

# 000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 OUT OF THE MEMORY BARRIER: A HIGHLY MEM- ORY EFFICIENT TRAINING SYSTEM FOR LLMs WITH MILLION-TOKEN CONTEXTS

Anonymous authors

Paper under double-blind review

## ABSTRACT

Training Large Language Models (LLMs) on long contexts is severely constrained by prohibitive GPU memory overhead, not training time. The primary culprits are the activations, whose memory footprints scale linearly with sequence length. We introduce OOMB, a highly memory-efficient training system that directly confronts this barrier. Our approach employs a chunk-recurrent training framework with on-the-fly activation recomputation, which maintains a constant activation memory footprint ( $\mathcal{O}(1)$ ) and shifts the primary bottleneck to the growing KV cache. To manage the KV cache, OOMB integrates a suite of synergistic optimizations: a paged memory manager for both the KV cache and its gradients to eliminate fragmentation, asynchronous CPU offloading to hide data transfer latency, and page-level sparse attention to reduce both computational complexity and communication overhead. The synergy of these techniques yields exceptional efficiency. Our empirical results show that for every additional 10K tokens of context, the end-to-end training memory overhead increases by a mere 10MB for Qwen2.5-7B. This allows training Qwen2.5-7B with a 4M-token context on a single H200 GPU, a feat that would otherwise require a large cluster using context parallelism. This work represents a substantial advance in resource efficiency for long-context LLM training. The source code is available for review at an anonymous repository.

## 1 INTRODUCTION

Long-context modeling remains a formidable challenge in training Large Language Models (LLMs). The principal obstacle is not only training time but also the prohibitive GPU memory overhead associated with long sequences (Liu et al., 2025b; Wang et al., 2024). Research indicates that significant long-context capabilities can be achieved with a surprisingly small number of training steps (Peng et al., 2024; Ding et al., 2024), and that fine-tuning on as little as 5% of the original pretraining data may suffice for commercial-grade performance (Gao et al., 2025; Bai et al., 2024).

Despite this modest data requirement, the number of tokens processed in a single iteration grows dramatically. This causes GPU memory consumption from activations and the KV cache to scale linearly with context length, rapidly exhausting available resources. For instance, a model with a  $4 \times$  GQA (Ainslie et al., 2023) ratio requires 64GB for the KV cache alone when processing a 256K context. This demand overwhelms an A100 GPU before even accounting for other network activations, often making 32K tokens a practical limit for single-GPU training (Grattafiori et al., 2024; Yang et al., 2025). This immense memory pressure renders popular techniques like ZeRO3 (Rajbhandari et al., 2020) and tensor parallelism (Shoeybi et al., 2020) insufficient.

While various training-free methods for context extension exist (Jin et al., 2024; Han et al., 2024), their practical utility is limited. Prior work has established that common evaluation metrics like perplexity and targeted retrieval are unreliable indicators of true long-range reasoning (Gao et al., 2024; Bai et al., 2024). Achieving robust performance necessitates dedicated contiguous pretraining.

To this end, this paper directly confronts the central problem: how to train longer-context models with fewer resources without compromising performance. We introduce **Out Of the Memory Barrier** (OOMB), a chunk-recurrent training framework that processes sequences in segments. During the forward pass, each chunk's activations are computed and then immediately discarded. For

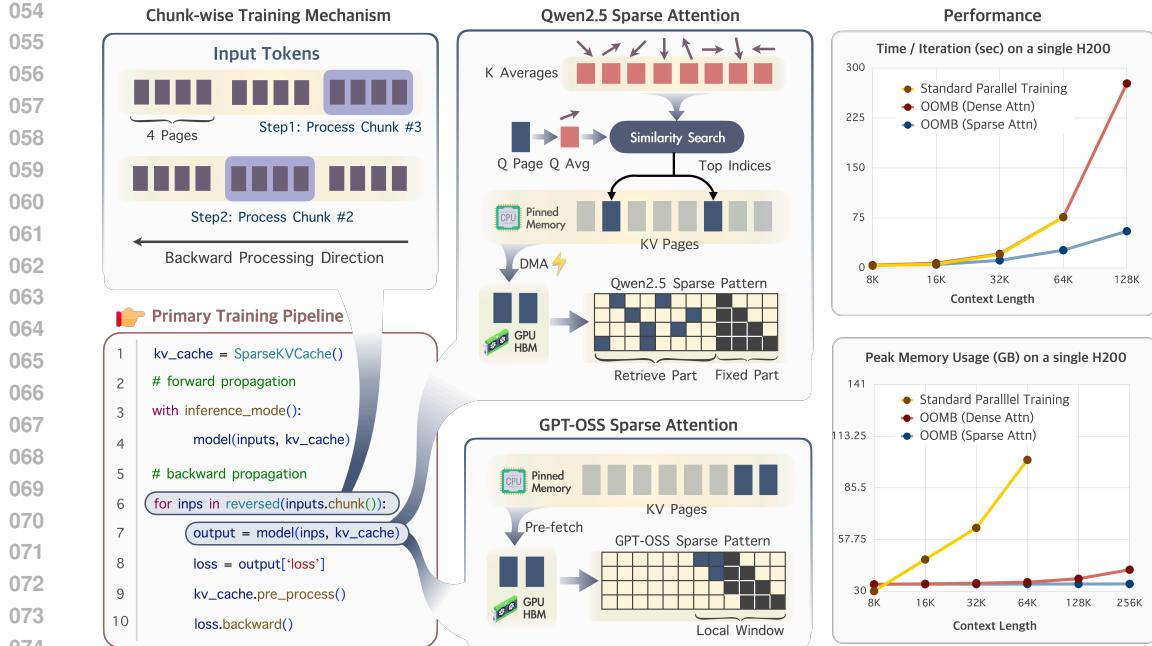


Figure 1: An overview of the OOMB training framework. (Left) OOMB processes sequences chunk-by-chunk with activation recomputation, maintaining a constant activation memory footprint ( $\mathcal{O}(1)$ ) and shifting the bottleneck to the KV cache. (Center) The growing KV cache is managed through a combination of paged memory, asynchronous CPU offloading, and page-level sparse attention. (Right) Performance benchmarks show that the integrated system drastically reduces peak GPU memory, enabling efficient training on long contexts with minimal memory overhead.

the backward pass, they are recomputed on-the-fly. This strategy maintains a constant activation memory footprint, regardless of the total sequence length. This approach, however, shifts the primary bottleneck to the KV cache, which must be retained throughout the training step and scales linearly with context length.

To overcome this challenge, OOMB integrates a system of co-designed optimizations for efficient KV cache management:

- **Paged KV Cache and Gradient Management.** We introduce a paged memory manager, inspired by (Kwon et al., 2023), for both the KV cache and its gradients. This design eliminates expensive memory operations and mitigates fragmentation when appending new key-value pairs.
- **Specialized Kernels.** We implement custom operators that execute all operations on the KV cache and its gradients directly, making them opaque to PyTorch’s autograd system. This avoids storing the KV cache as an activation and allows for direct gradient accumulation.
- **Asynchronous KV Cache CPU Offloading.** We designed an asynchronous mechanism to preemptively offload the KV cache and its gradients, whose memory grows with context length, to CPU memory. The resulting data transfer latency is effectively masked by computation.
- **Page-level Sparse Attention.** Our paged memory architecture natively supports page-level sparse attention (Yuan et al., 2025; Lu et al., 2025). This reduces the computational complexity of attention and minimizes the communication overhead of our offloading mechanism.

The synergy of these techniques yields exceptional GPU memory efficiency. Our results show that for every additional 10K context tokens, the end-to-end training memory overhead increases by a mere 10MB for Qwen2.5-7B. This allows training Qwen2.5-7B with a 4-million-token context on a single H200 GPU. In contrast, context parallelism methods (Liu et al., 2024; Lin, 2025; Shoeybi et al., 2020) require large clusters for similar tasks (Xu et al., 2025), highlighting a substantial advance in resource efficiency.

108 Table 1: Comparison of Long-Context Training Methods. OOMB achieves  $\mathcal{O}(1)$  activation memory  
 109 complexity, where  $N$  denotes the context length. This key advantage allows OOMB, when combined  
 110 with other techniques, to practically enable single-GPU training on contexts exceeding 4M tokens.

Method	GPUs	Activation	Exact Attn	Max Tokens
SeCO (Li et al., 2025)	1×H200	$\mathcal{O}(1)$	✓	128K
LongLoRA (Chen et al., 2024)		$\mathcal{O}(N)$	✗	128K
ZERO3 Offload		$\mathcal{O}(N)$	✓	128K
Ring Flash Attention	8×H200	$\mathcal{O}(N/8)$	✓	256K
Tensor Parallelism		$\mathcal{O}(N/8)$	✓	256K
OOMB + Dense Attn	1×H200	$\mathcal{O}(1)$	✓	4M
OOMB + Sparse Attn		$\mathcal{O}(1)$	✗	4M+

## 2 RELATED WORK

**Efficient Long-Context Extension.** A core challenge in extending language models to long contexts is their failure to generalize to positional encodings not seen during pretraining (Han et al., 2024). While continued training can resolve this issue, the associated GPU memory costs are often prohibitive. This has motivated training-free context extension methods, which typically modify positional encodings to accommodate longer sequences (Chen et al., 2023; Han et al., 2024; Jin et al., 2024; An et al., 2024). However, these approaches introduce significant trade-offs. Strategies based on interpolation or extrapolation of positional encodings can degrade performance on short-text tasks by reducing the resolution of positional information (Gao et al., 2024). Others that reuse position indices achieve high scores on perplexity benchmarks but have been shown to fail on tasks requiring deep contextual understanding, indicating a superficial grasp of the extended context (Gao et al., 2024).

Given these limitations, state-of-the-art models still rely on fine-tuning to expand their context windows (Grattafiori et al., 2024; Yang et al., 2025; Liu et al., 2025a). The prevailing trend has shifted from algorithmic efficiency (Chen et al., 2024) toward overcoming memory barriers with massive computational resources (Liu et al., 2024; Li et al., 2024; Jacobs et al., 2024; Shoeybi et al., 2020). For instance, recent efforts have required a 256-GPU cluster to extend a model’s context window to four million tokens (Xu et al., 2025).

**Serial vs. Parallel Training Paradigms.** Parallel training processes an entire sequence in a single pass, maximizing GPU utilization and scalability. However, its memory footprint scales linearly with sequence length, creating a significant bottleneck for long-context models. In contrast, serial training is highly memory-efficient, as it only activates a small part of the network at any given time, but at the cost of reduced parallelism.

For long-context fine-tuning, where dataset sizes are often moderate, memory consumption rather than throughput is the primary constraint (Peng et al., 2024; Gao et al., 2024). This observation motivates a hybrid approach that processes sequences in chunks: parallel within each chunk and serial between them. This strategy is standard in modern LLM inference engines (Kwon et al., 2023; Ye et al., 2025) where it incurs negligible latency. Its application to training, however, remains underdeveloped. An early exploration, SeCO (Li et al., 2025), introduced chunk-wise training but lacked critical optimizations at the operator and memory management levels. Consequently, it could only train a 16K context model on a single consumer-grade GPU. Under identical conditions, our fully-optimized system extends this capability by more than tenfold.

**Context Parallelism.** Context Parallelism (CP) and our proposed method both segment long sequences but diverge significantly in their scalability and overhead profiles. CP, inspired by Ring Attention (Liu et al., 2024) and implemented in frameworks like Megatron-LM (Shoeybi et al., 2020) and DeepSpeed Ulysses (Jacobs et al., 2024), distributes a single sequence across multiple GPUs. This design introduces substantial inter-GPU communication overhead that scales with the number of devices. In contrast, the overhead of our serial approach is limited to repeated KV cache accesses and the latency from CPU offloading. The trade-off is clear: CP excels with large GPU clusters and specialized high-bandwidth networks, whereas our method maintains a moderate and predictable overhead, eliminating the need for such extensive infrastructure.

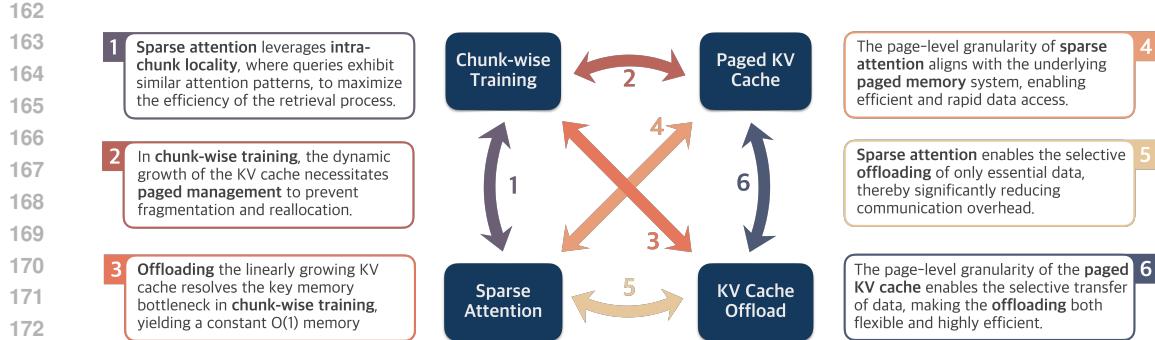


Figure 2: The synergistic architecture of OOMB’s efficient KV cache management relies on four deeply interdependent core components: chunk-wise training, paged key-value cache, sparse attention, and key-value cache offload. Each optimization enables and enhances the others, collectively resolving the memory bottleneck of long-context training.

### 3 PRELIMINARY: CHUNK-WISE TRAINING

The core strategy of OOMB is rooted in the principles of sequential processing, which have long been used to make training Recurrent Neural Networks (RNNs) memory efficient. We first revisit this foundation and then describe how it can be adapted to the Transformer architecture, setting the stage for the challenges that OOMB is designed to solve.

**Activation Recomputation in Sequential Models.** In a standard RNN, the forward pass iteratively computes an output  $y_i$  and a new hidden state  $m_i$  at each timestep  $i$ :

$$y_i, m_i, \mathbf{a}_i \leftarrow \mathcal{F}_{\text{RNN}}(x_i, m_{i-1}; \Theta) \quad (1)$$

where  $\mathbf{a}_i$  represents the intermediate activations cached for the backward pass. For long sequences, storing all activations  $\{\mathbf{a}_1, \dots, \mathbf{a}_T\}$  creates a memory bottleneck that scales linearly with sequence length. Activation recomputation (Sohoni et al., 2022; Bencheikh et al., 2024) circumvents this by discarding  $\mathbf{a}_i$  during the forward pass and regenerating it on-the-fly just before its use in the backward pass. This trades a modest amount of computation for a significant reduction in memory.

**Adapting Sequential Processing to Transformers.** While Transformers are inherently parallel, they can be trained using a similar chunk-wise, sequential paradigm. We partition a long input sequence into  $S$  chunks,  $\{X_1, X_2, \dots, X_S\}$ , and process them serially. The forward pass for chunk  $i$  attends to the KV caches from all preceding chunks, which act analogously to an RNN’s hidden state:

$$Y_i, M_i, \mathbf{A}_i = \mathcal{F}_{\text{Transformer}}(X_i, \{M_1, M_2, \dots, M_{i-1}\}; \Theta) \quad (2)$$

Here,  $M_i$  is the KV cache generated by chunk  $i$ , and  $\mathbf{A}_i$  represents its intermediate activations. By applying the same activation recomputation strategy, we discard  $\mathbf{A}_i$  after the forward step and recompute it for the backward pass:

$$d\Theta, \{dM_1, \dots, dM_{i-1}\} \leftarrow \text{backward}(dY_i, dM_i, \mathbf{A}_i) \quad (3)$$

This chunk-recurrent approach ensures that the activation memory footprint remains constant, determined only by the size of a single chunk. Consequently, this paradigm fundamentally shifts the memory bottleneck: the challenge is no longer the activations, but the management of the KV cache  $\{M_1, \dots, M_S\}$  and its gradients, which still grow linearly with the sequence length.

### 4 THE OOMB TRAINING SYSTEM

OOMB builds upon the chunk-wise training paradigm by integrating three synergistic components: paged KV cache management, sparse attention, and asynchronous KV cache offloading. As illustrated in Figure 2, these components are highly interdependent, and their combined effect is critical to the system’s overall efficiency. This section provides a detailed description of each component.

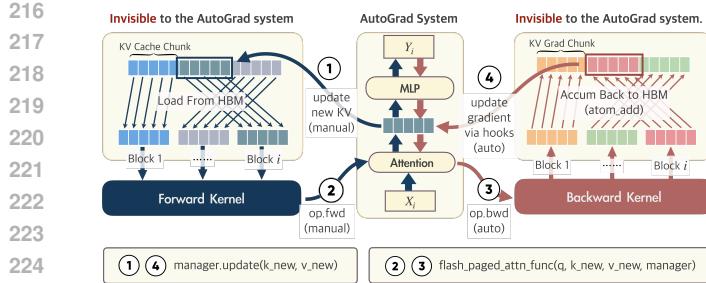


Figure 3: Our paged KV cache management system for training. A custom kernel handles both forward and backward passes, updating gradients in-place to bypass the PyTorch autograd system for greater efficiency.

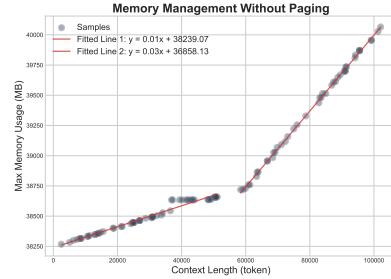


Figure 4: A non-paged implementation can result in memory usage nearly 3 $\times$  higher than the theoretical requirement.

#### 4.1 PAGED MEMORY MANAGEMENT FOR KV CACHE AND GRADIENTS

Chunk-wise training requires the KV cache to grow incrementally, a process analogous to autoregressive decoding. Standard tensor concatenation for this purpose is inefficient, leading to frequent memory reallocations, copy operations, and severe memory fragmentation (Kwon et al., 2023). As shown in Figure 4, this inefficiency causes memory consumption to grow disproportionately with context length, limiting scalability.

To resolve this, OOMB incorporates a paged memory manager for both the KV cache and its gradients. Since existing paged attention solutions are designed for inference and lack backpropagation support, we developed a custom Triton kernel that handles both forward and backward passes (Figure 3).

Our implementation is designed to bypass the PyTorch autograd engine. This approach provides two key benefits. First, custom CUDA kernels accumulate gradients in-place using `atomic_add`, which eliminates intermediate buffers and reduces memory I/O. Second, decoupling the KV cache from the autograd graph prevents PyTorch from storing it as an activation. This design simplifies memory management and facilitates efficient offloading by maintaining a leaner computation graph.

#### 4.2 SPARSE ATTENTION

The quadratic complexity of attention is a primary bottleneck in long-context training. OOMB addresses this by integrating page-level sparse attention, which reduces computational cost and is naturally supported by our paged KV cache architecture (Chen et al., 2024; Li et al., 2025). We implement two variants of sparse attention tailored to different model architectures.

**Approximating Dense Attention for Qwen2.5.** For models with dense attention like Qwen2.5 (Yang et al., 2025), we employ a Top-K page retrieval method to approximate the full attention pattern (Tang et al., 2024; Lu et al., 2025). The selected page indices from the forward pass are cached for reuse during backpropagation. The retrieval process begins by computing a single representative vector  $K_{\text{avg}} \in \mathbb{R}^{n \times D}$  for each of the  $n$  key pages by averaging. Given queries  $Q \in \mathbb{R}^{(m \times P) \times D}$  from the current chunk, where  $m$  is the number of query pages in the chunk and  $P$  is the page size (tokens per page), we compute similarity scores between each query token and all page-representative vectors. A voting mechanism then aggregates these scores for each of the  $m$  query pages to identify the most relevant key pages:

$$\text{Score}[i, j, k] \leftarrow Q[i, j]^T K_{\text{avg}}[k], \quad \text{Score} \leftarrow \text{softmax}(\text{Score}), \quad \text{Score} \leftarrow \text{sum}(\text{Score}). \quad (4)$$

This retrieval process, visualized in Figure 5, incurs negligible computational overhead with page sizes ( $P$ ) of 128 on H200 GPUs.

**Native Sparse Attention for GPT-OSS.** For natively sparse models like GPT-OSS (Agarwal et al., 2025), which alternate between dense and local attention layers, our implementation is straightforward. The dense layers use the same Top-K retrieval method described above, while the local attention layers retrieve only the most recent KV pages.

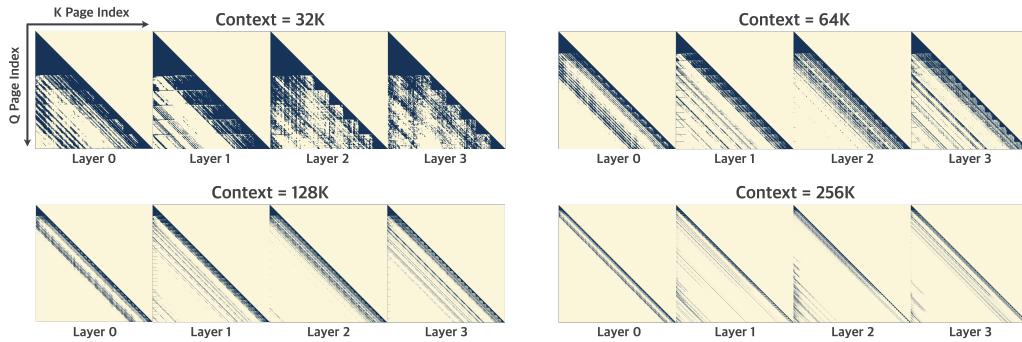


Figure 5: Visualization of the page-level sparse attention patterns for the Qwen2.5-7B model. Each subplot displays the retrieved key pages (x-axis) for each query page (y-axis) across the initial four layers at context lengths scaling from 32K to 256K.

### 4.3 ASYNCHRONOUS KV CACHE OFFLOADING

The KV cache is the only component in OOMB whose memory footprint scales with sequence length. To enable training on extremely long contexts, we introduce an asynchronous offloading mechanism that transfers the KV cache between GPU and CPU memory. The offloading strategy is adapted to the attention type to maximize the overlap between data transfer and computation.

**Offloading for Dense and Local Attention.** For these attention patterns, the required KV pages for a given layer are known in advance. We employ a pre-fetching strategy. During the forward pass for layer  $i - 1$ , the KV cache for layer  $i$  is asynchronously fetched from the CPU. A symmetric process occurs during the backward pass. This approach effectively hides most of the data transfer latency, resulting in an end-to-end overhead lower than 5%.

**Offloading for Sparse Attention.** In retrieval-based sparse attention, the required KV pages are not known until the query vectors for the current layer have been computed. Consequently, during the forward pass, we initiate page retrieval and the corresponding asynchronous data transfer immediately after the query projection. The small volume of data required by sparse attention allows this transfer to be overlapped with the subsequent key and value projection computations. For the backward pass, the required page indices are already cached from the forward pass, allowing us to use the same pre-fetching strategy as in dense attention.

Both offloading schemes, illustrated in Figure 6, are implemented using pinned CPU memory and dedicated CUDA streams to ensure true asynchronicity. Data transfers are handled by the GPU’s Direct Memory Access (DMA) engine to minimize latency.

## 5 EXPERIMENTS

### 5.1 EXPERIMENTAL SETUP

Unless otherwise specified, all experiments adhere to the following configuration.

**Model and Dataset.** We use the Qwen2.5-7B model (Yang et al., 2025), a powerful foundation model, for all efficiency and scalability benchmarks. The training data is sourced from the arXiv dataset (Soboleva et al., 2023). To construct sequences of the required lengths for our long-context experiments, we concatenate samples from the dataset.

**Baseline Methods.** For efficiency evaluation, we compare our method against two baselines: (1) a standard parallel training setup using FlashAttention (Dao, 2024) and layer-wise gradient checkpointing; and (2) Ring Flash Attention (RFA) (Lin, 2025), a state-of-the-art implementation of Ring Attention (Liu et al., 2024).

Additional details are provided in Appendix B.

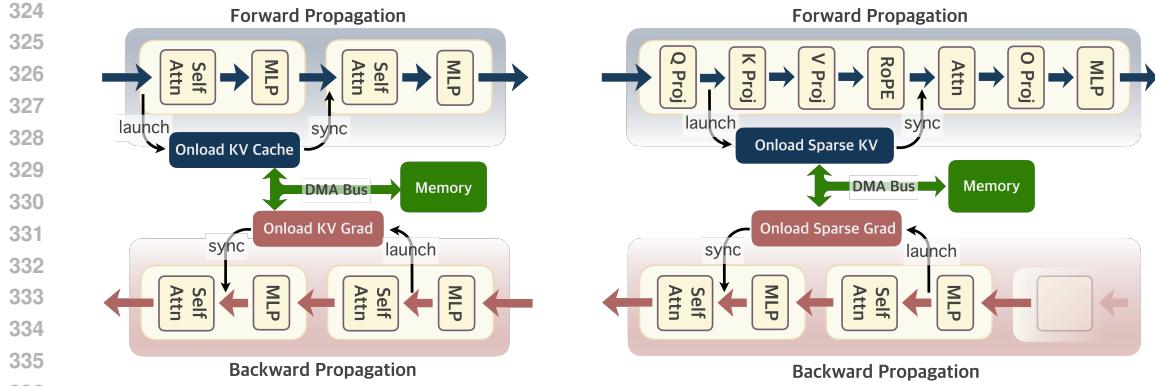


Figure 6: Asynchronous KV cache offloading in OOMB. (Left) For dense and local attention, the KV cache for the next layer is pre-fetched during computation of the current layer. (Right) For sparse attention, the forward pass overlaps data transfer with key-value projections, while the backward pass reverts to the pre-fetching strategy.

## 5.2 MEMORY, TIME, AND SCALABILITY ANALYSIS

We evaluate OOMB’s efficiency against two paradigms: standard parallel training and context parallelism.

**Comparison with Parallel Training.** Table 2 presents a comprehensive efficiency comparison between OOMB and a parallel training baseline using FlashAttention. Our analysis highlights several key findings:

- At a 32K context length, OOMB with dense attention already outperforms the baseline, and this performance advantage widens as the context length increases.
- The asynchronous offloading mechanism imposes negligible latency overhead on dense attention, demonstrating its high efficiency.
- Sparse attention provides substantial acceleration on longer sequences, yielding speedups of approximately 3.7 $\times$  at 64K and up to 13.5 $\times$  at 256K.
- For sparse attention with offloading, a larger chunk size mitigates latency; a 12K chunk size, for example, adds only a 12% overhead.
- Our offloading system dramatically improves memory scalability for both dense and sparse attention, keeping peak memory usage nearly constant irrespective of context length.
- Even without CPU offloading, OOMB reduces the memory growth rate by more than fivefold compared to the parallel training baseline.

**Comparison with Context Parallelism Methods.** We also benchmarked OOMB against Ring Flash Attention (RFA) (Lin, 2025), a state-of-the-art implementation of Context Parallelism. This paradigm manages memory by splitting sequences across GPUs but introduces significant inter-device communication overhead. As shown in Table 3, OOMB achieves higher per-device training throughput, outperforming the heavily optimized RFA.

## 5.3 ACCURACY VALIDATION OF SPARSE ATTENTION

We empirically validated the accuracy of our sparse attention mechanism and its impact on model training. First, we analyzed the gradient approximation error for Qwen2.5-7B (Yang et al., 2025) across 16 configurations, composed of 4 context lengths and 4 retrieval budgets. As shown in Figure 7 (Top-right), within the model’s 128K native context limit, a larger retrieval budget results in a smaller approximation error. Beyond this limit, the error differences between budgets become less significant. The loss values for these settings, presented in Table 4a, further indicate that sparse attention does not substantially degrade model performance.

378  
379  
380  
381  
382  
383  
384 Table 2: Performance and Ablation of OOMB for Qwen2.5-7B on an H200 GPU. Per-iteration latency  
385 and peak memory for various OOMB configurations (ablatiing chunk size, token budget, gradient  
386 checkpointing, and CPU offload) and a FlashAttention (Dao, 2024) baseline. **Red intensity** indicates:  
387 (1) for latency, the slowdown vs. the Ckpt-only counterpart; (2) for memory, the growth rate as  
388 context length increases. Single-GPU offload results are reported under full 8-GPU system load.

Method	Budget	Configuration		Per-iteration Latency (s)						Peak Memory Usage (MB)					
		Ckpt.	Offload	8K	16K	32K	64K	128K	256K	8K	16K	32K	64K	128K	256K
Baseline	-	✓	-	2.4	5.7	20.4	76.1	OOM	OOM	30298	47025	64985	1002546	OOM	OOM
<b>▼ Chunk Size = 4K</b>															
OOMB + Dense Attn	-	✓	✗	2.6	6.0	19.3	68.9	260.6	1015.9	34310	36326	39804	44390	55143	76650
		✗	✓	2.4	6.0	18.6	63.4	233.9	899.6	57369	57529	57743	58175	59065	60788
		✓	✓	3.0	6.7	20.7	70.7	260.4	985.8	33686	33848	34076	34504	35404	37129
OOMB + Sparse Attn	4K+512	✓	✗	2.7	4.4	8.8	18.0	36.6	75.7	34317	36335	39826	44409	55177	76719
		✗	✓	3.0	4.5	9.5	19.6	40.0	83.3	57348	57372	57377	57380	57410	57448
	4K+2048	✓	✓	2.8	5.2	10.8	22.6	46.5	96.9	33669	33694	33999	33712	33732	33769
		✓	✗	2.7	4.9	10.6	21.9	44.9	93.2	34317	36335	39827	44411	55178	76723
		✗	✓	2.9	5.5	11.5	24.5	51.4	107.2	57348	57401	57413	57456	57479	57515
OOMB + Sparse Attn	4K+ 8192	✓	✓	3.1	6.2	13.4	28.6	60.9	127.5	33669	33722	33735	33778	33800	33837
		✓	✗	2.6	5.9	14.3	31.5	66.2	138.1	34317	36335	39827	44413	55183	76733
	4K+ 32768	✗	✓	2.9	6.7	16.2	37.0	83.4	178.6	57348	57421	57487	57614	57647	57752
		✓	✓	2.9	7.4	18.2	42.8	94.0	201.4	33669	33743	33809	33857	34083	34083
		✓	✗	2.5	6.0	19.3	61.1	148.1	324.9	34317	36335	39826	44416	55196	76768
OOMB + Sparse Attn	8K+ 32768	✗	✓	2.4	6.4	20.6	67.1	154.7	392.1	57348	57421	57520	57716	57716	58238
		✓	✓	3.0	7.2	22.7	79.8	187.7	427.4	33669	33743	33842	34036	34426	34503
<b>▼ Chunk Size = 8K</b>															
OOMB + Dense Attn	-	✓	✗	2.8	5.7	19.0	68.4	259.0	1014.5	31628	40795	43483	48860	59613	81119
		✗	✓	2.3	5.5	17.6	64.3	237.4	922.2	71025	85705	85898	86282	87051	86589
OOMB + Sparse Attn	8K+512	✓	✓	3.0	6.1	19.7	70.6	264.3	1021.4	30374	38293	38485	38870	39639	41177
		✗	✓	2.5	5.1	11.4	24.1	49.6	101.9	31826	40796	43488	48872	59639	81181
	8K+2048	✓	✓	2.2	5.5	12.1	25.3	52.2	107.8	70979	85602	85624	85660	85694	85726
		✓	✓	2.5	6.3	13.4	28.7	59.9	123.7	30380	38193	38215	38253	38284	38270
		✓	✗	2.5	5.4	12.4	26.3	54.4	111.6	31826	40797	43489	48873	59640	81183
OOMB + Sparse Attn	8K+ 8192	✗	✓	2.4	5.4	12.6	27.9	58.2	120.7	30380	38201	38226	38262	38317	38364
		✓	✓	2.6	6.6	14.7	31.9	67.3	142.1	30380	38204	38286	38319	38427	38608
	8K+ 32768	✓	✗	2.7	5.8	16.0	36.6	78.1	162.6	31826	40797	43488	48874	59646	81192
		✗	✓	2.4	5.7	16.3	38.7	85.2	183.4	70979	85613	85695	85813	85952	85990
		✓	✓	3.0	6.4	17.9	41.7	94.0	204.7	30380	38204	38304	38508	38731	38856
<b>▼ Chunk Size = 12K</b>															
OOMB + Dense Attn	-	✓	✗	2.8	6.1	19.6	70.9	262.5	1024.0	31627	44589	46605	52654	62735	84913
		✗	✓	2.3	5.7	18.9	68.2	258.7	1017.2	71306	113827	115169	114398	115190	117984
OOMB + Sparse Attn	12K+512	✓	✓	2.8	6.1	19.6	70.9	262.5	1024.0	31575	42705	42879	44081	45858	45858
		✗	✓	2.7	4.7	12.5	28.5	60.5	125.3	31626	44589	46609	52064	62760	84973
	12K+2048	✓	✓	2.4	4.7	12.3	27.9	58.2	107.8	70979	113753	113765	113810	113881	113890
		✓	✓	2.9	5.4	14.0	32.8	72.8	146.4	30379	42639	42681	42698	42737	42774
		✓	✗	2.8	4.9	13.5	31.3	66.7	138.1	31626	44589	46609	52065	62761	84975
OOMB + Sparse Attn	12K+ 8192	✗	✓	2.4	5.1	13.9	32.8	69.8	146.7	70979	113753	113817	113880	113923	113963
		✓	✓	3.0	5.7	15.2	36.4	78.9	163.5	30379	42609	42703	42766	42809	42845
	12K+ 32768	✓	✗	2.8	5.7	17.6	43.4	90.6	188.7	31626	44593	46609	52067	62765	84985
		✗	✓	2.4	5.8	18.4	43.6	90.5	197.9	70979	113753	113830	113997	114017	114138
		✓	✓	3.0	6.6	21.5	48.2	105.1	224.1	30379	42639	42716	42860	42968	43018

417  
418  
419  
420  
421 Table 3: Per-device training throughput (tokens/sec) comparison against context-parallel methods.  
422 All OOMB configurations use a 4K chunk size with checkpointing and CPU offload; the sparse  
423 attention version adds an 8192 retrieval budget. TP: Tensor Parallel.

Context	Ring Attn (report)		4×H200	OOMB (Dense Attn)		OOMB (Sparse Attn)	
	8×A100	16×TPUv4		1×H200	4×H200 (TP)	1×H200	4×H200 (TP)
64K	-	-	872.51	936.22	933.84	1560.38	1542.85
128K	-	-	463.37	504.12	500.26	1394.16	1371.41
256K	50	49	218.41	266.13	261.47	1301.60	1265.59

426  
427  
428 Next, we performed training runs at 256K and 1M context lengths using 8K and 32K retrieval budgets.  
429 The results in Figure 7 (*Top-left*) show that at 256K, both budgets yield rapid convergence to a  
430 loss below 1.2 with highly similar loss curves. At a 1M context, the 32K budget exhibited minor  
431 instability late in training, an effect we attribute to the learning rate and limitations in positional  
432 encoding extrapolation. Nevertheless, both configurations converged successfully to low loss values.

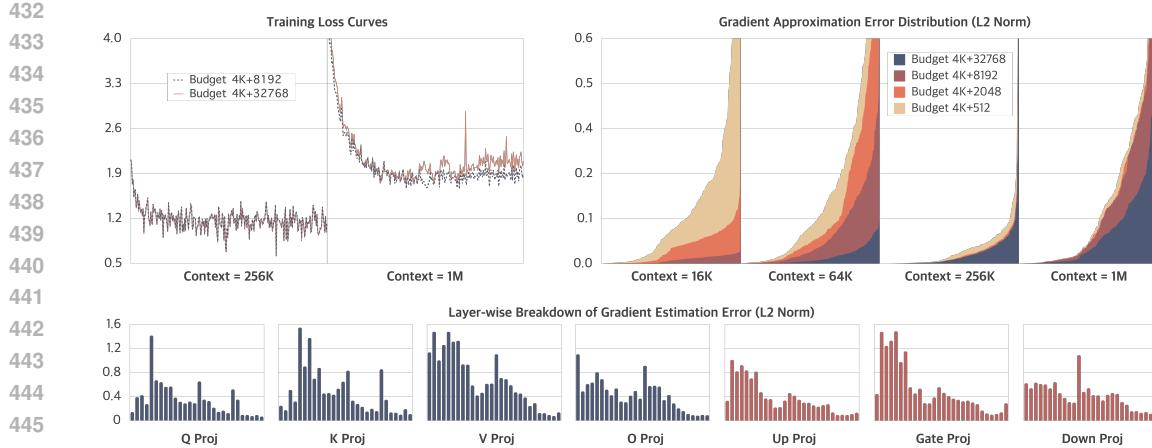


Figure 7: (Top-left) Training loss curves for Qwen2.5-7B (Yang et al., 2025) with 256K and 1M context lengths, employing sparse attention with retrieval budgets of 4K+8192 and 4K+32768. (Top-right) Distribution of the L2 norm of gradient approximation error for sparse attention across various context lengths (16K to 1M) and retrieval budgets. (Bottom) Layer-wise L2 norm of the gradient estimation error for different projection matrices, analyzed with a 64K context length and a 4K+8192 retrieval budget. Our work offers a preliminary validation of this sparse training; for more optimal sparse attention strategies, we refer readers to the established literature.

As detailed in Table 4b, this approach is practical, allowing a 1M-context model to be trained for 256 steps in approximately 10 hours on a  $4 \times \text{H200}$  system with tensor parallelism.

Collectively, these findings confirm that sparse attention is a viable and effective method for long-context training, as it preserves model performance while achieving a constant  $\mathcal{O}(1)$  memory footprint.

Table 4: (a) Training loss for the 16 configurations (4 context lengths  $\times$  4 sparse budgets) in the top-right subplot of Figure 7. (b) Resource usage during training for the 4 configurations (2 context lengths  $\times$  2 sparse budgets) in the top-left subplot of Figure 7. These experiments were conducted on four H200 GPUs using tensor parallelism, without gradient checkpointing or CPU offload.

(a) Loss values under different settings

Context	Dense Attention	Sparse Attention			
		4K+512	4K+2048	4K+8192	4K+32768
16K	1.32031	1.50000	1.36718	1.32031	1.32031
64K	1.22387	1.47656	1.32812	1.21875	1.21875
256K	2.21875	2.34375	2.28125	2.21875	2.21875
1M	4.31250	4.31250	4.50000	4.71875	4.78125

(b) Resource usage ( $4 \times \text{H200}$  tensor parallelism)

Metric	4K+8192		4K+32768	
	256K	1M	256K	1M
Latency/Iter (s)	43.454	176.305	87.340	422.383
Peak Memory (MB)	35637	67944	35671	68145

## 6 CONCLUSION AND LIMITATIONS

We introduced OOMB, a highly memory-efficient training system that overcomes the primary memory barrier in long-context LLM training. By integrating a chunk-recurrent framework with activation recomputation, a paged KV cache manager, asynchronous CPU offloading, and page-level sparse attention, OOMB maintains a nearly constant memory footprint regardless of sequence length. Our results demonstrate that OOMB enables training a Qwen2.5-7B model with a 4-million-token context on a single H200 GPU, a task that would otherwise require a large-scale cluster. This work represents a significant advance in making long-context model training more accessible and resource-efficient.

Despite its efficiency, OOMB has several limitations. The chunk-wise serial processing introduces a latency overhead compared to fully parallel training paradigms, which may be a consideration for throughput-sensitive training scenarios. Additionally, our use of sparse attention is an approximation of the full attention mechanism, and its impact on model performance for tasks requiring dense global context requires further investigation. The effectiveness of the asynchronous offloading is also dependent on the available CPU-GPU interconnect bandwidth.

486 ETHICS STATEMENT  
487488 The primary contribution of this work is a system designed to improve the computational efficiency of  
489 training large language models on long contexts. By significantly reducing the hardware requirements  
490 for such tasks, our research aims to democratize access to long-context model development for  
491 researchers and institutions with limited computational resources. A direct positive implication of this  
492 efficiency is the potential reduction in the overall energy consumption and carbon footprint associated  
493 with training these models, promoting more sustainable AI research practices.494 We acknowledge the potential for dual use, as is common with foundational AI research. The  
495 tools we develop could be applied to train models for malicious purposes, such as generating  
496 misinformation. While this work focuses on the system’s technical aspects, we advocate for the  
497 responsible development and deployment of models trained using our framework. Our experiments  
498 were conducted on the publicly available arXiv dataset, which consists of scientific literature and  
499 does not contain personally identifiable or sensitive information.500  
501 REPRODUCIBILITY STATEMENT  
502503 To facilitate the reproducibility of our findings, we have made the complete source code for the  
504 OOMB training system available in the anonymous repository cited in the abstract. A detailed  
505 description of our experimental setup, including model specifications, datasets, hyperparameters,  
506 and evaluation protocols, is provided in Section 5.1 and Appendix B. The core methodology and its  
507 components are thoroughly explained in Section 4. We believe these resources offer a sufficient basis  
508 for researchers to verify our results and extend our work.509  
510 REFERENCES  
511512 Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, et al.  
513 gpt-oss-120b gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.514 Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, et al. GQA:  
515 Training generalized multi-query transformer models from multi-head checkpoints. In *EMNLP*,  
516 2023.517 Chenxin An, Fei Huang, Jun Zhang, Shansan Gong, Xipeng Qiu, et al. Training-free long-context  
518 scaling of large language models. In *ICML*, 2024.519 Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, et al. LongAlign: A recipe for long context  
520 alignment of large language models. In *EMNLP*, 2024.521 Wadjih Bencheikh, Jan Finkbeiner, and Emre Neftci. Optimal gradient checkpointing for sparse and  
522 recurrent architectures using off-chip memory. *arXiv preprint arXiv:2412.11810*, 2024.523 Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of  
524 large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.525 Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, et al. Longlora: Efficient fine-tuning  
526 of long-context large language models. In *ICLR*, 2024.527 Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *ICLR*,  
528 2024.529 Yiran Ding, Li Lyra Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, et al. Longrope:  
530 extending llm context window beyond 2 million tokens. In *ICML*, 2024.531 Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. How to train long-context language  
532 models (effectively). *arXiv preprint arXiv:2410.02660*, 2024.533 Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. How to train long-context language  
534 models (effectively). In *ICLR*, 2025.

540 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. The  
 541 llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

542

543 Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, et al. Lm-infinite: Zero-shot extreme  
 544 length generalization for large language models. In *NAACL*, 2024.

545

546 Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Reza Yazdani Aminadabi,  
 547 et al. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence  
 548 transformer models. In *PODC*, 2024.

549

550 Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, et al. Llm maybe longlm:  
 551 Selfextend llm context window without tuning. In *ICML*, 2024.

552

553 Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph  
 554 Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model  
 555 serving with pagedattention. In *SOSP*, 2023.

556

557 Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Xuezhe Ma, et al. DISTFLASHATTN: Distributed  
 558 memory-efficient attention for long-context LLMs training. In *CoLM*, 2024.

559

560 Wenhao Li, Yuxin Zhang, Gen Luo, Daohai Yu, and Rongrong Ji. Training long-context llms  
 561 efficiently via chunk-wise optimization. In *ACL*, 2025.

562

563 Zhuzi Lin. Ring flash attention: Ring attention implementation with flash attention. Github, 2025.

564

565 Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, et al. Deepseek-v3 technical report.  
 566 *arXiv preprint arXiv:2412.19437*, 2025a.

567

568 Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-  
 569 infinite context. In *ICLR*, 2024.

570

571 Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang,  
 572 Chenchen Zhang, Ge Zhang, Jiebin Zhang, et al. A comprehensive survey on long context  
 573 language modeling. *arXiv preprint arXiv:2503.17407*, 2025b.

574

575 Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, et al. Moba: Mixture of block attention  
 576 for long-context llms. *arXiv preprint arXiv:2502.13189*, 2025.

577

578 Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window  
 579 extension of large language models. In *ICLR*, 2024.

580

581 Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations  
 582 toward training trillion parameter models. In *SC*, 2020.

583

584 Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, et al. Megatron-  
 585 lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint  
 586 arXiv:1909.08053*, 2020.

587

588 Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey.  
 589 SlimPajama: A 627B token cleaned and deduplicated version of RedPajama. HuggingFace, 2023.

590

591 Nimit S. Sohoni, Christopher R. Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré.  
 592 Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2022.

593

594 Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest:  
 595 Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*,  
 596 2024.

597

598 Xindi Wang, Mahsa Salmani, Parsa Omidi, Xiangyu Ren, Mehdi Rezagholizadeh, et al. Beyond the  
 599 limits: a survey of techniques to extend the context length in large language models. In *IJCAI*,  
 600 2024.

601

602 Chejian Xu, Wei Ping, Peng Xu, Zihan Liu, Boxin Wang, et al. From 128k to 4m: Efficient training  
 603 of ultra-long context large language models. *arXiv preprint arXiv:2504.06214*, 2025.

594 An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, et al. Qwen2.5  
 595 technical report. *arXiv preprint arXiv:2412.15115*, 2025.  
 596

597 Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, et al. Flashinfer: Efficient and  
 598 customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.  
 599

600 Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, et al. Native sparse attention:  
 601 Hardware-aligned and natively trainable sparse attention. 2025.  
 602

## 603 A USE OF LARGE LANGUAGE MODELS (LLMs)

604 During the writing of this paper, we utilized LLM solely for language editing to improve clarity and  
 605 readability. We critically reviewed and revised all AI-generated suggestions to ensure the final text  
 606 accurately reflects our original intent. All intellectual contributions, including the research design,  
 607 methodology, analysis, and conclusions, are our exclusive work, and we take full responsibility for  
 608 the academic integrity of this publication.  
 609

## 611 B EXPERIMENTAL DETAILS

613 All experiments were conducted using full-parameter fine-tuning in `bfloat16` precision. We  
 614 employed the Adam optimizer with a learning rate of  $5 \times 10^{-5}$  and betas of (0.9, 0.98). To isolate the  
 615 impact of sequence length on system performance, the per-GPU batch size was set to one, and all other  
 616 model and training hyperparameters were maintained at their default settings. For OOMB-specific  
 617 configurations, the KV cache page size was set to 128 for H200 GPUs.  
 618

619 System performance was evaluated using two primary metrics: peak GPU memory and per-iteration  
 620 latency. Peak memory per device was recorded using PyTorch’s `max_memory_allocated()`  
 621 function, while latency was measured precisely with CUDA events. To ensure reliability and mitigate  
 622 measurement noise, all reported results represent the minimum value from three independent runs.  
 623

## 624 C ADDITIONAL EXPERIMENTS AND ABLATION STUDIES

625 This section provides supplementary experiments that further validate the performance, scalability,  
 626 and correctness of the OOMB framework.  
 627

### 628 PERFORMANCE AND ABLATION STUDIES

630 **Scalability to 4M+ Tokens.** A key objective of OOMB is to enable training on sequence lengths that  
 631 are infeasible with conventional methods on limited hardware. We benchmarked the per-iteration  
 632 training time and memory usage of OOMB on contexts up to 8 million tokens. Figure 8 shows that  
 633 OOMB maintains exceptional memory efficiency even at this scale. When using sparse attention, the  
 634 training time scales in a near-linear fashion, making ultra-long context training practically achievable.  
 635

636 **Ablation on Chunk Size.** The chunk size is a key hyperparameter in OOMB that balances com-  
 637 putational parallelism and activation memory. We evaluated the effect of varying the chunk size  
 638 from 512 to 4096. As shown in Figure 9, larger chunk sizes improve computational efficiency by  
 639 enabling better hardware utilization. However, these performance gains diminish beyond a size of  
 640 4096. Memory consumption remains stable across all tested sizes. This analysis supports 4096 as  
 641 a well-balanced default value, as it provides strong computational performance without excessive  
 642 memory usage for activations.  
 643

644  
 645  
 646  
 647

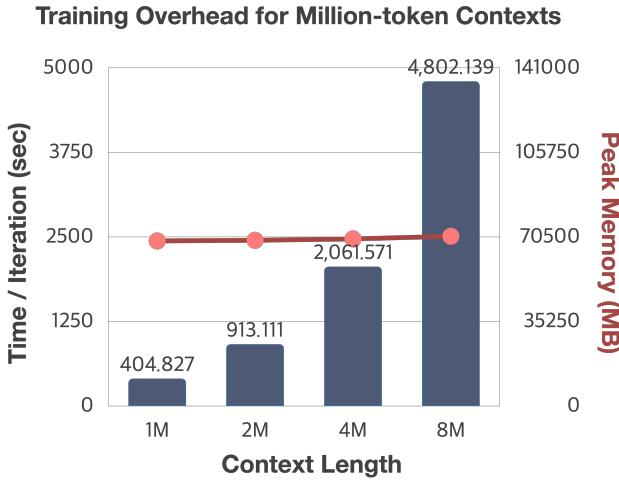


Figure 8: **Scalability to 4M+ Tokens.** OOMB demonstrates perfect memory efficiency up to a 8-million-token context length. With sparse attention, training time scales more linearly compared to dense attention. This experiment was conducted on  $4 \times$ H200 GPUs using tensor parallelism.

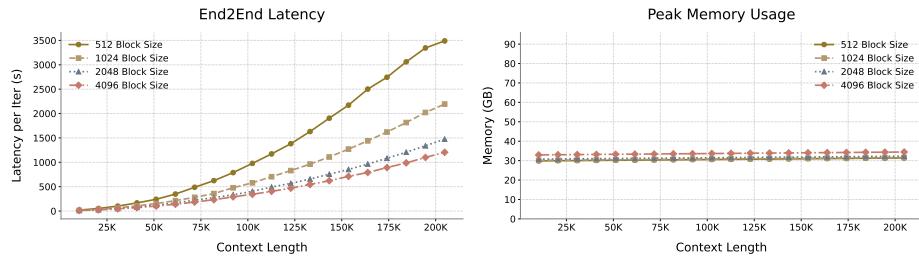


Figure 9: **Ablation on Chunk Size.** Increasing the chunk size improves computational throughput with diminishing returns, while peak memory usage remains stable. A chunk size of 4096 offers a strong balance between efficiency and memory. This experiment was conducted on a single A100 80G GPU.