APE

Faster and Longer Context-Augmented Generation via Adaptive Parallel Encoding

Xinyu Yang @ CMU
Catalyst | 2025-02-20

Paper link
Find me on X @Xinyu2ML

Sequence models as a universal abstraction

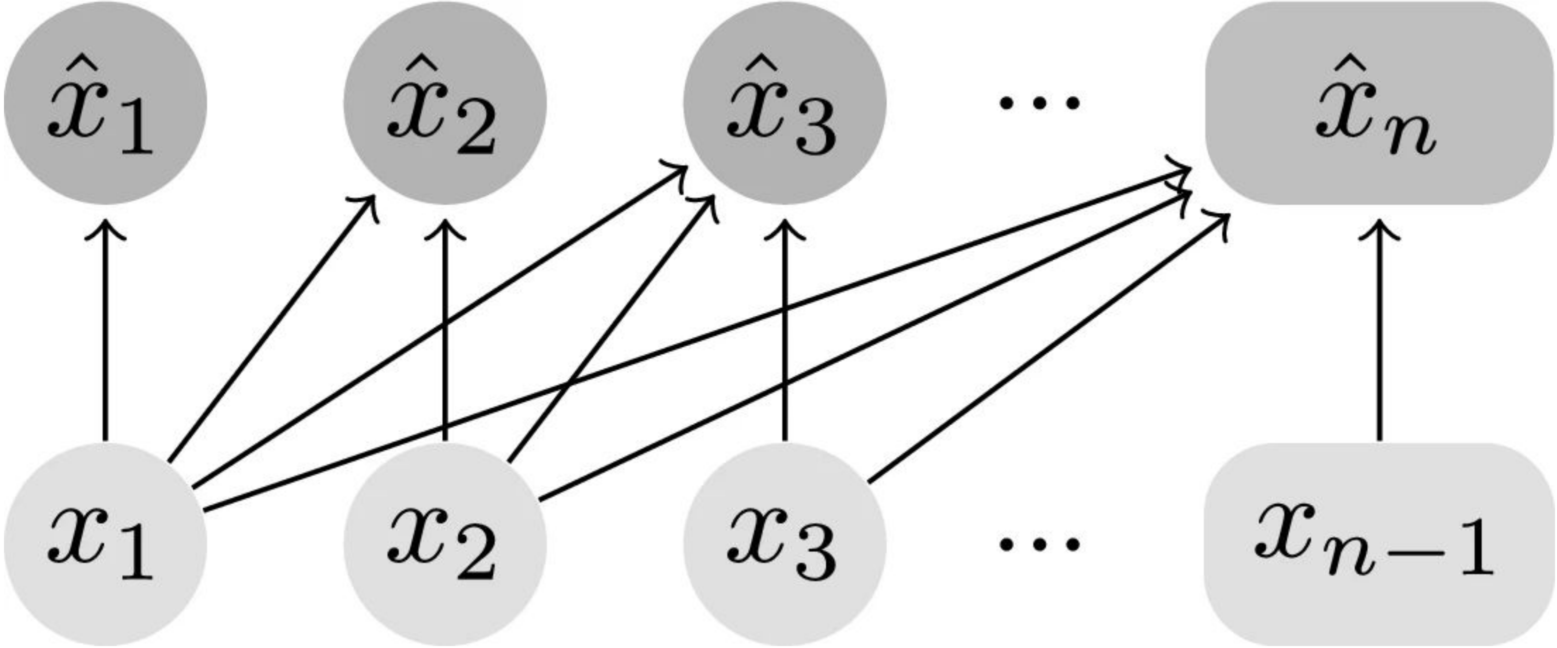
Sequences can represent various information:

Bit String | Text | Audio | Video | DNA

Autoregressive generative models (i.e. Transformers) are sequence models:

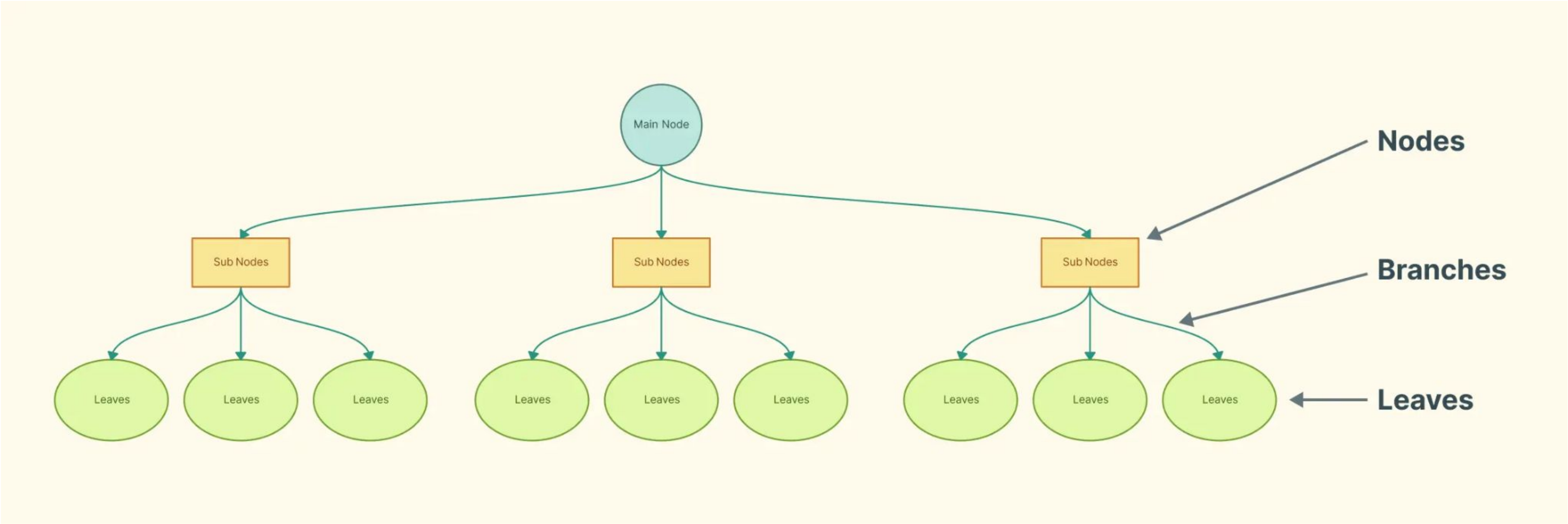
However, sequence models limit parallelism ability.

Each token is related to all past tokens, requiring sequential encoding:



Do we need to model everything into one sequence?

Information naturally possesses some structure that can be encoded in parallel:

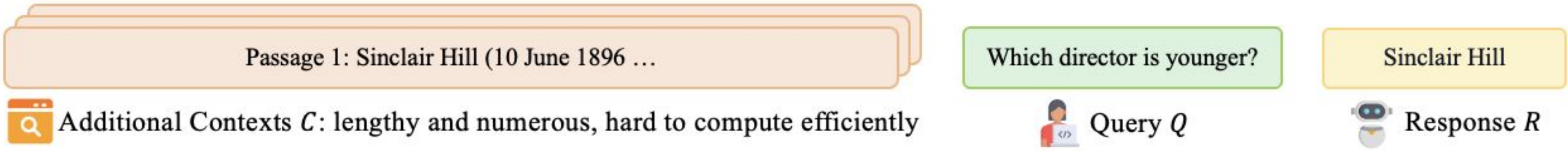


Therefore, are there practical scenarios where such a structure can be utilized?

We focus on context-augmented generation.

Context-augmented generation, including RAG and ICL, is a common case that combines LLMs and external databases, where the information includes:

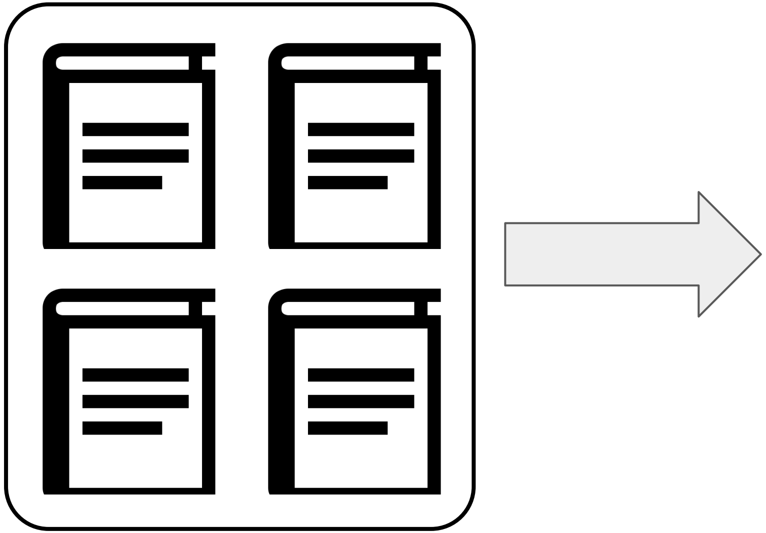
- Contexts: Multiple independent texts retrieved from external sources
- Query: Questions inputted by the user.
- Response: Answers generated by the LLMs



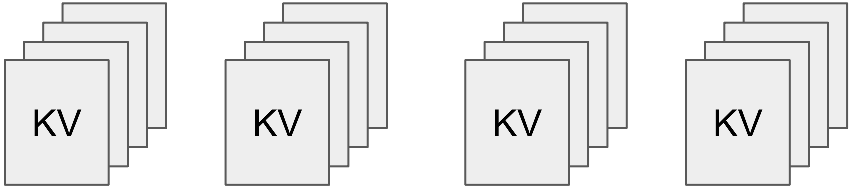
Obviously, different contexts are independent and can be encoded in parallel.

Parallel encoding offers many benefits.

In addition to speedup, parallel encoding enables combination among contexts.



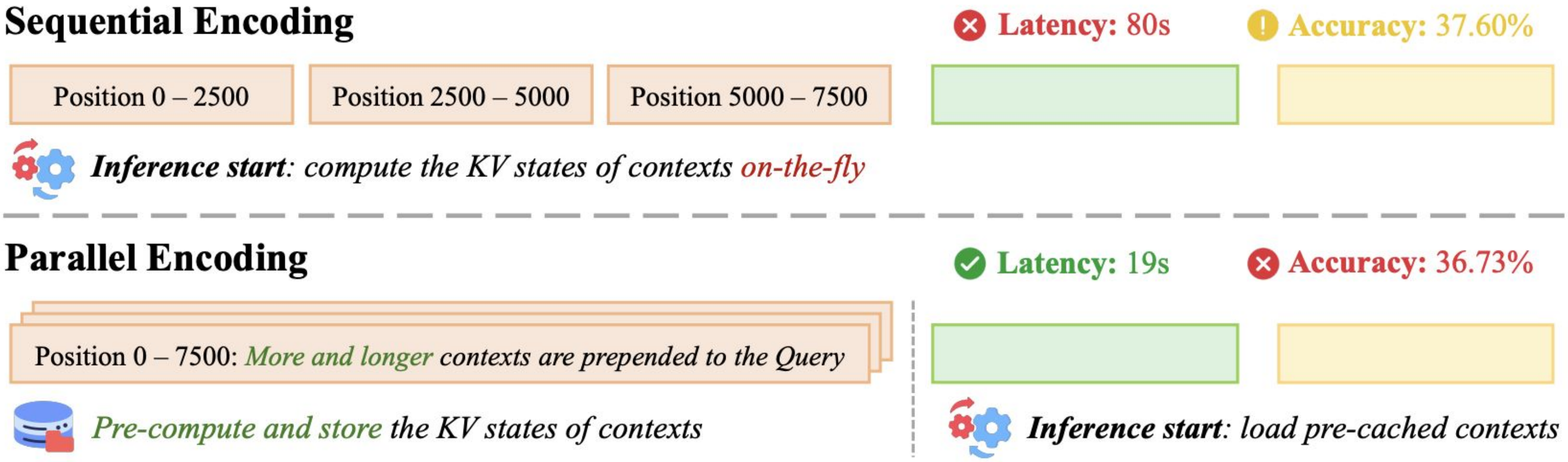
Pre-caching texts for Fast Inference



Re-using positions for long Context

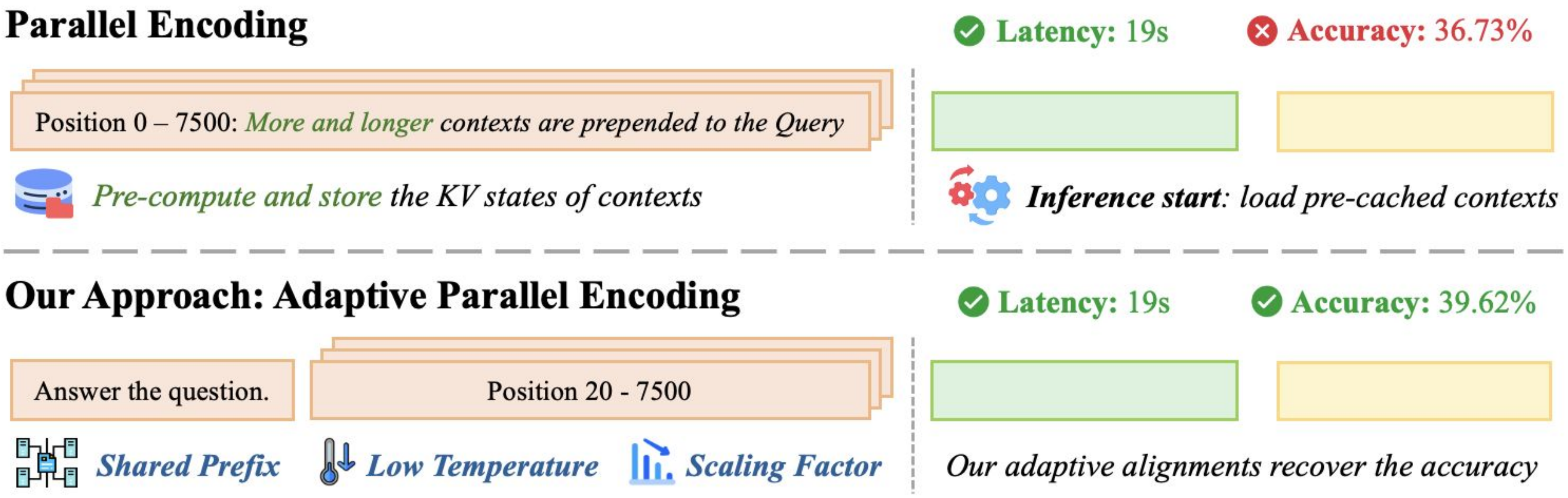
However, parallel encoding is inaccurate.

Despite these benefits, it remains inaccurate, as there is no guarantee that independent KV states from different contexts can be compared or combined.



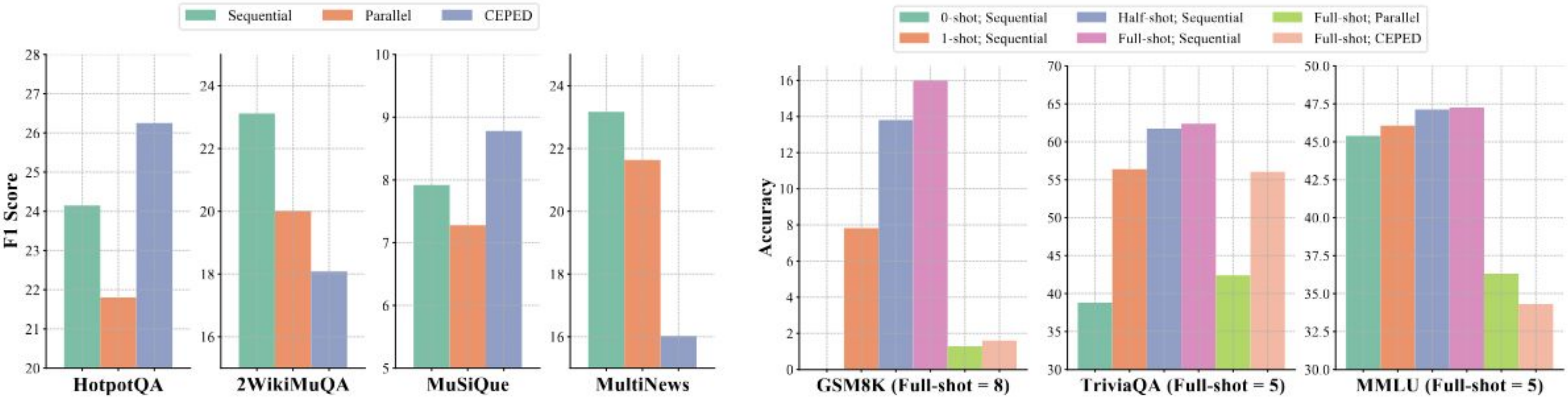
Adaptive Parallel Encoding recovers the drop.

To address these challenges, we propose APE that aligns the distribution of parallel encoding with sequence encoding with three inference-time steps.



Q1: Why we choose inference-time steps?

Some previous work try to train something to improve parallel encoding, however, they suffers from performance degradation on complex reasoning tasks.

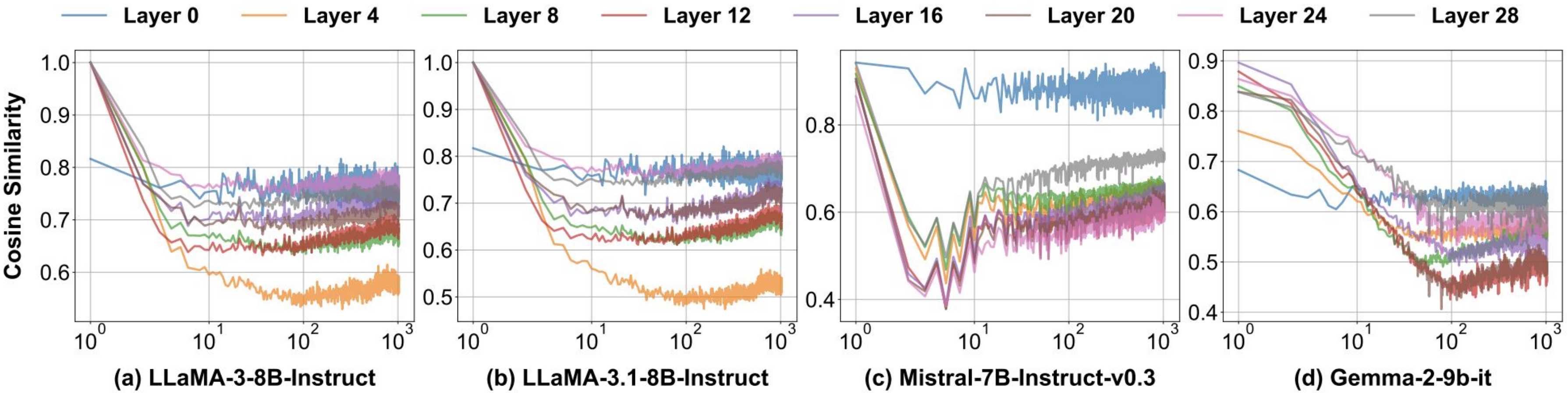


(a) Retrieval-augmented Generation

(b) In-context Learning

Q2: Why inference-time steps are enough?

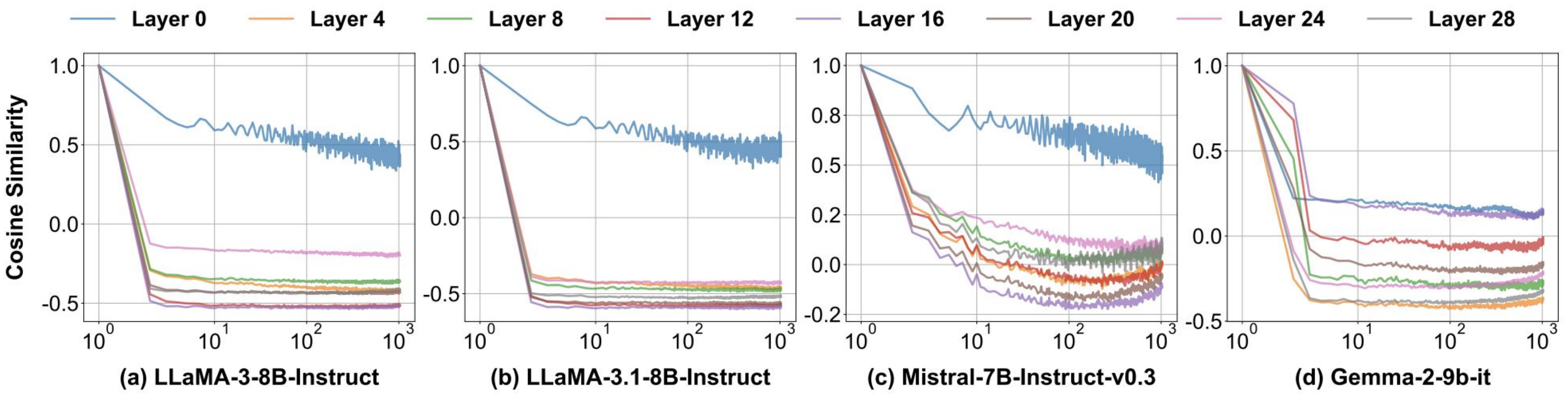
Double check the performance of parallel encoding, it decreases but does not drop to zero. ⇨ Something connects the KV states from different contexts.



The secret lies in the “attention sink”, which exhibits similar direction for different inputs.

But why the later tokens also can be compared?

Having similar direction for the initial tokens doesn’t mean later tokens also has similar directions. However, they may have some connection with the initial token.



Key states from different contexts are similar.

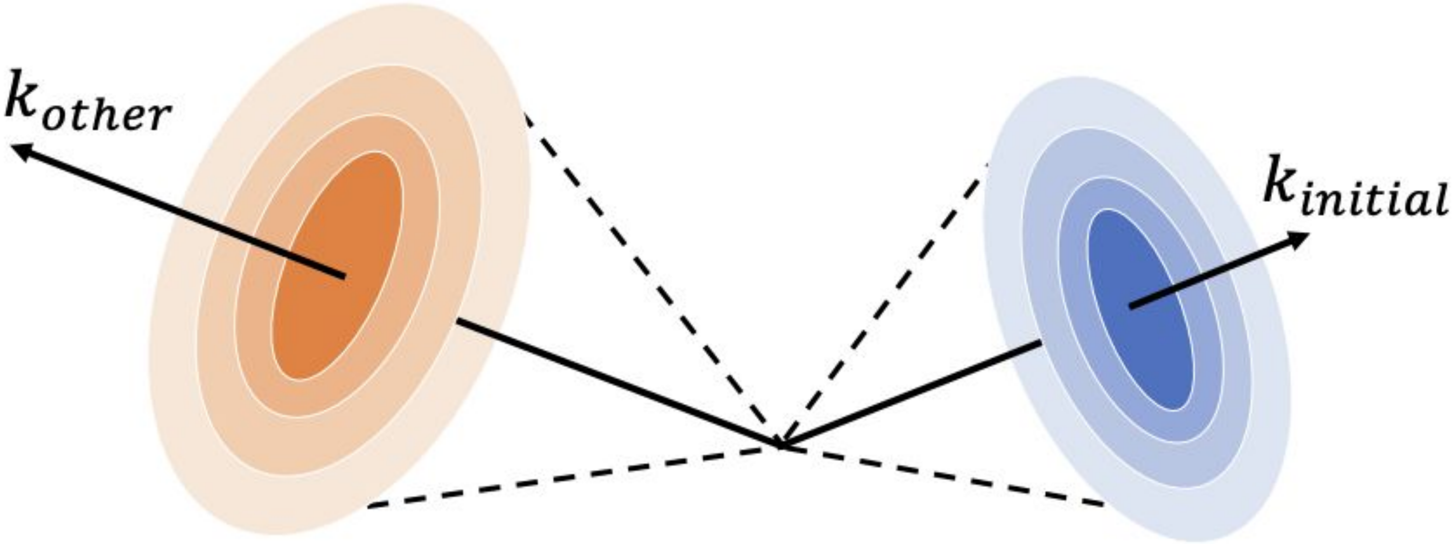
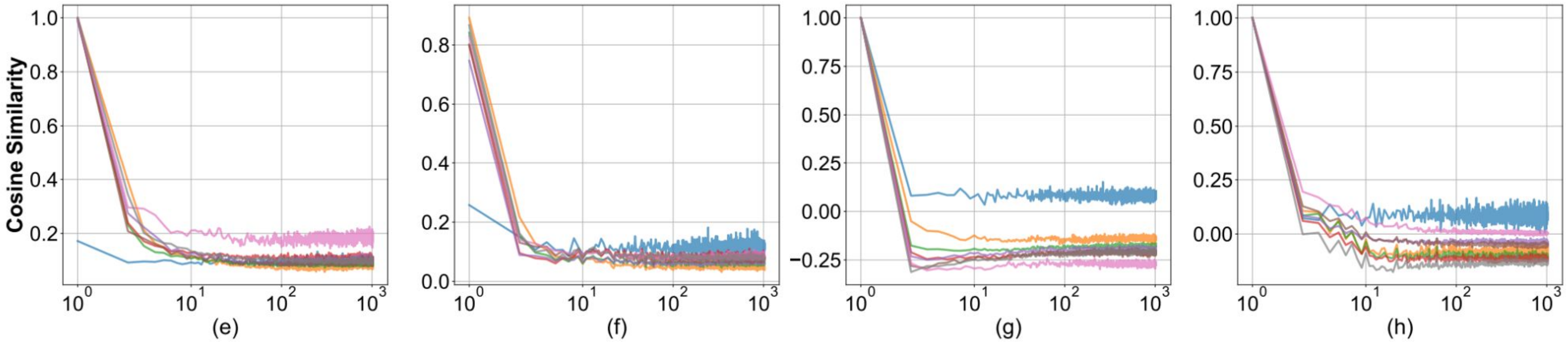


Figure 5 Geometry of Key States.

Next, we move to the value states.

Similarly, the direction of value states is similar across contexts, as they are decided by the initial states, which exhibit similar directions across examples.



Similarity between tokens from different samples in each positions

Similarity between the initial token and tokens in different positions

Value states from different contexts can be combined.

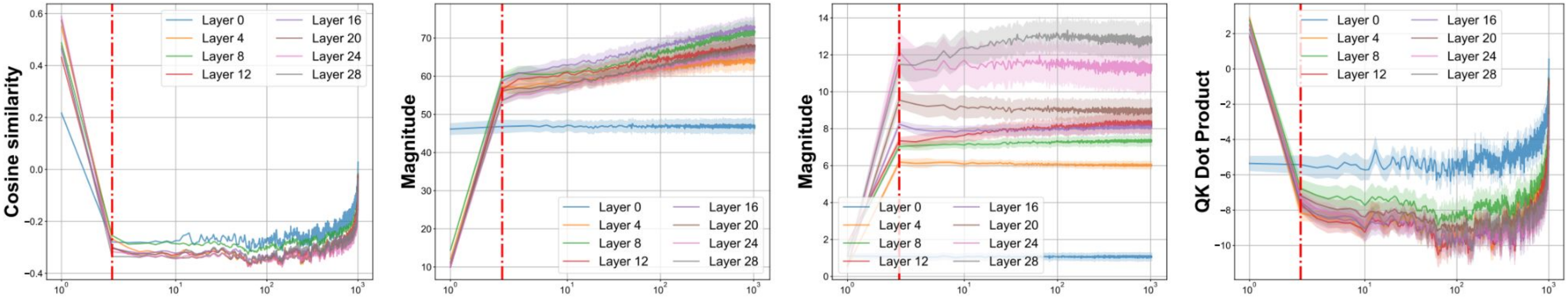
Due to the normalization in Softmax operator, value states naturally share similar magnitudes. Therefore, value states from different contexts can be combined.

In a standard Softmax attention, we attend the query to all past KV states using the following formula:

$$O = \text{Softmax}(\frac{QK^T}{\sqrt{d}})V \quad Q \in \mathbb{R}^{n \times d} \quad K, V \in \mathbb{R}^{m \times d}, \tag{2}$$

So, what is the source of the performance drop?

- The initial positions result in an abnormal region in the whole context.
- The dot products between the query state and all past key states encounter a notable increase when the states are positioned close to each other.



(a) Query-Key Similarity (b) Key Magnitude (c) Value Magnitude (d) Query-Key Product

Step 1: Prepending Shared Prefix.

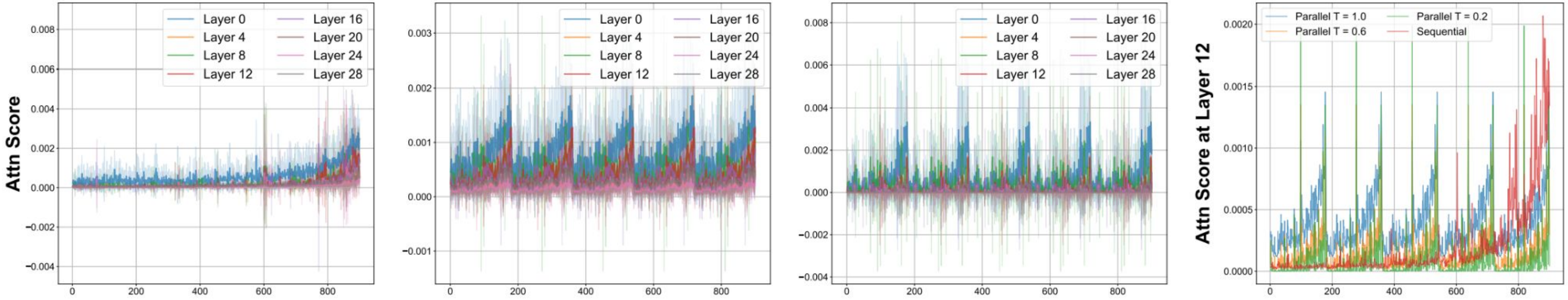
The first step is very direct: Since the first few tokens exhibit abnormal directions and magnitudes, we prepend a shared prefix to avoid duplication of these tokens.

- If the model has a system prompt, we directly use this system prompt as a shared prefix for all contexts.
- If the model does not have a system prompt, we additionally add a few “\n” before all contexts.

Both strategies can work for different context-augmented generation settings, as no task-specific information are provided in the shared prefix.

Step 2: Adjusting Attention Temperature.

To mitigate the impact of repeating neighboring tokens, we adjust the attention temperature to a value smaller than one to sharpen the attention distribution.



(a) Sequential (b) Parallel (T = 1.0) (c) Parallel (T = 0.2) (d) Parallel vs. Sequential

Step 3: Adding Scaling Factor.

However, Step 2 will also change the whole attention allocated to all context tokens, as shown by the LogSumExp value with different T in the figure.

The magnitude of this value increases when T decreases for different layers. To compensate for these changes, we will add a scaling factor smaller than one to reduce this absolute value.

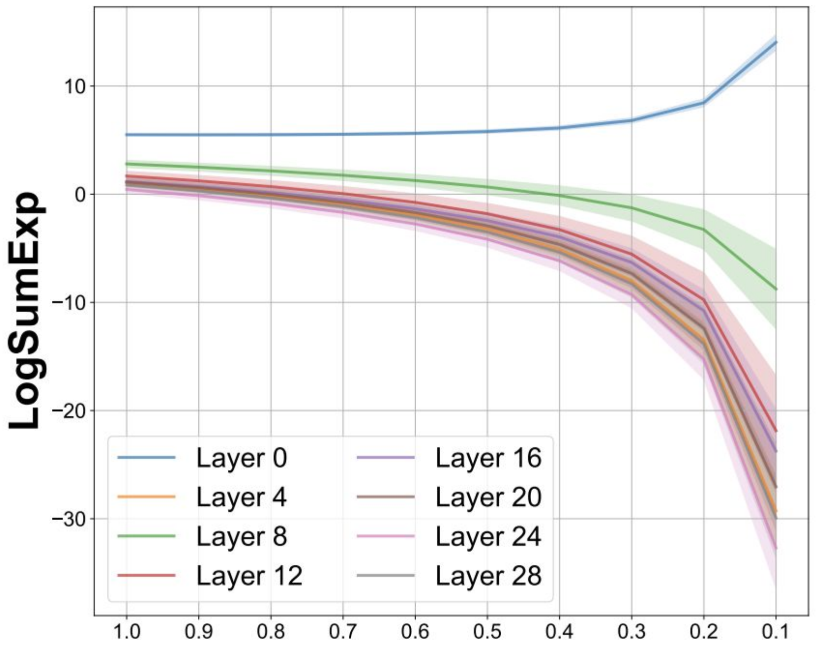


Figure 8 Parallel w/ Different T.

Efficient Implementation

These new hyperparameters make our APE incompatible with flash attention. To combine the computation for context and non-context tokens, we choose to employ flash attention twice—once for each part—and then merge the results.

```
def ape_attention(query, key, value, temperature, scale):
    # split key and value states into context and non-context parts
    key_context, key_other = key
    value_context, value_other = value
    attn_output_context, lse_context = flash_attn(query, key, value, temperature = temperature)
    attn_output_other, lse_other = flash_attn(query, key, value)
    lse_context = lse_context*(scale)
    attn_weights = [lse_context, lse_other]
    attn_weights = Softmax(attn_weights)
    value_states = [attn_output_context, attn_output_other]
    attn_output = attn_weights @ value_states
```

Performance Analysis: RAG

APE maintains 98% of the sequential encoding performance on ChatRAG-Bench.

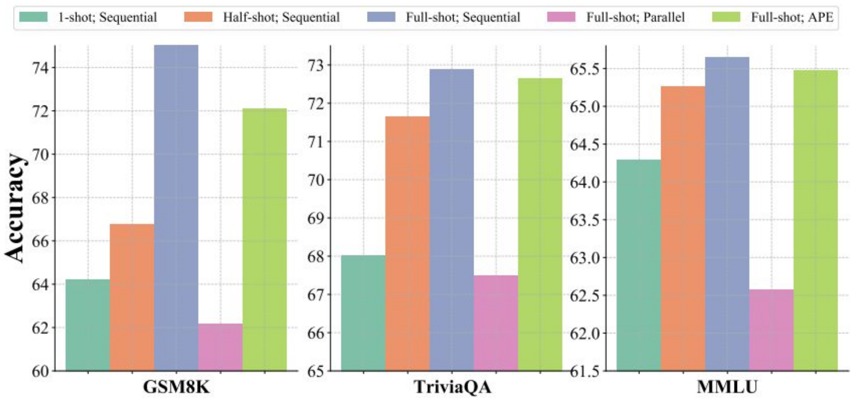
Method	INSCIT	Doc2Dial	TopicCQA	Qrecc	QuAC	Average
Contriever, Sequential	19.97	23.85	30.49	46.75	26.57	29.53
Contriever, APE	19.88	23.28	28.84	46.28	26.80	29.02
Δ	-0.09	-0.57	-1.65	-0.47	+0.23	-0.51
GTE-base, Sequential	21.58	32.35	33.41	46.54	30.69	32.91
GTE-base, APE	20.85	30.99	31.92	45.83	30.35	31.99
Δ	-0.73	-1.36	-1.49	-0.71	-0.34	-0.92
Dragon-multiturn, Sequential	25.42	36.27	36.10	49.01	35.12	36.38
Dragon-multiturn, APE	23.84	34.93	33.80	48.70	34.92	35.24
Δ	-1.58	-1.34	-2.30	-0.31	-0.20	-1.14
All texts, APE	**27.22**	36.13	35.72	**49.15**	**35.70**	**36.78**

By retrieving more texts, APE improves performance on LongBench.

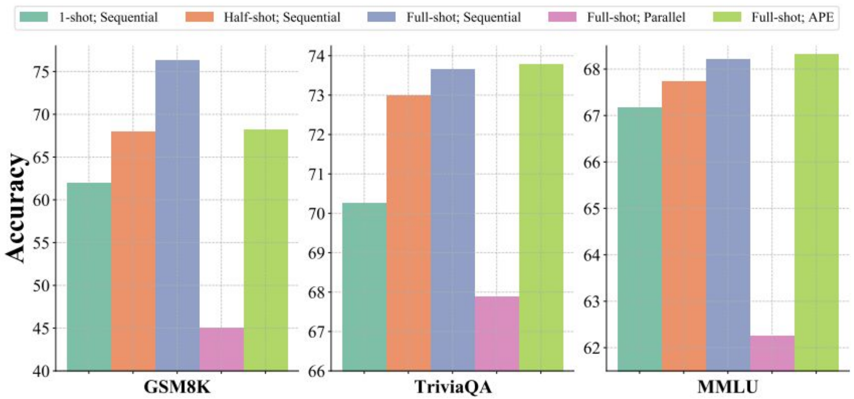
Model	MuSiQue	Qasper	2WikiMQA	DuRead	HotpotQA	NarratQA	MFQA_zh	MFQA_en	Avg.
LLaMA-3-8B-Instruct	20.70	41.05	30.02	9.55	45.90	20.98	**58.54**	45.04	33.97
C200×20, Sequential	**27.93**	**42.71**	38.35	12.65	49.60	22.78	57.82	**48.94**	37.60
C4000×20, PCW	18.82	42.59	40.99	21.57	47.09	23.29	54.40	45.05	36.73
C4000×20, APE	26.19	42.32	**44.43**	**23.13**	**49.71**	**30.71**	55.03	45.41	**39.62**
Mistral-7B-Instruct-v0.3	10.05	31.08	22.12	17.68	32.09	19.68	32.03	40.38	25.64
C200×20, Sequential	11.58	21.98	24.44	20.80	32.79	16.06	34.43	38.40	25.06
C4000×20, PCW	17.58	35.57	32.97	18.70	37.05	14.10	34.69	40.14	28.85
C4000×20, APE	**20.30**	**36.81**	**34.37**	**21.89**	**42.33**	**20.49**	**40.20**	**44.03**	**32.55**
Gemma-2-9b-it	22.57	39.99	48.06	27.40	47.49	23.11	50.81	45.35	38.10
C200×10, Sequential	30.69	42.86	**53.55**	28.04	52.05	24.45	50.25	48.34	41.28
C2000×20, PCW	26.27	46.69	47.59	23.43	48.95	27.11	**56.69**	49.81	40.82
C2000×20, APE	**33.38**	**47.72**	49.49	**28.43**	**56.62**	**30.41**	56.52	**50.84**	**44.18**
LLaMA-3.1-8B-Instruct	22.18	**46.81**	40.58	**34.61**	43.97	23.08	61.60	51.89	38.98
128K, Sequential	28.35	47.20	40.81	33.34	53.46	30.57	61.97	53.25	42.24
C200×20, Sequential	**30.62**	42.33	44.39	33.51	49.97	23.87	56.87	**55.14**	40.22
C4000×20, PCW	21.23	41.52	44.87	31.11	49.47	19.98	60.90	51.19	38.44
C4000×20, APE	26.88	43.03	**50.11**	32.10	**55.41**	**30.50**	**62.02**	52.51	**42.86**

Performance Analysis: ICL

APE maintains 93% of the sequential encoding performance on ICL tasks. It is the only parallel encoding method works for complex reasoning ICL scenarios.



(a) LLAMA-3-8B-INSTRUCT



(b) LLAMA-3.1-8B-INSTRUCT

Performance Analysis: Many-shot CAG

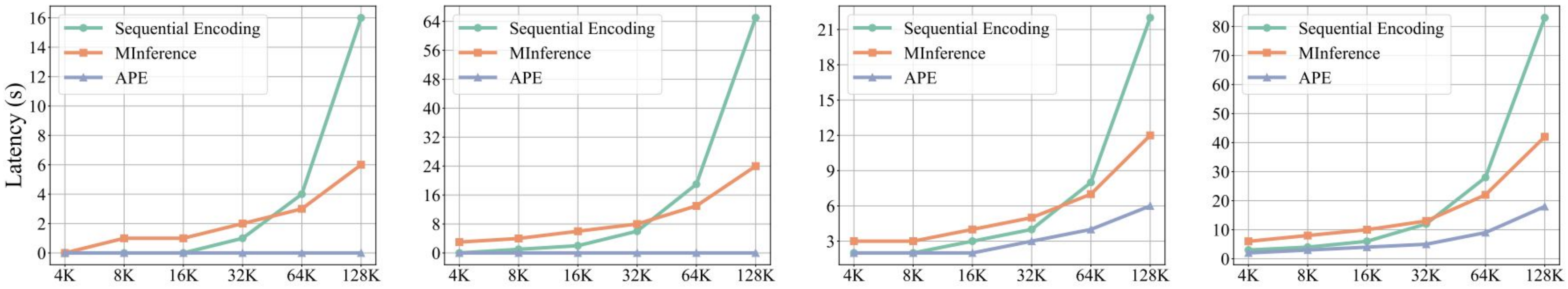
APE successfully handles hundreds of contexts in parallel without degradation.

- Parallel encoding leads to performance drop.
- Putting all contexts into nearby positions improve performance.

	Retrieval-augmented Generation				In-context Learning			
Method	ArguAna	FEVER	NQ	SciFact	Date	Salient	Tracking7	Web
Sequential, Zero-shot	11.15	7.78	17.78	7.74	20.00	8.89	1.12	8.89
Sequential, Few-shot	11.20	9.78	17.81	9.49	36.64	38.89	6.67	38.89
Sequential, Half-shot	15.34	13.12	19.64	16.12	45.55	42.22	8.89	55.56
Sequential, Full-shot	12.84	14.19	**24.54**	**16.88**	**46.67**	**46.67**	**8.89**	**58.89**
APE, Full-shot	**16.32**	**14.70**	21.91	15.72	43.33	45.55	**8.89**	**58.89**

Efficiency Analysis

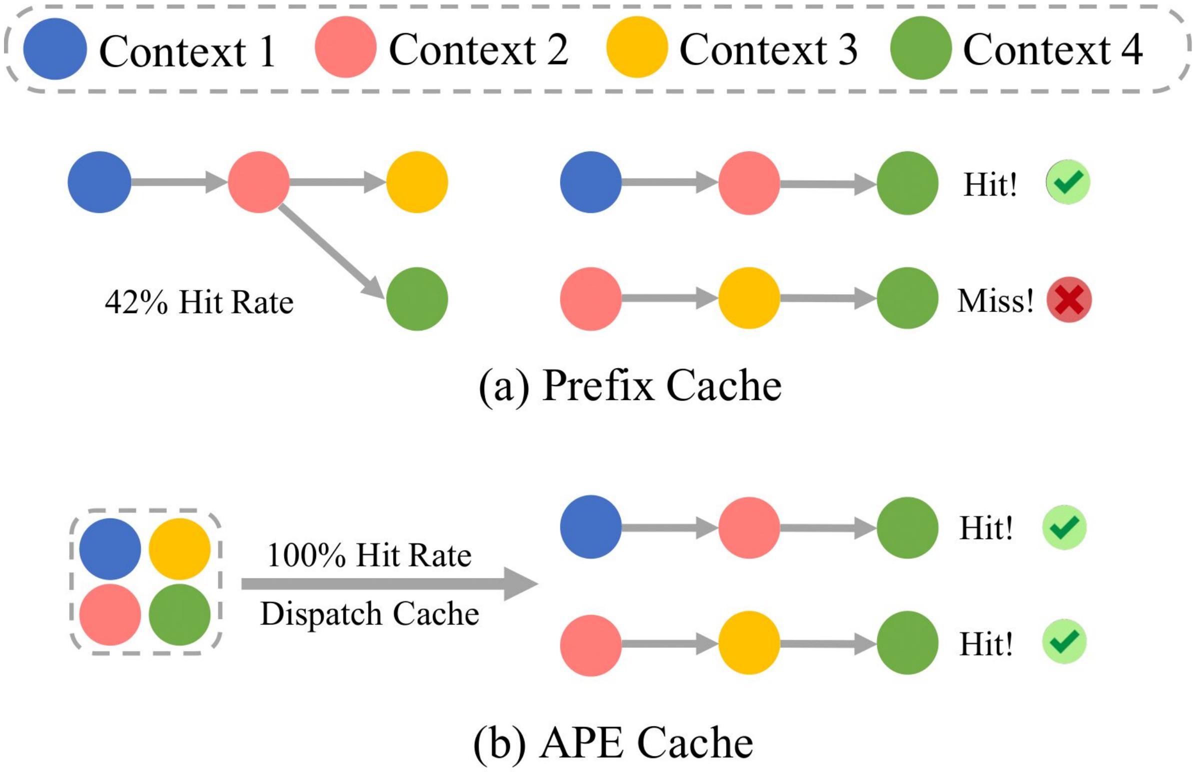
APE reduces the prefilling time to nearly zero. Therefore, it achieve an end-to-end 4.5× speedup when prefilling 128K-length context and generating 256 tokens.



(a) Prefill Time (bsz=1) (b) Prefill Time (bsz=4) (c) Total Time (bsz=1) (d) Total Time (bsz=4)

APE Cache Design

Moreover, APE unlock new potential for real-world CAG serving systems. By storing all external contexts into KV states, APE maintains a 100% cache hit rate for different user queries and corresponding retrieved contexts. In contrast, prefix cache can only have a 42% hit rate with only 4 contexts.



Future Directions

APE represents an initial exploration into enhancing parallelism for autoregressive generative models, which only leverages the inherent parallelizable structure of the data. Therefore, the future directions of APE should include:

- Building real-world APE cache system to serving RAG scenarios.
- Extending APE to multi-modal CAG scenarios with more information.
- Encouraging parallelism during generation. For example, when we do repeated sampling or subtask solving, different trajectories can be computed in a parallel way, while the merging mechanism in APE/Parallel Encoding can be used to combine these information in a smart way.

If you enjoyed this talk, you can find me on the Catalyst Slack channel!