CS769 Advanced NLP

KV Cache in LLMs

Junjie Hu

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Slides adapted from Sebastian Raschka

https://junjiehu.github.io/cs769-fall25/

Goal for Today

1. Intro to KV Cache for Efficient Inference of LLMs
2. KV Cache Compression
    - H2O
    - StreamingLLM
    - SnapKV
    - PyramidKV
3. Future Research

Long Context is Needed!

- The context window “arms race”: in ~2 years, max tokens rose from ~4K to ~1M
- Key scenarios needing long context:
  - Long documents understanding;
  - Agents with long interactive history

What Is a KV Cache?

- A KV cache stores intermediate **key (K)** and **value (V) computations** for reuse during **inference** (after training)

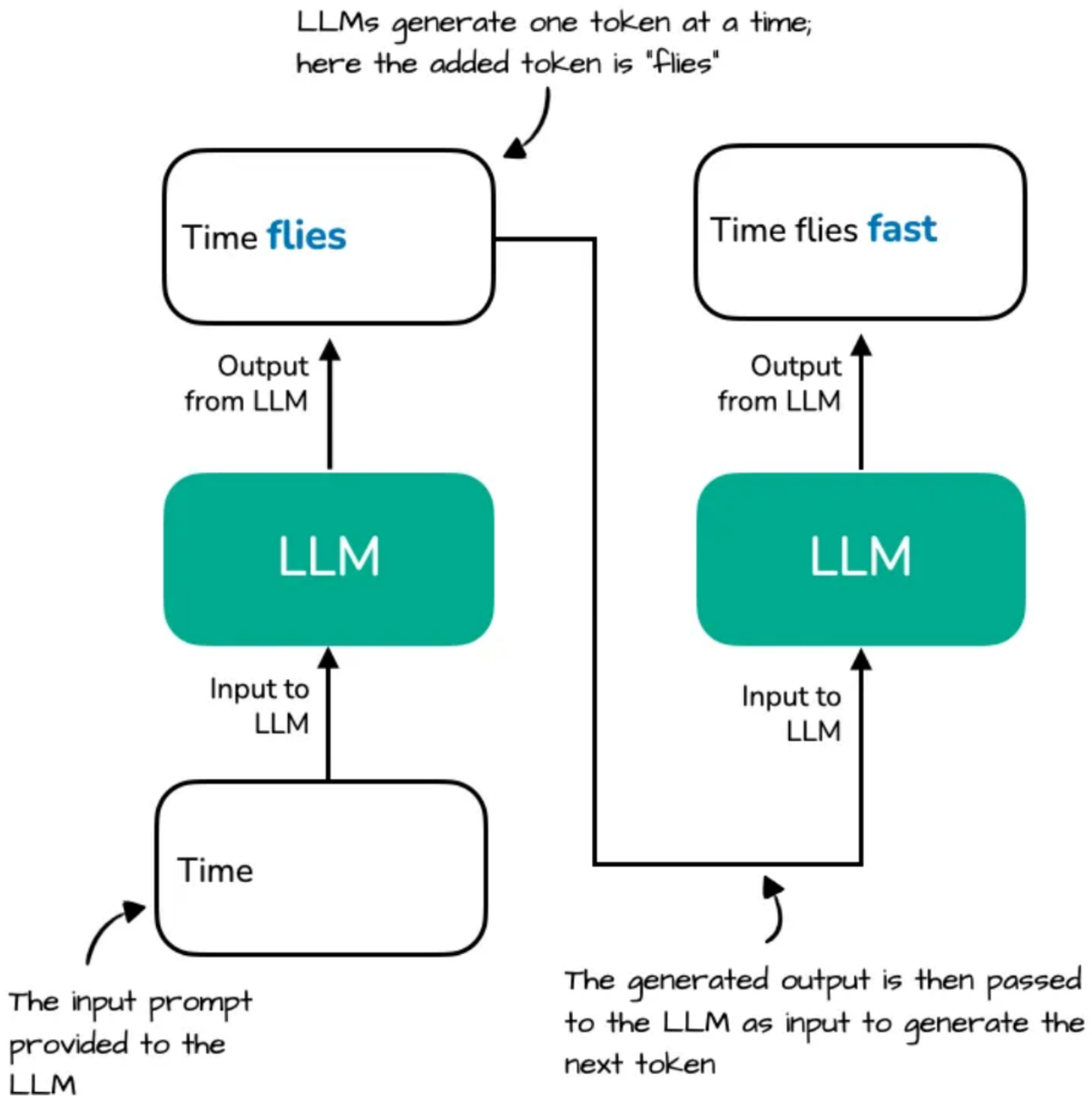
  ➡ Add complexity to code implementation

  ➡ Increase memory requirement

  ➡ Speed up decoding at inference

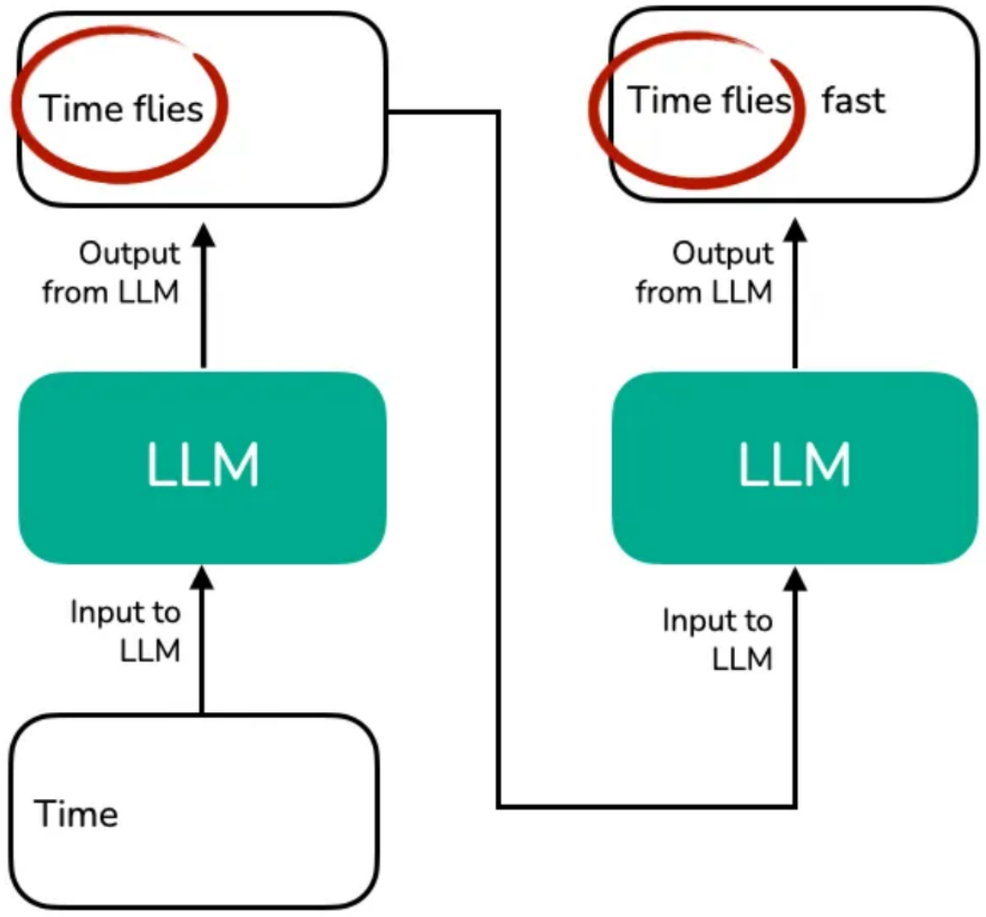
KV Cache Illustrations

- In autoregressive decoding, LLMs generate one word at a time.

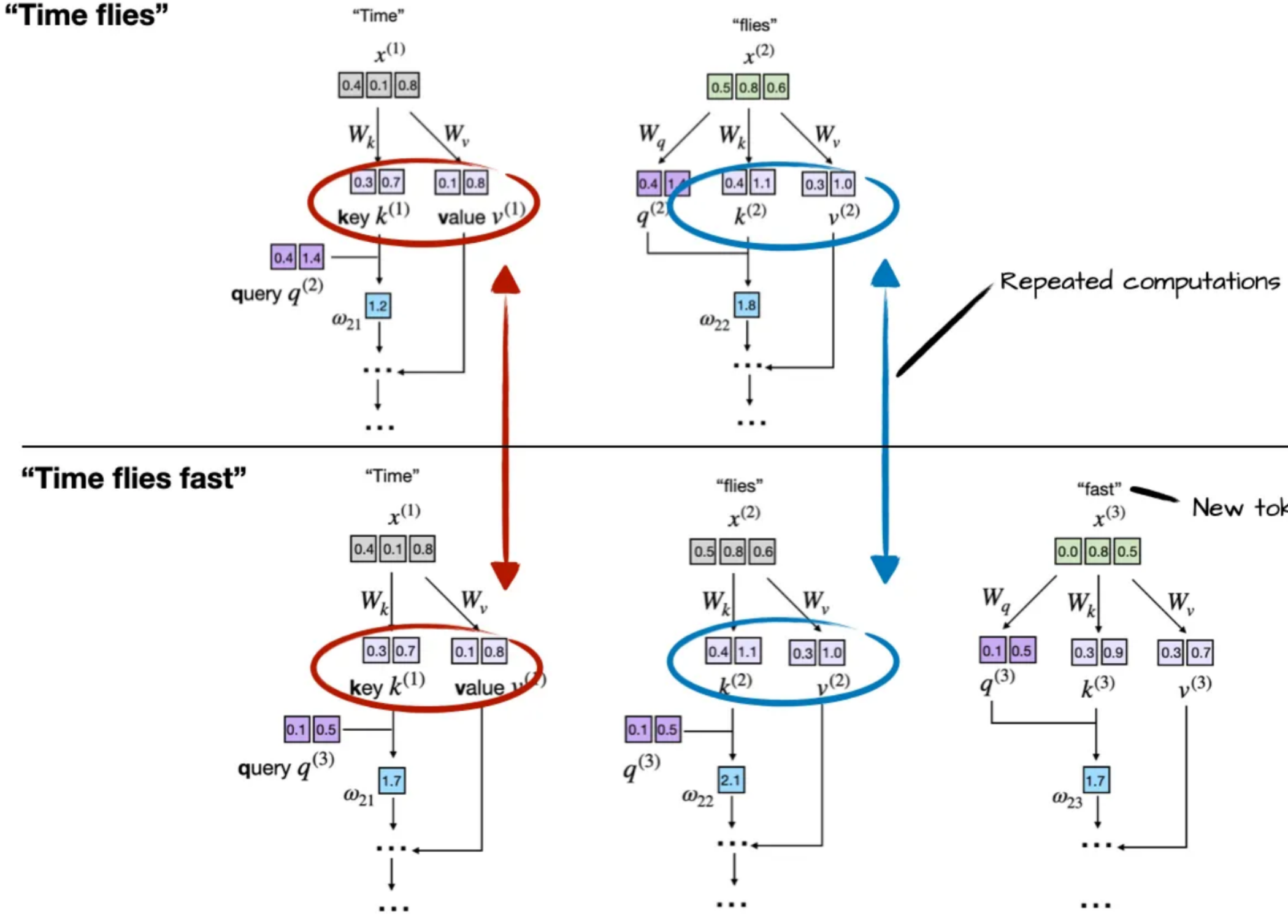


KV Cache Illustrations

- In autoregressive decoding, LLMs generate one word at a time. There exists **redundancy in the generated LLM text output across decoding steps**.



Repeated Computations of KV vectors



Repeated Computations of KV vectors

- Decoding w/o KV cache: “Time” and “files” are recomputed at every new generation step

Generation Step	Input Tokens	Computed K/V
1	`"Time"`	`"Time"`
2	`"Time flies"`	`"Time"`, `"flies"`
3	`"Time flies fast"`	`"Time"`, `"flies"`, `"fast"`

- Decoding w/ KV cache: “Time” is computed once and reused twice, and “files” is computed once and reused once.

Generation Step	Input Tokens	Computed K/V	Cached K/V
1	`"Time"`	`"Time"`	-
2	`"Time flies"`	`"flies"`	`"Time"`
3	`"Time flies fast"`	`"fast"`	`"Time", "flies"`

Transformer Attention and KV Cache Mechanics

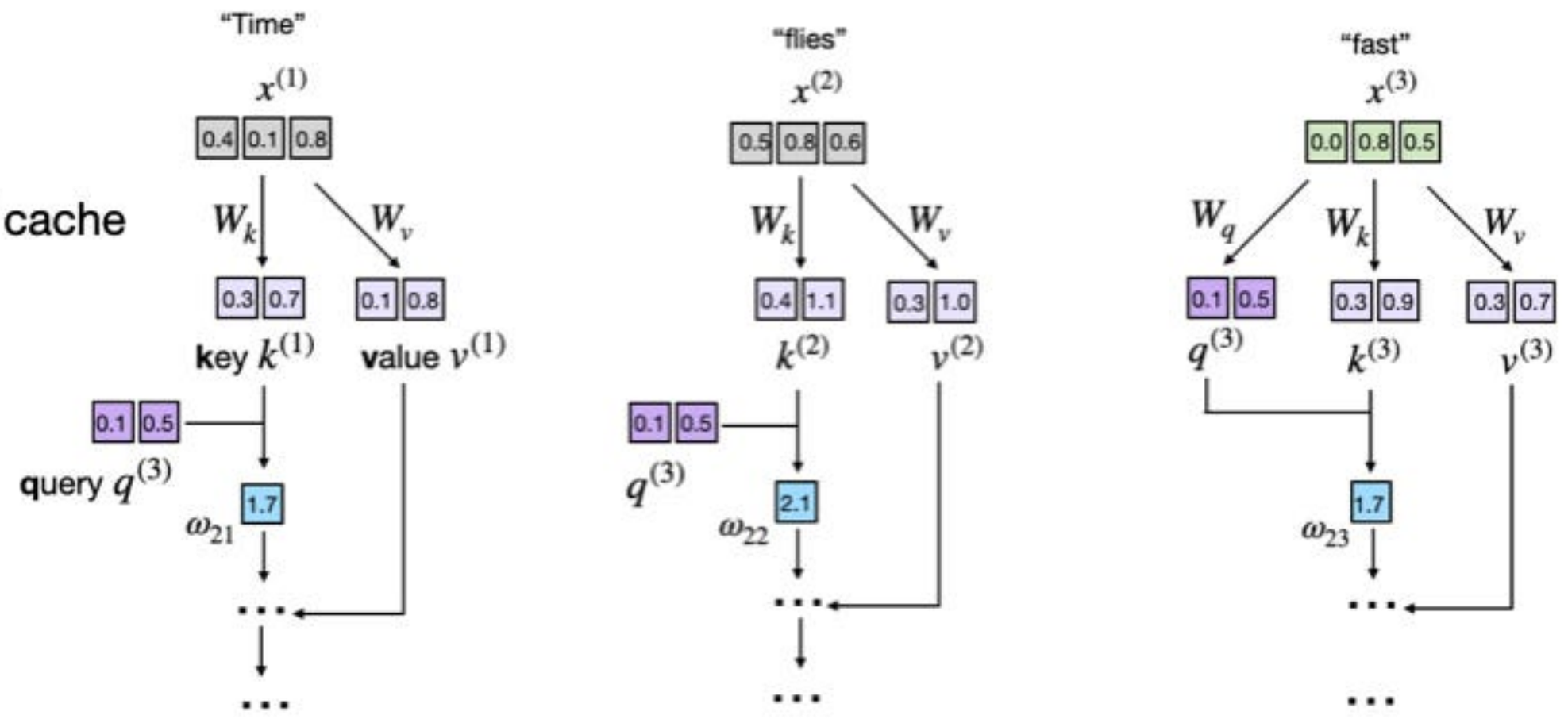
- **Self-Attention Mechanism:** For a query vector Q_t at time step t, the attention is computed over all previous Keys $K_{1:t}$ and corresponding Values

- **KV Cache in Autoregressive Decoding:** At step t, the new token's KV vectors (K_t, V_t) are calculated and appended to the cache $\mathbf{K}_{1:t} = [\mathbf{K}_{1:t-1}; K_t]$ and $\mathbf{V}_{1:t} = [\mathbf{V}_{1:t-1}; V_t]$.

$$\text{Attention}(Q_t, \mathbf{K}_{1:t}, \mathbf{V}_{1:t}) = \text{softmax}\left(\frac{Q_t \mathbf{K}_{1:t}^T}{\sqrt{d_k}}\right)\mathbf{V}_{1:t}$$

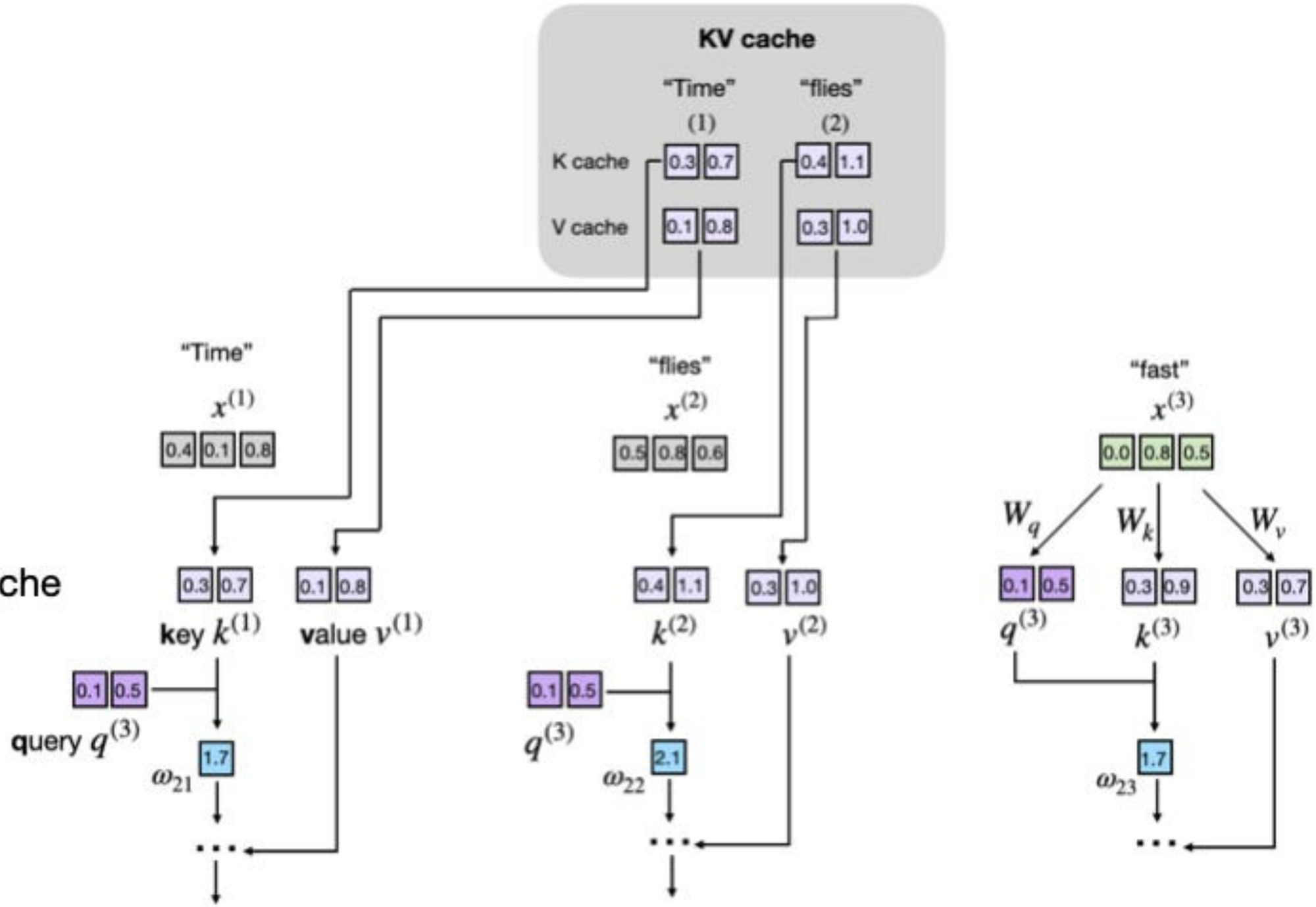
KV Cache Illustrations

Without KV cache



KV Cache Illustrations

With KV cache



The LLM Inference Bottleneck: KV Cache

- **The Challenge of LLM Inference:** Autoregressive decoding generates one token at a time, where each new token's computation depends on all previous tokens.

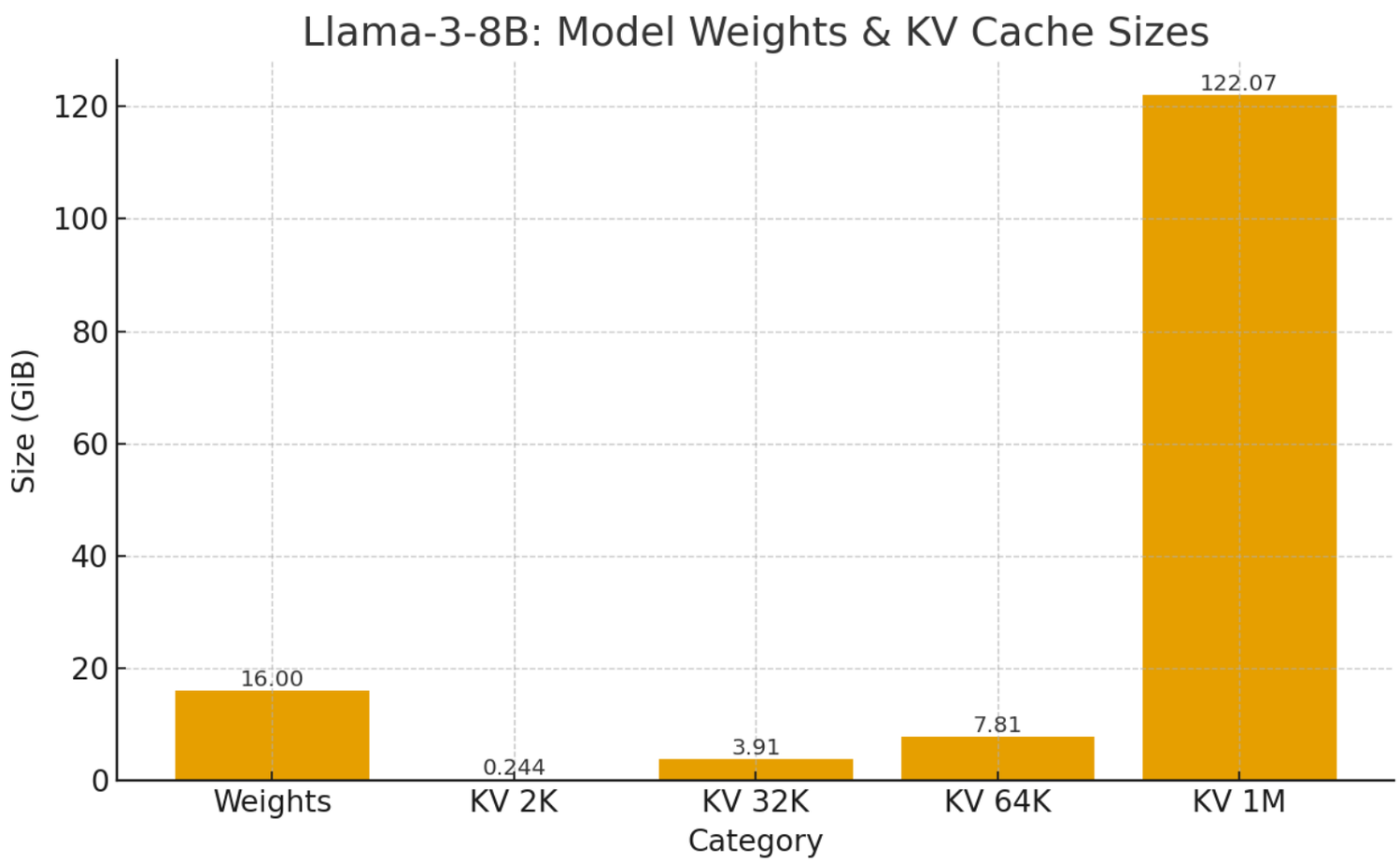
- **The KV Cache Solution:** To avoid re-computing the self-attention mechanism over the entire history at every step, the pre-computed **Key and Value vectors** from all preceding tokens are stored in memory—this is the **KV Cache**.

- **The New Bottleneck (Memory):** While the KV Cache makes computation $\mathcal{O}(L)$ (linear in sequence length) instead of $\mathcal{O}(L^2)$, the memory consumption for storing the KV Cache grows **linearly with the sequence length and the model size**.

    - Reuses cached **per-layer key/value tensors** for past tokens — memory grows $\mathcal{O}(L \times d_{model} \times n_{heads} \times n_{layers})$.

KV Cache Dominates GPU RAM

- Example:
  - Serving a LLaMA-3 70B model: batch size=512, prompt length= 2048 -> requires 512 GB of GPU RAM just for KV cache.
  - Serving a LLaMA-3 8B model: prompt length=1M -> much more than the storage requirement for the model weights



KV Cache Compression

- **Memory footprint** and **bandwidth** become major bottlenecks for long-context or many-batch inference
    - For long contexts (100K+ tokens) KV cache dominates GPU memory and can make single-GPU inference impossible
    - CPU->GPU transfer when offloading, network transfer for distributed serving
- **Objective:** Develop methods to **compress** or **selectively prune** the KV Cache with minimal loss in model accuracy.

KV Cache Compression

KV Cache: Prefill vs Decode

- **Prefill stage**: process the input, produce and store KV tensors for all input tokens.

- **Decoding stage**: for each generated token, model attends to cached K/V.

- Implication: some methods operate at prefill (compress stored cache), some during decoding (online eviction/compression).

Taxonomy of compression strategies

- **Token selection / dropping** (keep important tokens only)
- **Quantization / low-bit / tensor decomposition** (reduce per-token footprint)
- **Low-rank & factorization** (approximate KV matrices)
- **Encoding / gzip-style tensor encoders** for network transfer
- **Streaming & architectural approaches** (attention sinks, head splitting)
- **Redundancy-aware / semantic chunking**

Evaluation metrics

- Memory reduction (%)
- Generation quality (perplexity, task accuracy, human eval)
- Latency & throughput (prefill time, tokens/sec decode)
- Implementation complexity and compatibility (no finetune vs requires finetune)

H_2O — Heavy-Hitter Oracle

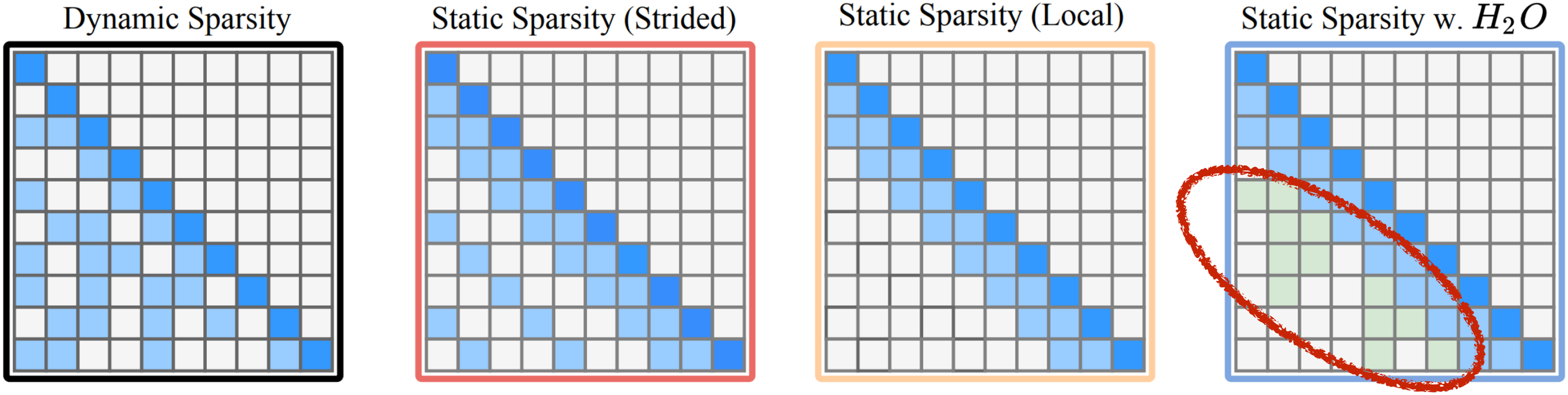
[Zhang et al, NeurIPS 2023]

- **Observation**: attention mass is highly skewed — a small subset of tokens accrue most attention (**"heavy hitters"**).
- H_2O identifies **heavy-hitter tokens** and keeps their KV entries; discards or compresses others.

H_2O — Heavy-Hitter Oracle

[Zhang et al, NeurIPS 2023]

- **Online** selection method: compute **token importance (attention accumulation)** and maintain a compact cache**.**

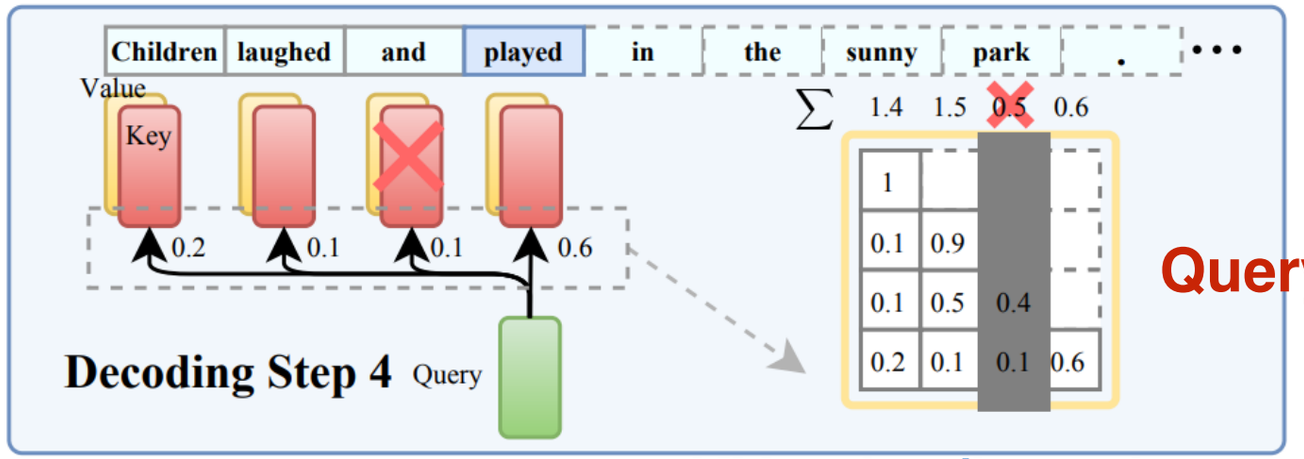


“heavy hitters” tokens

H_2O Illustration

- For **each KV pair**, compute the **accumulated attention score** from **every Query vector** to **this KV pair** over time.
- Keep the **top k scoring** KV vectors

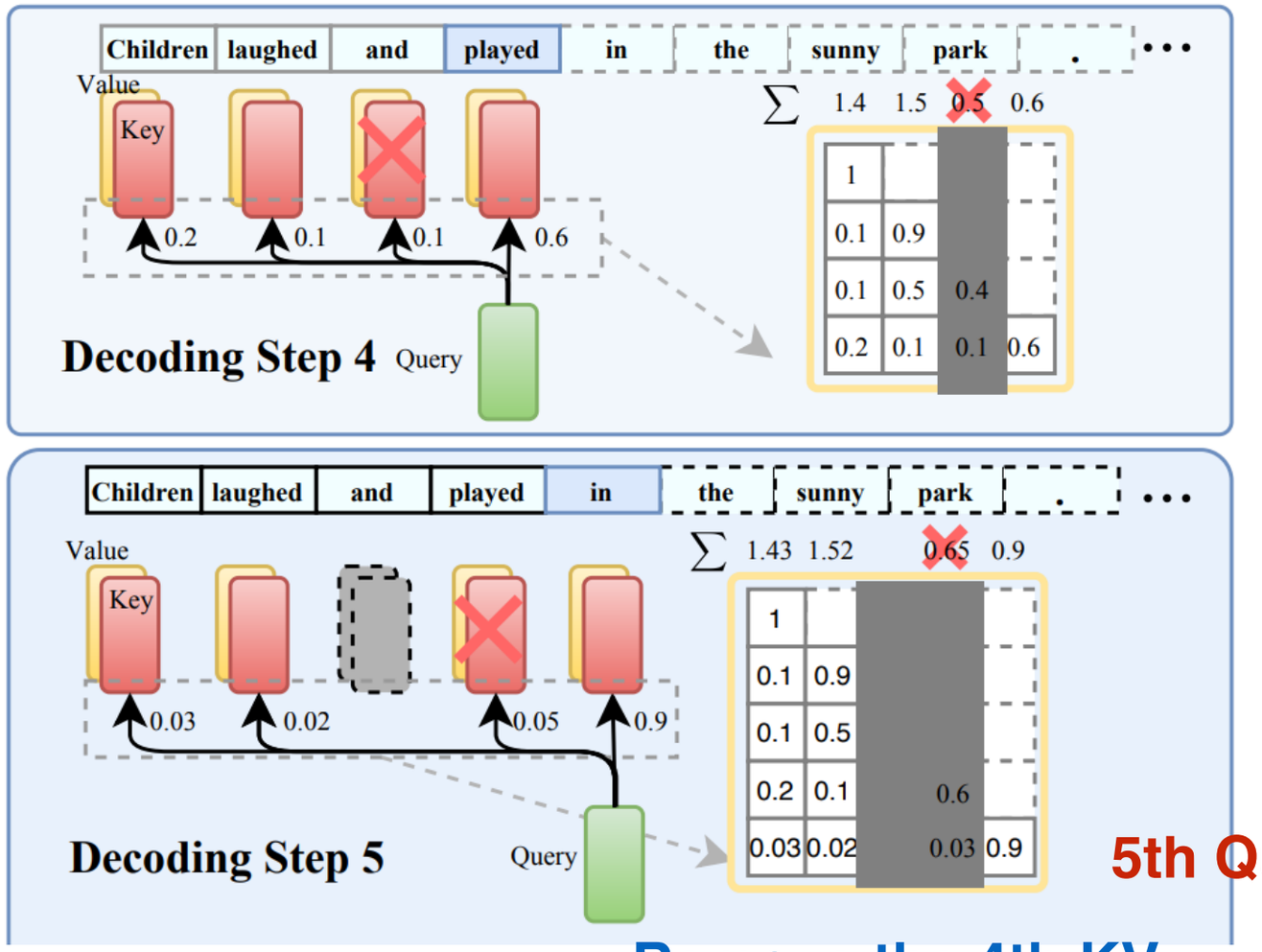
Example: k=3



H_2O Illustration

- For **each KV pair**, compute the **accumulated attention score** from **every Query vector** to **this KV pair** over time.
- Keep the **top k scoring** KV vectors

Example: k=3

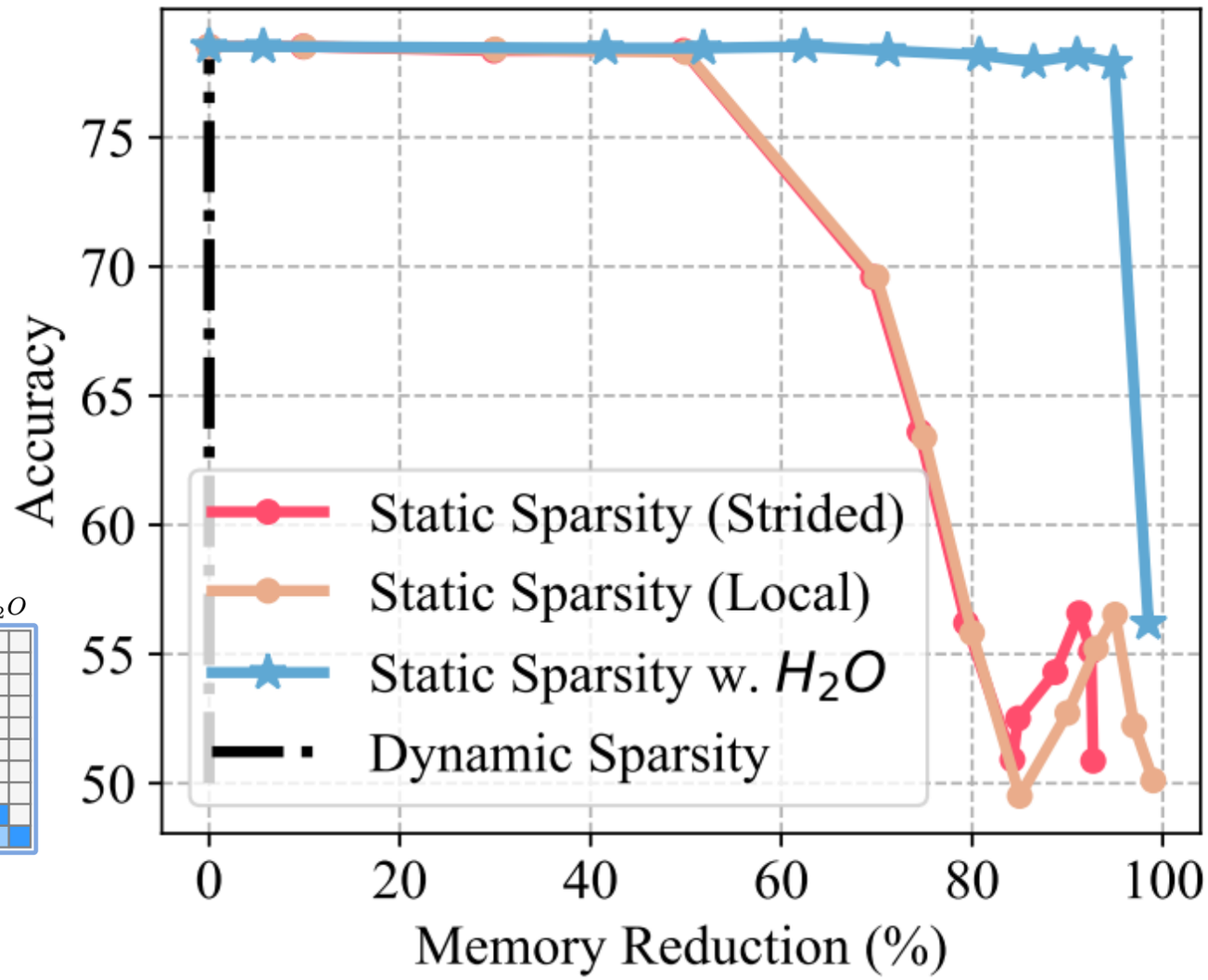


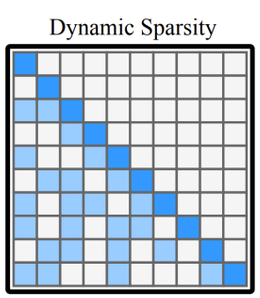
5th Query (new)

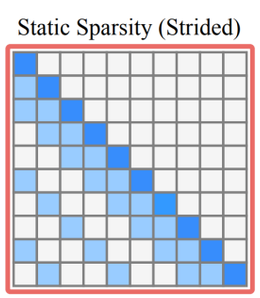
Remove the 4th KV vectors

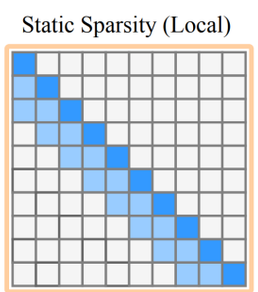
Huge Memory Reduction

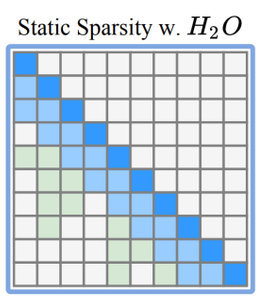
- Reduce 90% KV cache memory while keeping ~78% accuracy compared to full KV cache.











H_2O: pros & cons

+ Very effective memory reduction in practice for many LMs

+ Simple to integrate into inference loops

- Requires scoring tokens (some compute overhead)

- Can fail if "non-heavy" tokens are crucial for downstream reasoning

StreamingLLM

[Xiao et al. ICLR 2024]

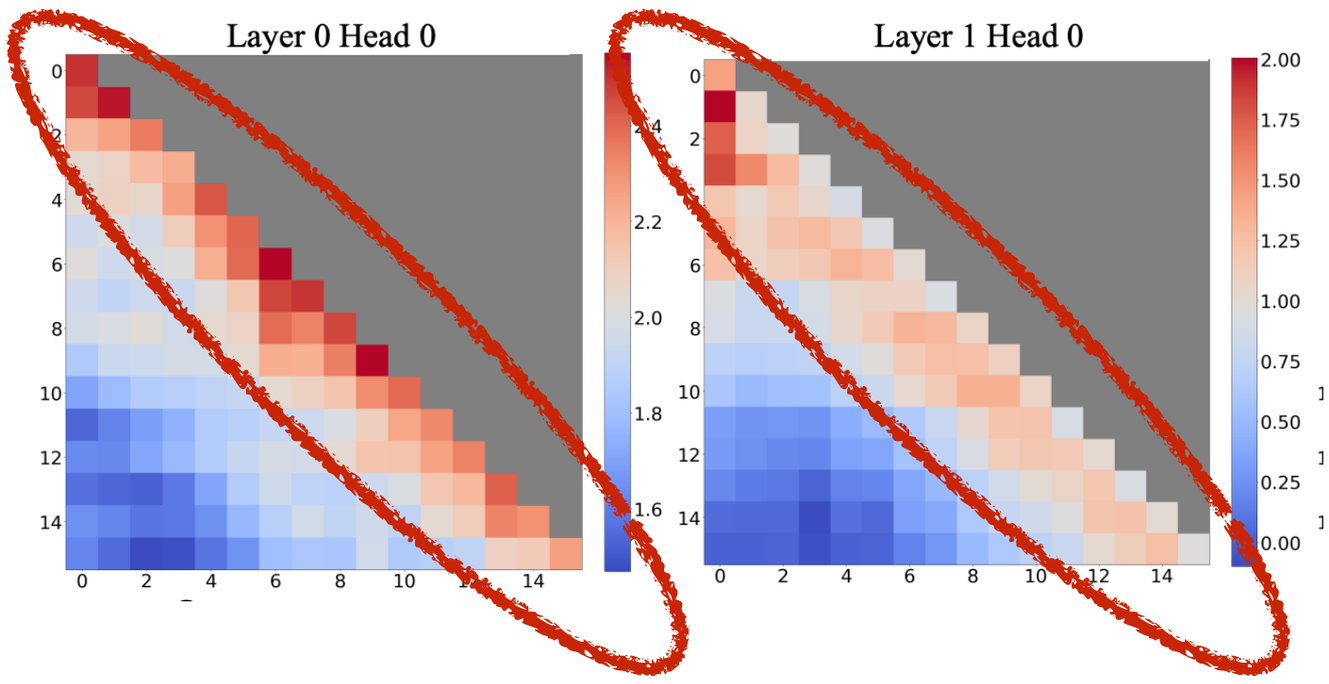
- **Key empirical observation**: models develop ***attention sinks*** — specific tokens (**often early ones**) that attract sustained attention.

- StreamingLLM: combine a **sliding window of recent tokens** with **a tiny set of sink tokens** (initial or learned placeholders) kept indefinitely in the cache.

- Achieves effectively “infinite” context streaming with bounded KV memory and without finetuning.

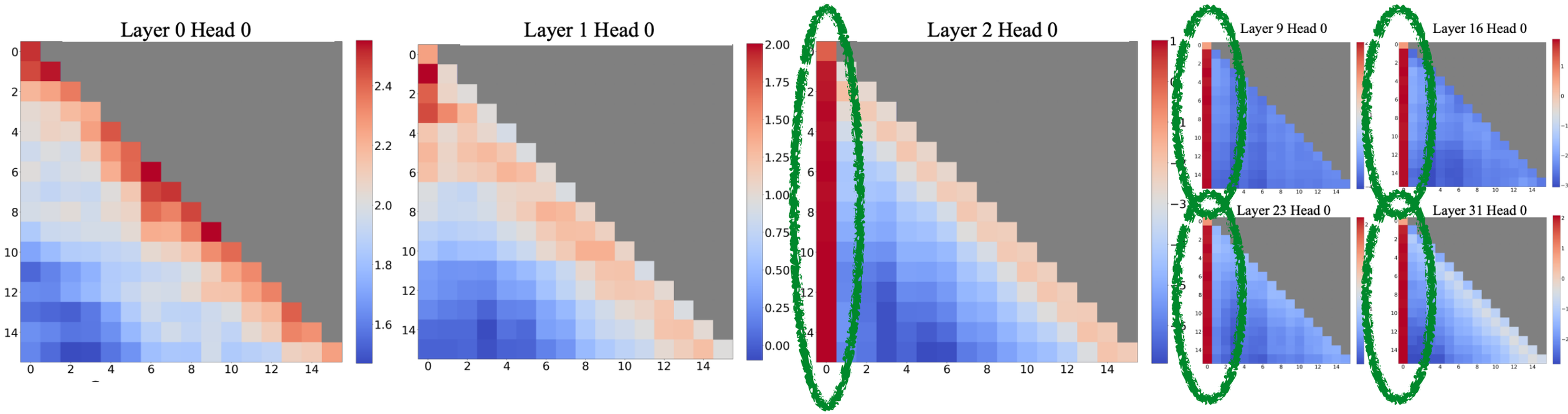
Attention Visualization

- Visualization of the average attention logits in Llama-2-7B over 256 sentences, each with a length of 16. Observations include:
  - (1) The attention maps in the first two layers (layers 0 and 1) exhibit **the "local" pattern**, with recent tokens receiving more attention.



Attention Visualization

- Visualization of the average attention logits in Llama-2-7B over 256 sentences, each with a length of 16. Observations include:
  - (1) The attention maps in the first two layers (layers 0 and 1) exhibit **the "local" pattern**, with recent tokens receiving more attention.
  - (2) Beyond the bottom two layers, the model heavily attends to the **initial token across all layers and heads**.

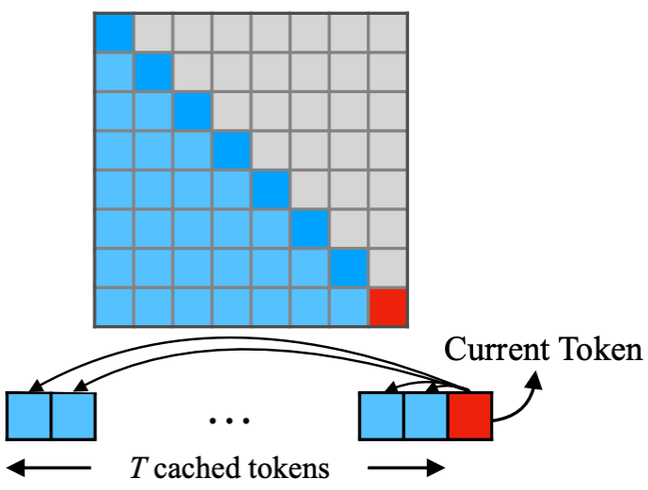


StreamingLLMs

- Yellow: Attention Sink
- Light Blue: Sliding Window

T: total decoding time steps
L: sliding window size

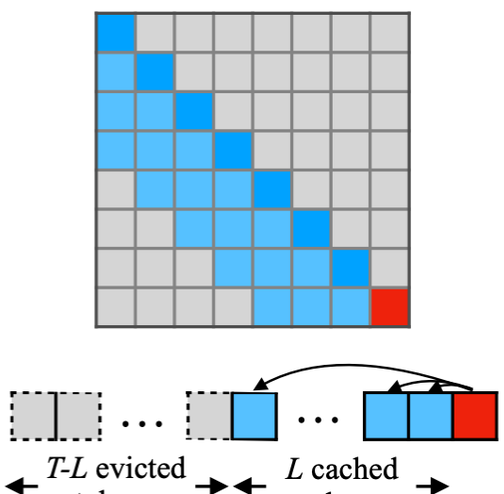
(a) Dense Attention



$O(T^2)$✗ **PPL:** 5641✗

Has poor efficiency and performance on long text.

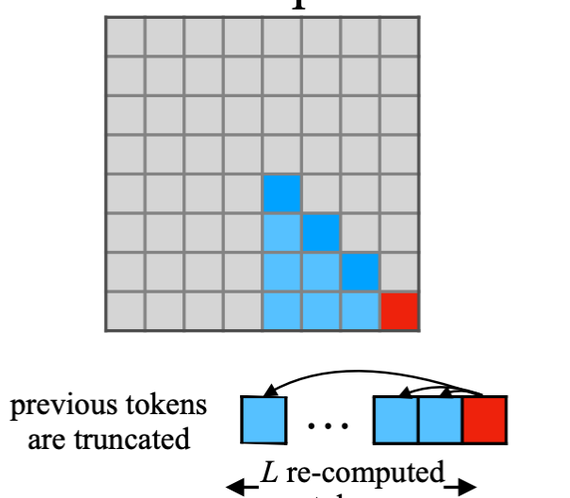
(b) Window Attention



$O(TL)$✓ **PPL:** 5158✗

Breaks when initial tokens are evicted.

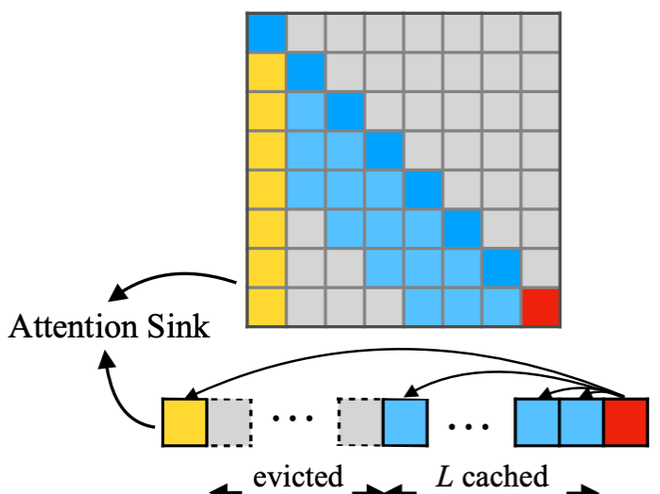
(c) Sliding Window w/ Re-computation



$O(TL^2)$✗ **PPL:** 5.43✓

Has to re-compute cache for each incoming token.

(d) **StreamingLLM (ours)**

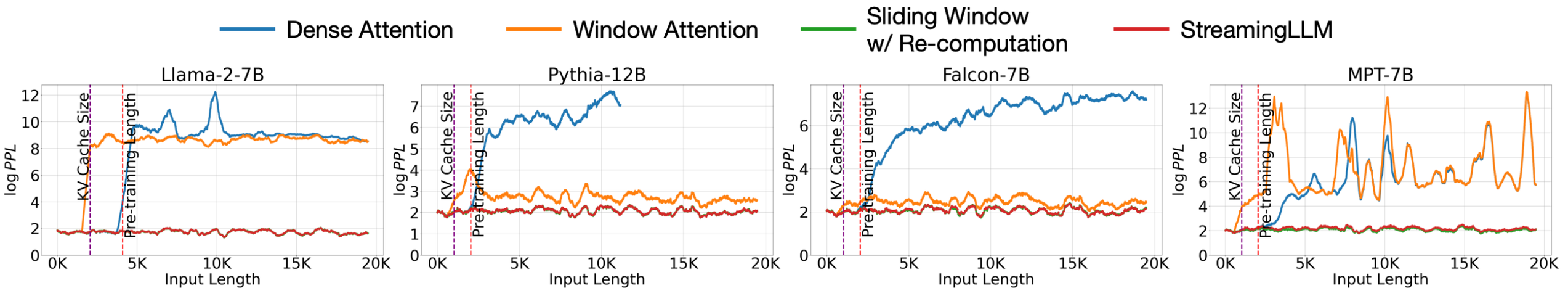


$O(TL)$✓ **PPL:** 5.40✓

Can perform efficient and stable language modeling on long texts.

Generate tokens beyond pre-training token length

- Results: LM perplexity on texts with 20K tokens across various LLM.
  - (1) **Dense attention** fails once the input length surpasses the pre-training attention window size.
  - (2) **Window attention** collapses once the input length exceeds the cache size, i.e., the initial tokens are evicted.
  - (3) **StreamingLLM** demonstrates stable performance, with its perplexity nearly matching that of the sliding window with re-computation baseline.



StreamingLLM: benefits and implementation notes

- Enables coherent generation across millions of tokens with bounded memory.

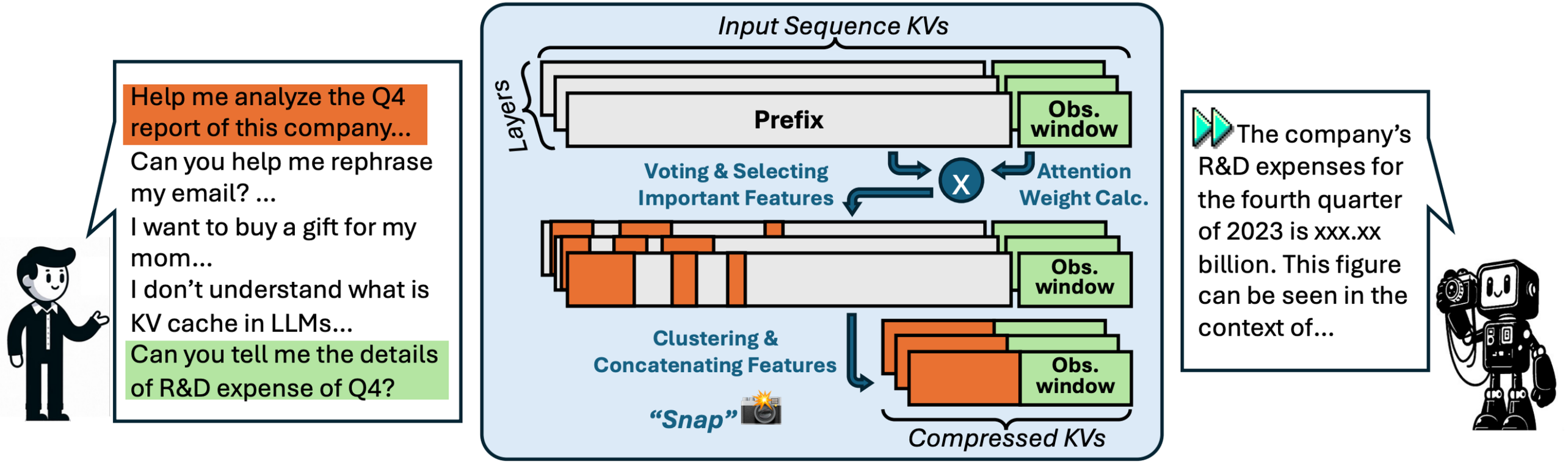
- No model finetuning required (in the rolling variant).

- Implementation: keep small global sink KV entries + windowed KV; small engineering changes to attention computation.

SnapKV

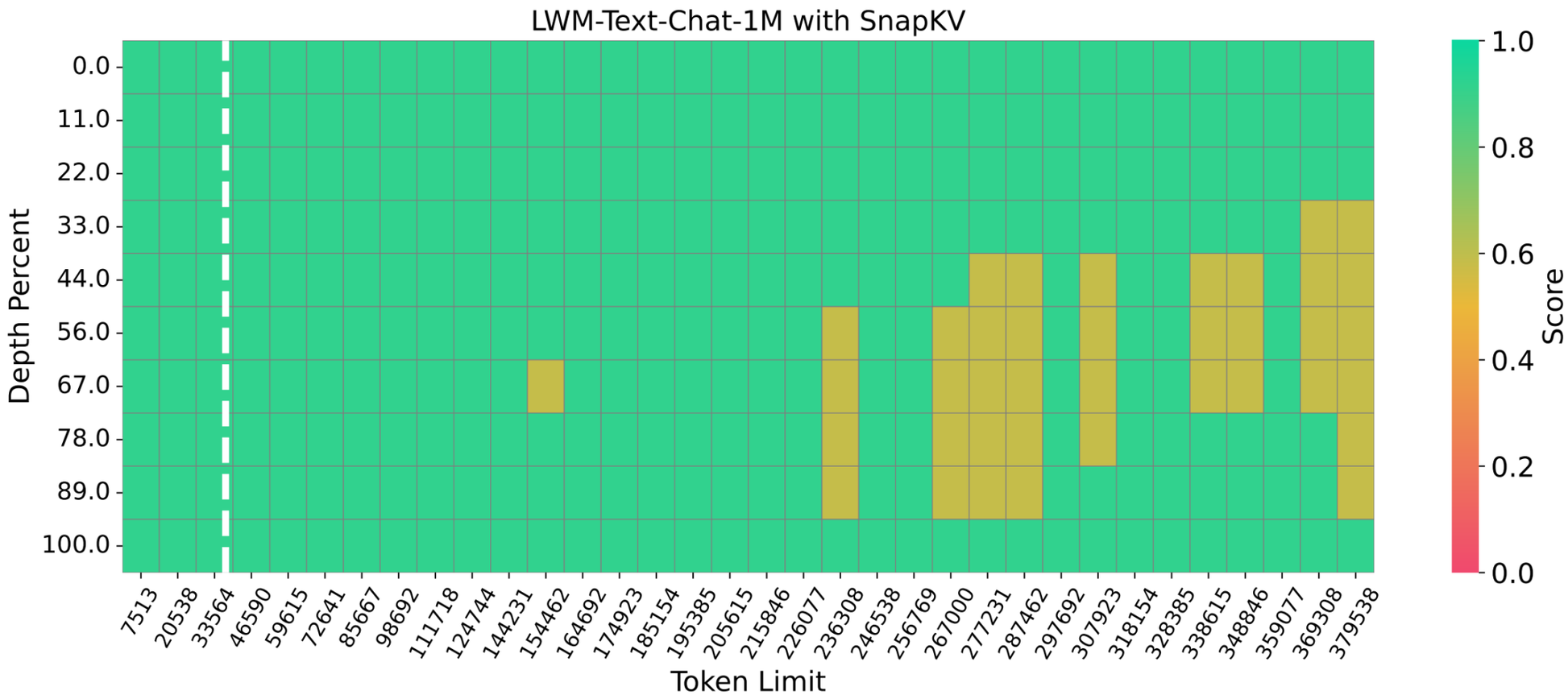
[Li et al, NeurIPS 2024]

- LLM Knows What You are Looking for Before Generation
- Key observation: each attention head consistently focuses on **specific parts of the prompts (orange)**. This pattern can be obtained from an **“observation” window (green)** at the end of the prompt.



Needle-in-a-Haystack

- Needle-in-a-Haystack test performance comparison on single A100-80GB GPU
- The x-axis denotes the length of the document (the “haystack”) from 1K to 380K tokens; the y-axis indicates the position that the “needle” (a short sentence) is located within the document.
- For example, 50% indicates that the needle is placed in the middle of the document. Here LWMChat with SnapKV is able to retrieve the needle correctly before 140k and with only a little accuracy drop after. Meanwhile, the original implementation encounters OOM error with 33k input tokens (white dashed line).

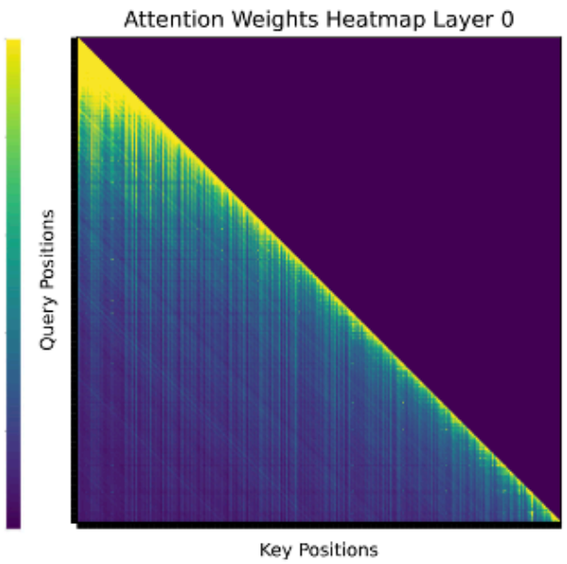


Pyramidal Information Funneling

[Cai et al, COLM 2025] (Oral)

1. **Figure**: Attention Heatmap of LLAMA-3 to do QA task based on multi documents
2. **Six Figure**: Each represents attention visualization for one layer. (layer=0, 6, 12, 18, 24, 30)

Finding 1: Uniform Distribution in Bottom Layer: Attention distribution in layer 0 has not a focus

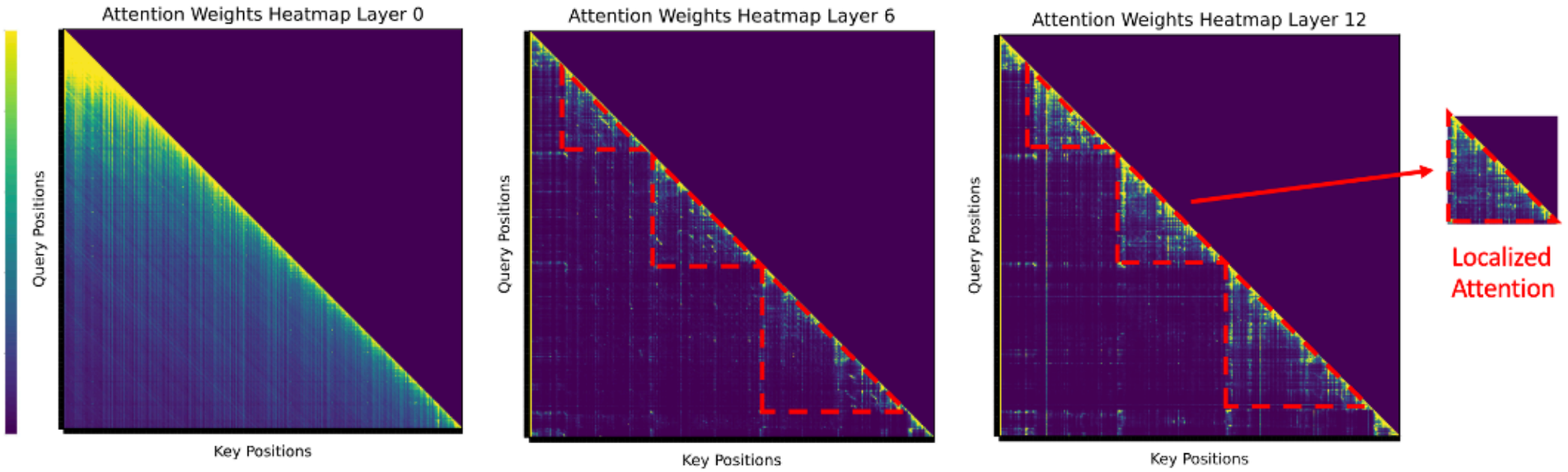


Pyramidal Information Funneling

[Cai et al, COLM 2025] (Oral)

1. **Figure**: Attention Heatmap of LLAMA-3 to do QA task based on multi documents
2. **Six Figure**: Each represents attention visualization for one layer. (layer=0, 6, 12, 18, 24, 30)

Finding 2: Localized Aggregation: Each **red triangle** represents one document, Attention is aggregated to the first tokens inside the document

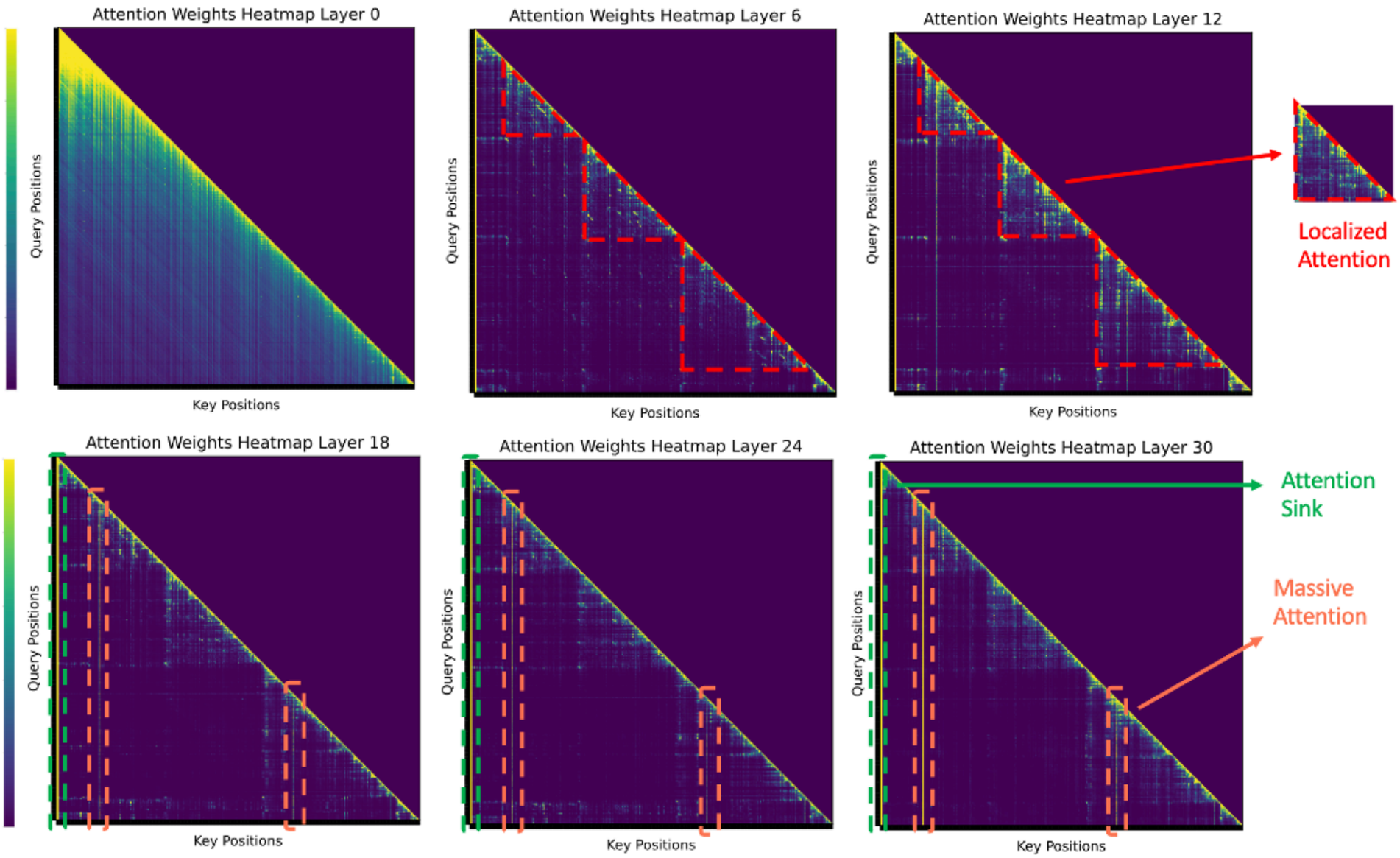


Pyramidal Information Funneling

[Cai et al, COLM 2025] (Oral)

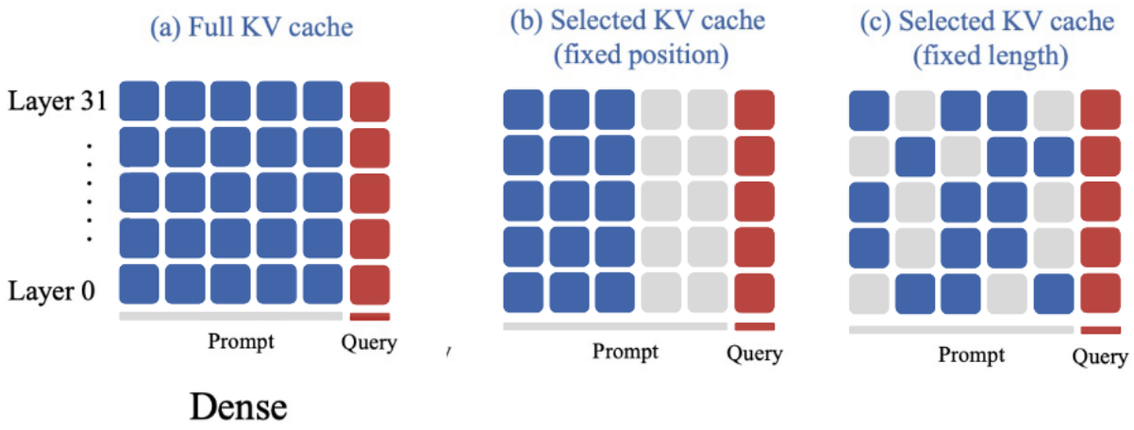
1. **Figure**: Attention Heatmap of LLAMA-3 to do QA task based on multi documents
2. **Six Figure**: Each represents attention visualization for one layer. (layer=0, 6, 12, 18, 24, 30)

Finding 3: Global Aggregation: In top layers, Attention is aggregated to Attention Sink and some massive activations



Existing Works on KV Cache Compression

- **Dense**: Use full KV cache
- **StreamingLLM (SLM)**: keeps only the first few and most recent tokens — inspired by the *attention sink* phenomenon.
- **SnapKV / H2O (Heavy Hitter Oracle):** selects a fixed number of KV entries per layer based on attention scores within an instruction window.



***RQ1*: Does each layer show the same behavior?**

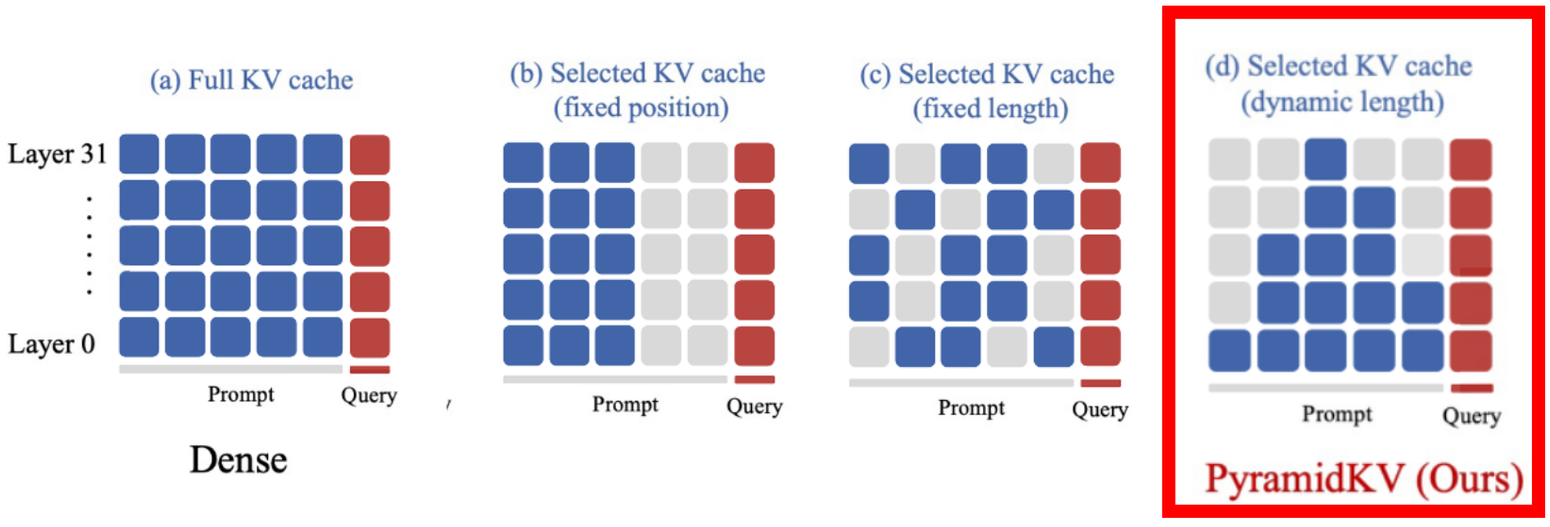
***RQ2*: Is it optimal to keep the same number of KV Cache in each layer?**

Problem: All assume same KV size should be kept the same at all depths

PyramidKV

[Cai et al, COLM 2025] (Oral)

- **Dense**: Use full KV cache
- **StreamingLLM (SLM)**: keeps only the first few and most recent tokens — inspired by the *attention sink* phenomenon.
- **SnapKV / H2O (Heavy Hitter Oracle):** selects a fixed number of KV entries per layer based on attention scores within an instruction window.
- **PyramidKV**: **higher layers** with **fewer KV** Cache budget and **lower layers** with **more KV** Cache budget

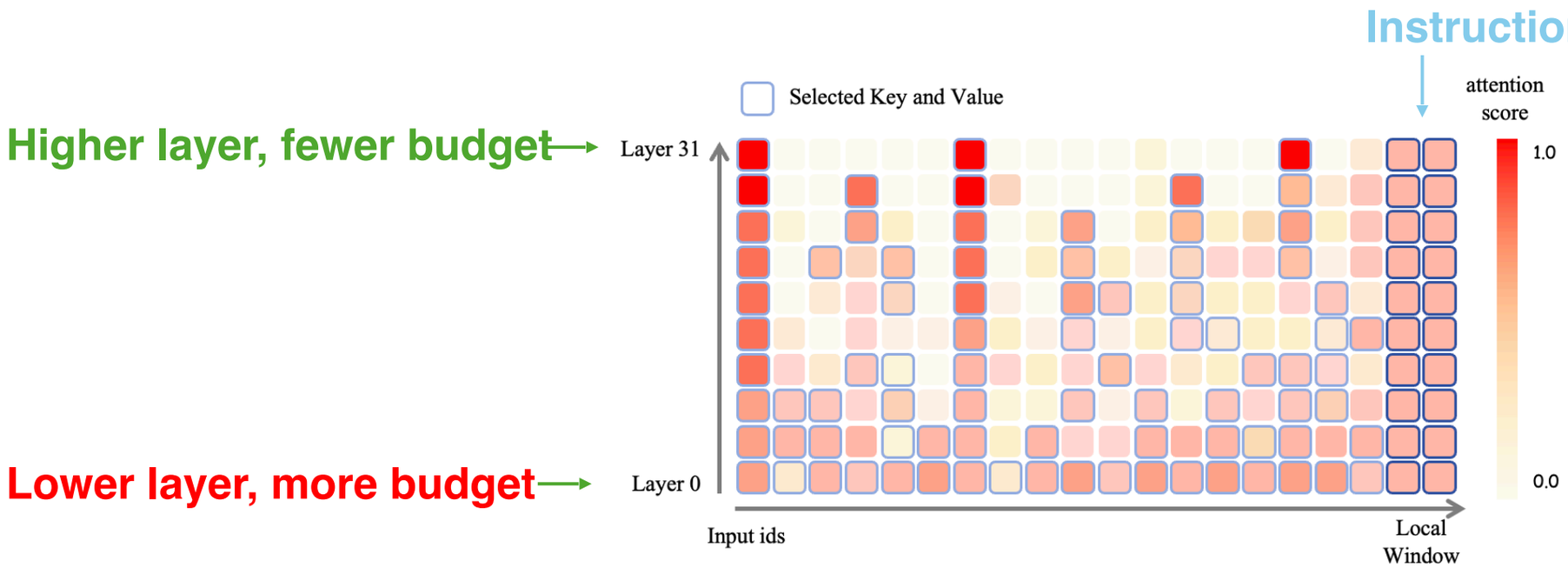


In our method, the budget of each layer is DIFFERENT

PyramidKV

[Cai et al, COLM 2025] (Oral)

- Allocate KV cache budget adaptively across layers



1. Instruction tokens - retain the KV cache for the **last α tokens** of the input across all layers

2. Higher layer, fewer budget

3. Lower layer, more budget

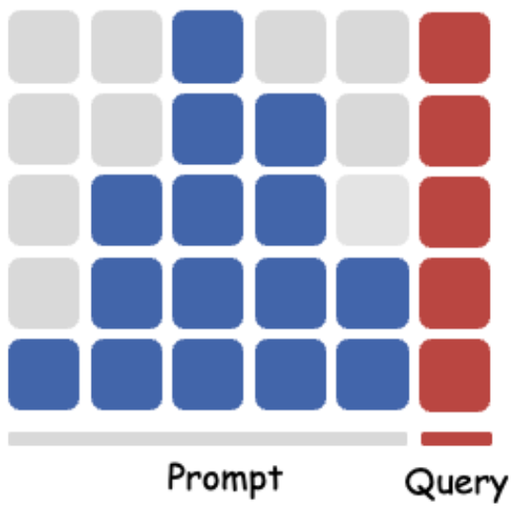
PyramidKV – Budget Allocation

- Budget Computation:
  - k^{total} - total KV budget (has subtracted the instruction tokens already, avg k- α per layer)
  - m – number of layers
  - β – determine the shape of the pyramid.

top $k^{m-1} = k^{\text{total}}/(\beta \cdot m)$

Intermediate $k^l = k^{m-1} - \frac{k^{m-1} - k^0}{m} \times l.$

bottom $k^0 = (2 \cdot k^{\text{total}})/m$



PyramidKV – Token Selection

- Compute **Important Score** of each token based on the attention matrix
  - The total attention each token (at the index of *i* at the head *h*) receives from the instruction tokens

$$s_i^h = \sum_{j \in [n-\alpha, n]} A_{ij}^h$$

- We then select top k tokens based on the importance score to retain in each layer (k is the budget computed in budget allocation step)

Experiment Settings

- **Dataset**: LongBench [8] - 17 datasets across 6 tasks with average length from 1235 to 18409

- **Models**:
  - LLaMa-3-8B-Instruct,
  - Mistral-7B-Instruct,
  - LLaMa-3-70B-Instruct

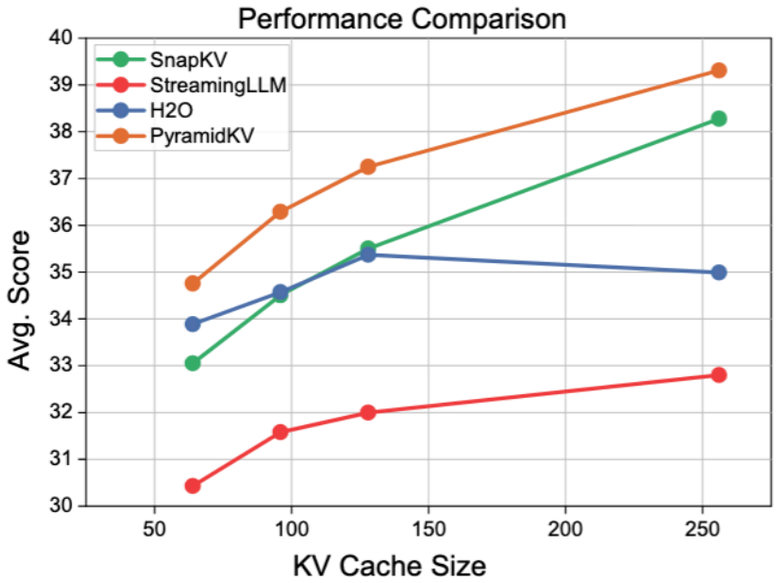
- **Baseline algorithms**: SnapKV, H2O, StreamLLM

- **Setting**:
  - Use the same average KV cache for each layer for all method to ensure fair comparison

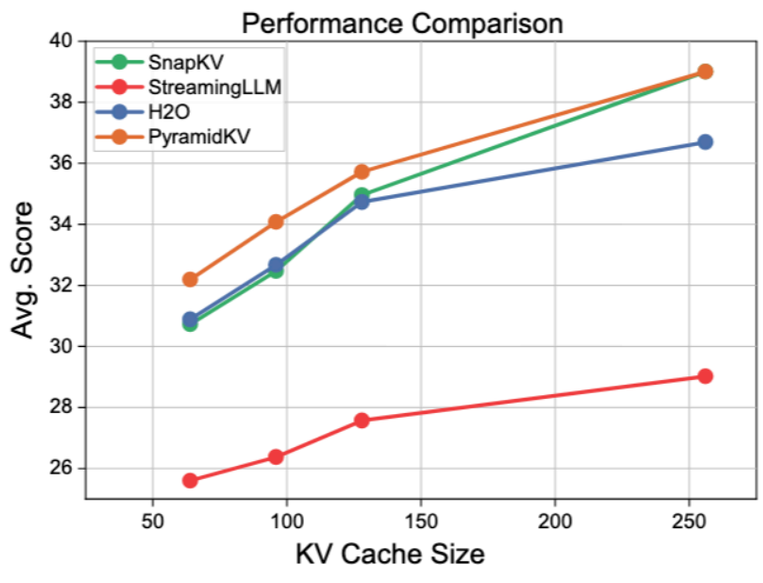
Main Results – Limited Budget

- X: KV cache size
- Y: Average performance score (higher is better) across 17 datasets in LongBench

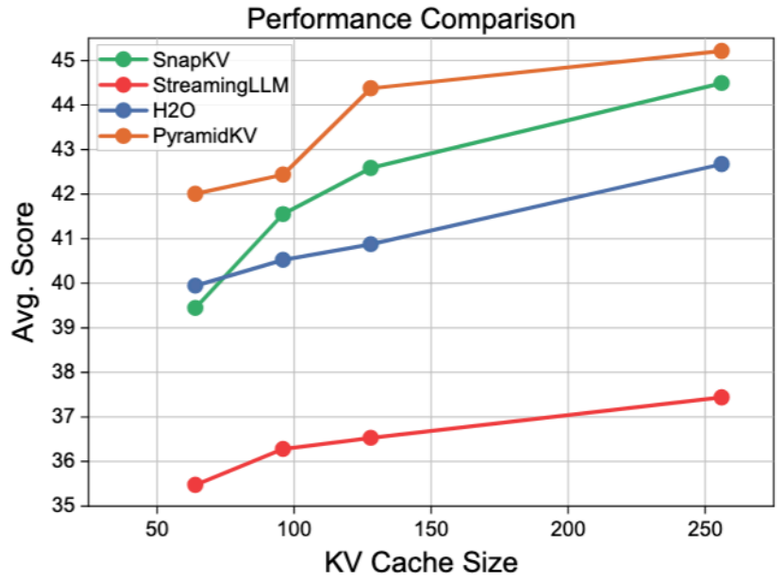
Llama-3-8b



Mistral-7B



Llama-3-70b



- **Key takeaway**: Our PyramidKV outperforms the baseline algorithms across budgets from 64 to 256 (average KV cache per layer)

Main Results – Tasks

Method	Qasper	MF-en	TREC
SnapKV	19.1	34.6	39.2
StreamingLLM	16.2	27.7	37.7
H2O	20.8	31.0	39.0
PyramidKV	**23.7**	**37.5**	**58.8**

Tasks

- **Qasper**: QA over NLP papers
- **MF-en**: QA in specific domains
- **TREC**: ICL classification task with 50 classes

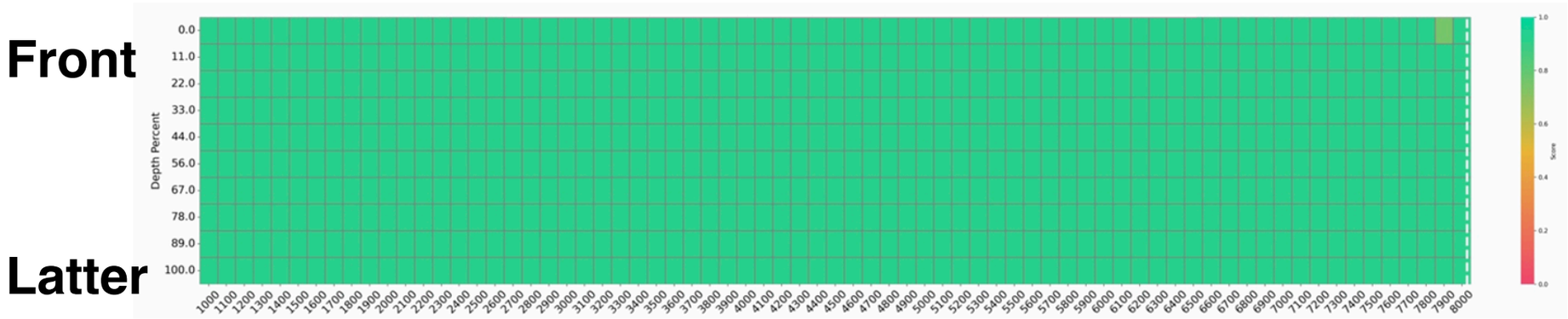
Evaluation setting: Average performance score across 3 models, with KV cache size=64

Key takeaway

- In the **extreme memory efficient scenario (i.e., KV cache size = 64)**, PyramidKV outperforms baselines on some tasks by a large margin
- PyramidKV works well in specific domain QA (i.e., MF-en), or In-Context Learning (i.e., TREC)

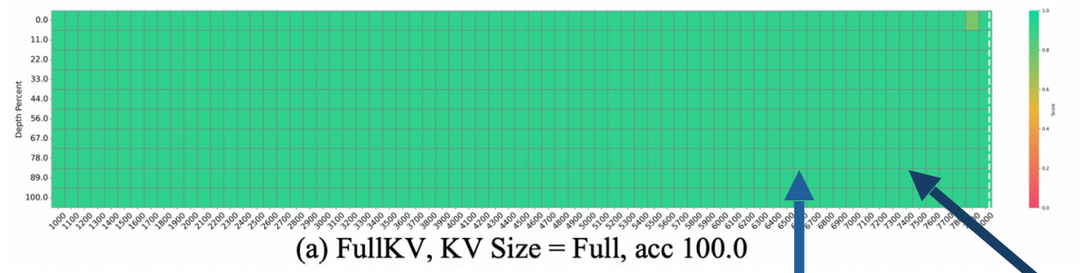
Needle-in-a-haystack

- **Input Data**: Insert factual information necessary for QA (i.e., a 70digit number for a special word “fantastic apple”) inside some irrelevant paraphrases
- **Evaluation**: Answer questions based on the information (i.e., what it the number for “fantastic apple?)
    - **Color**: **Green** represents success, **Red** represents Failure, **Yellow** represents partial success.
- **Y-Axis**: The relative location to insert the factual information

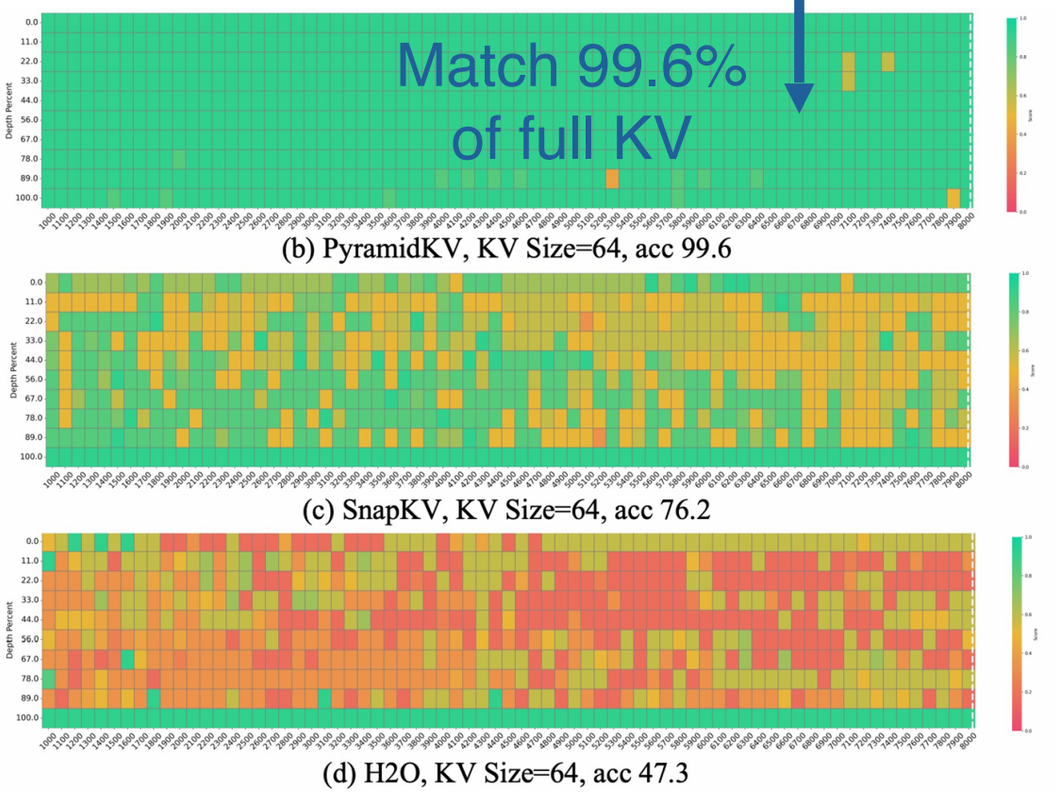


- **X-Axis**: Each column represents one specific Input length

Needle-in-a-haystack

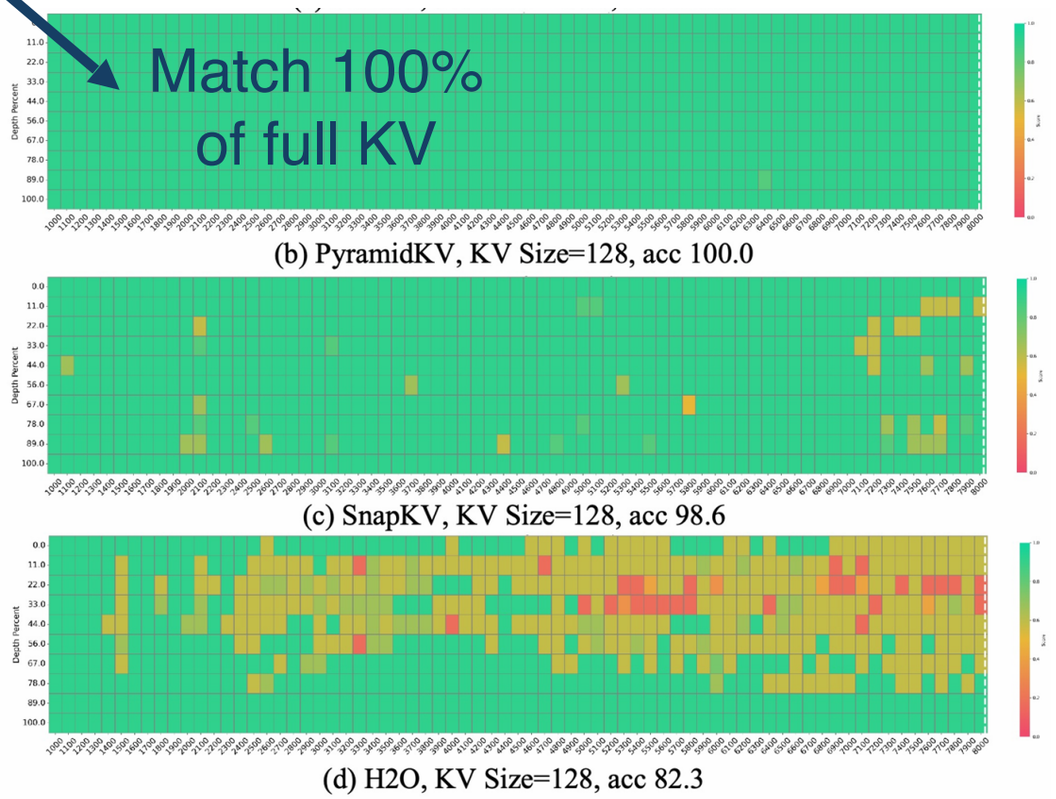


Model: LLaMA 3-70B, 8K Context Size



Match 99.6% of full KV

- **KV size = 64** (comp. rate = 0.7%), PyramidKV achieves 99.6 % accuracy



Match 100% of full KV

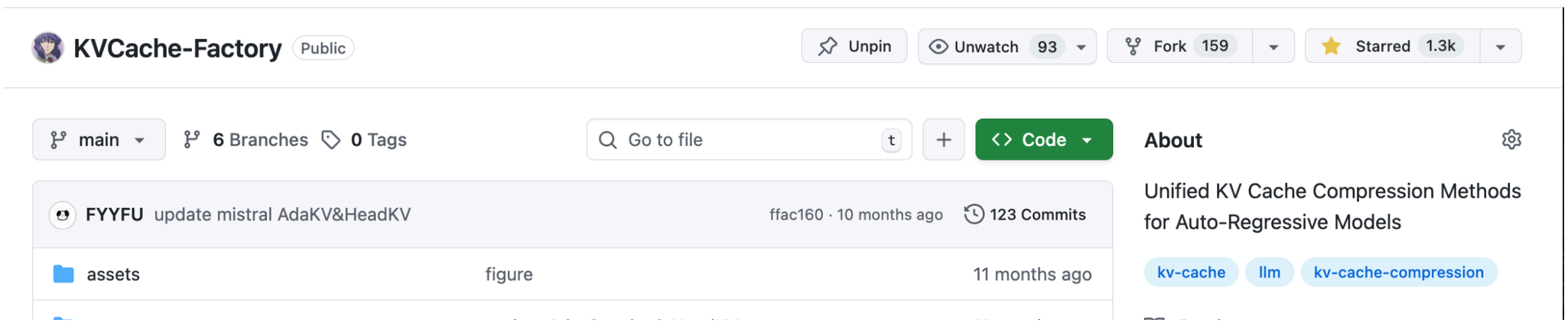
- **KV size = 128** (comp. rate = 1.4%), PyramidKV achieves 100 % accuracy

Open-Source Project

- **Implementation**
  - PyramidKV's codebase turned out to be a unified framework for diverse KV optimization methods
  - PyramidKV's codebase helps the community work on research at KV cache easily
  - https://github.com/Zefan-Cai/KVCache-Factory

KVCache-Factory

Easy-to-use KV Cache Compression



THE UNIVERSITY of WISCONSIN MADISON

R-KV: Redundancy-aware KV Cache Compression for Reasoning Models
[Cai et al. NeurIPS 2025]

- **Overthinking issue:** Existing KV cache compression leads to loss of information, therefore increasing the overthinking issue.
- **Key idea of R-KV:** Estimate the information redundancy based on comparing all the key pairs.

You are given a math problem. Problem: In Mr. Roper's class of 30 students,
[Question and Instruction - 102 words]
First, the problem says that there are 30 students in total in the class. Out of
[Think - 203 words]
...
[Reflection for 13 times and 581 words in total]
...
But wait, the .. So, 10% of 30 is 3. So 3 students are leaving early.
[Think - 36 words]
But in the initial problem...So 3 students are leaving early.
[Think - 42 words]
But wait, the ...30 is 3. So 3 students are leaving early.
[Think - 36 words]
But in the initial problem, the... So 3 students are leaving early.
[Think - 42 words]
But wait, the .. early?" So, 10% of 30 is 3. So 3 students are leaving early.
[Think - 36 words]
But in the initial ..." So, 10% of 30 is 3. So, 3 students are leaving early.
[Think - 40 words]
But wait, the user wrote: ...10% of 30 is 3. So, 3 students are leaving early.
[Think - 31 words]
But in the initial ... of 30 is 3. Therefore, 3 students are leaving early.
[Think - 83 words]
I think that's all. The calculation is straightforward: 10% of 30 is 3.
[Conclusion- 11 words]

Overthinking: DeepSeek R1 using SnapKV

$$\overline{\boldsymbol{K}}_i^h = \frac{\boldsymbol{K}_i^h}{\|\boldsymbol{K}_i^h\|_2 + \epsilon} \in \mathbb{R}^d,$$

$$\boldsymbol{S}^h = \overline{\boldsymbol{K}}^h (\overline{\boldsymbol{K}}^h)^\top \in \mathbb{R}^{n \times n}$$

$$S_{i,i}^h \leftarrow 0, \forall i \in [0, n),$$

THE UNIVERSITY of WISCONSIN MADISON

Open Research Problems

- Provable guarantees: when can we safely evict tokens without losing reasoning correctness?

- Learning to compress: end-to-end differentiable KV compression that can be finetuned for tasks.

- Multimodal KV caches (visual tokens), and cross-modal redundancy handling.

- Better benchmarks for long reasoning under compressed KV.

Questions?