SnapKV: LLM Knows What You are Looking for Before Generation

Summary

Paper2: Snap KV

Goal

- **Minimizes KV cache size while still delivering comparable accuracy**
- **Compress prompt KV cache before decoding**

Idea

- **LLMs knows what you are looking for before generation**
  - two observations
  - Attention during prefill strongly predicts future decoding attention
  - Only a small subset of prompt tokens remains important

Method

- **Observation window**
- **SnapKV algorithm**

Experiments

- **Long Bench, Needle-in-a-Haystack**

Background

Attention sink

- **Attention sink**

Background

Previous KV Cache Eviction Method

- **Problem:**
  - **less efficient KV Cache(**Prompt length ↑ -> Decoding speed ↓, memory ↑)
  - Contain
- **Solution:** KV cache eviction
  - Existing Methods: exploit non-uniform attention
    - Streming LLM: preserving attention sinks and a sliding window of recent tokens
    - H2O: maintains cumulative attention scores, preserves heavy-hitter tokens(repeatedly receive high attention)
    - **Problem 1**: long context evaluation X
    - **Problem 2**: compressing KV cache appended during generation(do not compress prompt)

RotateKV

KIVI

- Granularity: key cache: per-channel, value cache: per-token
- key outlier, value: decode attention weight
- ,Video value per-channel .

KVQuant

- Granularity: key cache: per-channel, value cache: per-token
- key outlier, value: decode attention weight
- ,Video value per-channel .

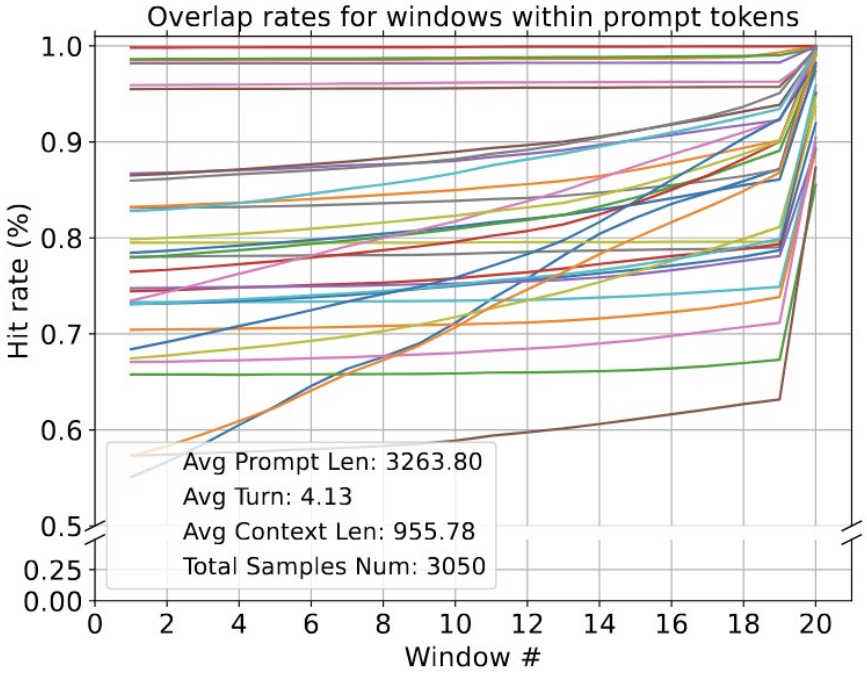
AdaKV

Proposed Methods

Key Idea: LLM knows what you are looking for before generation

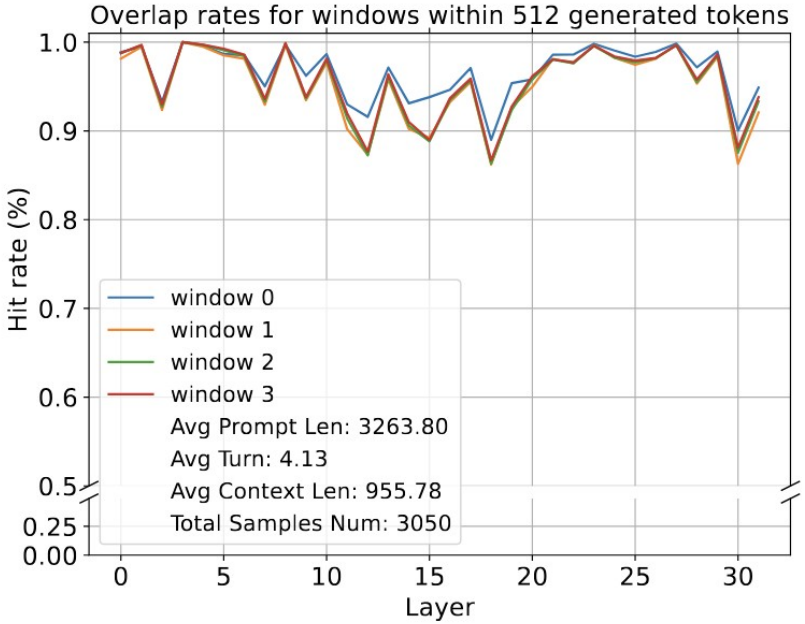
- **LLM knows what you are looking for before generation**
  - Attention allocation patterns in the Query-Key matrix during token generation

OBSERVATION 1: Pattern can be identified before generation



Overlap rates between attention features of the **input sequence** and the **actual** generation

OBSERVATION 2: Pattern is consistent during generation

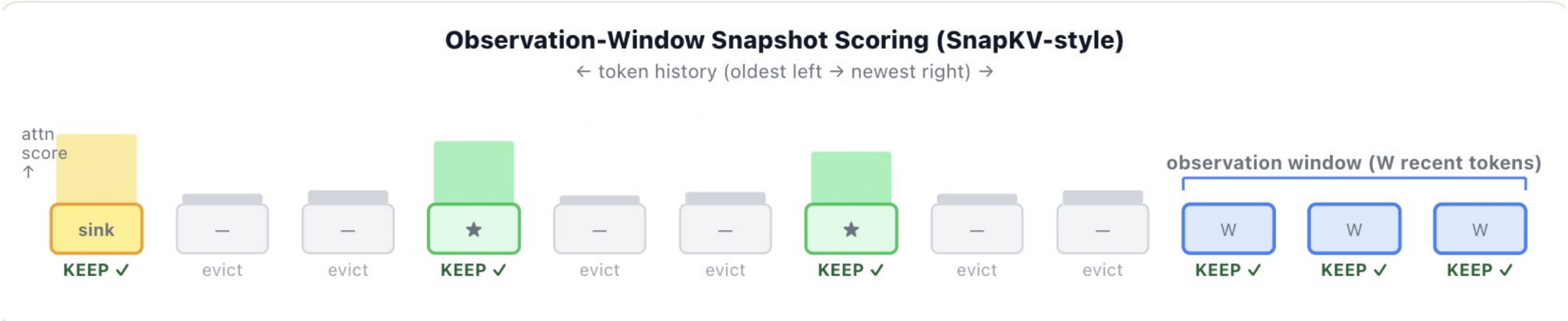


Layer-wise overlap between the last **input window** and the first four **generation windows**

Proposed Methods

Observation Window

- **Key Idea**
  - Estimate future token importance using only the last W prompt tokens



- **Process**
  - Observation window queries-> Attend to entire prompt -> Attention Scores -> Evict Tokens

Proposed Methods

SnapKV Algorithm

- **Maintaining a constant amount of prompt KVs during generation**
  - Identifying and selecting the most crucial attention features per head to create the compressed KV Cache

Process: Prompt → Observation Window → Attention Map → Pooling → Top-K Selection → Compressed KV Cache

Vote for important previous features

$$\mathbf{C} = \sum_{i=0}^{L_{\text{obs}}} \mathbf{W}_{\text{obs}}[:, i, :]$$

$$I = \text{Top}_k(\mathbf{C}, k)$$

← Pooling

$$Q = XW_q, K = XW_k, V = XW_v \qquad q_t, k_t, v_t \in \mathbb{R}^d$$

1) Q, K, V Projection

$$z_{t,s} = \frac{q_t k_s}{\sqrt{d}} + M_{t,s} \qquad M_{t,s} = \begin{cases} 0, & s \le t \\ -\infty, & s > t \end{cases} \qquad \alpha_{t,s} = \frac{\exp(z_{t,s})}{\sum_{j=1}^{t} \exp(z_{t,j})}$$

$$\alpha_{t,s} = \text{Attn}(x_t \to x_s)$$

2) Causal Attention Weights

$$w_{i,t} = Attn(x_i \to x_t) \qquad u_t = \sum_{i \in I_{inst}} w_{i,t} \qquad I = \text{TopK}(u_1, \dots, u_N; K)$$

3) Importance Scoring & Top-K Selection

Update and store compressed KVs

Attention features: selected + observation

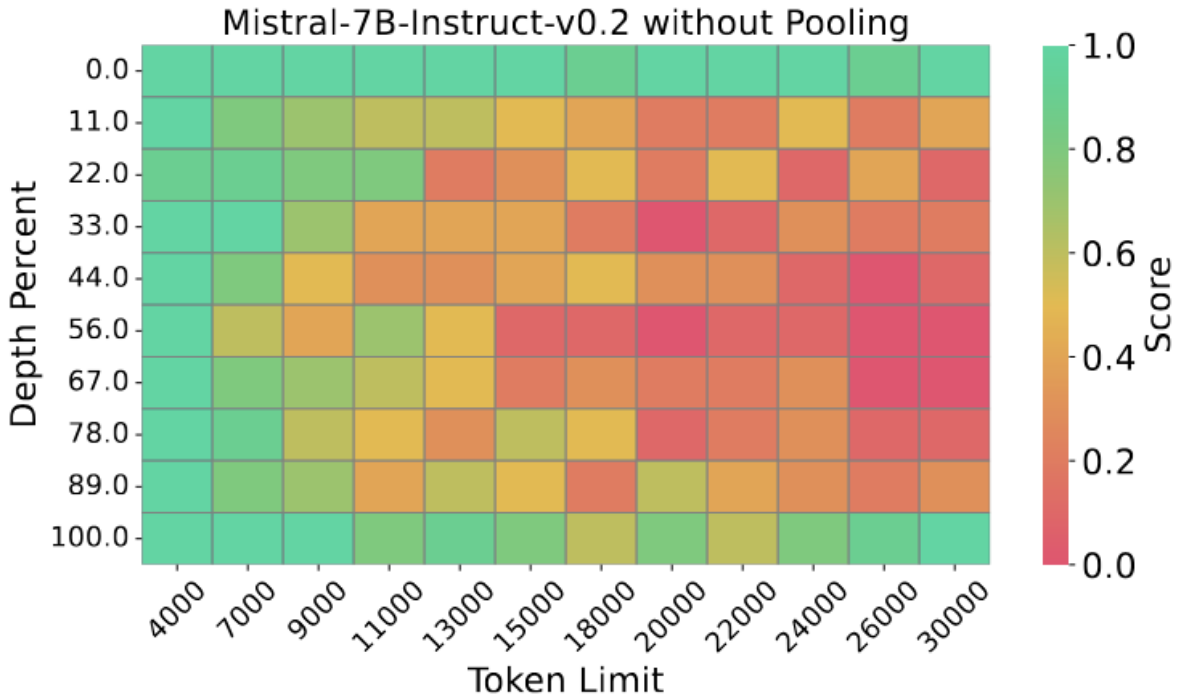
$$\tilde{K} = K[:, I, :], \tilde{V} = V[:, I, :]$$

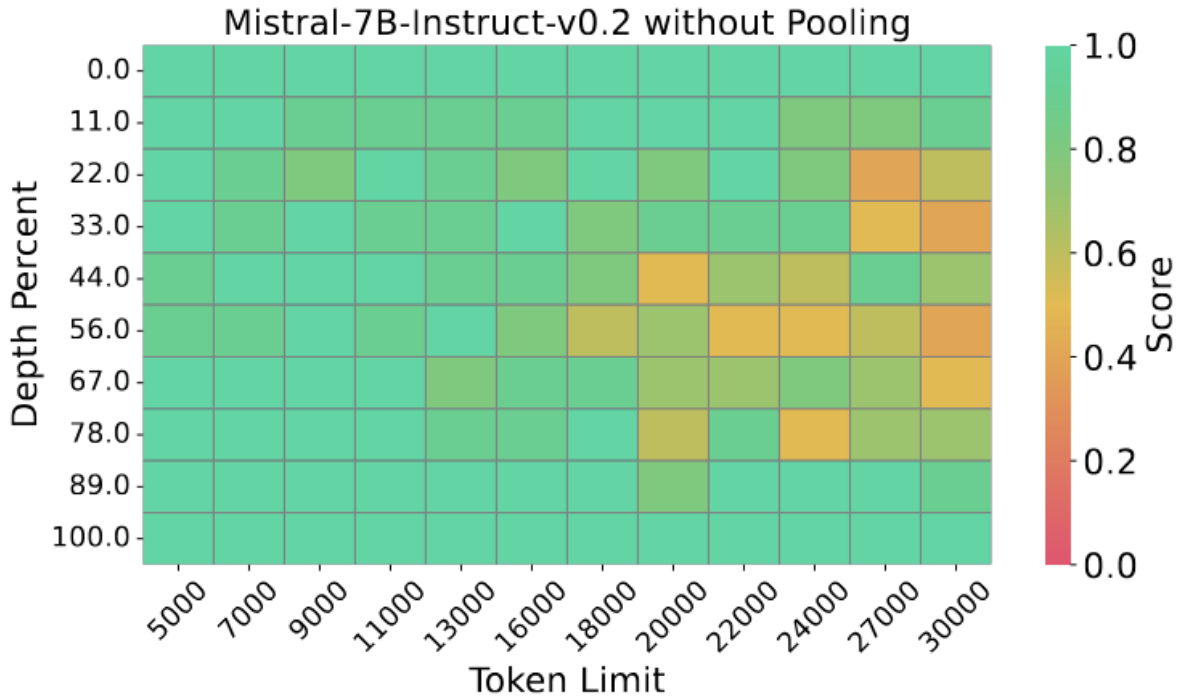
4) KV Cache Compression

Proposed Methods

Pooling

- **Efficient Clustering via Pooling**
  - Selecting only high-attention KV features -> loses surrounding contextual information -> reducing accuracy
  - pooling-based clustering preserves neighboring context



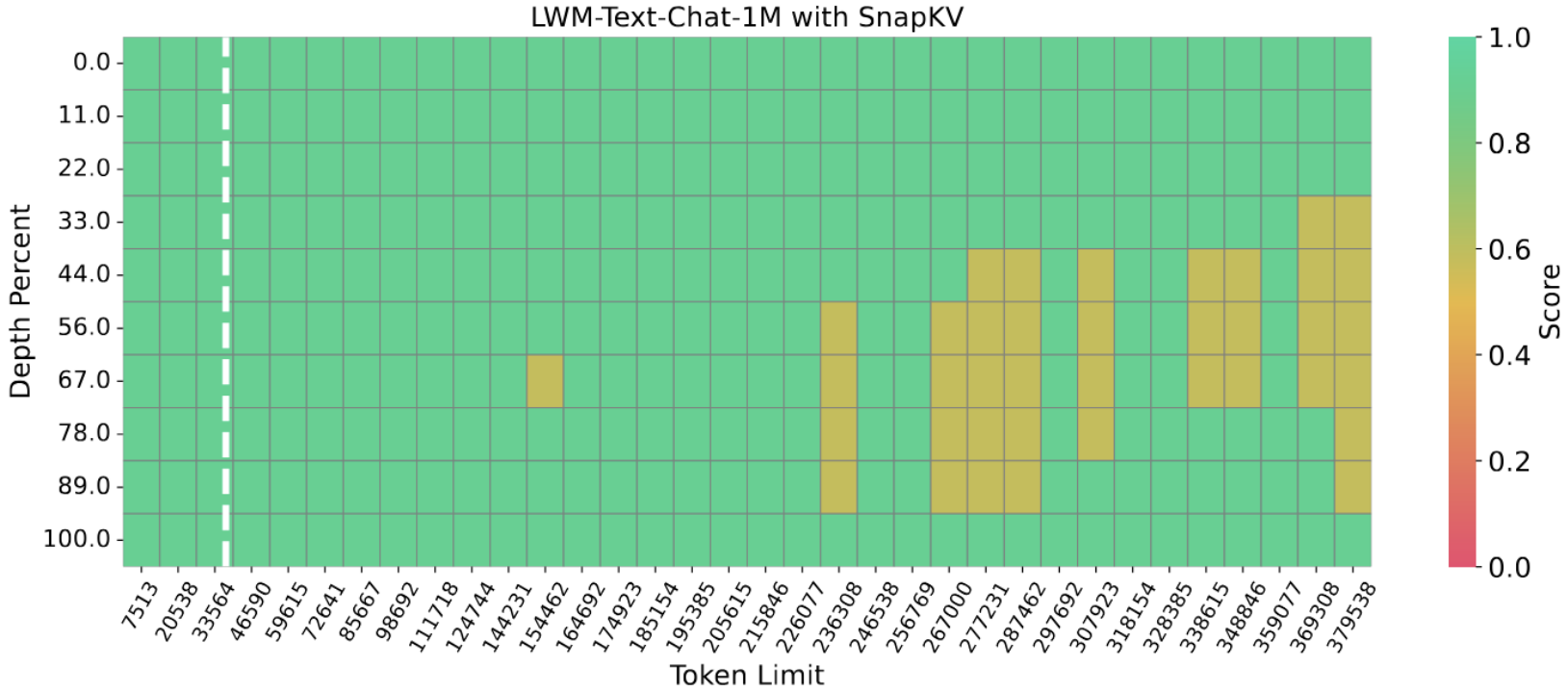


LongEval-Lines Retrieval Accuracy Comparison (**without** vs. **with Pooling**)
(Retrieve target values from long contexts)

Experiments

Needle-in-a-Haystack(NIAH)

- **Long-Context Performance Evaluation(Needle-in-a-Haystack)**
  - manage small details on extremely long input contexts with a 380x compression ratio



Needle-In-A-Haystack Retrieval Accuracy
(Retrieve pass key among long noise context)

Experiments

LongBench

- **Long-Context Performance Evaluation(LongBench)**
  - SnapKV can grasp the key information in the long context and give comprehensive summaries with details
  - the effectiveness of SnapKV in compressing the prompt KV cache

		Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning			Synthetic		Code	
	LLMs *	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	Musique	GovReport	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P
LWMChat	All KV	**18.18**	**25.56**	40.94	24.57	19.39	10.49	**27.97**	24.9	**24.81**	71.0	60.9	39.73	3.17	3.5	44.4	43.82
	SnapKV: 1024	18.02	23.73	40.25	24.61	**19.84**	10.77	19.79	24.44	23.53	70.0	**61.42**	39.64	1.67	3.0	43.34	44.0
	SnapKV: 2048	17.92	25.03	**41.38**	24.49	19.38	**11.34**	21.6	24.22	24.36	70.0	61.11	39.91	2.17	4.0	44.46	**44.92**
	SnapKV: 4096	17.92	25.47	40.76	**24.92**	19.53	11.27	25.34	**25.42**	24.58	70.5	61.08	39.62	**3.17**	**4.0**	**44.49**	44.08
	H2O: 4096	13.17	24.82	20.01	16.86	9.74	7.2	25.77	23.26	23.83	**71.0**	61.06	**40.33**	0.0	0.0	41.52	40.97
LongChat	All KV	**20.88**	**29.36**	**43.2**	33.05	24.58	**14.66**	**30.89**	22.76	**26.61**	**66.5**	**83.99**	**40.83**	0.0	30.5	54.89	**59.05**
	SnapKV: 1024	19.32	26.6	37.93	34.15	23.34	12.71	23.45	21.81	24.93	65.0	80.88	38.19	0.0	31.0	53.63	57.62
	SnapKV: 2048	19.28	28.81	40.26	**35.31**	23.75	13.44	26.3	22.29	25.73	66.0	79.93	39.59	0.0	**31.0**	**56.05**	58.61
	SnapKV: 4096	20.68	29.34	42.21	33.95	**24.88**	14.15	28.55	**23.11**	26.45	66.0	**81.25**	**40.52**	0.0	29.5	54.79	58.81
	H2O: 4096	19.31	28.3	37.75	30.51	23.06	11.76	27.55	21.37	26.49	66.0	75.8	39.92	0.0	25.5	53.56	55.53
Mistral	All KV	**26.82**	33.06	49.28	**42.77**	27.33	19.27	**32.85**	24.25	27.06	71.0	86.23	42.98	2.75	86.98	55.51	**52.88**
	SnapKV: 1024	25.54	29.51	49.25	40.94	25.7	**19.42**	25.89	23.82	26.11	69.5	**86.48**	42.06	2.98	**88.56**	55.65	51.87
	SnapKV: 2048	25.89	32.47	48.6	41.71	27.31	18.69	28.81	**24.5**	26.6	70.0	86.27	42.47	3.09	87.43	**55.93**	52.01
	SnapKV: 4096	26.41	**33.36**	**49.81**	42.32	**27.93**	18.76	30.74	24.19	**27.08**	**71.0**	86.25	**43.01**	2.73	86.18	55.62	52.65
	H2O: 4096	22.61	29.06	47.22	36.54	20.6	16.25	30.0	23.8	26.75	70.5	86.16	42.97	**3.46**	86.38	53.72	51.1
Mixtral	All KV	26.81	**37.06**	51.55	47.77	32.46	**26.59**	**34.25**	**26.05**	27.91	76.0	90.57	46.98	5.5	100.0	**69.07**	69.65
	SnapKV: 1024	26.01	34.65	51.58	**48.23**	32.67	25.92	27.77	25.0	27.25	74.5	90.42	46.48	5.5	99.5	69.02	68.98
	SnapKV: 2048	**27.12**	36.9	51.91	47.46	33.23	26.27	30.19	25.84	27.8	**76.0**	90.24	46.31	5.5	100.0	68.72	**70.01**
	SnapKV: 4096	26.46	37.03	**52.62**	47.71	**33.35**	26.45	32.64	25.87	**27.94**	75.5	**90.71**	**47.14**	5.5	**100.0**	68.81	69.56
	H2O: 4096	20.45	32.09	48.02	34.76	25.69	16.5	29.76	23.53	26.84	74.5	90.24	47.1	**7.06**	99.42	64.91	63.52

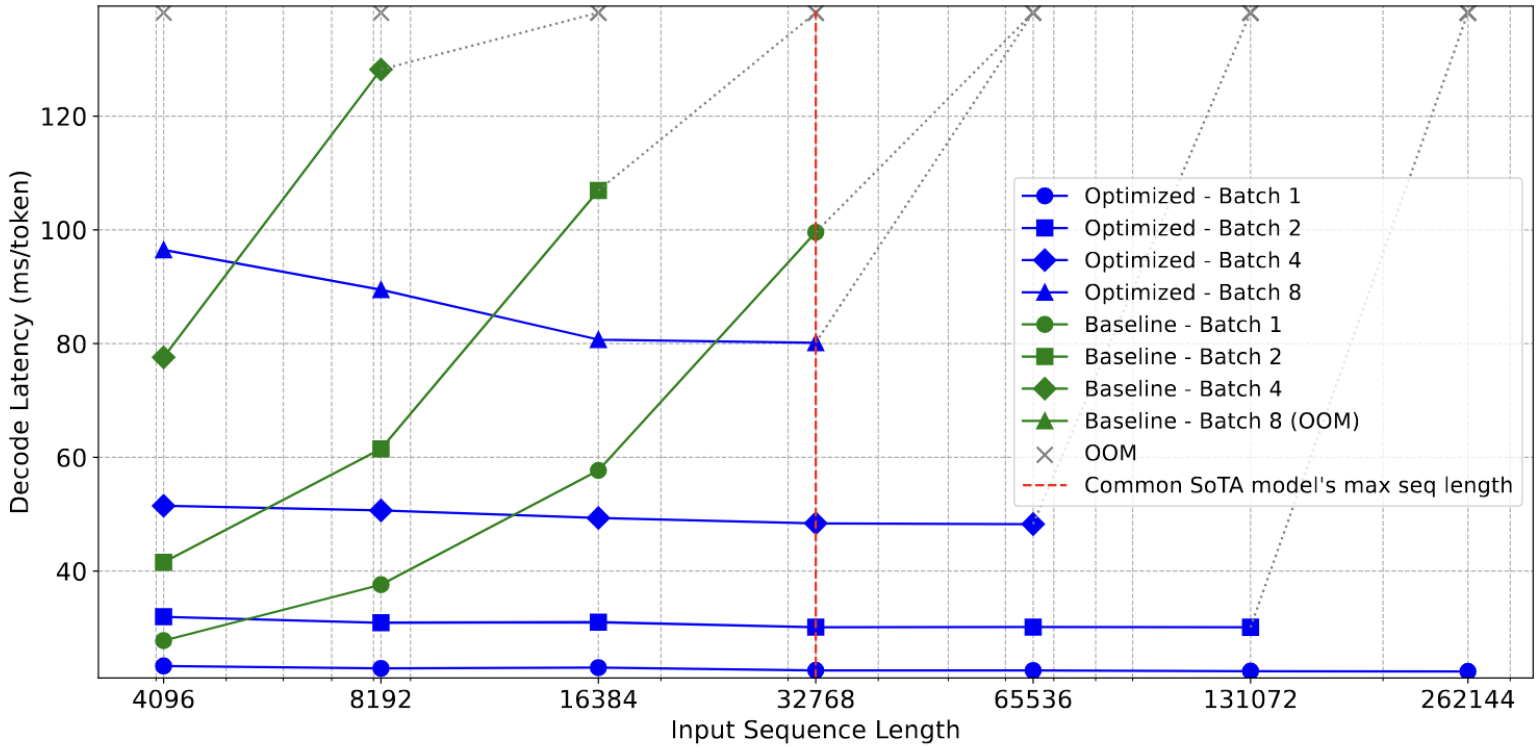
LongBench Score Comparison with Various LLMs

(LWM-Text-Chat-1M with 1 million context length, LongChat-7b-v1.5-32k, Mistral-7B-Instruct-v0.2, Mixtral-8x7B-Instruct-v0.1 with 32k context length)

Experiments

Decoding Speed and Memory Bound

- **Decoding Latency Evaluation**
  - **Speed**: baseline escalates **linearly** <->SnapKV maintains **constant** decoding speed(**3.6x** speedup)
  - **Memory**: SnapKV can decode significantly **longer sequences**(**16k**-> **131k** input tokens, **8.2x** improvement)



Decoding latency comparison of **baseline** and **SnapKV** optimized solutions of various **batch sizes**

Follow-up Work

Beyond SnapKV

- PyramidKV / Ada-KV: split the one-shot budget per layer and per head
- Scissorhands / TOVA: swap the decode-time scoring signal(accumulated attention score)
- KIVI / KVQuant / TurboQuant: shrink bits per entry instead of dropping tokens

Ada-KV

Beyond SnapKV

- PyramidKV / Ada-KV: split the one-shot budget per layer and per head
- Scissorhands / TOVA: swap the decode-time scoring signal(accumulated attention score)
- KIVI / KVQuant / TurboQuant: shrink bits per entry instead of dropping tokens