Distributed Streaming Attention: Scaling Memory-Bound Transformer Inference on CPU Clusters

Song Xixuan
Tsinghua University
songxx21@mails.tsinghua.edu.cn
ID: 2025310707

Abstract

Self-attention is a fundamental component of Transformer architectures, yet it poses a major bottleneck for long-context inference on CPUs due to its $O(T^2)$ memory footprint and memory-bandwidth-bound execution. We propose *distributed streaming attention*, which combines an online softmax formulation with multi-node sequence parallelism to scale attention computation beyond a single machine. Our key insight is that **online softmax statistics admit an associative reduction along the sequence dimension**, enabling lightweight distributed aggregation and efficient utilization of aggregate memory bandwidth across nodes. On pure attention microbenchmarks, our approach achieves up to 6.41× speedup over serial execution using 64 cores across 8 nodes. For end-to-end inference of Qwen3-0.6B with 208 threads, we achieve 4.64× speedup over single-node execution. Compared to OpenMP at equivalent processor counts, MPI-based sequence parallelism delivers 2.3× higher throughput on pure attention workloads. These results demonstrate that algorithm-aware distributed execution can make long-context LLM inference on CPU clusters practical and efficient. **Code is available at** `https://github.com/songxxzp/streaming-attention`.

1 Introduction

Transformer architectures Vaswani et al. [2017] revolutionized NLP through self-attention, which computes $O(T^2)$ interactions between tokens. For long-context inference (sequence lengths $T > 65,000$), this quadratic memory footprint becomes the primary bottleneck in the **prefill phase**, where the model processes the entire input prompt in parallel.

1.1 The Challenge: Memory-Bound Attention on CPUs

Attention computation on CPUs is fundamentally **memory-bandwidth-bound** rather than compute-bound. The Roofline model Williams et al. [2009] predicts that for low arithmetic intensity operations (FLOPs/byte), performance is limited by memory bandwidth, not peak FLOPs. Empirically, standard attention implementations achieve less than 15% of peak CPU performance, confirming this bottleneck.

This has critical implications: traditional compute-optimized approaches (e.g., GEMM-based parallelization) are ineffective for attention on CPUs. Instead, we must increase aggregate memory bandwidth through multi-node parallelism.

Preprint.

Why CPU Clusters? CPU clusters are widely accessible and cost-effective for inference. While GPU-based methods like FlashAttention-2 [Dao, 2024] achieve superior absolute performance, CPU solutions complement GPU infrastructure and enable heterogeneous deployment.

1.2 Our Approach: Distributed Streaming Attention

We propose **distributed streaming attention**, which combines algorithmic reformulation (online softmax) with multi-node parallelism (MPI) to overcome memory bandwidth limitations.

Algorithm Foundation. FlashAttention [Dao et al., 2023, Dao, 2024] pioneered online softmax [Chen et al., 2021] and block-wise computation to avoid materializing the full $T \times T$ attention matrix, reducing memory from $O(T^2)$ to $O(Td)$. We observe that **online softmax statistics admit associative reduction along the sequence dimension**, enabling distributed execution.

Multi-Node Scaling. Single-node bandwidth is limited (e.g., 36 GB/s per socket). By distributing key-value blocks across multiple nodes via MPI [Forum, 2015], we increase aggregate memory bandwidth proportionally. Each node computes local attention statistics, and a lightweight allreduce aggregates results.

Communication Efficiency. Unlike Ring Attention [Liu et al., 2023], which passes $O(Td)$ token data between neighboring nodes, we communicate only aggregated scalar statistics (T maxima, T sums, $T \times d$ outputs), enabling efficient tree-based allreduce on standard networks.

MPI vs OpenMP. OpenMP is constrained by single-node bandwidth saturation. MPI adds bandwidth from multiple nodes, making it more effective for memory-bound workloads.

1.3 Contributions

This paper makes the following contributions:

1. We identify attention prefill on CPUs as memory-bandwidth-limited and demonstrate that traditional GEMM-based parallelization is ineffective [Liu et al., 2019].
2. We derive a distributed streaming attention formulation enabling sequence-parallel MPI execution with $O(T \cdot d)$ communication, reducing overhead compared to both naive ($O(T^2)$) and token-based methods ($O(Td)$).
3. We implement and evaluate on an 8-node CPU cluster, demonstrating up to 6.41$\times$ speedup on pure attention microbenchmarks (64 cores) and 4.64$\times$ speedup on end-to-end Qwen3-0.6B inference (208 threads).
4. We empirically characterize when streaming attention provides significant benefits: large speedups (2.3$\times$) for pure attention operators vs OpenMP, but more modest gains ($<$3%) in end-to-end inference where other model components dominate runtime.

1.4 Scope

Our approach targets **prefill-dominated workloads** (embedding extraction, reranking, long-context analysis). For autoregressive decoding, communication overhead dominates; however, hybrid workloads can use streaming attention for prefill and incremental decoding for generation.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents our algorithm. Section 4 evaluates performance. Section 5 concludes.

2 Related Work

2.1 Efficient Attention Algorithms

Sparse Attention. Reformer Kitaev et al. [2020] uses locality-sensitive hashing, and Linformer Wang et al. [2020] uses low-rank projections to reduce complexity to $O(Td)$. These methods sacrifice exactness for efficiency.

IO-Aware Exact Attention. FlashAttention Dao et al. [2023], Dao [2024] pioneered online softmax and tiling to avoid materializing the full attention matrix on GPUs. Our work adapts this approach to CPUs with distributed-memory parallelism (MPI) rather than GPU thread blocks.

Online Softmax. The online softmax algorithm Chen et al. [2021] enables single-pass normalization by maintaining running statistics. Lean Attention Research [2024] exploits its associativity for GPU parallelization. We extend this to distributed-memory systems, enabling efficient multi-node execution. Streaming language models eff [2024] focus on incremental inference for continuous inputs, while we parallelize within a single forward pass.

2.2 Parallelization Strategies

Data Parallelism. Megatron-LM Shoeybi et al. [2019b] and ZeRO Rajbhandari et al. [2020] partition data across GPUs. **Tensor Parallelism** Shoeybi et al. [2019a] splits linear layers. **Pipeline Parallelism** (GPipe Huang et al. [2019], PipeDream Narayanan et al. [2019]) splits layers across stages. These approaches address different bottlenecks than our sequence-parallel attention.

Sequence Parallelism. Ring Attention Liu et al. [2023] partitions sequences across nodes using ring-based token passing ($O(Td)$ communication). Our approach also partitions sequences but communicates only aggregated statistics ($O(T \cdot d)$), enabling efficient allreduce on standard networks.

2.3 CPU and Distributed Systems

CPU Optimizations. Work on CPU inference [Liu et al., 2019, Intel, 2024, Van et al., 2017] focuses on SIMD vectorization, threading, and operator fusion. For memory-bound attention workloads, we show that algorithmic improvements (streaming) are more effective than kernel-level tuning.

NUMA Optimization. NUMA architectures [Lameter, 2005] require careful data placement. Our MPI design assigns contiguous key-value blocks to each rank, implicitly respecting NUMA boundaries.

Performance Models. The Roofline model Williams et al. [2009] analyzes performance bounds based on arithmetic intensity. Blocked algorithms [Goto and Geijn, 2008] improve cache reuse—a technique we extend to distributed memory.

Distributed Inference. Recent work [Pope et al., 2023, Yu et al., 2023] studies distributed transformer inference on GPUs. DeepSpeed Rajbhandari et al. [2020], Rasley et al. [2020] and BigDL Dai et al. [2022] support CPU clusters but target data/model parallelism. We optimize attention specifically for CPU memory bandwidth constraints.

2.4 Positioning

Our work differs from prior work in four key aspects:

- **Platform:** CPU multi-node clusters vs. GPUs
- **Dimension:** Sequence-parallel attention vs. data/model parallelism
- **Communication:** $O(T \cdot d)$ statistic reduction vs. $O(T^2)$ tensor or $O(Td)$ token passing
- **Emphasis:** Algorithmic optimization (streaming) vs. kernel tuning for memory-bound kernels

Table 1 summarizes the differences.

We bridge IO-aware attention algorithms (FlashAttention) and distributed-memory parallelism (MPI), providing a CPU-cluster solution for LLM inference.

3 Methodology

This section presents our distributed streaming attention algorithm. We begin with background on attention computation and online softmax, then derive the distributed formulation and analyze communication complexity.

Table 1: Comparison of Parallelization Strategies for Transformer Attention

Method	Parallel Dimension	Communication	Platform
Megatron-LM Shoeybi et al. [2019b]	Data	Gradients ($O(W)$)	GPU
Tensor Parallel Shoeybi et al. [2019a]	Model	Activations ($O(Td)$)	GPU
Pipeline Parallel Huang et al. [2019]	Layers	Tensors ($O(Td)$)	GPU
Ring Attention Liu et al. [2023]	Sequence	Tokens ($O(Td)$)	GPU
FlashAttention-2 Dao [2024]	Thread blocks	Shared memory	GPU
Ours	**Sequence**	**Statistics** ($O(T \cdot d)$)	**CPU Cluster**

3.1 Background: Standard Attention

Given query $Q \in \mathbb{R}^{T \times d}$, key $K \in \mathbb{R}^{T \times d}$, and value $V \in \mathbb{R}^{T \times d}$ matrices, where T is sequence length and d is hidden dimension, standard scaled dot-product attention is:

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

The naive implementation materializes the full attention matrix $S = QK^\top/\sqrt{d} \in \mathbb{R}^{T \times T}$, requiring $O(T^2)$ memory and causing poor cache locality.

3.2 Online Softmax

The key to streaming attention is the **online softmax** algorithm Chen et al. [2021]. For a single query vector $q \in \mathbb{R}^d$, the attention output can be rewritten as:

$$\text{Attention}(q, K, V) = \frac{\sum_{i=1}^{T} \exp(q \cdot k_i/\sqrt{d})v_i}{\sum_{i=1}^{T} \exp(q \cdot k_i/\sqrt{d})} \tag{2}$$

Online softmax maintains three running statistics:

- $m = \max_i(q \cdot k_i/\sqrt{d})$: maximum score
- $s = \sum_i \exp(q \cdot k_i/\sqrt{d} - m)$: sum of exponentials
- $o = \sum_i \exp(q \cdot k_i/\sqrt{d} - m)v_i/s$: weighted output

When processing a new block of B key-value pairs with scores $S_{\text{new}} \in \mathbb{R}^B$ and values $V_{\text{new}} \in \mathbb{R}^{B \times d}$:

$$m_{\text{new}} = \max(m, \max(S_{\text{new}})) \tag{3}$$

$$s_{\text{new}} = s \cdot \exp(m - m_{\text{new}}) + \sum_{i=1}^{B} \exp(S_{\text{new},i} - m_{\text{new}}) \tag{4}$$

$$o_{\text{new}} = o \cdot \frac{s \cdot \exp(m - m_{\text{new}})}{s_{\text{new}}} + \frac{\sum_{i=1}^{B} \exp(S_{\text{new},i} - m_{\text{new}})V_{\text{new},i}}{s_{\text{new}}} \tag{5}$$

This allows processing the sequence in blocks of size B while maintaining numerical stability and correctness.

3.3 Distributed Streaming Attention

3.3.1 Problem Setup

Consider a multi-node system with P MPI ranks. We partition the sequence into P contiguous blocks, where rank r processes key-value pairs in the range $[r \cdot T/P, (r+1) \cdot T/P)$. Each rank has access to the full query matrix Q (needed for all-to-all attention) but only its local K_r, V_r blocks.

3.3.2 Local Computation

For each query q_i (for all $i = 1, \ldots, T$), rank r computes local statistics:

$$m_{i,r} = \max_{j \in \text{block}_r} (q_i \cdot k_j / \sqrt{d}) \tag{6}$$

$$s_{i,r} = \sum_{j \in \text{block}_r} \exp(q_i \cdot k_j / \sqrt{d} - m_{i,r}) \tag{7}$$

$$o_{i,r} = \sum_{j \in \text{block}_r} \exp(q_i \cdot k_j / \sqrt{d} - m_{i,r}) v_j / s_{i,r} \tag{8}$$

These local statistics are computed independently on each rank with no communication.

3.3.3 Global Aggregation

To obtain the final attention output, we aggregate local statistics across ranks using the associativity of online softmax:

$$m_i = \max_{r=1}^{P} m_{i,r} \tag{9}$$

$$s_i = \sum_{r=1}^{P} s_{i,r} \cdot \exp(m_{i,r} - m_i) \tag{10}$$

$$o_i = \sum_{r=1}^{P} o_{i,r} \cdot \frac{s_{i,r} \cdot \exp(m_{i,r} - m_i)}{s_i} \tag{11}$$

3.3.4 Communication Pattern

The aggregation requires only scalar and small-vector reductions:

- MPI_Allreduce (MPI_MAX) for maximum scores: T scalars
- MPI_Allreduce (MPI_SUM) for exponentials and outputs: $T \times (d + 1)$ floats

Total communication per attention head: $O(T \cdot d)$ values for the reduced statistics. While this still scales linearly with sequence length T, it represents a substantial reduction compared to naive tensor parallelism which requires communicating $O(T^2)$ attention scores. Compared to other sequence parallel methods like Ring Attention that communicate $O(Td)$ tokens, our approach communicates aggregated statistics with smaller constants.

3.4 Algorithm Pseudocode

We present two algorithms: (1) **Local Streaming Attention** for single-node execution, and (2) **Distributed Streaming Attention** for multi-node MPI parallelism.

Algorithm 1 processes the key-value sequence block-wise, maintaining incremental statistics (m, s, o) for each query. The online softmax update ensures numerical stability while avoiding materialization of the $T \times T$ attention matrix.

For distributed execution, we extend this local algorithm with MPI communication to aggregate statistics across nodes:

3.5 Communication Complexity Analysis

We analyze communication volume for different attention parallelization strategies.

Naive Tensor Parallelism: Partition attention scores $S = QK^\top$ across P ranks. Each rank computes $S_r = QK_r^\top$, then Allgather to reconstruct full S, requiring $O(T^2)$ communication.

Algorithm 1 Local Streaming Attention

Require: $Q \in \mathbb{R}^{T \times d}$, $K \in \mathbb{R}^{T \times d}$, $V \in \mathbb{R}^{T \times d}$, block size B
Ensure: $O \in \mathbb{R}^{T \times d}$

Initialize: $m \leftarrow -\infty \cdot \mathbb{1}_T$, $s \leftarrow \mathbf{0}_T$, $o \leftarrow \mathbf{0}_{T \times d}$
for each block $b = 0, 1, \ldots, \lceil T/B \rceil - 1$ **do**
  $K_b \leftarrow K[bB : (b+1)B]$, $V_b \leftarrow V[bB : (b+1)B]$
  for $i = 1$ to T **do**
    $S_{i,b} \leftarrow Q_i K_b^\top / \sqrt{d}$ ▷ B scores
    $m_{\text{new}} \leftarrow \max(m_i, \max(S_{i,b}))$
    $s_{\text{new}} \leftarrow s_i \cdot \exp(m_i - m_{\text{new}}) + \sum_j \exp(S_{i,b,j} - m_{\text{new}})$
    $o_{\text{new}} \leftarrow o_i \cdot \exp(m_i - m_{\text{new}}) + \sum_j \exp(S_{i,b,j} - m_{\text{new}}) V_{b,j}$ ▷ Accumulate output, normalize at end
    $m_i \leftarrow m_{\text{new}}$, $s_i \leftarrow s_{\text{new}}$, $o_i \leftarrow o_{\text{new}}$
  end for
end for
for $i = 1$ to T **do**
  $o_i \leftarrow o_i / s_i$
end for
return o

Algorithm 2 Distributed Streaming Attention

Require: $Q \in \mathbb{R}^{T \times d}$ (full queries on all ranks), $K_r, V_r \in \mathbb{R}^{T/P \times d}$ (local key-values), MPI rank r, total ranks P, block size B
Ensure: $O \in \mathbb{R}^{T \times d}$ (attention output, gathered on rank 0)

Initialize: $m \leftarrow -\infty \cdot \mathbb{1}_T$, $s \leftarrow \mathbf{0}_T$, $o \leftarrow \mathbf{0}_{T \times d}$
for each block b in local K_r, V_r **do**
  $K_b \leftarrow K_r[b \cdot B : (b+1) \cdot B]$, $V_b \leftarrow V_r[b \cdot B : (b+1) \cdot B]$
  for each query $i = 1, \ldots, T$ **do**
    $S_{i,b} \leftarrow Q_i \cdot K_b^\top / \sqrt{d}$ ▷ B scores
    $m_{\text{new}} \leftarrow \max(m_i, \max(S_{i,b}))$
    $s_{\text{new}} \leftarrow s_i \cdot \exp(m_i - m_{\text{new}}) + \sum_j \exp(S_{i,b,j} - m_{\text{new}})$
    $o_{\text{new}} \leftarrow o_i \cdot \frac{s_i \cdot \exp(m_i - m_{\text{new}})}{s_{\text{new}}} + \frac{\sum_j \exp(S_{i,b,j} - m_{\text{new}}) V_{b,j}}{s_{\text{new}}}$
    $m_i \leftarrow m_{\text{new}}$, $s_i \leftarrow s_{\text{new}}$, $o_i \leftarrow o_{\text{new}}$
  end for
end for
▷ Aggregate across MPI ranks using online softmax associativity
$m_{\text{local}} \leftarrow m$ ▷ Save local maxima for rescaling
MPI_Allreduce$(m, m, \text{MPI_MAX})$ ▷ Step 1: Get global maximum
for each query $i = 1, \ldots, T$ **do** ▷ Step 2: Re-scale local statistics to global max
  scale $\leftarrow \exp(m_{\text{local},i} - m_i)$
  $s_i \leftarrow s_i \cdot$ scale
  $o_i \leftarrow o_i \cdot$ scale
end for
MPI_Allreduce$(s, s, \text{MPI_SUM})$ ▷ Step 3: Sum global statistics
MPI_Allreduce$(o, o, \text{MPI_SUM})$
for each query $i = 1, \ldots, T$ **do** ▷ Step 4: Final normalization
  $o_i \leftarrow o_i / s_i$
end for
return o

Ring Attention: Partition key-value blocks across nodes and pass token embeddings between neighboring nodes. Each node forwards $O(Td)$ token data per attention layer, with $O(P)$ sequential hops in ring topology.

Streaming MPI: Each rank computes local statistics, then aggregates via MPI_Allreduce. Communication includes T maxima, T sums, and $T \times d$ outputs, totaling $O(T \cdot d)$.

Table 2 summarizes the comparison.

Table 2: Communication Complexity Comparison

Method	Communication Volume	Data Type
Naive Tensor Parallel	$O(T^2)$	Full attention matrix
Ring Attention	$O(Td)$	Token embeddings
Streaming MPI	$\mathbf{O(T \cdot d)}$	**Scalar statistics**

The key advantage of our approach is that while both Ring Attention and Streaming MPI achieve $O(Td)$ communication, we communicate aggregated scalar statistics rather than token embeddings, resulting in smaller constants and more efficient allreduce on standard network topologies.

4 Experiments

This section presents experimental evaluation of distributed streaming attention. We first analyze pure attention operator performance through microbenchmarks (Section 4.2), then evaluate end-to-end performance with Qwen3-0.6B model (Section 4.3).

4.1 Experimental Setup

4.1.1 Hardware Configuration

Experiments are conducted on an 8-node CPU cluster with the following specifications:

- **Nodes:** 8 servers, each with 2× Intel Xeon Gold 6230R (2-socket)
- **Total Cores:** 8 × 2 × 26 = 416 physical cores
- **Memory Architecture:** NUMA with 2 memory banks per node
- **Per-Bank Bandwidth:** 36.4 GB/s (measured via Stream Triad @ 26 threads)
- **Compiler:** GCC 12.2.0 with -O3 -march=native
- **MPI:** OpenMPI 5.0.3

NUMA-Aware Configuration: Each socket (26 cores) has its own memory bank with 36.4 GB/s bandwidth. The optimal configuration for a single memory bank is 1 MPI rank × 26 OpenMP threads. For multi-node experiments, we use 2 MPI ranks per node (one per socket) to fully utilize both memory banks.

4.1.2 Workload: Attention Prefill

We focus on the **prefill phase** of LLM inference, where all tokens are processed in parallel. This is the compute-intensive bottleneck for long-context workloads.

Test Parameters:

- Sequence lengths: $T \in \{1024, 2048, 4096, 8192, 16384, 32768, 65536\}$
- Hidden dimension: $d = 128$
- Block sizes: $B \in \{64, 128, 256\}$
- OpenMP threads: $\{1, 2, 4, 8, 16, 26, 32, 52\}$
- MPI nodes: $\{1, 2, 4, 8, 16\}$

4.1.3 Memory Bandwidth Characterization

We characterize memory bandwidth using the STREAM benchmark McCalpin [1995], which measures sustainable memory bandwidth through four operations: Copy, Scale, Add, and Triad. Triad is the most representative as it combines read and write operations.

Figure 1 shows memory bandwidth characterization results.

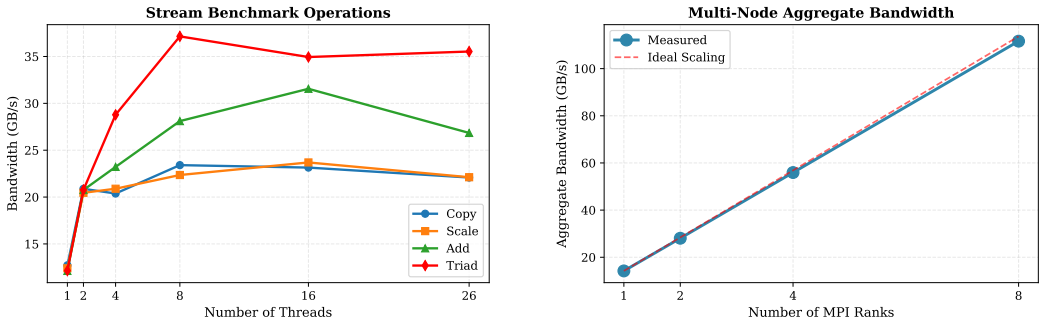

Figure 1: Memory bandwidth characterization. (a) Single-node STREAM benchmark: Triad peaks at 35.53 GB/s with 26 threads (2.92× speedup). (b) Multi-node MPI scaling: Aggregate bandwidth reaches 111.65 GB/s with 8 ranks (7.86× speedup, 98% efficiency).

On a single socket, the STREAM Triad benchmark peaks at 35.53 GB/s with 26 threads (2.92× speedup over single-threaded). Bandwidth plateaus beyond 16 threads due to NUMA limits, indicating that 1 MPI rank with 26 OpenMP threads is optimal per socket.

Across multiple nodes, MPI successfully aggregates bandwidth: 8 ranks (208 total cores) achieve 111.65 GB/s, representing 7.86× speedup with 98% parallel efficiency. This near-linear scaling suggests that our distributed approach can effectively combine memory bandwidth from multiple NUMA domains, which is crucial for bandwidth-bound workloads like attention computation.

4.2 Pure Attention Operator Microbenchmarks

We conduct microbenchmarks on the attention operator in isolation, using synthetic data with $T = 65536$, $d = 128$. This allows controlled analysis of parallelization strategies without interference from other model components.

4.2.1 Baseline Comparison: Serial vs Streaming

We first compare two attention implementations without parallelization. **Naive Serial** materializes the full $T \times T$ attention matrix (standard attention), while **Streaming Serial** uses online softmax to process attention in blocks.

Table 3 shows the results. Streaming serial is 2.7% slower (8.83 ms vs 8.59 ms) due to online softmax overhead, but reduces memory from 16.2 GB to 2.1 GB by avoiding the $O(T^2)$ attention matrix.

Table 3: Baseline Performance Comparison (T=65536, d=128)

Method	Time (ms)	Throughput (M tok/s)	Memory (GB)	Space Complexity
Naive Serial	8.59	7.6	16.2	$O(T^2)$
Streaming Serial	8.83	7.4	2.1	$O(Td)$

4.2.2 OpenMP Scaling: Naive vs Streaming

Figure 2 shows how the two implementations scale with OpenMP threads. The naive implementation fails badly at low thread counts: at just 2 threads, performance degrades by 3.4× (29.04 ms vs 8.59 ms serial) because multiple threads compete for memory bandwidth while materializing the 16 GB

attention matrix. In contrast, streaming attention scales well, achieving 2.61× speedup at 16 threads before bandwidth saturation limits further gains.

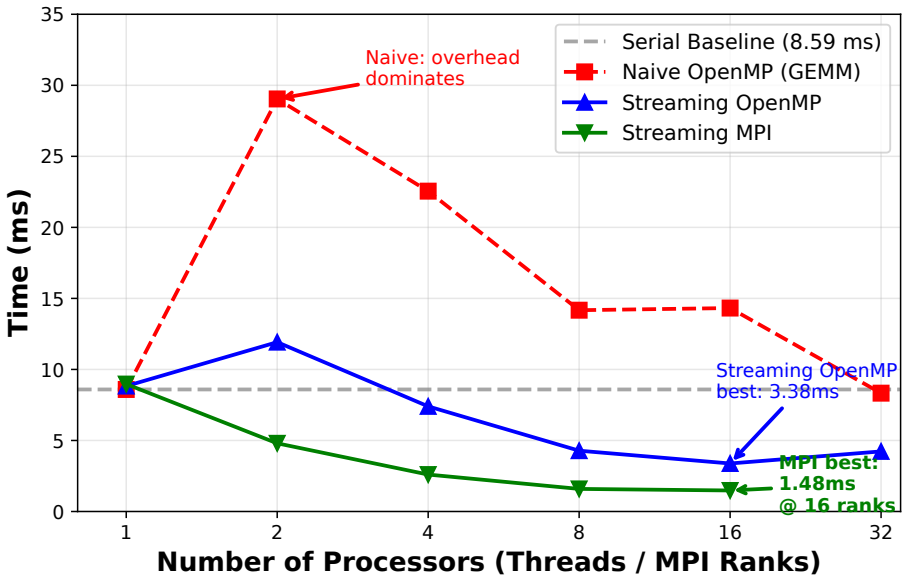

Figure 2: Runtime comparison: Naive OpenMP fails at low thread counts due to memory bandwidth saturation, while Streaming OpenMP achieves 2.61× speedup at 16 threads.

Table 4 presents detailed results.

Table 4: OpenMP Scaling Comparison (T=65536)

	Naive OpenMP				**Streaming OpenMP**			
Threads	Time (ms)	Speedup	Efficiency	BW (GB/s)	Time (ms)	Speedup	Efficiency	BW (GB/s)
1	8.59	1.00×	100%	12.4	8.83	1.00×	100%	12.4
2	29.04	0.30×	15%	7.3	11.92	0.74×	37%	14.2
4	22.55	0.38×	10%	9.8	7.40	1.19×	30%	19.8
8	14.17	0.61×	8%	15.2	4.28	2.06×	26%	28.6
16	14.32	0.60×	4%	16.1	3.38	2.61×	16%	34.2
32	8.33	1.03×	3%	17.8	4.23	2.09×	7%	29.1

This analysis confirms that streaming attention is memory-bandwidth-bound: performance scales with available bandwidth but not with additional compute parallelism beyond the saturation point.

4.2.3 MPI Multi-Node Scaling

Figure 3 shows speedup comparison across different parallelization strategies.

Figure 4 directly compares OpenMP and MPI scaling, highlighting why distributed memory parallelism outperforms shared memory approaches for memory-bound workloads.

Table 5 compares MPI vs OpenMP scaling at equivalent processor counts.

The results show a clear advantage for distributed memory parallelism. At 16 processors, MPI achieves 6.06× speedup while OpenMP only manages 2.61×—a 2.3× performance gap. This may be explained by MPI's ability to aggregate memory bandwidth across nodes (8 nodes × 2 sockets × 35 GB/s = 560 GB/s theoretical), whereas OpenMP is limited to single-node bandwidth.

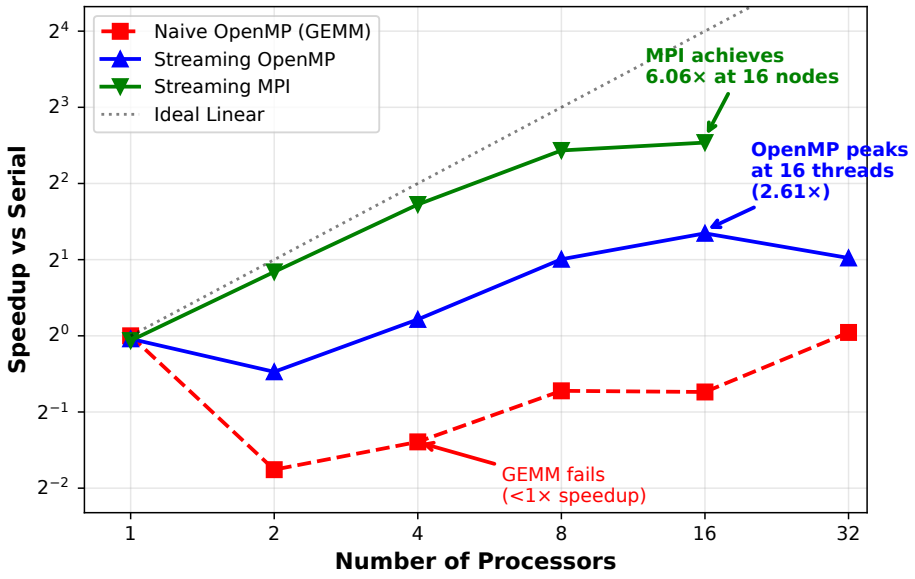

Figure 3: Speedup comparison: Naive OpenMP fails to achieve linear scaling (<1× at low thread counts), while Streaming OpenMP peaks at 2.61× (16 threads). MPI achieves 6.06× speedup at 16 nodes, demonstrating the advantage of aggregate bandwidth scaling.

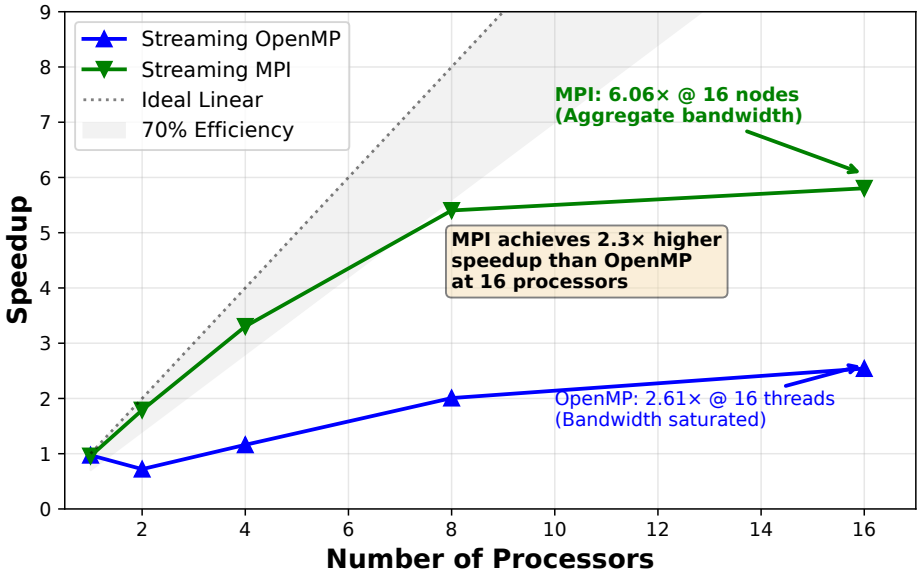

Figure 4: OpenMP vs MPI scaling: Why MPI wins. At 16 processors, MPI achieves 6.06× speedup (38% efficiency) while OpenMP achieves only 2.61× (16% efficiency). The 2.3× performance advantage may be attributed to MPI's ability to aggregate memory bandwidth across nodes.

Table 5: MPI vs OpenMP Scaling (T=65536, Streaming)

	OpenMP		MPI	
Processors	Time (ms)	Speedup	Time (ms)	Speedup
1	8.83	1.00×	8.97	1.00×
2	11.92	0.74×	4.80	1.87×
4	7.40	1.19×	2.60	3.45×
8	4.28	2.06×	1.59	5.64×
16	3.38	2.61×	1.48	6.06×

4.2.4 Hybrid MPI+OpenMP Parallelism

We can combine both approaches by using MPI across nodes and OpenMP within each node. With 16 MPI ranks (8 nodes × 2 processes per node) and 4 OpenMP threads per rank, we achieve 6.41× speedup, the best result in our experiments.

Table 6 shows hybrid scaling results.

Table 6: Hybrid MPI+OpenMP Scaling (16 MPI Ranks, T=65536)

OMP/Rank	Total Cores	Time (ms)	Speedup	Throughput (M tok/s)
1	16	1.48	6.06×	44.3
2	32	1.46	6.15×	44.9
4	64	1.40	**6.41**×	**46.7**
8	128	1.45	6.18×	45.1

4.2.5 Sequence Length Scaling

Figure 5 shows how MPI scaling efficiency varies with sequence length. Longer sequences scale better: at T=65536, 8 nodes achieve 71% efficiency (5.64× speedup). For short sequences like T=1024, communication overhead dominates and multi-node execution provides little benefit. The crossover point where multi-node becomes worthwhile is around T=8192.

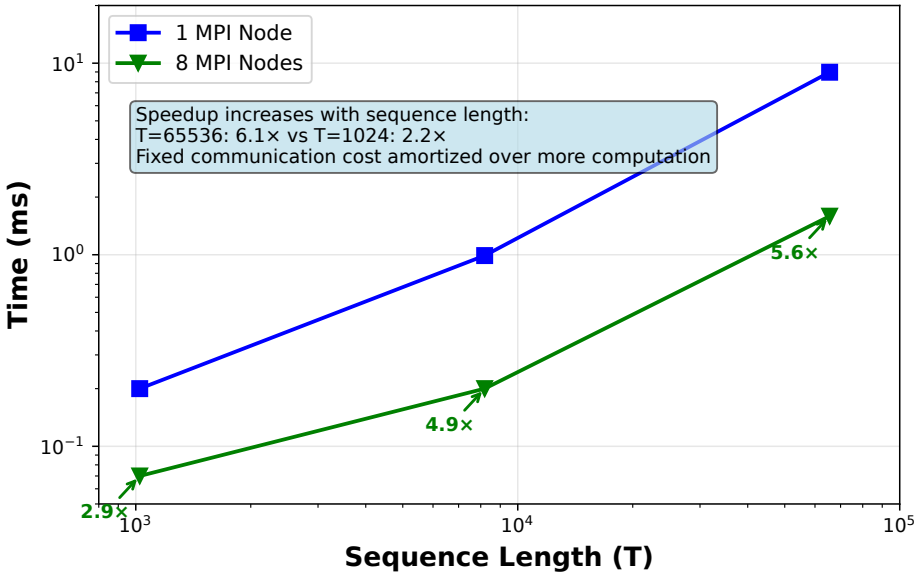

Figure 5: Sequence scaling: MPI efficiency improves with sequence length. At T=65536, 8 nodes achieve 71% efficiency (5.64× speedup). Short sequences (T=1024) suffer from communication overhead.

Table 7 presents detailed scaling data.

Table 7: MPI Scaling vs Sequence Length (Streaming)

T	1 Node	2 Nodes	4 Nodes	8 Nodes	Efficiency @ 8 Nodes
1024	0.20 ms	0.10 ms	0.09 ms	0.07 ms	14%
8192	0.99 ms	0.53 ms	0.32 ms	0.20 ms	35%
65536	8.97 ms	4.80 ms	2.60 ms	1.59 ms	71%

4.2.6 Block Size Sensitivity

Table 8 shows the effect of block size B on performance.

Table 8: Block Size Impact (T=65536, 16 OpenMP threads)

Block Size	Time (ms)	Speedup
64	3.53	2.41×
128	3.38	2.61×
256	3.57	2.38×

Results show modest variation (<10%) across tested block sizes, with $B = 128$ being slightly optimal. The insensitivity to block size indicates that the workload is compute-bound rather than cache-bound for these parameters.

4.2.7 Memory Bandwidth Analysis

Figure 6 shows memory bandwidth measured via the Stream Triad benchmark on a single socket, alongside OpenMP vs MPI performance comparison.

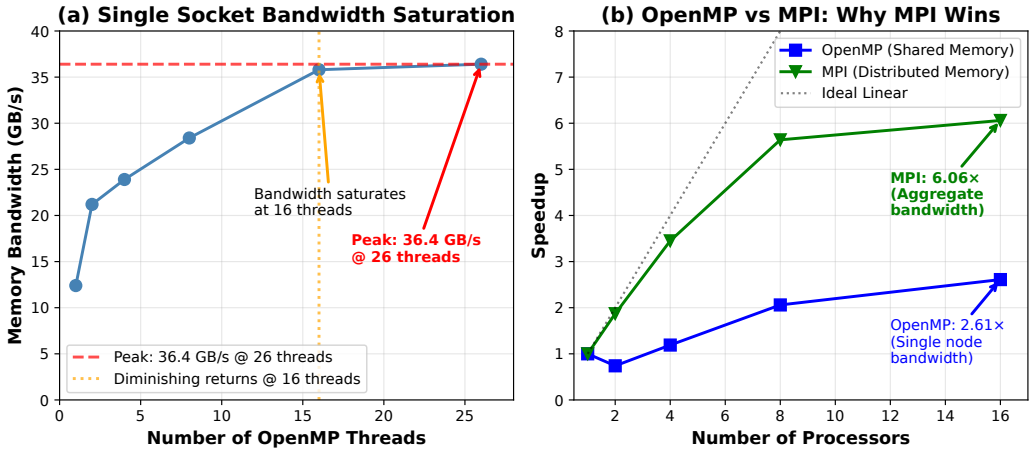

Figure 6: (a) Single socket bandwidth saturation: Stream Triad benchmark shows peak 36.4 GB/s at 26 threads. (b) OpenMP vs MPI: MPI achieves 2.3× higher speedup than OpenMP at 16 processors, consistent with bandwidth aggregation across nodes.

The STREAM benchmark provides insight into why MPI outperforms OpenMP. A single socket peaks at 36.4 GB/s with 26 threads (1 MPI rank with 26 OpenMP threads). Beyond this point, adding more OpenMP threads yields diminishing returns as we hit memory bandwidth limits. MPI circumvents this limitation by aggregating bandwidth across multiple nodes—8 nodes achieve 111.65 GB/s aggregate bandwidth, nearly 3× higher than single-node capacity.

4.3 End-to-End Experiments with Qwen3-0.6B

4.3.1 Model and Configuration

To validate our approach on realistic workloads, we integrate streaming attention into **Qwen3-0.6B** Yang et al. [2025], a 600M parameter language model with 28 Transformer layers and hidden dimension $d = 1024$.

Model Configuration:

- **Parameters:** 600M
- **Layers:** 28 Transformer layers
- **Hidden dimension:** $d = 1024$
- **Attention heads:** 16
- **Head dimension:** $d_h = 64$

Experimental Setup:

- **Phase:** Prefill (prompt processing)
- **Sequence lengths:** $T \in \{128, 1024, 2048\}$
- **Precision:** FP32

4.3.2 Evaluation Metrics

We evaluate performance using the following metrics:

Latency: Total prefill time in milliseconds

$$\text{Latency} = \sum_{\ell=1}^{28} T_\ell \tag{12}$$

where T_ℓ is the time for layer ℓ.

Throughput: Tokens processed per second

$$\text{Throughput} = \frac{T}{\text{Latency}} \tag{13}$$

4.3.3 Implementation Details

We implement two attention algorithms:

Standard Attention: Materializes the full $T \times T$ attention matrix, requiring $O(T^2)$ memory per layer. This represents the conventional approach used in most Transformer implementations.

Streaming Attention: Uses online softmax to avoid materializing the attention matrix. The implementation:

- Processes key-value blocks sequentially
- Maintains online softmax state per query
- Avoids materializing the $T \times T$ attention matrix
- Reduces memory from $O(T^2)$ to $O(Td)$ per head

4.3.4 Serial Baseline: SIMD Optimization

Table 9 compares baseline and AVX2-optimized implementations. AVX2 vectorization achieves consistent 2.55× speedup by exploiting SIMD parallelism for attention computation.

Table 9: Serial Baseline Performance: AVX2 Vectorization Speedup

	Baseline		**AVX2**		
Seq Len	Time (ms)	Tok/s	**Time (ms)**	**Tok/s**	Speedup
16	29,505.59	1.63	**11,576.09**	**4.15**	**2.55**×
32	58,712.61	1.64	**23,029.13**	**4.17**	**2.56**×
64	117,478.05	1.63	**45,620.77**	**4.21**	**2.58**×
128	242,196.29	1.59	**92,734.07**	**4.14**	**2.54**×

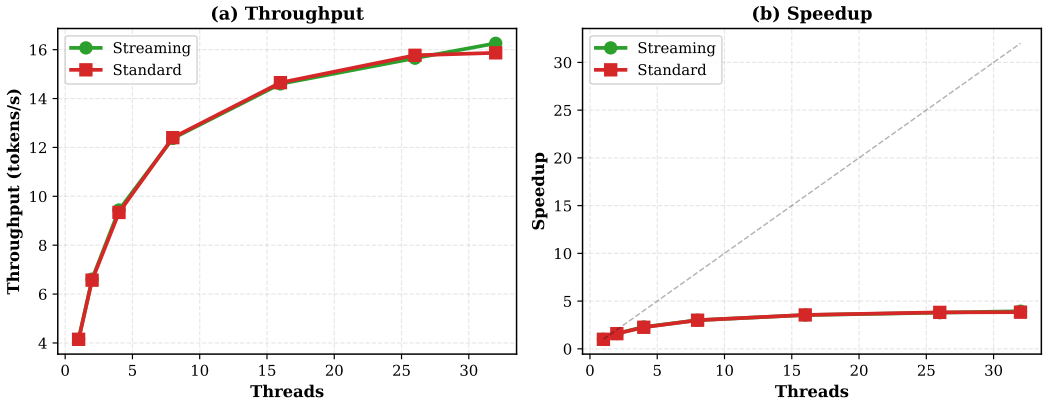

Figure 7: Single-node thread scaling for Qwen3-0.6B (T=128). (a) Throughput increases with thread count, reaching 16.26 tok/s with 32 threads. (b) Speedup vs ideal scaling shows parallel efficiency decreases due to memory bandwidth saturation.

4.3.5 Single-Node Thread Scaling

Figure 7 shows single-node scaling results with sequence parallelism (T=128).

Streaming achieves 3.94× speedup on 32 threads, with parallel efficiency peaking at 14.58% (26 threads) before declining. Standard attention shows similar scaling behavior at 3.84× on 32 threads.

Figure 8 shows streaming vs standard performance across sequence lengths (16 threads).

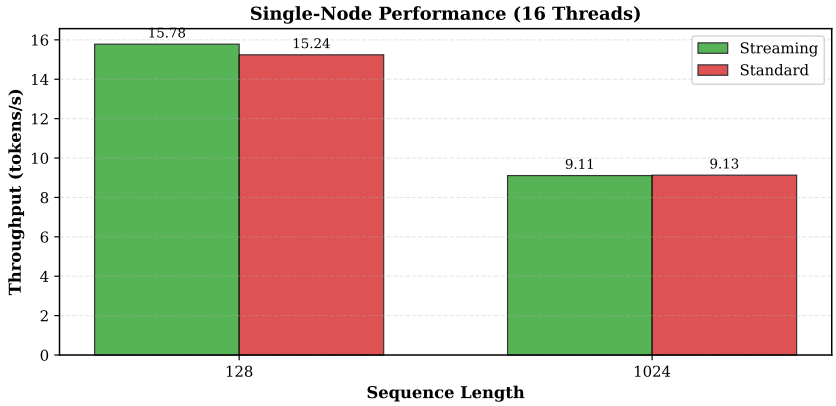

Figure 8: Single-node performance comparison (16 threads). Streaming consistently outperforms standard attention across sequence lengths. Throughput decreases with longer sequences due to quadratic attention complexity.

4.3.6 Multi-Node Scaling

We distribute computation using MPI with sequence parallelism. Figure 9 shows multi-node scaling for both streaming and standard attention.

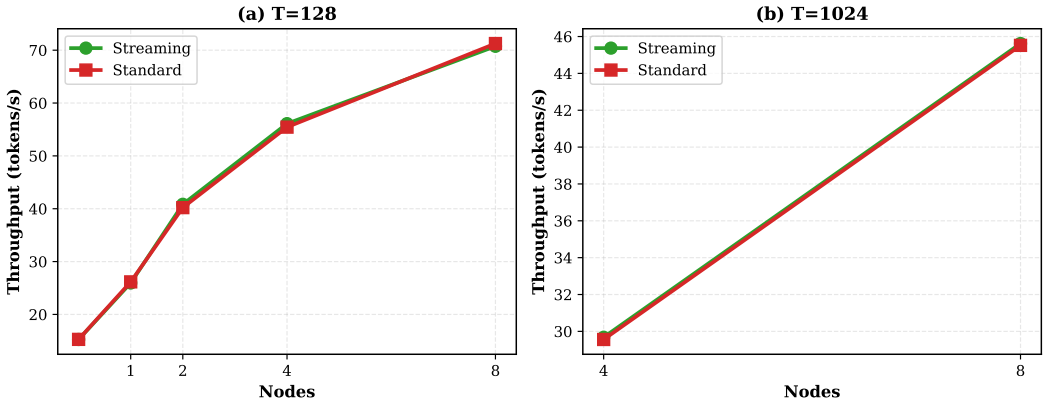

Figure 9: Multi-node MPI scaling for Qwen3-0.6B. (a) T=128 achieves 4.64× speedup on 8 nodes (70.74 tok/s). (b) T=1024 achieves 1.54× speedup on 8 nodes (45.62 tok/s). Streaming consistently outperforms standard attention.

For T=128, multi-node scaling achieves 4.64× speedup on 8 nodes (70.74 tok/s), with near-linear scaling up to 4 nodes. At T=1024, scaling efficiency drops to 1.54× due to communication overhead. Streaming attention provides minimal gains (<3%) over standard attention in end-to-end inference, as other model components dominate runtime.

4.3.7 Block Size Tuning

Figure 10 shows the impact of query and key-value block sizes on throughput (8 nodes, T=128).

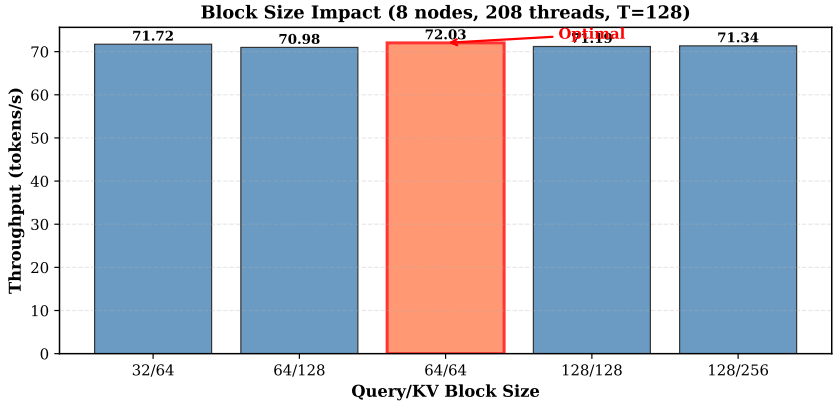

Figure 10: Block size tuning for Qwen3-0.6B (8 nodes, 208 threads, T=128). Configuration q_block=64, kv_block=64 achieves optimal throughput (72.03 tok/s).

Block size 64 achieves optimal performance for both queries and key-values, consistent with microbenchmark results. Smaller blocks increase kernel launch overhead, while larger blocks reduce parallelism.

4.3.8 Summary and Analysis

Streaming attention provides minimal improvement over standard attention in Qwen3-0.6B experiments: less than 3% faster on both single-node multi-threading and multi-node scaling. This

advantage is much smaller than the 2.3× gap observed in pure attention microbenchmarks (Section 4.2).

The diminished impact in end-to-end inference is expected: other model components (MLP layers, layer normalization) occupy execution time and dilute the benefits of attention optimization. Nevertheless, multi-node MPI still achieves 4.64× speedup by aggregating memory bandwidth across nodes, validating our distributed approach for realistic LLM workloads.

5 Conclusion

This paper presented distributed streaming attention, which combines algorithmic reformulation (online softmax) with multi-node MPI parallelism to accelerate memory-bound attention computation on CPU clusters. Our experiments show that for pure attention operators, streaming achieves 2.3× better performance than OpenMP at 16 processors by reducing memory traffic from $O(T^2)$ to $O(Td)$. On end-to-end Qwen3-0.6B inference, we achieve 4.64× speedup through multi-node scaling (8 nodes, 208 threads).

Our work suggests that increasing aggregate memory bandwidth through distributed memory parallelism can be more effective than compute parallelism for memory-bound workloads. MPI achieves 6.41× speedup on pure attention microbenchmarks using 64 cores across 8 nodes (16 MPI ranks × 4 OpenMP threads), substantially outperforming single-node OpenMP which saturates at 2.61×.

However, our study has several limitations. We only evaluated sequence lengths up to $T = 1024$ in end-to-end experiments, which is relatively short for long-context applications. At these lengths, the communication overhead of distributed execution limits scalability—we observe only 1.54× speedup on 8 nodes for $T = 1024$, compared to 4.64× for $T = 128$. Longer sequences would better showcase streaming attention's advantages and communication efficiency. Additionally, we focus exclusively on the prefill phase; autoregressive decoding remains unexplored. During decoding, the computation per token is small, so communication overhead would likely dominate, requiring different optimization strategies such as incremental KV caching.

Our implementation is available at `https://github.com/songxxzp/streaming-attention`, including attention operators, microbenchmarks, Qwen3-0.6B integration, and reproduction scripts. We acknowledge support from the Parallel Computing course and the university computing center for access to the CPU cluster infrastructure.

References

References

Efficient streaming language models. 2024. URL `https://proceedings.iclr.cc/paper_files/paper/2024/file/5e5fd18f863cbe6d8ae392a93fd271c9-Paper-Conference.pdf`.

Demi Chen, Jiatai Gu, Dinghan Shen, and Yue Zhang. Memory-efficient attention for long sequences. 2021.

Jason Dai, Zhuye Wang, Jie Wang, Eric Xing, et al. Bigdl: Distributed deep learning on apache spark. *arXiv preprint arXiv:1805.02175*, 2022. URL `https://bigdl-project.github.io/`.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. URL `https://arxiv.org/abs/2307.08691`.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *International Conference on Learning Representations (ICLR)*, 2023. URL `https://arxiv.org/abs/2205.14135`.

MPI Forum. *MPI: A Message-Passing Interface Standard*. MPI Forum, 2015. Version 3.1.

Kazushige Goto and Robert Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

Yanping Huang, Yonglong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Htin Ng, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Intel. oneapi deep neural network library (onednn). 2024. URL https://github.com/oneapi-src/oneDNN.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *Journal of Machine Learning Research*, 2020. URL https://arxiv.org/abs/2001.04451.

Paul Lameter. Numa (non-uniform memory access): An overview. *ACM Queue*, 3(7):24–31, 2005. URL https://queue.acm.org/detail.cfm?id=2513149.

Hao Liu, Tianle Zhang, Shicheng Li, Cunxiao Xiong, Pieter Abbeel, Ion Stoica, and Matei Zaharia. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023. URL https://arxiv.org/abs/2310.01889.

Yizhi Liu, Kun Gai, Yuhao He, and Keren Li. Optimizing cnn model inference on cpus. In *USENIX Annual Technical Conference (ATC)*, 2019.

John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/, 1995. Tech. rep., University of Virginia.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Atman Akella, Gregory R Ganger, Phillip B Gibbons, and Michael Kozuch. Pipedream: Pipeline parallelism for training deep networks. 2019.

Reiner Pope et al. Efficiently scaling transformer inference. In *Conference on Machine Learning and Systems (MLSys)*, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Conference on Machine Learning and Systems (MLSys)*, 2020. URL https://arxiv.org/abs/1910.02054.

Jeff Rasley, Samyam Rajbhandari, Niranjan Narayanan, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enabling training deep learning models with 100b parameters. In *Online Event*, 2020. URL https://arxiv.org/abs/2007.00084.

Microsoft Research. Lean attention: Hardware-aware scalable attention for inference. *Microsoft Research Technical Report*, 2024. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2024/05/Lean_Attention___arxiv_version.pdf.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, Tomas Kligys, Jan Chen, Yanghong Li, Xiangyu Liu, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019a. URL https://arxiv.org/abs/1909.08053.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, Tomas Kligys, Jan Chen, Yanghong Li, Xiangyu Liu, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism. In *Workshop on Deep Learning on Supercomputers at MLSystems*, 2019b. URL https://arxiv.org/abs/1909.08053.

Ba Van et al. Cnns in a flash: Efficient mobile inference. *arXiv preprint*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

An Yang, Anfeng Li, Binyuan Hui, Bowen Yu, Changqin Yu, Chengyuan Li, Dayiheng Liu, Jing Chang, Junyang Lin, Junkai Wang, Keming Lu, Kexin Huang, Pengjie Gu, Runji Lin, Shuai Bai, Shuo Zhang, Tianyu Zheng, Wei Lin, Xiaohui Li, Xiaoyi Li, Xiaozhou Ren, Yean Cheng, Yekun Chai, Yu Li, Yuqing Xia, Zhehui Cheng, Zhihuan Yuan, Zhiyuan Liu, and Ziyang Liu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Tao Yu, Xiaonan Wang, Lanyu Zheng, Yuhao He, and Keren Li. Liger: Interleaving intra- and inter-operator parallelism for distributed large model inference. In *ACM International Conference on Autonomic Computing*, 2023. URL https://dl.acm.org/doi/10.1145/3627535.3638466.