Core Context Aware Transformers for Long Context Language Modeling

ICML 2025

Yaofo Chen[1], Zeng You[1,2], Shuhai Zhang[1,3], Haokun Li[1,2], Yiru Li[1], Yaowei Wang[2,4], Mingkui Tan[1,3,5]

South China University of Technology[1], Peng Cheng Laboratory[2], Pazhou Laboratory[3], Harbin Institute of Technology[4], Key Laboratory of Big Data and Intelligent Robot, Ministry of Education[5]

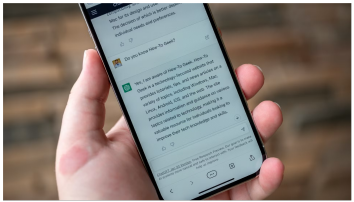
June 10, 2025

Outline

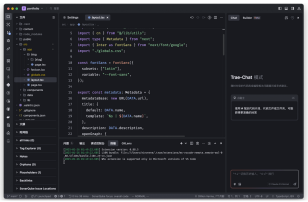
Outline

Large Language Model

Large language models (LLMs) have demonstrated exceptional proficiency across various applications, particularly in natural language understanding and generation.



chatbot



coding

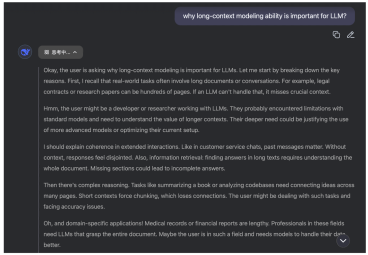
education

Figure: Applications of LLMs: empowering chatbots, coding assistance, and educational tools

Importance of Long-context Modeling

The ability to process longer contexts has been a key factor in improving the performance of LLMs across a wide range of tasks.

Legal document analysis



Chain-of-thought reasoning

Figure: Long context matters: enabling advanced long text analysis and chain-of-thought reasoning with large language models.

Challenges of Processing Long Context

When the context length becomes very large (*i.e.*, 128K), the redundant information in the context tends to increase, which not only **incurs unnecessary computational and storage overhead** but also **may hamper the modeling representation performance**.

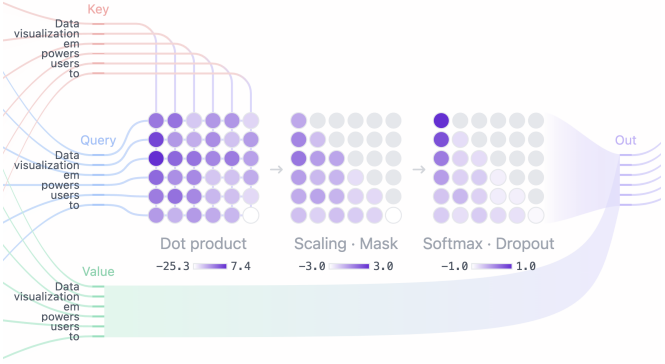


Figure: The computational complexity and KV cache of self-attention grows quadratically and linearly with the length of the context. This figure is from https://poloclub.github.io/transformer-explainer.

Preliminaries

Given a sequence of tokens $\mathbf{X} = [\mathbf{X}_1; \mathbf{X}_2; \ldots; \mathbf{X}_L] \in \mathbb{R}^{L \times d}$, where each token $\mathbf{X}_i \in \mathbb{R}^{1 \times d}$, the self-attention begins by transforming $\mathbf{X}$ into a query $\mathbf{Q}{=}\mathbf{X}\mathbf{W}^Q$, a key $\mathbf{K}{=}\mathbf{X}\mathbf{W}^K$, and a value $\mathbf{V}{=}\mathbf{X}\mathbf{W}^V$ through three linear projections $\mathbf{W}^{Q,K,V} \in \mathbb{R}^{d \times d}$. The self-attention is calculated by

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}. \tag{1}$$

The computational and storage complexity:

- The computational complexity of self-attention grows **quadratically** with the length of the context.
- During inference, we cache $\mathbf{K}$ and $\mathbf{V}$ for accelerating attention computation. The size and KV cache grows **linearly** with the length of the context.

Outline

Motivation

As the context length scales to extremely large magnitudes, it becomes impractical for a token to maintain significant semantic connections with all tokens in the context.

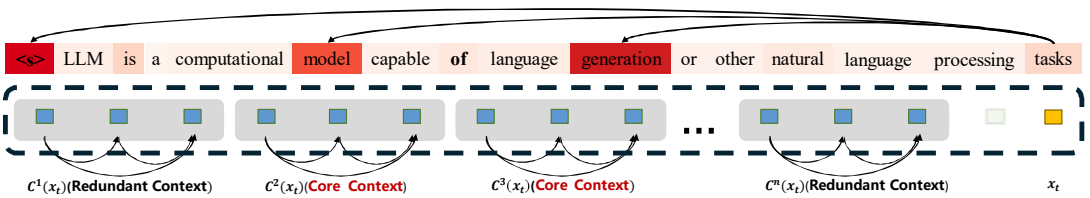


Figure: Illustration of core contexts and redundant contexts. We show attention scores of the last token relative to the other tokens in LLaMA2-7B (darker shadows indicate higher attention scores). The last token exhibits high attention scores towards core contexts. The remains are considered as redundant contexts, introducing unnecessary computational overhead for attention.

Motivation

The core context (*e.g.*, $C^2(\mathbf{x}_t)$ *w.r.t.* $\mathbf{x}_t$ in Figure) refers to the contextual information that is highly relevant to the token $\mathbf{x}_t$, which is crucial for the token's representation. Therefore, the model should **prioritize the core context over redundant parts**.

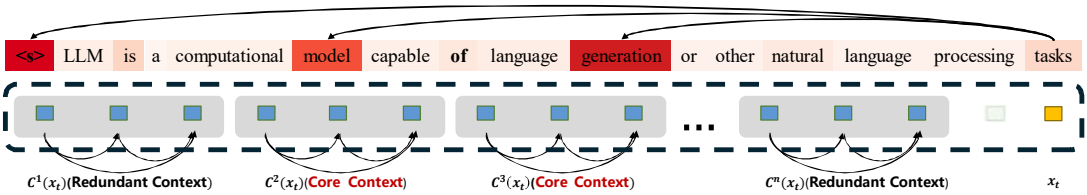


Figure: Illustration of core contexts and redundant contexts. We show attention scores of the last token relative to the other tokens in LLaMA2-7B (darker shadows indicate higher attention scores). The last token exhibits high attention scores towards core contexts. The remains are considered as redundant contexts, introducing unnecessary computational overhead for attention.

Overview

We propose a **Core Context Aware Attention** that employs globality-aware pooling and locality-preserving modules to capture both global and local context dependencies.

- The globality-aware pooling module operates by generating representative core tokens from segmented groups of the input sequence.
- The locality-preserving module is responsible for capturing the local information of s neighborhood tokens to ensure comprehensive coverage.
- We devise a differentiable fusion strategy to combine the insights from global and local modules to generate the final outputs.

Overview

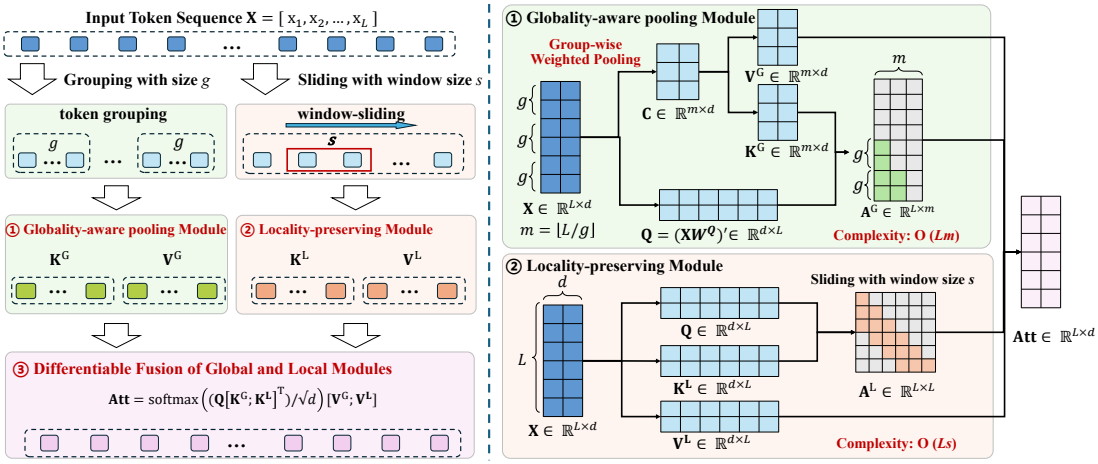


Figure: Illustration of CCA-Attention, which includes two components: Globality-aware pooling module and Locality-preserving module. We produce the final output $\mathbf{Att}$ by fusing these two modules.

Globality-aware Pooling Module: Core Tokens Generation

Motivation: The context redundancy indicates that computational resources can be dynamically allocated to core contexts while reducing emphasis on the remaining ones.

Solution: Given input tokens $\mathbf{X}=[\mathbf{x}_1;\mathbf{x}_2;\ldots;\mathbf{x}_L]\in\mathbb{R}^{L\times d}$, we segment it into $m=\lfloor L/g\rfloor$ groups, each group containing g tokens (denoting the i-th group by $\mathbf{X}^{\mathsf{G}}_{\mathcal{I}_i}\in\mathbb{R}^{g\times d}$).
To identify prominent tokens in the i-th group, we devise a group-wise weighted pooling strategy that employs the last token $\mathbf{x}_{ig}$ to evaluate the importance. Formally, we derive one core token $\mathbf{c}_i\in\mathbb{R}^{1\times d}$ from each group by

$$\mathbf{c}_i=\text{softmax}\left(\frac{\mathbf{Q}_{ig}\mathbf{K}'^{\top}_{\mathcal{I}_i}}{\sqrt{d}}\right)\mathbf{X}^{\mathsf{G}}_{\mathcal{I}_i}\in\mathbb{R}^{1\times d}, i=1,\ldots,m, \tag{2}$$

where $\mathbf{K}'_{\mathcal{I}_i}=\mathbf{X}^{\mathsf{G}}_{\mathcal{I}_i}\mathbf{W}^K$. The core token $\mathbf{c}_i$ encapsulates crucial information of the group. With m groups, we derive m core tokens $\mathbf{C}=[\mathbf{c}_1;\mathbf{c}_2;\ldots;\mathbf{c}_m]\in\mathbb{R}^{m\times d}$.

Globality-aware Pooling Module: Core Tokens as Efficient Proxy

To reduce the redundancy, we use the sequence of core tokens $\mathbf{C}$ instead of the original tokens $\mathbf{X}$ for attention computation. Formally, we adopt core tokens to calculate the key and value matrices for a specific query $\mathbf{Q}_i$ as follows

$$\begin{aligned} \widetilde{\mathbf{K}}^{\mathsf{G}}_{\mathcal{T}_i}=[\mathbf{K}^{\mathsf{G}}_1;\cdots;\mathbf{K}^{\mathsf{G}}_j]\in\mathbb{R}^{j\times d}, \widetilde{\mathbf{V}}^{\mathsf{G}}_{\mathcal{T}_i}=[\mathbf{V}^{\mathsf{G}}_1;\cdots;\mathbf{V}^{\mathsf{G}}_j]\in\mathbb{R}^{j\times d}, \\ \mathbf{K}^{\mathsf{G}}=\mathbf{C}\mathbf{W}^K\in\mathbb{R}^{m\times d}, \mathbf{V}^{\mathsf{G}}=\mathbf{C}\mathbf{W}^V\in\mathbb{R}^{m\times d}, \end{aligned} \tag{3}$$

The index j in the above Eqn. can be calculated as $j=\max(0,\lfloor(i-s)/g\rfloor)$. When the context is short ($i<(g+s)$), the key and value $\mathbf{K}^{\mathsf{G}}$, $\mathbf{V}^{\mathsf{G}}$ would be excluded from attention calculation since the redundancy in the context is negligible.

During inference, as tokens are generated sequentially, we derive a new core token via the aforementioned Eqn., once the number of generated tokens reaches g. Different from the full self-attention, we cache $\widetilde{\mathbf{K}}^{\mathsf{G}}$ and $\widetilde{\mathbf{V}}^{\mathsf{G}}$ for inference.

Locality-preserving Module

Challenge: The globality-aware pooling module focuses on coarse-grained global information, potentially overlooking fine-grained local context.

Solution: We introduce a locality-preserving module that complements the globality-aware pooling module by focusing on neighboring tokens to capture detailed local dependencies. The key and value matrices for a specific query $\mathbf{Q}_i$ in the locality-preserving module are defined as follows:

$$\begin{gathered}\widetilde{\mathbf{K}}^{\mathsf{L}}_{\mathcal{U}_i}=[\mathbf{K}^{\mathsf{L}}_k;\cdots;\mathbf{K}^{\mathsf{L}}_i],\quad \widetilde{\mathbf{V}}^{\mathsf{L}}_{\mathcal{U}_i}=[\mathbf{V}^{\mathsf{L}}_k;\cdots;\mathbf{V}^{\mathsf{L}}_i],\\ \mathbf{K}^{\mathsf{L}}=\mathbf{X}\mathbf{W}^K,\mathbf{V}^{\mathsf{L}}=\mathbf{X}\mathbf{W}^V\end{gathered} \tag{4}$$

During generation, it is challenging to maintain the number of tokens as a multiple of the group size g. To address this, we set the local window size to $s+((i-s) \bmod g)$.

Differentiable Fusion of Global and Local Modules

Challenge: Both globality-aware pooling and locality-preserving modules involve only a portion of tokens, leading to a limited comprehensive understanding of the context.

Solution: We seek to combine the involved tokens of these two attentions by concatenating the key and value matrices from both attentions, *i.e.*, $[\widetilde{\mathbf{K}}^{\mathsf{G}}_{\mathcal{T}_i}; \widetilde{\mathbf{K}}^{\mathsf{L}}_{\mathcal{U}_i}]$ and $[\widetilde{\mathbf{V}}^{\mathsf{G}}_{\mathcal{T}_i}; \widetilde{\mathbf{V}}^{\mathsf{L}}_{\mathcal{U}_i}]$, to leverage the combined information. Formally, the proposed CCA-Attention is computed as follows:

$$\mathbf{Att}_i = \mathrm{softmax}\Big(\frac{\mathbf{Q}_i[\widetilde{\mathbf{K}}^{\mathsf{G}}_{\mathcal{T}_i}; \widetilde{\mathbf{K}}^{\mathsf{L}}_{\mathcal{U}_i}]^\top)}{\sqrt{d}}\Big)[\widetilde{\mathbf{V}}^{\mathsf{G}}_{\mathcal{T}_i}; \widetilde{\mathbf{V}}^{\mathsf{L}}_{\mathcal{U}_i}]. \tag{5}$$

We represent the final output of our CCA-Attention as $\mathbf{Att} = [\mathbf{Att}_1; \mathbf{Att}_2; \ldots; \mathbf{Att}_L]$.

Algorithm

Require: Inputs $\mathbf{X}=[\mathbf{x}_1;\mathbf{x}_2;\ldots;\mathbf{x}_L]$, parameters $\mathbf{W}^{Q,K,V}$, group size g, local window size s.

Calculate the query $\mathbf{Q}=\mathbf{X}\mathbf{W}^Q$, #groups $m=\lfloor L/g \rfloor$
for i in $\{1,2,\ldots,m\}$ **do**
  $\mathbf{X}^{\mathsf{G}}_{\mathcal{I}_i}=[\mathbf{x}_{(i-1)g+1};\mathbf{x}_{(i-1)g+2};\ldots;\mathbf{x}_{ig}]$
  $\mathbf{c}_i=\text{softmax}\left(\frac{\mathbf{Q}_{ig}\mathbf{K}'^{\top}_{\mathcal{I}_i}}{\sqrt{d}}\right)\mathbf{X}^{\mathsf{G}}_{\mathcal{I}_i}$, where $\mathbf{K}'_{\mathcal{I}_i}=\mathbf{X}^{\mathsf{G}}_{\mathcal{I}_i}\mathbf{W}^K$
end for
Let $\mathbf{C}=[\mathbf{c}_1;\mathbf{c}_2;\ldots;\mathbf{c}_m]$
$\mathbf{K}^{\mathsf{G}}=\mathbf{C}\mathbf{W}^K$, $\mathbf{V}^{\mathsf{G}}=\mathbf{C}\mathbf{W}^V$ // *Globality-aware Pooling Module*
$\mathbf{K}^{\mathsf{L}}=\mathbf{X}\mathbf{W}^K$, $\mathbf{V}^{\mathsf{L}}=\mathbf{X}\mathbf{W}^V$ // *Locality-preserving Module*
for i in $\{1,2,\ldots,L\}$ **do**
  $\widetilde{\mathbf{K}}^{\mathsf{G}}_{\mathcal{T}_i}=\mathbf{K}^{\mathsf{G}}_{1:j}$, $\widetilde{\mathbf{V}}^{\mathsf{G}}_{\mathcal{T}_i}=\mathbf{V}^{\mathsf{G}}_{1:j}$, $j=\max(0,\lfloor(i-s)/g\rfloor)$
  $\widetilde{\mathbf{K}}^{\mathsf{L}}_{\mathcal{U}_i}=\mathbf{K}^{\mathsf{L}}_{k:i}$, $\widetilde{\mathbf{V}}^{\mathsf{L}}_{\mathcal{U}_i}=\mathbf{V}^{\mathsf{L}}_{k:i}$, $k=\max(1,i-s-((i-s) \bmod g))$
  $\mathbf{Att}_i=\text{softmax}(\mathbf{Q}_i[\widetilde{\mathbf{K}}^{\mathsf{G}}_{\mathcal{T}_i};\widetilde{\mathbf{K}}^{\mathsf{L}}_{\mathcal{U}_i}]^{\top})/\sqrt{d})[\widetilde{\mathbf{V}}^{\mathsf{G}}_{\mathcal{T}_i};\widetilde{\mathbf{V}}^{\mathsf{L}}_{\mathcal{U}_i}]$
end for
Return: Representations of tokens $\mathbf{Att}=[\mathbf{Att}_1;\ldots;\mathbf{Att}_L]$

Efficient and Parallel Implementation

We implement our CCA-Attention by leveraging Triton(github.com/triton-lang/triton) to perform low-level operator fusion between our globality-aware pooling and locality-preserving modules. This enables us to integrate our CCA-Attention as a standalone, cache-friendly operator, effectively eliminating redundant computations.

```
@triton.jit
def _locality_preserved_attn_fwd_inner(
    local_acc, l_i, m_i, q,  #
    K_block_ptr, V_block_ptr,  #
    Pre_K_block_ptr, Pre_V_block_ptr,
    Core_K_block_ptr, Core_V_block_ptr,
    start_m, qk_scale,  #
    BLOCK_M: tl.constexpr, HEAD_DIM: tl.constexpr, BLOCK_N: tl.constexpr,  #
    STAGE: tl.constexpr, offs_m: tl.constexpr, offs_n: tl.constexpr,  #
    N_CTX: tl.constexpr, fp8_v: tl.constexpr,
    window_size: tl.constexpr,
    group_size: tl.constexpr,
):
    # calculation for locality-preserved attention
    n_group_per_block = BLOCK_M // group_size
    n_group_per_block = BLOCK_N // group_size
    n_blocks_per_window = window_size // BLOCK_M
    # n_blocks_per_window = tl.cdiv(window_size, BLOCK_N)
    if STAGE == 1:
        lo, hi = tl.maximum(start_m-n_blocks_per_window, 0) * BLOCK_N, \
            tl.maximum(start_m-n_blocks_per_window+1, 0) * BLOCK_N
        la = tl.multiple_of(lo, BLOCK_N)
    elif STAGE == 2:
        lo, hi = tl.maximum(start_m-n_blocks_per_window+1, 0) * BLOCK_N, \
            tl.maximum(start_m + BLOCK_M, 0)
        la = tl.multiple_of(lo, BLOCK_N)
    elif STAGE == 3:
        lo, hi = start_m * BLOCK_M, (start_m + 1) * BLOCK_M
        la = tl.multiple_of(lo, BLOCK_M)
    K_block_ptr = tl.advance(K_block_ptr, (0, la))
    V_block_ptr = tl.advance(V_block_ptr, (la, 0))
    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        start_n = tl.multiple_of(start_n, BLOCK_N)
        k = tl.load(K_block_ptr)
        qk = tl.dot(q, k)  # [BLOCK_M,BLOCK_N]
        if STAGE == 1:
            mask = offs_m[:, None] - window_size < (start_n + offs_n[None, :])
            qk = qk * qk_scale + tl.where(mask, 0, -1.0e6)
            m_ij = tl.maximum(m_i, tl.max(qk, 1))  # [BLOCK_M,]
            qk -= m_ij[:, None]  # [BLOCK_M,BLOCK_N]
        elif STAGE == 2:
            m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
            qk = qk * qk_scale - m_ij[:, None]
        else:  # STAGE == 3
            mask = offs_m[:, None] >= (start_n + offs_n[None, :])
            qk = qk * qk_scale + tl.where(mask, 0, -1.0e6)
            m_ij = tl.maximum(m_i, tl.max(qk, 1))
            qk -= m_ij[:, None]
        p = tl.math.exp2(qk)  # [BLOCK_M,BLOCK_N]
        l_ij = tl.sum(p, 1)  # [BLOCK_M,]
        alpha = tl.math.exp2(m_i - m_ij)  # [BLOCK_M,]
        l_i = l_i * alpha + l_ij  # [BLOCK_M,]
        local_acc = local_acc * alpha[:, None]  # [BLOCK_M,HEAD_DIM]
        v = tl.load(V_block_ptr)  # [BLOCK_N, HEAD_DIM]
        if fp8_v:
            p = p.to(tl.float8e5)
        else:
```

```
@triton.jit
def _globality_pooling_attn_fwd_inner(
    global_acc, l_i, m_i, q,  #
    K_block_ptr, V_block_ptr,  #
    start_m, qk_scale,  #
    BLOCK_M: tl.constexpr, HEAD_DIM: tl.constexpr, BLOCK_N: tl.constexpr,  #
    STAGE: tl.constexpr, offs_m: tl.constexpr, offs_n: tl.constexpr,  #
    N_CTX: tl.constexpr, group_size:tl.constexpr, fp8_v: tl.constexpr
):
    N_GROUP = N_CTX//group_size
    lo, hi = 0, (start_m + 1) * BLOCK_M//group_size
    K_block_ptr = tl.advance(K_block_ptr, (0, lo))
    V_block_ptr = tl.advance(V_block_ptr, (lo, 0))
    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        # -- compute qk ----
        k = tl.load(K_block_ptr,boundary_check=(0,1))
        qk = tl.dot(q, k)
        offs_group = offs_m//group_size
        mask = offs_group[:, None] > (start_n + offs_n[None, :])
        qk = qk * qk_scale + tl.where(mask, 0, -1.0e6)
        # print(f'startm:{start_m},mask: {mask}, qk: {qk}')
        m_ij = tl.maximum(m_i, tl.max(qk, 1))
        qk -= m_ij[:, None]
        p = tl.math.exp2(qk)
        l_ij = tl.sum(p, 1)

        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        # -- update output accumulator --
        global_acc = global_acc * alpha[:, None]
        # update acc
```

Outline

Experimental Results

We apply our CCA-Attention to pretrained LLMs and compare it with state-of-the-art efficient attention methods in terms of long-context modeling and efficiency.

Dataset and Evaluation Metrics

- **LongBench** is a benchmark for the bilingual, multi-task, and comprehensive assessment of LLMs' long context understanding capabilities.
- **Exact Match Score (EM Score)** in multidocument QA is a metric for measuring the model's ability to find the key information within a long context.

Models and Compared Methods

- **Models**: We apply our proposed CCAAttention to LLaMA2-7B-32K, LLaMA2-7B-80K, LLaMA3.1-8B-128K, and Qwen2.5-7B-128K.
- **Compared Methods**: We compare our proposed CCA-Attention with vanilla attention, StreamingLLM, LM-infinite, and MInference.

Comparisons on Long Context Modeling

Comparisons on Longbench-E:

- CCA-LLM outperforms all efficient attention baselines across varying context lengths, attributed to its global-aware pooling for enhanced core-context focus.
- CCA-LLM attains the lowest inference latency and memory usage among competitors, significantly reducing storage requirements while maintaining speed.

Methods	S. QA	M. QA	Sum.	FS. Learning	Synthetic	Code	Avg.	FTL (s)	Mem. (GB)
LLaMA2-7B-32K (Vanilla Self-Attention)	2.75	1.85	**12.43**	**66.28**	0.34	48.99	**22.11**	9.15	35.58
• StreamingLLM	**4.75**	2.94	2.97	48.20	0.66	30.16	14.95	5.75 (1.6×)	22.94 (35%↓)
• LM-Infinite	2.04	2.33	1.98	57.45	0.3	48.46	18.76	4.72 (1.9×)	26.35 (26%↓)
• MInference	3.68	3.05	10.97	66.26	0.61	42.30	21.14	4.20 (2.2×)	33.52 (6%↓)
• CCA-LLM (Ours)	3.63	**3.98**	7.79	61.79	**2.64**	**51.36**	21.86	**2.59 (3.5×)**	**19.12 (46%↓)**
LLaMA2-7B-80K (Vanilla Self-Attention)	3.22	2.71	3.90	**64.98**	0.56	**59.16**	**22.42**	32.43	60.03
• StreamingLLM	2.07	2.32	0.37	45.03	**2.67**	37.17	14.94	9.04 (3.6×)	37.45 (37%↓)
• LM-Infinite	2.54	1.53	2.22	61.29	1.08	58.54	21.20	8.27 (3.9×)	41.54 (31%↓)
• MInference	2.44	3.49	4.41	64.26	0.28	57.60	22.08	8.14 (4.0×)	54.09 (10%↓)
• CCA-LLM (Ours)	**5.62**	**4.34**	**8.99**	59.60	0.48	54.40	22.24	**6.42 (5.7×)**	**33.86 (44%↓)**

* Refer our paper for results of LLaMA3.1-8B-Instruct-128K and Qwen2.5-7B-128K

Comparisons on Long Context Modeling

Comparisons on Long-document QA:

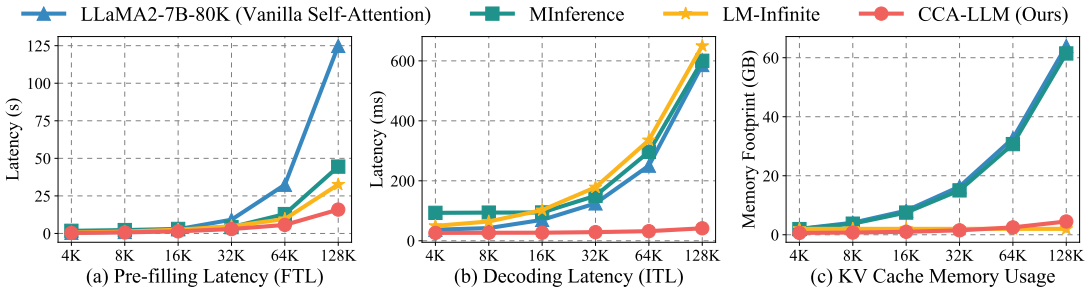
- CCA-LLM outperforms baselines on short sequences (e.g., 4K) by preserving both local and global dependencies without token discard.
- For extreme lengths (64K–128K), CCA-LLM delivers up to 7.9× faster inference and higher accuracy than vanilla self-attention, thanks to adaptive context compression.
- Our method consistently excels across diverse context scales (4K–128K), avoiding performance degradation by dynamically balancing core/redundant context.

Methods	4K	8K	16K	32K	64K	128K	Avg.	FTL (s)
LLaMA2-7B-80K (Vanilla Self-Attention)	**39.4**	**37.8**	**37.6**	**36.2**	34.6	30.3	**36.0**	124.85
• StreamingLLM	33.6	26.0	32.2	30.6	27.4	25.1	29.2	34.74 (3.6×)
• LM-Infinite	31.6	25.6	32.4	32.2	28.2	26.3	29.4	32.57 (3.8×)
• MInference	39.0	32.4	37.4	36.0	32.3	28.9	34.3	20.18 (6.2×)
• CCA-LLM (Ours)	39.3	33.2	35.4	31.4	**35.3**	**32.0**	34.4	**15.89 (7.9×)**

Comparisons on Computational Efficiency

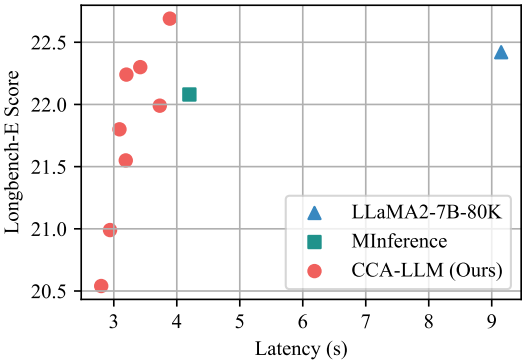
We compare our CCA-LLM with existing methods in terms of inference latency and memory footprint during inference on a single NVIDIA A800 GPU.

- CCA-LLM achieves **7.9**× faster pre-filling speed than full self-attention (15.89s vs. 124.85s at 128K) and accelerates both pre-filling and decoding phrase.
- With **93%** KV cache usage reducing (4.5GB vs. 64GB at 128K), CCA-LLM slashes GPU memory demands while maintaining end-to-end efficiency.



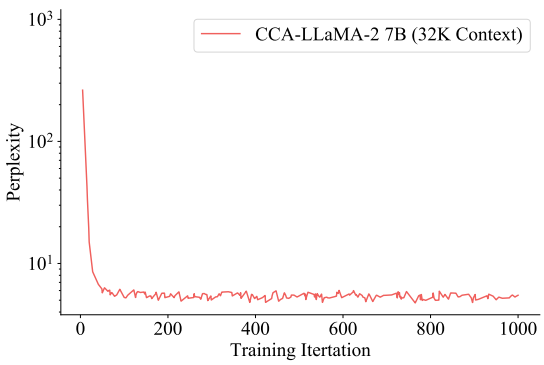
Demonstration of Inference Flexibility

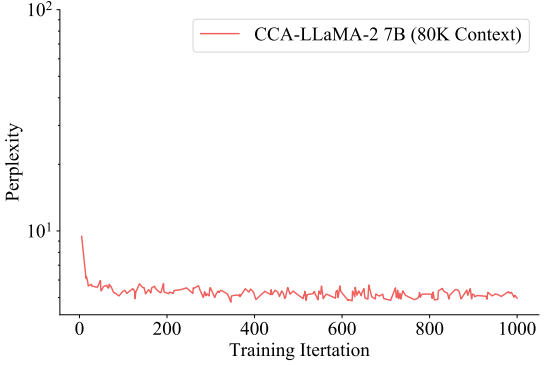
- With **no retraining needed**, CCA offers a tunable tradeoff between accuracy and speed, enabled by its context-aware compression and scale-invariant local attention.
- We can **dynamically adapt group/window sizes** to maintain high throughput during peak traffic, eliminating server bottlenecks while retaining performance.



Training Convergence Curve

- Our method enables both LLaMA2-7B-32K and 80K models to achieve rapid convergence within 100 steps while maintaining training stability over 1,000 steps.
- The consistent convergence performance demonstrates the effectiveness of CCA-Attention as a readily integrable component for existing LLMs, regardless of their initial context window sizes.





Effect of Different Updating Strategies

We can fine-tune the models by two updating strategies: 1) updating all the parameters (full finetuning) and 2) only updating $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$ (partial finetuning).

- Our method achieves optimal performance when fully fine-tuning all parameters, enabling complete adaptation to the proposed attention mechanism.
- Partial fine-tuning offers a resource-efficient solution with competitive accuracy, ideal for scenarios with computational constraints.

Strategies	Single-Doc. QA	Multi-Doc. QA	Sum.	FS. Learning	Synthetic	Code	Avg.
Partial Finetuning	5.39	3.62	9.21	60.41	1.34	51.77	21.96
Full Finetuning	5.62	4.34	8.99	59.60	0.48	54.40	**22.24**

Outline

Conclusion

- Extremely long contexts in LLMs introduce redundant computations while sparse attention methods compromise token connectivity. Our Core Context Aware Attention addresses this by dynamically compress context into core tokens.
- CCA combines (1) a globality-aware pooling module to compress context into adaptive core tokens and (2) a locality-preserving module for fine-grained context – achieving linear complexity without retraining.
- CCA enables 7.9$\times$ faster inference on 128K contexts while outperforming baselines, offering a plug-and-play solution for efficient long-context modeling in LLMs.

Thanks for Your Attention

Q&A