Efficient Streaming Language Models with Attention Sinks

Paper written by:

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, Mike Lewis, (2023)

(https://arxiv.org/abs/2309.17453)

The problem they're trying to solve

- Transformer-based LLMs only work with finite context length
- Output quality drops significantly beyond training context length
  - (This is a more difficult problem)
- Required computation also grows with increasing input length
  - Memory for KV cache also grows rapidly

In this paper

- Technique for "streaming" input to LLM of possibly infinite length
  - LLM doesn't fail catastrophically while being computationally efficient

Recap of the attention layer

- Input is a sequence $x_1, x_2, \ldots, x_T$
  - In traditional transformer, x_i includes positional encoding: $x_i = t_i + f(i)$
- For each item x_i, query, key, and value q_i, k_i, v_i are generated
- Output given by

$$a_i = softmax_{j:j<i}(q_i \cdot k_j)v_j$$

- If KV values are cached, needs $O(T)$ compute for each i
  - Also fails catastrophically if T is more than training context length

Sliding window

Simple solution for long inputs:

- Slide the context window so that you only consider last L tokens

$$x_1, x_2, \ldots, \boxed{x_{T-L+1}, x_{T-L+2}, \ldots, x_{T-1}, x_T}$$

Use the last L tokens as input to LLM

The downside:

- Need to recompute all the KV values due to positional encoding
  - Needs $O(L^2)$ compute
- Very poor performance if we reuse the KV values

Relative positional encoding

- Many models like Llama use relative positional encoding
  - Encoding is not directly integrated into the inputs x_i

$$\ldots, x_{T-4}, x_{T-3}, x_{T-2}, x_{T-1}, x_T$$

Positions: -4 -3 -2 -1 0

For next token:

$$\ldots, x_{T-4}, x_{T-3}, x_{T-2}, x_{T-1}, x_T, x_{T+1}$$

Positions: -5 -4 -3 -2 -1 0

KV values cached before positional encoding

- Apply encoding while running attention layer
- Llama uses RoPE
  - https://arxiv.org/abs/2104.09864

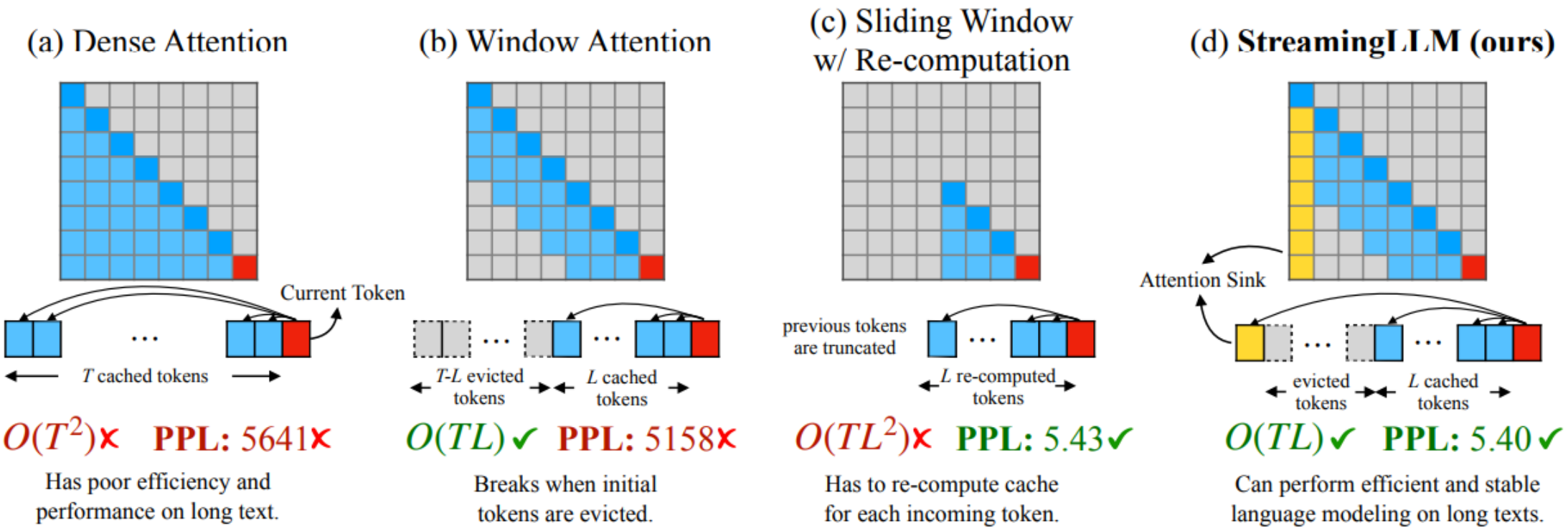
Attention sink

- Current paper manages to reuse KV cache with sliding window
  - Without performance loss due to mismatched positional encoding
  - Works for LLMs with relative positional encoding

Key observation

- Most of the attention score is held in the first few “sink” tokens
  - Preserve these tokens and reuse the KV cache for last L tokens
  - $O(L)$ computation without much loss in performance

Pictorial summary of attention sink



Empirical observation of attention sink

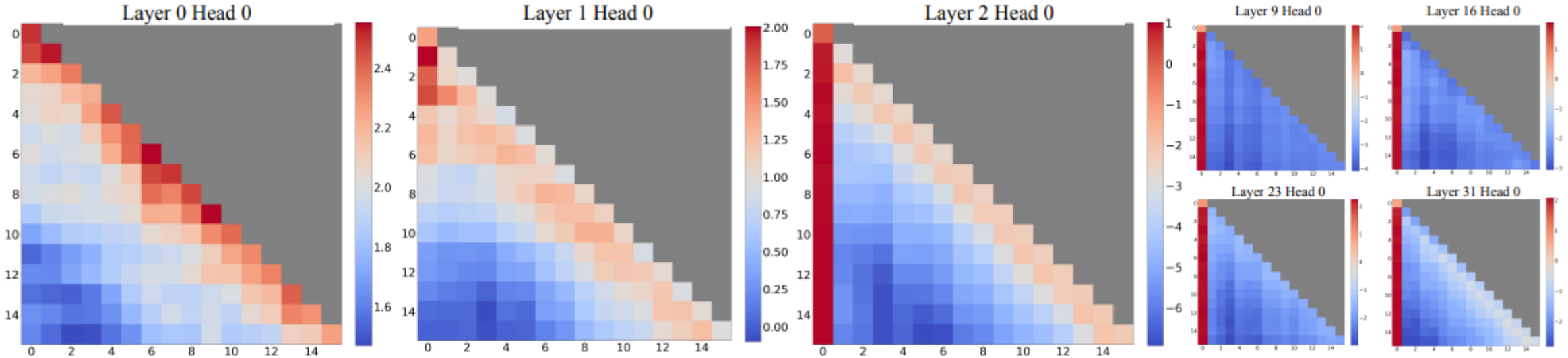
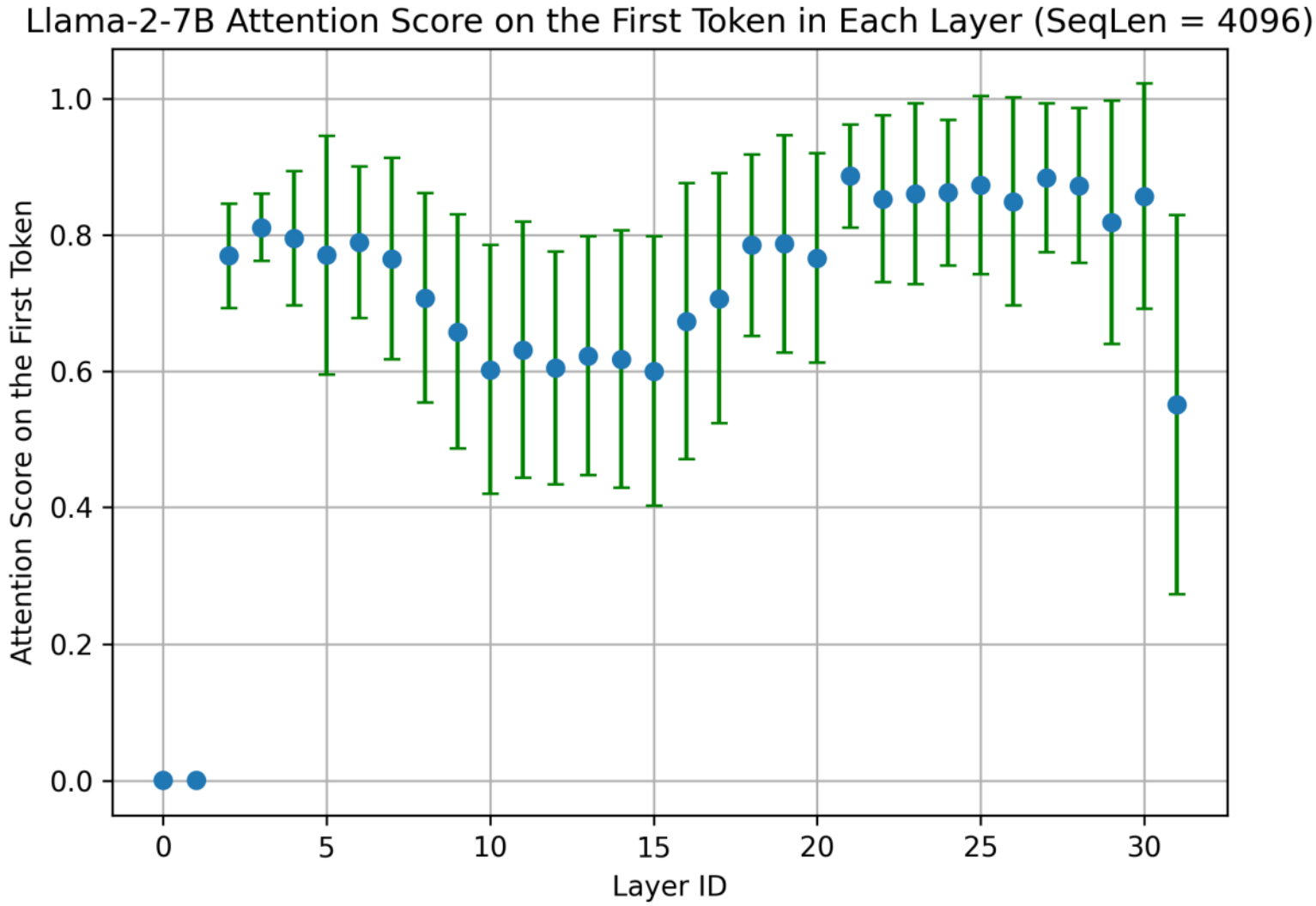


Figure 2: Visualization of the *average* attention logits in Llama-2-7B over 256 sentences, each with a length of 16. Observations include: (1) The attention maps in the first two layers (layers 0 and 1) exhibit the "local" pattern, with recent tokens receiving more attention. (2) Beyond the bottom two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

Attention scores on longer sequences



Attention sink details

- The first tokens are not semantically important
- Four tokens enough on pre-trained LLMs

Table 2: Effects of reintroduced initial token numbers on StreamingLLM. (1) Window attention (0+y) has a drastic increase in perplexity. (2) Introducing one or two initial tokens usually doesn't suffice to fully restore model perplexity, indicating that the model doesn't solely use the first token as the attention sink. (3) Introducing four initial tokens generally suffices; further additions have diminishing returns. Cache config x+y denotes adding x initial tokens to y recent tokens. Perplexities are evaluated on 400K tokens in the concatenated PG19 test set.

Cache Config	0+2048	1+2047	2+2046	4+2044	8+2040
Falcon-7B	17.90	12.12	12.12	12.12	12.12
MPT-7B	460.29	14.99	15.00	14.99	14.98
Pythia-12B	21.62	11.95	12.09	12.09	12.02
Cache Config	**0+4096**	**1+4095**	**2+4094**	**4+4092**	**8+4088**
Llama-2-7B	3359.95	11.88	10.51	9.59	9.54

Table 1: Window attention has poor performance on long text. The perplexity is restored when we reintroduce the initial four tokens alongside the recent 1020 tokens (4+1020). Substituting the original four initial tokens with linebreak tokens "\n" (4"\n"+1020) achieves comparable perplexity restoration. Cache config x+y denotes adding x initial tokens with y recent tokens. Perplexities are measured on the first book (65K tokens) in the PG19 test set.

Llama-2-13B	PPL (↓)
0 + 1024 (Window)	5158.07
4 + 1020	5.40
4"\n"+1020	5.60

Attention sink details (contd.)

- Positional encoding determined w.r.t. cache position
  - Positions assigned continuously from 0 to L starting from the sink tokens
  - Most recent token assigned position L, not T

- Can be easily integrated with paged attention
  - Just don’t discard the first page

Pre-training with sink token

- As an alternative to storing the first few tokens, a zero token can be inserted before every sample in training
  - Can also have learnable token instead of a zero token

Table 3: Comparison of vanilla attention with prepending a zero token and a learnable sink token during pre-training. To ensure stable streaming perplexity, the vanilla model required several initial tokens. While Zero Sink demonstrated a slight improvement, it still needed other initial tokens. Conversely, the model trained with a learnable Sink Token showed stable streaming perplexity with only the sink token added. Cache config x+y denotes adding x initial tokens with y recent tokens. Perplexity is evaluated on the first sample in the PG19 test set.

Cache Config	0+1024	1+1023	2+1022	4+1020
Vanilla	27.87	18.49	18.05	18.05
Zero Sink	29214	19.90	18.27	18.01
Learnable Sink	1235	**18.01**	18.01	18.02

Another alternative is to change the softmax function:

$$\text{SoftMax}_1(x)_i = \frac{e^{x_i}}{1 + \sum_{j=1}^{N} e^{x_j}},$$

Experiment results

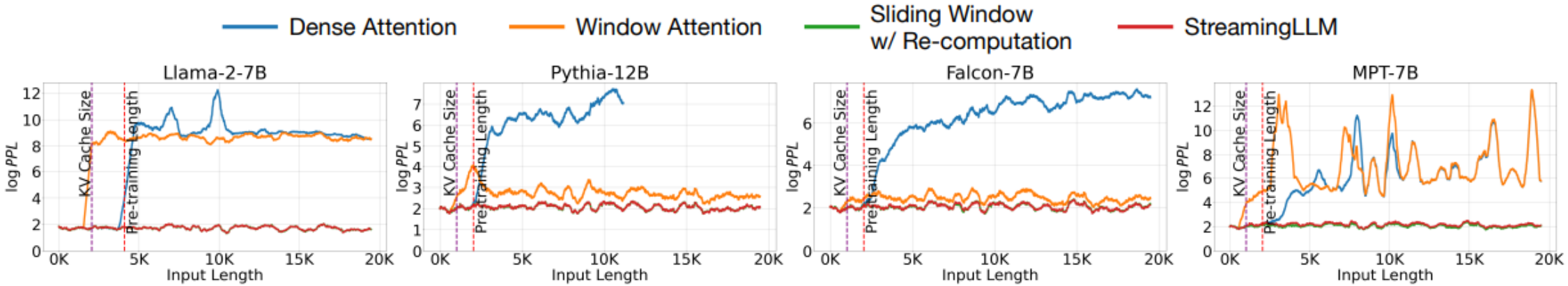


Figure 3: Language modeling perplexity on texts with 20K tokens across various LLM. Observations reveal consistent trends: (1) Dense attention fails once the input length surpasses the pre-training attention window size. (2) Window attention collapses once the input length exceeds the cache size, i.e.,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

Experiment results (contd.)

Pre-training with sink token

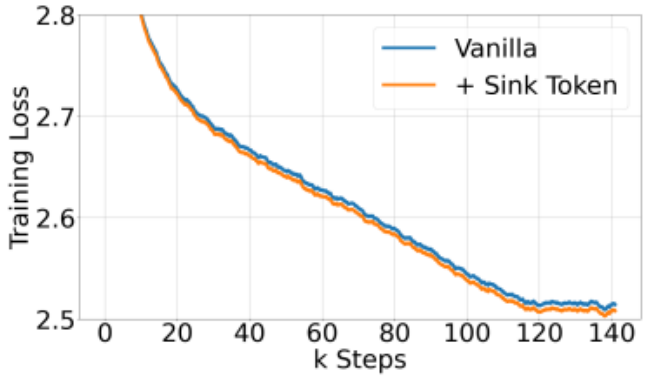


Figure 6: Pre-training loss curves of models w/ and w/o sink tokens. Two models have a similar convergence trend.

Table 4: Zero-shot accuracy (in %) across 7 NLP benchmarks, including ARC-[Challenge, Easy], HellaSwag, LAMBADA, OpenbookQA, PIQA, and Winogrande. The inclusion of a sink token during pre-training doesn't harm the model performance.

Methods	ARC-c	ARC-e	HS	LBD	OBQA	PIQA	WG
Vanilla	18.6	45.2	29.4	39.6	16.0	62.2	50.1
+Sink Token	**19.6**	**45.6**	**29.8**	**39.9**	**16.6**	**62.6**	**50.8**

Experiment results (contd.)

Performance gains

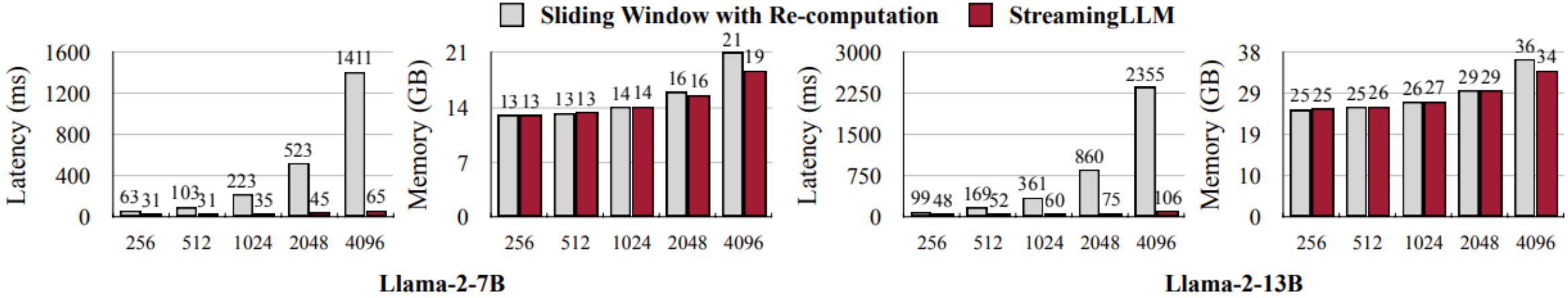


Figure 10: Comparison of per-token decoding latency and memory usage between the sliding window approach with re-computation baseline and StreamingLLM, plotted against the cache size (attention window size) on the X-axis. StreamingLLM delivers a remarkable speedup of up to 22.2× per token and retains a memory footprint similar to the re-computation baseline.

Why does the sink phenomenon happen?

- The first tokens are seen by every attention calculation, while later tokens are only seen by subsequent tokens
  - Similar observations for last tokens in encoder transformers (periods at the end of sentence, etc.)

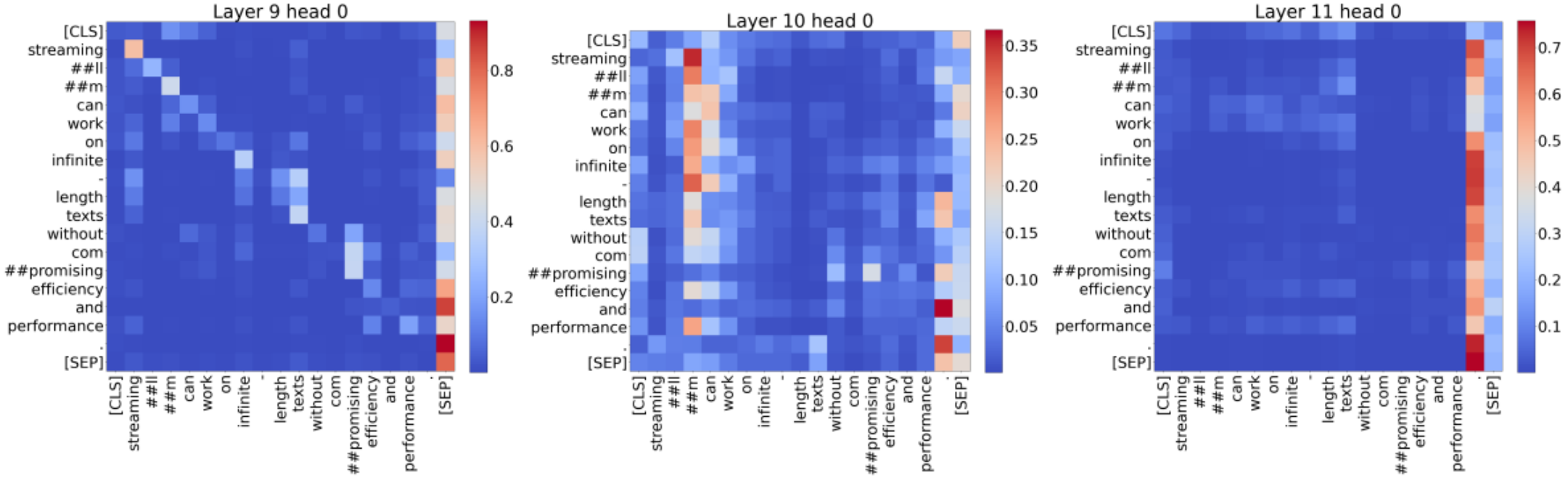


Figure 14: Visualization of attention maps for sentence *"StreamingLLM can work on infinite-length texts without compromising efficiency and performance."* in BERT-base-uncased.

Further work to be done?

- This technique doesn’t help extend context length
- Relation with different kinds of positional encoding seems unclear
- Window attention not significantly worse for some LLMs