Context-Aggregated Linear Attention (CALA): Enhancing Linear Attention with Local Context

Author: *Michael Morgan*
AI Assistant: *Gemini 2.5 Pro*
Date: April 17, 2025

Abstract

Transformer models, while highly successful, face scalability challenges due to the quadratic complexity of their self-attention mechanism. Linear attention methods address this by approximating the softmax kernel or leveraging matrix associativity, achieving O(N) complexity but potentially sacrificing the ability to capture fine-grained token interactions based on single query-key vector pairs. Conversely, methods like Multi-Token Attention (MTA) enhance expressiveness by conditioning attention on multiple tokens via convolutions, but reintroduce significant computational costs. We propose Context-Aggregated Linear Attention (CALA), a novel attention mechanism designed to synthesize the efficiency of linear attention with the enhanced expressiveness of context-aware methods. CALA maintains O(N) time and space complexity by augmenting a linear attention backbone. Crucially, *before* the main linear attention computation, CALA incorporates a step that efficiently aggregates local context (from a sliding window) into the query and key representations using a localized, efficient attention or pooling mechanism. This allows the final linear attention step to operate on context-enriched features, enabling attention weights to be implicitly conditioned on multi-token information without quadratic complexity or heavy convolutional overhead. We detail the CALA architecture, analyze its linear complexity, contrast it with existing efficient and context-aware attention methods, and outline its potential for efficiently modeling long sequences with improved representational capacity.

1. Introduction

The Transformer architecture (Vaswani et al., 2017) has become a cornerstone of modern AI, particularly in sequence modeling tasks. Its power largely derives from the self-attention mechanism, which computes pairwise interactions between all tokens in a sequence, enabling the modeling of long-range dependencies in constant path length. However, the standard scaled dot-product attention exhibits $O(N^2)$ time and space complexity with respect to sequence length N, severely limiting its applicability to very long sequences.

This limitation has spurred extensive research into efficient attention mechanisms. Broadly, these fall into several categories:

- **Sparse Attention:** These methods reduce the number of query-key pairs computed, using fixed patterns (e.g., local windows, dilation, global tokens in Longformer (Beltagy et al., 2020)), learned sparsity, or combinations including random patterns (Big Bird, Zaheer et al., 2020). While improving asymptotic complexity (often to O(N) or O(N log N)), they introduce structural biases and can sometimes be less efficient in practice due to irregular memory access. ETC (Ainslie et al., 2020) uses global-local patterns effectively for structure.
- **Linearized/Kernelized Attention:** These methods approximate the softmax function via kernel methods (e.g., using random features in Performers (Choromanski et al., 2020) or RFA (Peng et al., 2021)) or use linear activation functions combined with the associative property of matrix multiplication (`Attention(Q, K, V) ≈ φ(Q)(φ(K)^T V)`) as in Linear Transformers (Katharopoulos et al., 2020) or Linformer's low-rank projection (Wang et al., 2020). These achieve true O(N) complexity but typically base their approximation on single query-key pairs, potentially simplifying the interaction landscape.
- **IO-Aware Optimization:** Techniques like FlashAttention (Dao et al., 2022; 2023) optimize the standard attention computation for modern GPU hardware by minimizing costly reads/writes to high-bandwidth memory (HBM) through tiling and kernel fusion. They accelerate exact attention but do not change its quadratic complexity.

While efficiency is crucial, another line of research investigates the expressive limitations of standard attention. Multi-Token Attention (MTA) (Golovneva et al., 2025) argues that conditioning attention solely on single query-key pairs is a bottleneck. MTA introduces convolutions over queries, keys, or attention matrices, allowing the attention weight between query `i` and key `j` to depend on their respective neighbors. This demonstrated improved performance but reintroduced significant computational overhead via the convolutions.

In this work, we aim to integrate the efficiency of linear attention with the enhanced expressiveness suggested by MTA. We propose Context-Aggregated Linear Attention (CALA), which maintains O(N) complexity while allowing attention weights to be implicitly conditioned on local multi-token context. CALA achieves this by inserting an efficient *local context aggregation* step after initial Q/K projections but *before* the main linear attention computation. This step modifies the Q and K vectors based on their local neighborhood, creating context-enriched representations that are then processed by a standard linear attention mechanism.

Our contributions are:

1. A novel attention mechanism, CALA, that modifies query and key representations based on local context using an efficient aggregation step within a linear attention framework.

2. A detailed architectural description of CALA, including considerations for causal and bidirectional settings.
3. An analysis of CALA's O(N) time and space complexity and its relationship to prior work.
4. A discussion of the potential benefits for long-sequence modeling, balancing efficiency and expressiveness.

2. Background and Related Work

We build upon the standard Transformer (Vaswani et al., 2017) and the body of work on efficient attention mechanisms reviewed briefly in the Introduction. We highlight specific connections below.

2.1 Linear Attention (Kernelized/Associative)

Methods like Linear Transformers (Katharopoulos et al., 2020), Performers (Choromanski et al., 2020), RFA (Peng et al., 2021), and Linformer (Wang et al., 2020) avoid the $O(N^2)$ bottleneck by either approximating the softmax kernel `exp(q_i^T k_j / sqrt(d_k))` with a kernel `sim(q_i, k_j)` that factorizes via feature maps `φ` (i.e., `sim(q_i, k_j) ≈ φ(q_i)^T φ(k_j)`), or by using low-rank projections. This allows computing the output via associativity: `sum_j (φ(q_i)^T φ(k_j)) v_j = φ(q_i)^T (sum_j φ(k_j) v_j^T)`. The sum over `j` can be computed once (or maintained as an RNN state) and reused for all queries `i`, yielding O(N) complexity. CALA uses this as its final computation step but operates on modified Q and K.

2.2 Sparse Attention (Local/Global Patterns)

Big Bird (Zaheer et al., 2020), Longformer (Beltagy et al., 2020), and ETC (Ainslie et al., 2020) demonstrate the effectiveness of combining local windowed attention with access to a few global tokens. This ensures information can propagate across the sequence while most computations remain local and efficient. CALA differs by aiming to make *all* tokens potentially context-aware (within a local window) rather than designating specific global tokens or using fixed sparse patterns.

2.3 Context-Aware Mechanisms (MTA, Convolutions)

MTA (Golovneva et al., 2025) explicitly uses convolutions to allow attention scores `A_{ij}` to depend on neighboring queries and keys. Other works have incorporated convolutions into Transformers, often applied to the input embeddings or before/after attention blocks (e.g., Conformer (Gulati et al., 2020) in speech). CALA aims for a similar outcome – making attention context-aware – but achieves it by modifying the Q/K representations using an efficient, non-convolutional (potentially attention-based) local aggregation, integrated within a linear attention framework.

2.4 Hardware Optimization (FlashAttention)

FlashAttention (Dao et al., 2022; 2023) brilliantly optimizes standard attention by leveraging the GPU memory hierarchy. It uses tiling and kernel fusion to compute exact attention while avoiding materialization of the N×N matrices in slow HBM. CALA operates at the algorithmic level, changing the attention computation itself to be O(N). However, the *implementation* of CALA's components, particularly the local aggregation and the matrix multiplies in the final linear step, could potentially benefit from FlashAttention-like optimization principles.

3. Context-Aggregated Linear Attention (CALA)

3.1 High-Level Idea

CALA introduces a preparatory step into the linear attention pipeline. After projecting inputs to Q, K, V, it computes "aggregated" representations `Q'` and `K'` where the representation for each position `i` incorporates information from its local neighborhood (e.g., tokens `i-w` to `i-1` in the causal case). This aggregation uses an efficient mechanism. The resulting context-aware `Q'` and `K'` are then used, along with the original V, in a final O(N) linear attention step.

3.2 Architectural Steps

Let `X ∈ R^{N x d_m}` be the input sequence. Let `w` be the local aggregation window size.

1. **Standard Projections:** `Q, K, V = XW_q, XW_k, XW_v` (`Q, K, V ∈ R^{N x d}`).
2. **Local Feature Mapping (** Apply a simple, non-negative feature map (e.g., `φ_local(x) = elu(x) + 1`) suitable for cheap local similarity:
   `Q'_{local} = φ_local(Q)`, `K'_{local} = φ_local(K)`.
3. **Local Context Aggregation:** Compute context vectors `C_Q` and `C_K` (`∈ R^{N x d}`). For each position `i`:
   - Define the causal neighborhood `Neigh(i) = {max(0, i-w), ..., i-1}`.
   - Compute local attention weights `alpha_{ij}` for `j ∈ Neigh(i)`, e.g., `alpha_{ij} ∝ dot(Q'_{local, i}, K'_{local, j})` (normalized over `j ∈ Neigh(i)`).
   - Aggregate neighboring features:
     `C_{Q, i} = sum_{j ∈ Neigh(i)} alpha_{ij} * Q_{j}` *(Note: using original Q)*
     `C_{K, i} = sum_{j ∈ Neigh(i)} alpha_{ij} * K_{j}` *(Note: using original K)*
     (Alternative: Aggregate `Q'_{local}`/`K'_{local}` instead of `Q`/`K`. Using dot-product attention locally might be feasible for small `w`; for larger `w`, a local linear approximation or simple pooling (e.g., max/mean) could be used for efficiency).
4. **Combine and Normalize:** Integrate context. Additive combination with Layer Normalization is a robust default:

`Q'_{agg} = LayerNorm(Q + C_{Q, i})`
`K'_{agg} = LayerNorm(K + C_{K, i})`

5. **Global Feature Mapping (** Apply a kernel-approximating feature map (e.g., FAVOR+) to the aggregated representations:
   `Q'' = φ_global(Q'_{agg}),K'' = φ_global(K'_{agg})` (`Q'', K'' ∈ R^{N x d'}`).
6. **Final Linear Attention:** Compute output O using `Q'', K'', V` via O(N) methods:
   - *Bidirectional:* `O = Normalize( Q'' @ (K''^T @ V) ).`
   - *Causal (RNN):* Update states `S_i, Z_i`, compute `Output_i = Normalize( Q''_i @ S_i ).`

3.3 Implementation Considerations
The efficiency bottleneck is Step 3. A naive implementation with explicit loops over `i` and windows is O(N*w). Direct application of standard attention functions here would be O(N*w²), which is only efficient for very small `w`. If using dot-products locally, optimizations like those for sliding-window convolutions or banded matrix multiplication are relevant. If using a linear approximation locally, the cost could be reduced. Fused kernels minimizing redundant reads/writes for the windowed operations are essential for good practical performance.

4. Analysis

4.1 Computational Complexity
If Step 3 is implemented efficiently (e.g., O(N*w) or O(N*log w) via FFT-based convolutions if pooling is used, or O(N*w) with local dot-products), the overall time complexity is O(N), dominated by steps 1, 5, and 6. Space complexity remains O(N).

4.2 Novelty and Expressiveness
CALA modifies the *inputs* (Q, K) to the main attention based on *local content*, differing from methods that modify attention *scores* based on *distance* (RPEs) or use global convolutions (MTA). By enriching Q and K before the linear attention step, CALA allows the model to base its final attention weighting on features that already incorporate neighborhood context, potentially enabling more nuanced dependencies than standard linear attention methods operating on single-token features. It hypothesizes that efficient local aggregation can approximate some benefits of MTA's more expensive global convolutions within a strictly linear framework.

5. Potential Limitations and Future Work

CALA is presented as an architectural proposal. Its practical effectiveness requires empirical validation and, crucially, efficient implementation of the local context aggregation step. Without optimized kernels, Step 3 could dominate runtime and negate the benefits of the O(N) final step.

Future directions include:

- Developing optimized CUDA kernels for Step 3.
- Empirically evaluating CALA on long-context tasks against relevant baselines.
- Ablating the design choices: `φ_local`, `φ_global`, aggregation mechanism (attention vs. pooling), combination method (additive vs. gating), window size `w`.
- Comparing the learned local aggregation behavior to convolutions in MTA.
- Integrating FlashAttention principles into the implementation.

6. Conclusion

We proposed Context-Aggregated Linear Attention (CALA), a novel O(N) attention mechanism that enhances linear attention by incorporating local context into query and key representations before the main attention calculation. By using an efficient local aggregation step, CALA aims to capture some of the multi-token expressiveness benefits seen in methods like MTA, while preserving the linear scalability required for very long sequences. This architectural design offers a new direction for balancing efficiency and representational capacity in Transformers, though its practical realization depends on effective low-level optimization of the proposed local aggregation step.

References

[1] Ainslie et al. (2020). ETC: Encoding long and structured data in transformers. https://arxiv.org/abs/2004.08483

[2] Ba et al. (2016). Layer normalization. https://arxiv.org/abs/1607.06450

[3] Bahdanau et al. (2014). Neural machine translation by jointly learning to align and translate. https://arxiv.org/abs/1409.0473

[4] Beltagy et al. (2020). Longformer: The long-document transformer. https://arxiv.org/abs/2004.05150

[5] Child et al. (2019). Generating long sequences with sparse transformers. https://arxiv.org/abs/1904.10509

[6] Choromanski et al. (2020). Rethinking attention with performers. https://arxiv.org/abs/2009.14794

[7] Dao et al. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. https://arxiv.org/abs/2205.14135

[8] Dao, T. (2023). FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. https://arxiv.org/abs/2307.08691

[9] Golovneva et al. (2025). Multi-Token Attention. https://arxiv.org/abs/2504.00927

[10] Gulati et al. (2020). Conformer: Convolution-augmented transformer for speech recognition. https://arxiv.org/abs/2005.08100

[11] Katharopoulos et al. (2020). Transformers are RNNs: Fast autoregressive transformers with linear attention. https://arxiv.org/abs/2006.16236

[12] Peng et al. (2021). Random feature attention. https://arxiv.org/abs/2103.02143

[13] Vaswani et al. (2017). Attention is all you need. https://arxiv.org/abs/1706.03762

[14] Wang et al. (2020). Linformer: Self-attention with linear complexity. https://arxiv.org/abs/2006.04768

[15] Zaheer et al. (2020). Big Bird: Transformers for longer sequences.
https://arxiv.org/abs/2007.14062

Appendices

Appendix A: Detailed Pseudo-code for CALA

We provide pseudo-code for a single layer of the CALA mechanism within a multi-head attention framework. For simplicity, we present the bidirectional case first, followed by modifications for the causal case (RNN formulation).

```
# Inputs:
# X: Input sequence tensor, shape (batch_size, seq_len, d_model)
# Wq, Wk, Wv: Projection matrices for Q, K, V per head
# Wo: Output projection matrix
# W_agg_q, W_agg_k, W_agg_v: Optional weights if LocalAttention uses projections
# W_local_attn_out: Optional output projection for LocalAttention
# W_phi_global: Parameters for global feature map (if any)
# W_out_final: Final output projection for multi-head
# w: Local context window size
# h: Number of heads
# d: Dimension per head (d = d_model / h)
# d_prime: Dimension of global feature map phi_global

# --- Multi-Head Wrapper ---
def multi_head_cala(X, num_heads=h, window=w, ...):
    batch_size, seq_len, d_model = X.shape
    d_head = d_model // num_heads

    # 1. Standard Projections (Linear layers applied to last dim)
    Q = X @ Wq # Shape: (batch_size, seq_len, num_heads * d_head)
    K = X @ Wk # Shape: (batch_size, seq_len, num_heads * d_head)
    V = X @ Wv # Shape: (batch_size, seq_len, num_heads * d_head)

    # Reshape for multi-head processing
    Q = Q.view(batch_size, seq_len, num_heads, d_head).transpose(1, 2)
    K = K.view(batch_size, seq_len, num_heads, d_head).transpose(1, 2)
    V = V.view(batch_size, seq_len, num_heads, d_head).transpose(1, 2)
    # Now Q, K, V have shape (batch_size, num_heads, seq_len, d_head)

    # Apply CALA per head (can be parallelized over heads)
    outputs = []
    for i in range(num_heads):
        # Note: Slicing weights per head if they are stored combined
        head_output = cala_single_head(Q[:, i, :, :], K[:, i, :, :], V[:, i, :, :],
window, ...)
        outputs.append(head_output)
    # Concatenate heads and project
    O_concat = torch.cat(outputs, dim=-1) # Shape: (batch_size, seq_len, num_heads
* d_head)
    O_final = O_concat @ W_out_final # Shape: (batch_size, seq_len, d_model)
    return O_final

# --- CALA Single Head (Bidirectional Example) ---
def cala_single_head(Q_head, K_head, V_head, window, d_prime, ...):
    batch_size, seq_len, d_head = Q_head.shape

    # 2. Local Feature Mapping (phi_local)
    # Example: elu + 1
    Q_local_prime = F.elu(Q_head) + 1.0
    K_local_prime = F.elu(K_head) + 1.0

    # 3. Local Context Aggregation (Requires efficient windowed implementation)
    # Placeholder for aggregation result tensors
    C_Q = torch.zeros_like(Q_head)
    C_K = torch.zeros_like(K_head)

    # This loop is conceptual; actual implementation needs vectorization/kernels
    for i in range(seq_len):
        start_idx = max(0, i - window)
        # --- Causal: only keys/values up to i-1 --- #
        # keys_local = K_local_prime[:, start_idx:i, :]
        # queries_for_agg = Q_head[:, start_idx:i, :] # Or Q_local_prime
        # keys_for_agg = K_head[:, start_idx:i, :] # Or K_local_prime
        # --- Bidirectional: include window around i (handle boundaries) --- #
        # For simplicity, showing conceptual causal aggregation:
        if i > 0:
             keys_local = K_local_prime[:, start_idx:i, :]
             vals_Q_local = Q_head[:, start_idx:i, :] # Aggregate original Q
             vals_K_local = K_head[:, start_idx:i, :] # Aggregate original K

             # Option A: Simple Local Dot-Product Attention (Potentially O(w^2))
             local_scores_q = (Q_local_prime[:, i:i+1, :] @
keys_local.transpose(-2, -1)) / sqrt(d_head)
             local_alpha_q = F.softmax(local_scores_q, dim=-1)
             C_Q[:, i, :] = local_alpha_q @ vals_Q_local

             local_scores_k = (K_local_prime[:, i:i+1, :] @
keys_local.transpose(-2, -1)) / sqrt(d_head)
             local_alpha_k = F.softmax(local_scores_k, dim=-1)
             C_K[:, i, :] = local_alpha_k @ vals_K_local

             # Option B: Local Linear Attention / Pooling (Potentially O(w))
             # C_Q[:, i, :] = EfficientLocalAggregation(Q_local_prime[:, i, :],
keys_local, vals_Q_local)
            # C_K[:, i, :] = EfficientLocalAggregation(K_local_prime[:, i, :],
keys_local, vals_K_local)

    # 4. Combine and Normalize
    Q_agg = LayerNorm(Q_head + C_Q)
    K_agg = LayerNorm(K_head + C_K)

    # 5. Global Feature Mapping (phi_global)
    # Example: Placeholder for FAVOR+ or similar
    Q_global_prime = phi_global(Q_agg) # Shape: (batch_size, seq_len, d_prime)
    K_global_prime = phi_global(K_agg) # Shape: (batch_size, seq_len, d_prime)

    # 6. Final Linear Attention (Bidirectional using Associativity)
    K_prime_V = K_global_prime.transpose(-2, -1) @ V_head # Shape: (batch_size,
d_prime, d_head)
    O_unnormalized = Q_global_prime @ K_prime_V # Shape: (batch_size, seq_len,
d_head)

    # Normalization Factor (Denominator)
    Z_sum = torch.sum(K_global_prime, dim=1, keepdim=True) # Shape: (batch_size, 1,
d_prime)
    # Need to handle potential broadcasting issues here carefully depending on
phi_global
    # Simple sum used here; actual normalization depends on kernel properties
(e.g., dot product with 1 vector)
    Denominator = Q_global_prime @ Z_sum.transpose(-2, -1) # Shape: (batch_size,
seq_len, 1)
    Denominator = Denominator + 1e-6 # Epsilon for stability

    O_head = O_unnormalized / Denominator # Shape: (batch_size, seq_len, d_head)

    return O_head

# --- Causal Modification (RNN Formulation for Step 6) ---
# Requires iterating through sequence length 's'
# Initialize states: S = zeros(batch_size, d_prime, d_head), Z = zeros(batch_size,
d_prime, 1)
# Inside a loop for step 'i' from 0 to seq_len-1:
# ... (Steps 1-5 compute Q_global_prime[:, i, :], K_global_prime[:, i, :]) ...
# Update states:
# Z = Z + K_global_prime[:, i:i+1, :].transpose(-2, -1) # Shape: (batch_size,
d_prime, 1)
# S = S + K_global_prime[:, i:i+1, :].transpose(-2, -1) @ V_head[:, i:i+1, :] #
Shape: (batch_size, d_prime, d_head)
# Compute output for step i:
# O_unnormalized_i = Q_global_prime[:, i:i+1, :] @ S
# Denominator_i = Q_global_prime[:, i:i+1, :] @ Z + 1e-6
# Output_i = O_unnormalized_i / Denominator_i
```

Appendix B: Analysis of `LocalAttention` Mechanisms

The efficiency and effectiveness of the Local Context Aggregation step (Step 3 in CALA) depend critically on the choice of the `LocalAttention` mechanism used within the sliding window `w`. Here we analyze potential options:

1. **Scaled Dot-Product (SDP) Attention (Local):**
   - **Mechanism:** Computes standard attention using `Q'_{local, i}` and `K'_{local, j}` for `j` within the window `Neigh(i)`. Requires O(w^2) interactions per query `i`.
   - **Complexity:** O(N * w^2 * d) for the full sequence if implemented naively.
   - **Pros:** Most expressive, directly computes pairwise similarities within the window.
   - **Cons:** Becomes computationally expensive as window size `w` increases. Not asymptotically O(N) unless `w` is constant. May require significant memory bandwidth for fetching windowed keys if not implemented carefully.
   - **Feasibility:** Suitable only for very small window sizes (`w` << `d`).
2. **Linear Attention Approximation (Local):**
   - **Mechanism:** Applies a linear attention mechanism *within* the local window. For example, using `φ_local` features: `Agg_Q_i ≈ φ_local(Q_i) @ sum_{j ∈ Neigh(i)} (φ_local(K_j)^T @ Q_j)`. This avoids the O(w^2) dot-product matrix.
   - **Complexity:** O(N * w * d' * d) where `d'` is the dimension of `φ_local`. If `d'` is similar to `d`, this is O(N * w * d^2), still potentially slow. If using fast feature maps (e.g., very low `d'`), can be closer to O(N * w * d).
   - **Pros:** Reduces complexity compared to local SDP, especially if `w` is large.
   - **Cons:** Introduces an approximation within the aggregation step itself. Adds complexity of choosing and implementing local feature maps. Still requires efficient windowed sums.
   - **Feasibility:** A potential middle ground, especially if efficient local feature maps are used.
3. **Pooling Mechanisms (Max/Mean Pooling):**
   - **Mechanism:** Directly aggregates features within the window without attention weighting. E.g., `Agg_Q_i = MaxPool(Q_j for j ∈ Neigh(i))` or `Agg_Q_i = MeanPool(Q_j for j ∈ Neigh(i))`.
   - **Complexity:** O(N * w * d).

  - **Pros:** Computationally simplest and potentially fastest, especially if hardware has optimized pooling implementations.
  - **Cons:** Least expressive. Loses information about specific interactions within the window (e.g., which neighbor was most important). Max pooling is non-linear, mean pooling is linear.
  - **Feasibility:** Easiest to implement, but might not capture sufficient context.

Choice Rationale: The pseudo-code in Appendix A uses local SDP for conceptual clarity. However, for practical efficiency, especially with larger `w`, **local linear attention approximation** or even **pooling** are likely necessary. The optimal choice requires empirical investigation and depends on the trade-off between capturing local detail and computational cost. For CALA to be significantly faster than alternatives, this step must be highly optimized, likely requiring implementations closer to O(N * w * d) or O(N * w * log w).

Appendix C: Derivation of Normalization Factors for Final Linear Attention

The final step (Step 6) of CALA computes linear attention using the context-aggregated, globally feature-mapped queries `Q''` and keys `K''`, and original values `V`. The core idea of linear attention normalization relies on decomposing the softmax.

Recall standard softmax attention: `A_{ij} = exp(s_{ij}) / sum_k exp(s_{ik})`, where `s_{ij}` is the scaled dot product. The output is `O_i = sum_j A_{ij} V_j`.

In linear attention, we approximate `exp(s_{ij}) ≈ φ(q_i)^T φ(k_j)`. The output becomes:
`O_i ≈ (sum_j φ(q_i)^T φ(k_j) V_j) / (sum_k φ(q_i)^T φ(k_k))`
`O_i ≈ (φ(q_i)^T sum_j (φ(k_j) V_j^T)) / (φ(q_i)^T sum_k φ(k_k))`

Let `φ(q_i) = Q''_i`, `φ(k_j) = K''_j`.

Causal Case (RNN Formulation):
The states maintained are:

- `S_i = sum_{j=1 to i} (K''_j^T V_j)` (Numerator state, matrix shape `d' x d`)
- `Z_i = sum_{j=1 to i} K''_j^T` (Denominator state, vector shape `d' x 1`)
  These are updated iteratively: `S_i = S_{i-1} + K''_i^T V_i`, `Z_i = Z_{i-1} + K''_i^T`.

The output at step `i` is:

`Output_i = (Q''_i @ S_i) / (Q''_i @ Z_i + epsilon)`

The normalization factor for query `i` is explicitly computed as `Q''_i @ Z_i`. Since `Q''_i` and `Z_i` are available at step `i`, this is computed directly.

Bidirectional Case:
The output is computed as:
`O = Normalize( Q'' @ (K''^T @ V) )`
The unnormalized output matrix is `O_{unnorm} = Q'' @ (K''^T @ V).`
The normalization denominator matrix `D` (diagonal) should have entries `D_{ii} = sum_j sim(Q''_i, K''_j) ≈ sum_j Q''_i @ K''^T_j = Q''_i @ (sum_j K''^T_j).`
Let `Z_{global} = sum_{j=1 to N} K''^T_j` (a single vector of shape `d' x 1`). This is computed once.
Then the vector of denominators for all `i` is `D_{vec} = Q'' @ Z_{global}` (shape `N x 1`).
The normalized output is `O_i = O_{unnorm, i} / (D_{vec, i} + epsilon).`

Key Point: The normalization mechanism remains fundamentally the same as in standard linear attention, but it operates on the context-aggregated and globally-mapped features `Q''` and `K''`. The states `S` and `Z` (or global sum `Z_{global}`) incorporate the richer features derived from Steps 3-5. Numerical stability typically involves adding a small epsilon to the denominator. The LayerNorm steps applied earlier also contribute significantly to overall stability.

Appendix D: Potential Kernel Fusion Strategy

Achieving high practical performance with CALA necessitates minimizing data movement between slow HBM and fast SRAM, primarily by fusing operations into custom CUDA kernels. Here's a high-level strategy inspired by FlashAttention:

Goal: Compute one layer of CALA, processing the sequence `X` block by block, keeping intermediate results in SRAM as much as possible.

Assumptions:

- We process queries `Q` in blocks of size `B_q`.
- Keys `K` and Values `V` are also processed in blocks (`B_k`).
- SRAM size `M` is sufficient to hold blocks of Q, K, V and intermediate features.

High-Level Fused Kernel Logic (Conceptual Bidirectional Example):

```
// Outer loop: Iterate through query blocks
for i_block from 0 to N/B_q - 1:
    // Load query block and corresponding original K, V
    Q_block = LOAD_Q_FROM_HBM(i_block) // Shape (B_q, d)
    K_block = LOAD_K_FROM_HBM(i_block) // Shape (B_q, d)
    V_block = LOAD_V_FROM_HBM(i_block) // Shape (B_q, d)

    // Initialize block outputs and accumulators in SRAM
    O_block_accum = zeros(B_q, d)
    Z_block_denom = zeros(B_q, 1) // For final normalization sum

    // --- Local Aggregation Preparation (within SRAM if possible) ---
    // Apply phi_local to Q_block, K_block -> Q_local_prime, K_local_prime
    Q_local_prime_block = COMPUTE_ON_CHIP(phi_local(Q_block))
    K_local_prime_block = COMPUTE_ON_CHIP(phi_local(K_block))

    // --- Inner loop: Iterate through key/value blocks ---
    for j_block from 0 to N/B_k - 1:
        // --- Load necessary data for this block pair to SRAM ---
        // 1. Key/Value block
        Kj_block = LOAD_K_FROM_HBM(j_block) // Shape (B_k, d)
        Vj_block = LOAD_V_FROM_HBM(j_block) // Shape (B_k, d)

        // 2. Required data for Local Aggregation (potential bottleneck)
        //    Needs K'_local for local keys, and Q/K for values within windows
        //    -> Requires efficient windowed loading/caching strategy
        K_local_prime_j_window = LOAD_WINDOWED_K_LOCAL_PRIME(...) // From HBM or
cache
        Q_j_window = LOAD_WINDOWED_Q(...) // From HBM or cache
        K_j_window = LOAD_WINDOWED_K(...) // From HBM or cache

        // --- Compute CALA steps on-chip (SRAM) ---
        // 3. Local Context Aggregation for Q_block (using K_local_prime_j_window,
Q_j_window, K_j_window)
        C_Q_block = COMPUTE_ON_CHIP(LocalAggregation_Q(Q_local_prime_block, ...))
        C_K_block = COMPUTE_ON_CHIP(LocalAggregation_K(K_local_prime_block, ...))

        // 4. Combine & Normalize for Q_block, Kj_block
        Q_agg_block = COMPUTE_ON_CHIP(LayerNorm(Q_block + C_Q_block))
        K_agg_j_block = COMPUTE_ON_CHIP(LayerNorm(Kj_block + C_K_block_j)) // Need
C_K for Kj_block

        // 5. Global Feature Mapping for Q_block, Kj_block
        Q_double_prime_block = COMPUTE_ON_CHIP(phi_global(Q_agg_block))
        K_double_prime_j_block = COMPUTE_ON_CHIP(phi_global(K_agg_j_block))
        // 6. Partial Final Linear Attention calculation
        Partial_Kprime_V = K_double_prime_j_block.T @ Vj_block
        Partial_O_unnorm = Q_double_prime_block @ Partial_Kprime_V
        Partial_Z_sum = torch.sum(K_double_prime_j_block, dim=0) // Needs reduction

        // --- Accumulate results in SRAM ---
        O_block_accum += Partial_O_unnorm
        Z_block_denom += Q_double_prime_block @ Partial_Z_sum.T // Accumulate
denominator terms

    // Normalize final block output using accumulated denominator
    O_block_final = O_block_accum / (Z_block_denom + 1e-6)

    // Write final output block to HBM
    WRITE_O_TO_HBM(i_block, O_block_final)

// --- End ---
```

Challenges and Refinements:

- **Local Aggregation IO:** Step 3's dependency on windowed data (`K_local_prime_j_window`, etc.) is the most complex IO pattern. Efficient caching or pre-fetching into SRAM is critical. Fusing the loading and computation for this step is paramount.
- **Intermediate Storage:** Storing `C_Q_block`, `C_K_block`, `Q_agg_block`, etc., for the whole `Q_block` requires significant SRAM. Tiling might need to be finer-grained.
- **Synchronization:** Accumulating `O_block_accum` and `Z_block_denom` across inner loop iterations is straightforward.
- **Causal Case:** Would require careful management of states `S` and `Z` across blocks, likely serializing the outer loop (`i_block`) but parallelizing within the block or across heads.

This pseudo-code highlights the potential for fusion but underscores the complexity, especially managing the IO for the local aggregation step efficiently alongside the main attention computation. A real implementation would require sophisticated tiling and data movement orchestration.