lionel Messi scored 13 goals at FIFA World Cups.\n"

Doc 2

"Cristiano scored 8 goals at FIFA World Cups.\n"

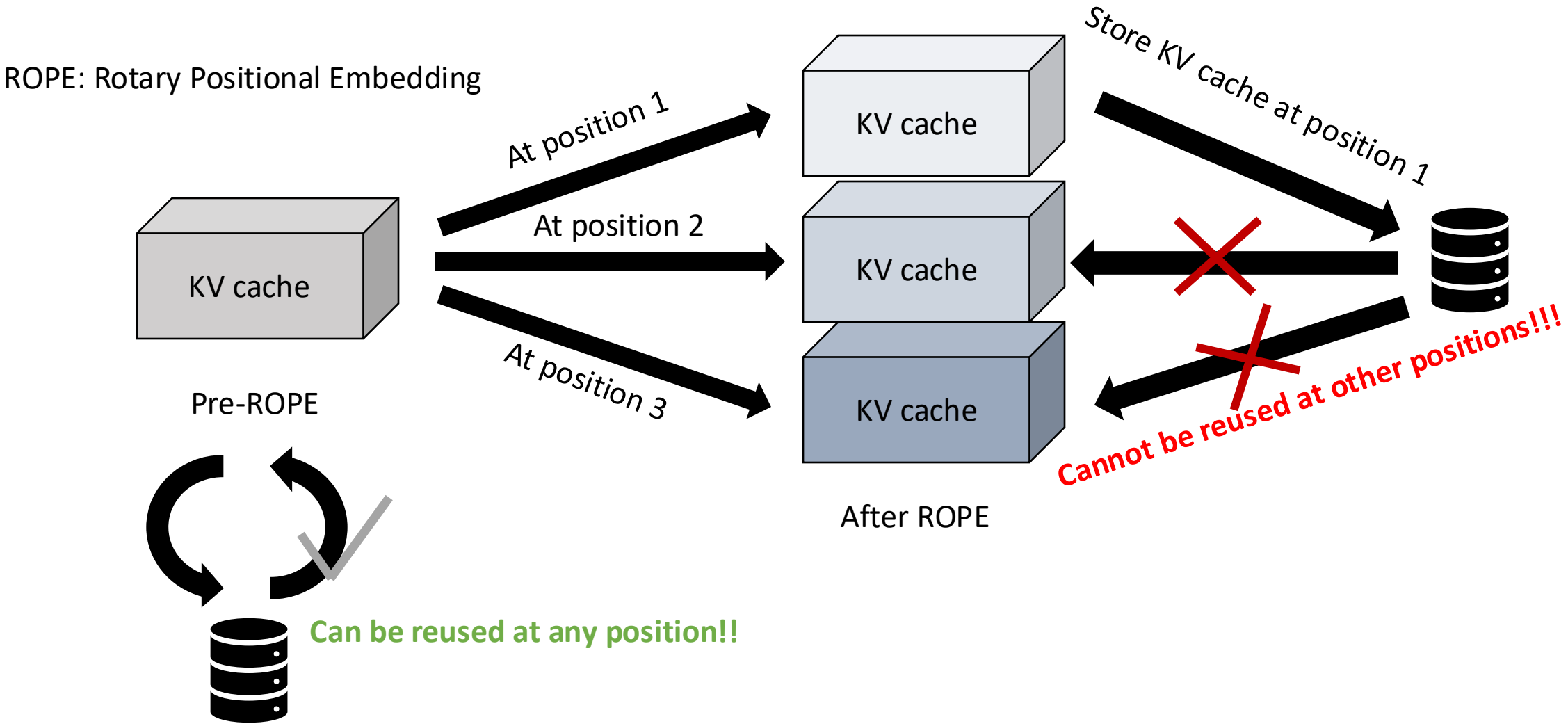
Query

"Who scored more goals at FIFA World Cups, Messi or Ronaldo?\n"



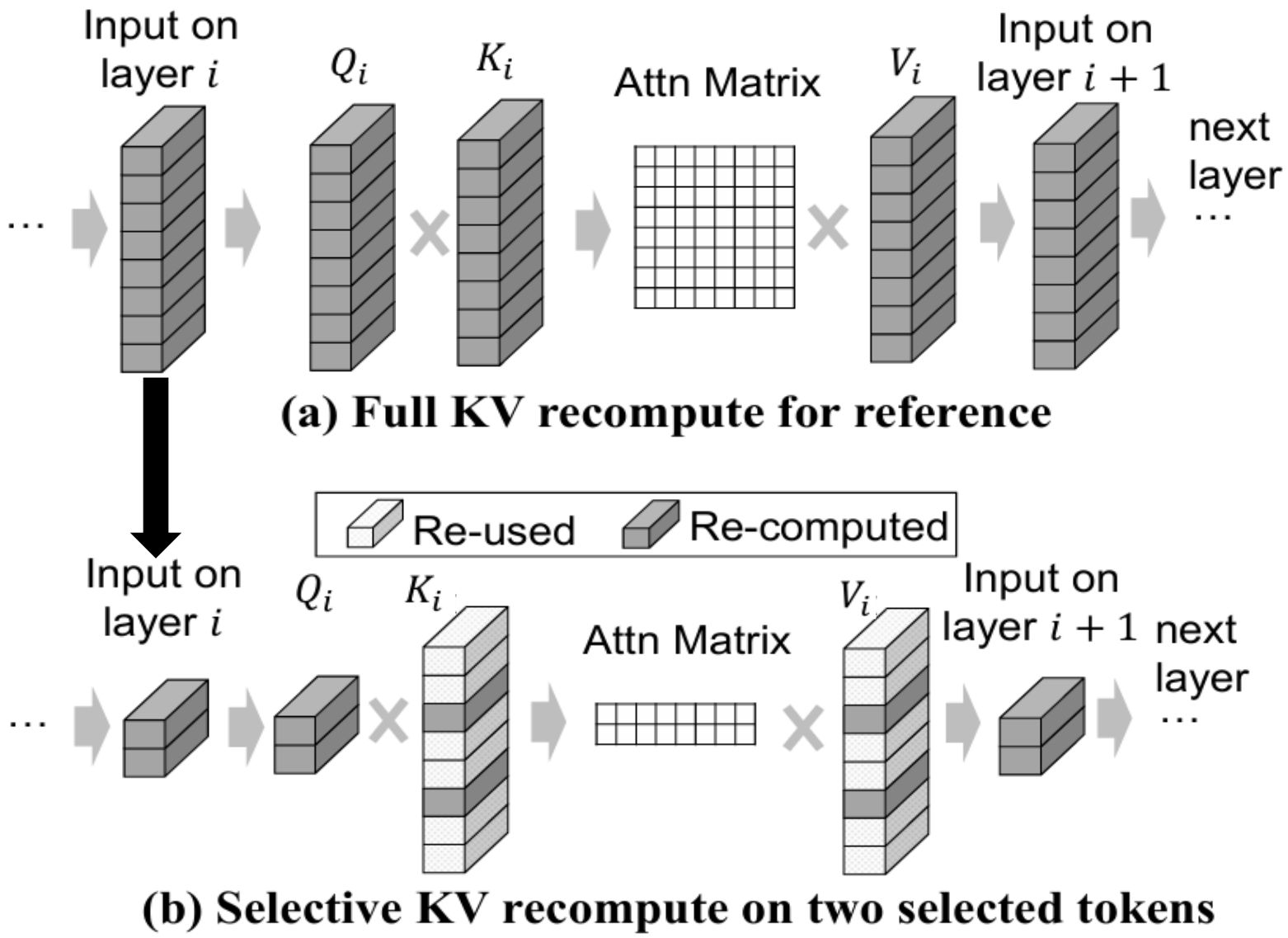
Reuse most KV cache while maintaining generation quality!

CacheBlend: Recover positional embedding



Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

CacheBlend: Selective recomputation



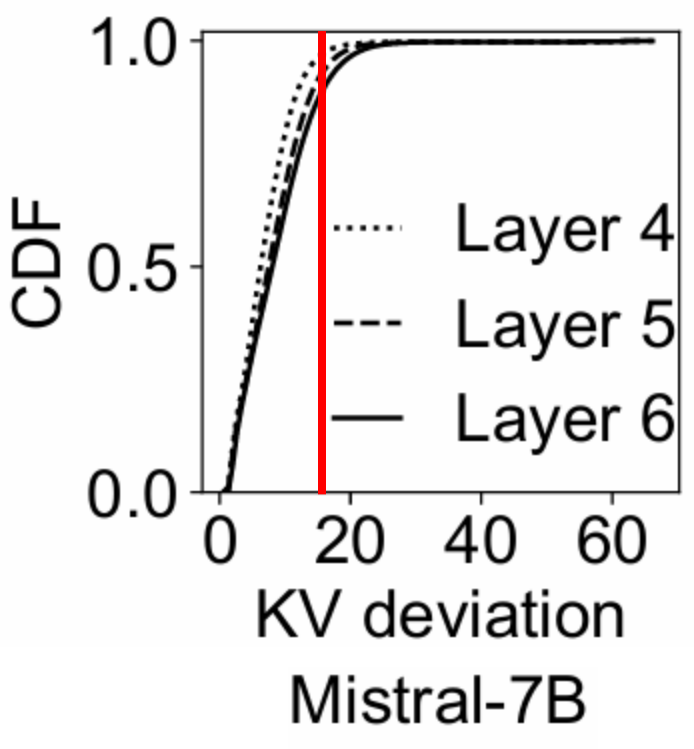
(# total tokens)

(# selected tokens)

How to select tokens?

Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

Insight 1: The fraction of High-KV-Deviation tokens is small



Definition of KV deviation:

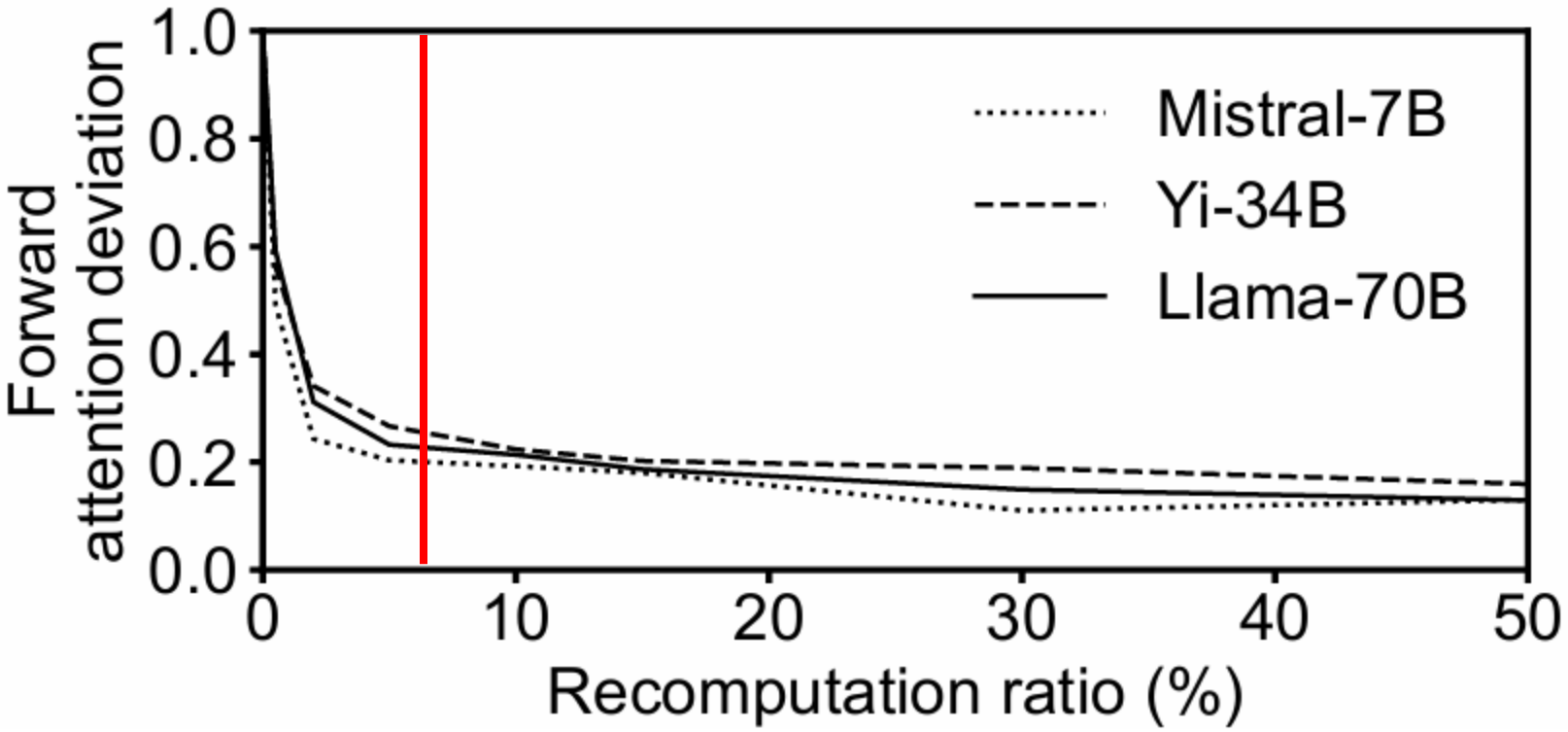
$$\|\mathrm{KV}_{\mathrm{pre}} - \mathrm{KV}_{\mathrm{re}}\|_2$$

$\mathrm{KV}_{\mathrm{pre}}$: **pre**-computed KV cache

$\mathrm{KV}_{\mathrm{re}}$: **re**-computed KV cache

Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

CacheBlend: Recompute a small fraction of High-KV-Deviation tokens greatly reduces forward attention deviation



Definition of forward attention:
Attention between the context documents and the query

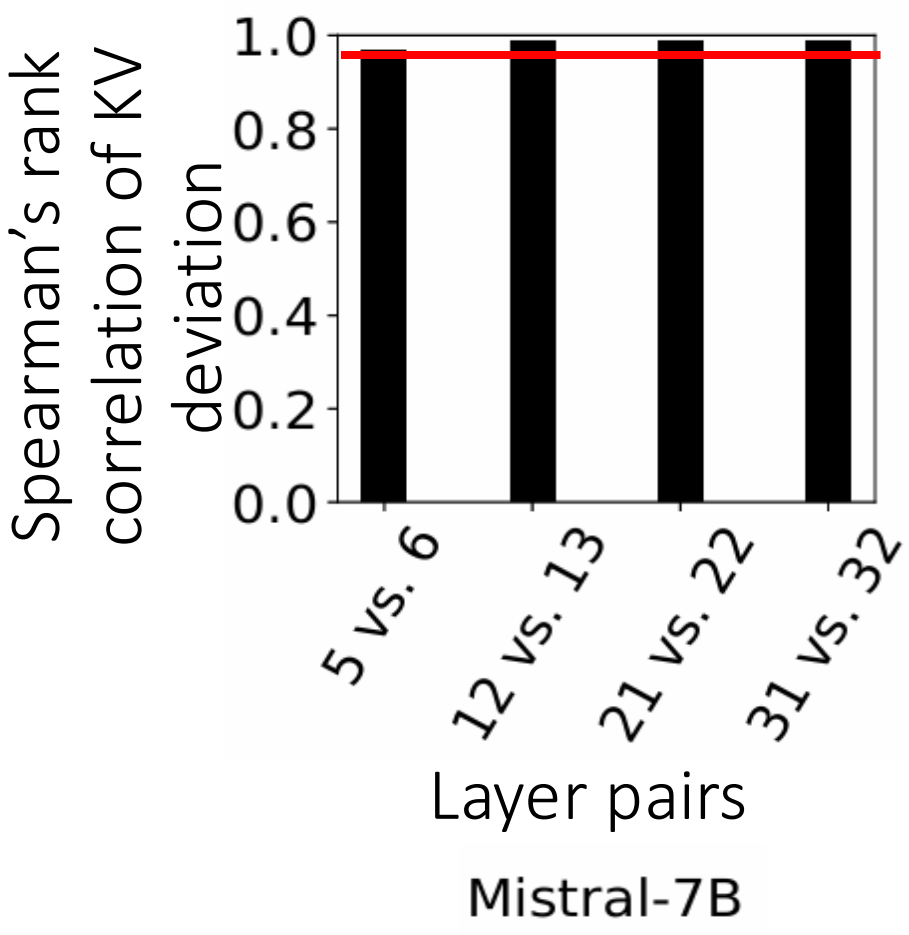
Definition of forward attention deviation:

$$\|\mathrm{Attn}(\mathrm{KV}_{\mathrm{pre}}, Q) - \mathrm{Attn}(\mathrm{KV}_{\mathrm{re}}, Q)\|_2$$

Q : the user query

Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

CacheBlend: High-KV-Deviation tokens are similar across layers



Definition of Spearman's rank correlation:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

Rank of KV deviation at layer l: [1, 2, 3, 4, 5]

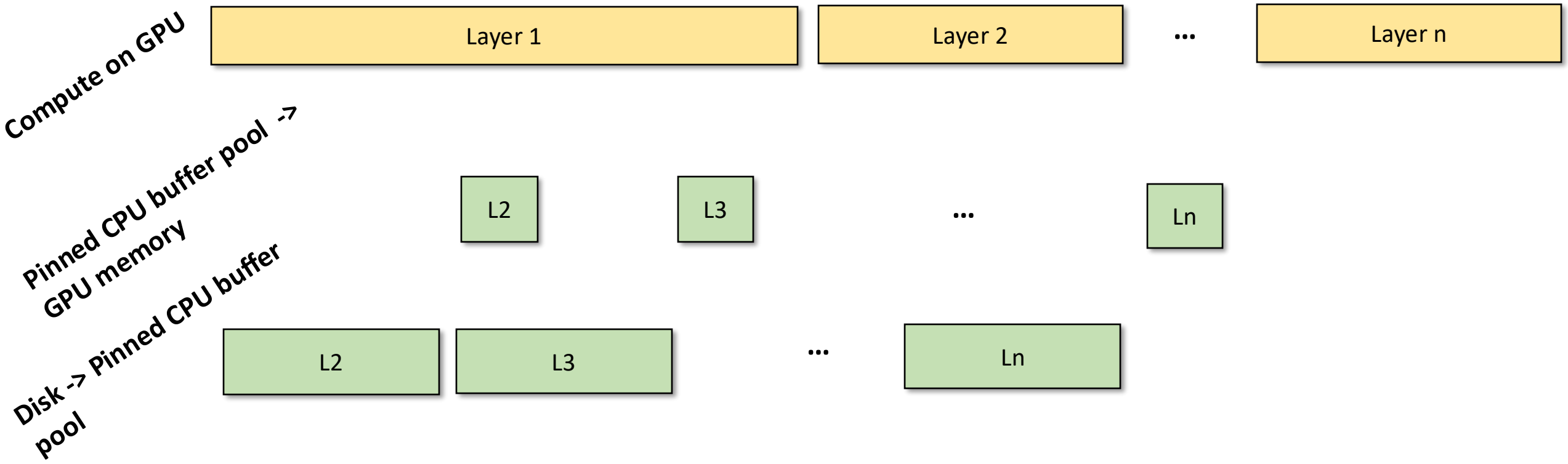
Rank of KV deviation at layer l+1: [2, 1, 3, 5, 4]

n = 5

d = [1, 2, 3, 4, 5] - [2, 1, 3, 5, 4] = [-1, 1, 0, -1, 0]

Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

Loading Controller: Compute & load pipeline



Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

Implementation of CacheBlend

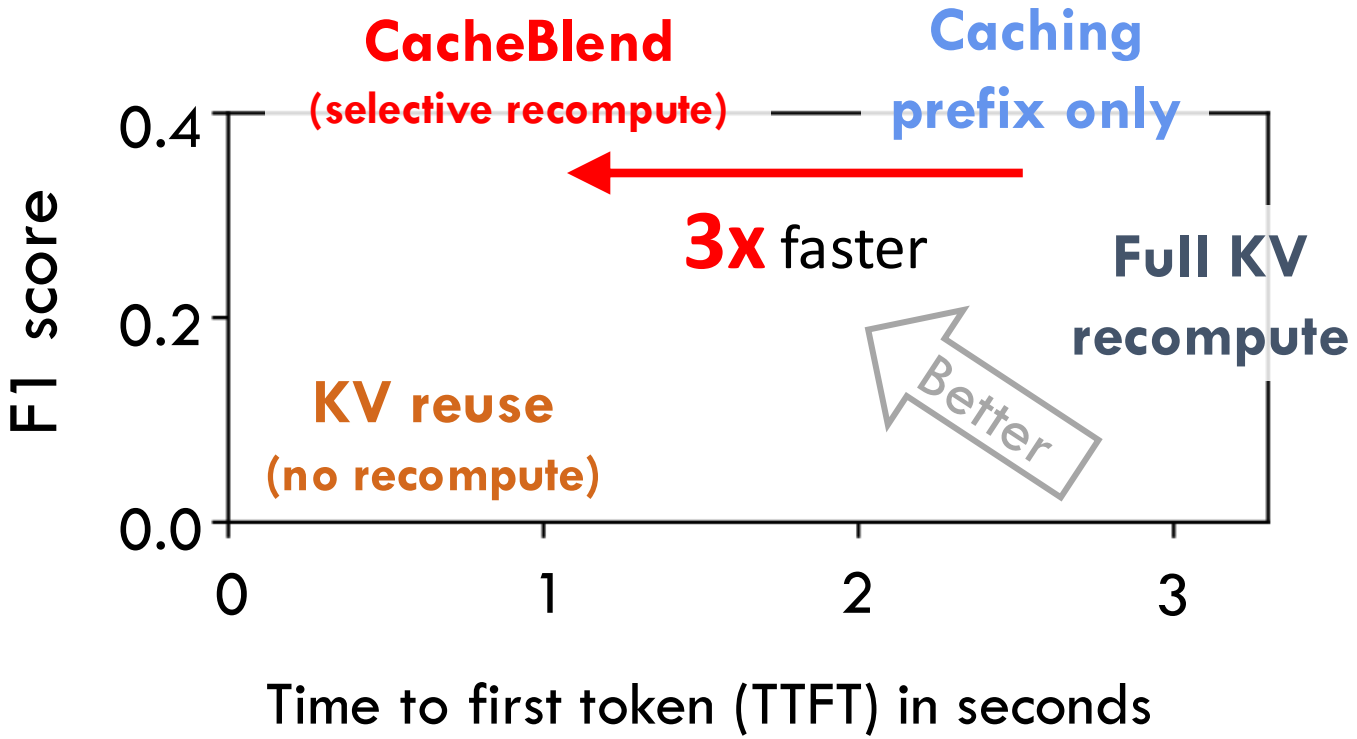
https://github.com/LMCache/LMCache/blob/dev/lmcache/blend/executor.py

```
def blend(self, layer_id, retrieved_k, retrieved_v,
valid_mask, original_positions,
fresh_q, fresh_k, fresh_v, positions, query_start_loc,
token_dim):
    # Layer-specific handling:
    # Layer 0: Return fresh QKV directly
    …
    # Layer 1: Select tokens and prepare for blending
    …
    # Other layers: Perform actual blending with
    positional encoding
    …
```

Evaluation Setup

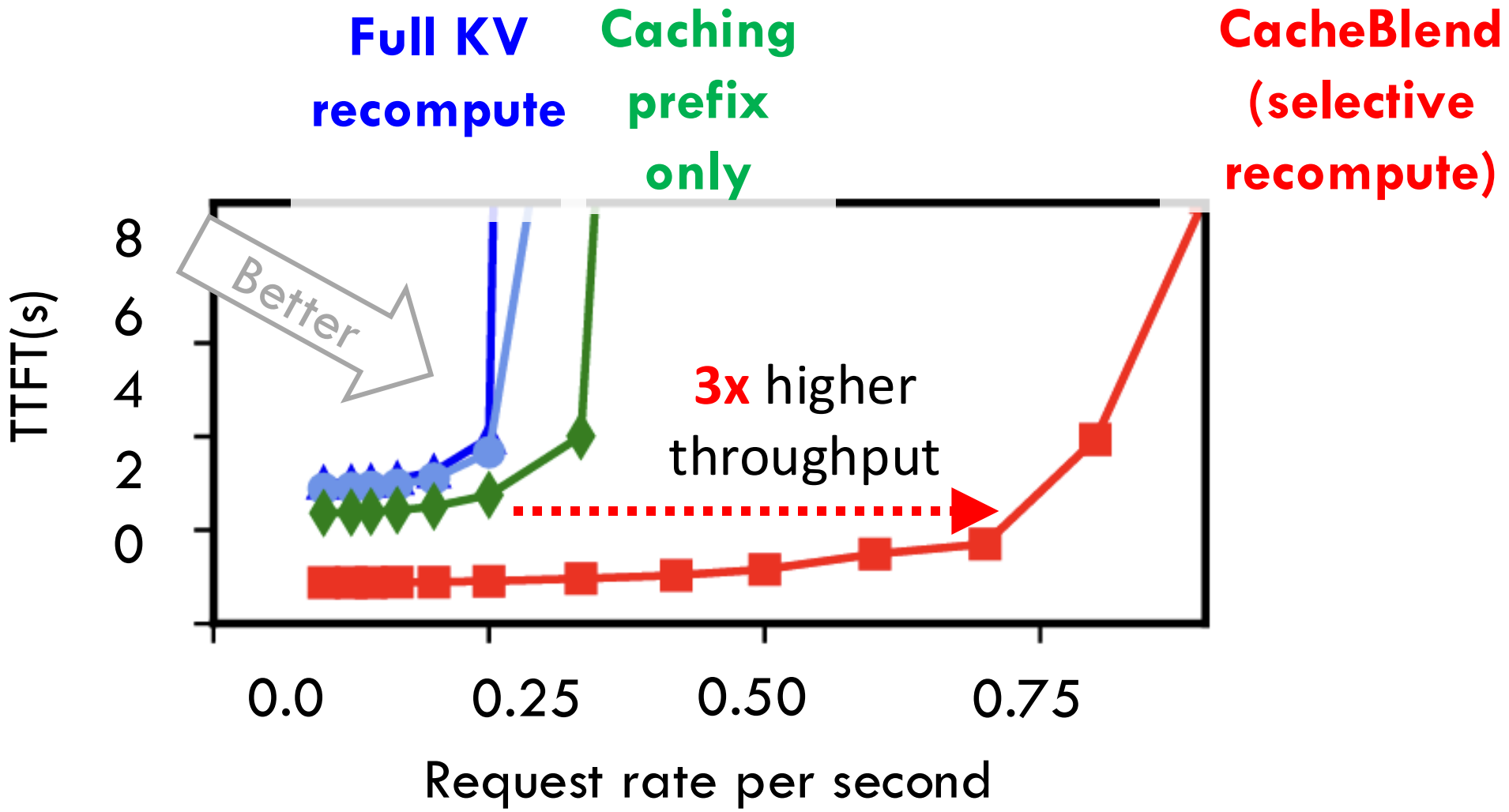
- Dataset: "2WikiMQA"
  - Top 6 chunks (512 tokens each)
- 1.5K sampled queries
- Model: Llama 70B
- GPU: 2x A40
- KV cache initially stored on disk
- Workload Generator: Sending requests to request queue at varying rate (Query-per-Second)
- Baselines:
  - Full Recompute
  - Full KV reuse
  - Caching Prefix

CacheBlend achieves ~3X latency reduction



Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

CacheBlend achieves ~3X higher throughput



Yao, Jiayi, et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion." *Proceedings of the Twentieth European Conference on Computer Systems*. 2025.

This Lecture: Prefill Optimization

- ***CacheGen (SIGCOMM'24):*** Compressing KV cache into compact bitstreams for faster transferring
  - Compressing deltas
  - Layer-wise quantization
  - Smart Arithmetic Coding
- ***CacheBlend (EuroSys'25):*** Blending different KV cache for different chunks in RAG for reducing inference latency
  - Selective recompute highly deviated tokens
  - Loading controller to pipeline recompute with loading

Walkthrough of a Simple Example in LMCache

```
chunk_size: 256
local_cpu: True
max_local_cpu_size: 10

remote_url: "lm://localhost:65432"
remote_serde: "cachegen"
```

Set LMCache config file

Start LMCache storage backend

```
python3 -m lmcache.experimental.server
localhost {port_number} cpu
```

Run LMCache with CacheGen

```
LMCACHE_USE_EXPERIMENTAL=True
LMCACHE_CONFIG_FILE=example.yaml
python3 offline_inference.py
```

Inside offline_inference.py

```python
import copy
import json
import os
import time

from transformers import AutoTokenizer
from vllm import LLM, SamplingParams
from vllm.config import KVTransferConfig

from lmcache.experimental.cache_engine import LMCacheEngineBuilder
from lmcache.integration.vllm.utils import ENGINE_NAME

model_name = "meta-llama/Meta-Llama-3.1-8B-Instruct"
```

→ Importing LMCache library

```
kv_transfer_config = '{\
    "kv_connector":"LMCacheConnector",\
    "kv_role":"kv_both"}'
```

Setting KV transfer config

```
kv_transfer_config = KVTransferConfig.from_cli(kv_transfer_config)

context_length = get_context_length(tokenizer, context_messages)
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.0, top_p=0.95, max_tokens=128)
prompts = gen_prompts(tokenizer, context_messages, user_inputs_batch)
# Create an LLM.
llm = LLM(model=model_name,
          gpu_memory_utilization=0.8,
          enable_chunked_prefill=True,
          tensor_parallel_size=2,
          max_model_len=32768,
          enforce_eager=False,
          kv_transfer_config=kv_transfer_config)
outputs = llm.generate(prompts, sampling_params)
```

Initializing vLLM engine with transfer config

Key Functions in LMCache

In LMCache/lmcache/experimental/cache_engine.py

```
def store(self,
          tokens: torch.Tensor,
          mask: Optional[torch.Tensor] = None,
          **kwargs) -> None:
    """Store the tokens and mask into the cache engine.

    :param torch.Tensor tokens: The tokens of the corresponding KV caches.

    :param Optional[torch.Tensor] mask: The mask for the tokens. Should
        have the same length as tokens. And the mask should ALWAYS be like
        FFFFFTTTTTTT, where True means the tokens needs to be matched,
        and the Falses will ALWAYS be at the PREFIX of the tensor.

    :param **kwargs: The additional arguments for the storage backend which
        will be passed into the gpu_connector.
        Should include KV cache specific information (e.g., paged KV buffer
        and the page tables).

    :raises: ValueError if the number of Falses in the mask is not a
        multiple of the chunk size.
    """
```

```
def retrieve(self,
             tokens: torch.Tensor,
             mask: Optional[torch.Tensor] = None,
             **kwargs) -> torch.Tensor:
    """Retrieve the KV caches from the cache engine. And put the retrieved
    KV cache to the serving engine via the GPU connector.

    :param torch.Tensor tokens: The tokens of the corresponding KV caches.

    :param Optional[torch.Tensor] mask: The mask for the tokens. Should
        have the same length as tokens. And the mask should ALWAYS be like
        FFFFFTTTTTTT, where True means the tokens needs to be matched,
        and the Falses will ALWAYS be at the PREFIX of the tensor.

    :param **kwargs: The additional arguments for the storage backend which
        will be passed into the gpu_connector.
        Should include KV cache specific information (e.g., paged KV buffer
        and the page tables).

    :return: the boolean mask indicating which tokens are retrieved. The
        length of the mask should be the same as the tokens. On CPU.

    :raises: ValueError if the number of Falses in the mask is not a
        multiple of the chunk size.
    """
```

Compression Interface in LMCache

https://github.com/LMCache/LMCache/blob/dev/lmcache/experimental/storage_backend/naive_serde/cachegen_encoder.py

```
def serialize(self, memory_obj: MemoryObj) -> BytesBufferMemoryObj:
    """
    Serialize a KV_BLOB MemoryObj to CACHEGEN_BINARY MemoryObj.

    Input:
        memory_obj: the memory object to be serialized.

    Returns:
        MemoryObj: the serialized binary memory object.
    """
```

https://github.com/LMCache/LMCache/blob/dev/lmcache/experimental/storage_backend/naive_serde/cachegen_decoder.py

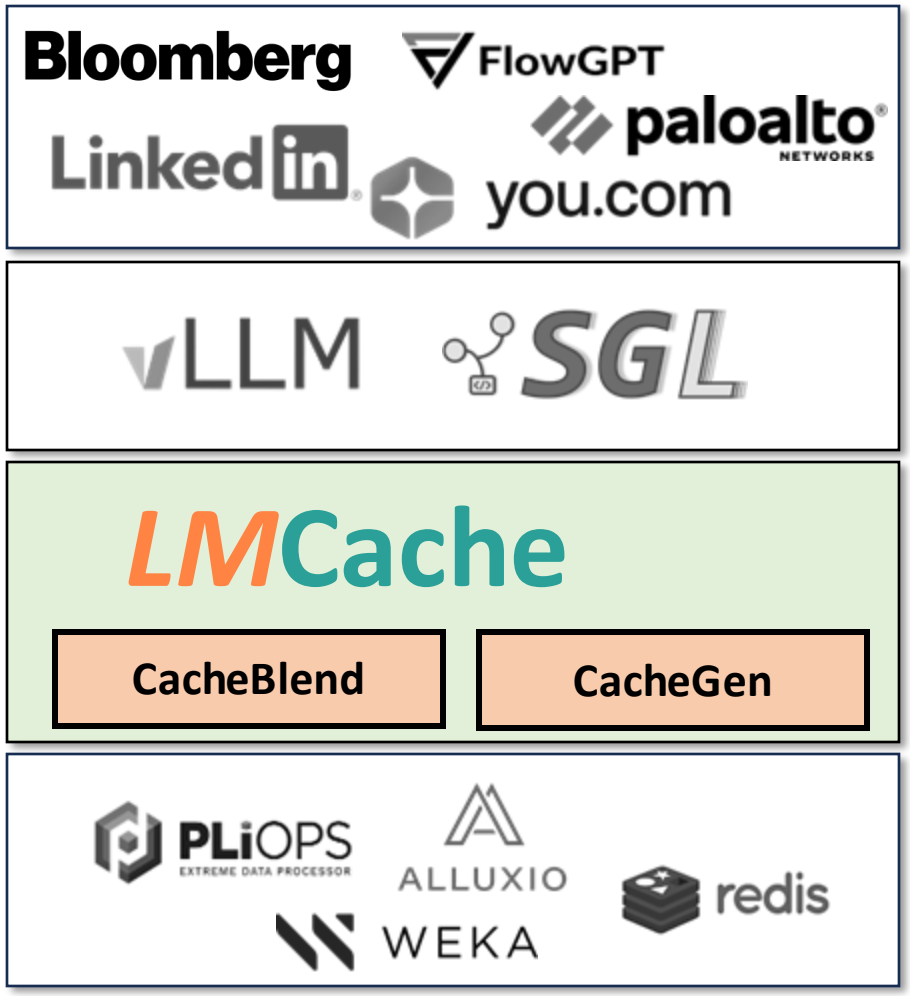
```
def deserialize(
        self,
        buffer_memory_obj: BytesBufferMemoryObj) -> Optional[MemoryObj]:
    encoder_output = CacheGenGPUEncoderOutput.from_bytes(
        buffer_memory_obj.byte_array)

    encoder_output.max_tensors_key = encoder_output.max_tensors_key.cuda()
    encoder_output.max_tensors_value = (
        encoder_output.max_tensors_value.cuda())

    ntokens = encoder_output.max_tensors_key.shape[1]
    layers_in_key = encoder_output.max_tensors_key.shape[0]
    key, value = decode_function_gpu(
        encoder_output.cdf,
        encoder_output.data_chunks,
        layers_in_key,
        ntokens,
        self.get_output_buffer(
            encoder_output.cdf.shape[0] // 2,
            encoder_output.cdf.shape[1],
            ntokens,
        ),
    )
```

Key Takeaways

- Two problems in KV Cache management:
  - How to quickly transfer KV cache to GPU memory? → **Efficiently compress and stream a KV cache [SIGCOMM'24]**
  - How to quickly combine multiple KV caches? → **Quickly combine KV caches of multiple contexts [EuroSys'25 Best Paper]**

Check out *LM*Cache

https://github.com/LMCache/LMCache



Join our Slack channel!

Contact me (yuhanl@uchicago.edu) if you are interested!