SeCOv2: Highly Memory-Efficient Training for LLMs with Million-Token Contexts

Anonymous Author(s)
Affiliation
Address
email

Abstract

Training Large Language Models (LLMs) on long contexts is hindered by the prohibitive GPU memory cost of activations. We introduce SeCOv2, a block-recurrent framework that overcomes this limitation by processing sequences in blocks, akin to an RNN. By discarding and recomputing block activations on-the-fly, our method maintains a constant activation memory footprint, irrespective of sequence length. To address the subsequent bottleneck from the linearly growing KV cache (a bottleneck ignored by v1), we introduce three co-designed optimizations: (1) a paged memory manager for the KV cache and its gradients that eliminates redundant memory operations and fragmentation; (2) Op-level cache management independent of the PyTorch autograd system; and (3) an asynchronous mechanism to offload inactive KV caches to the CPU, reducing their GPU memory consumption by over 10×. The synergy of these techniques yields a remarkably low memory overhead, adding only 70-160 MB per 10,240 additional context tokens for 7-8B models. This efficiency allows us to train a Qwen2.5-7B model with a 4M token context on a single 96GB GPU, a substantial advance over existing open-source frameworks. To further enhance time efficiency, we also combined SeCOv2 with tensor parallelism and page-level sparse attention, accelerating the training process.

1 Introduction

Long-context modeling remains a formidable challenge in the training of Large Language Models (LLMs), a problem rooted not in training time, but in the prohibitive GPU memory overhead associated with long sequences [28, 38]. Research indicates that significant long-context capabilities can be achieved with a surprisingly small number of training steps (as few as one thousand) [30, 10] and that reaching a commercial standard may only require fine-tuning on approximately 5% of the original pretraining token count [14, 4].

Despite this modest data requirement, the number of tokens processed in a single training iteration grows dramatically, causing GPU memory consumption to scale linearly with context length and rapidly exhaust available resources. For instance, a model with a 4× Grouped-Query Attention (GQA) [2] ratio requires 64GB of HBM for the KV cache alone when processing a 256K context, a demand that single-handedly overwhelms a 96GB H20 GPU. This figure does not even account for the substantial memory footprint of other network activations, which often makes 128K tokens a practical ceiling for many models [15, 40]. The immense memory pressure from the KV cache and activations dwarfs the optimizer state, rendering popular memory-saving techniques like ZeRO3 offloading [33] and PyTorch FSDP [1] insufficient to overcome this fundamental bottleneck.

While various training-free methods for long-context extension have been proposed [20, 18, 3], their practical utility is often overstated. As established in prior work [13], common evaluation metrics such as Perplexity (PPL) and targeted retrieval tasks like Needle-in-a-Haystack (NIAH)

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

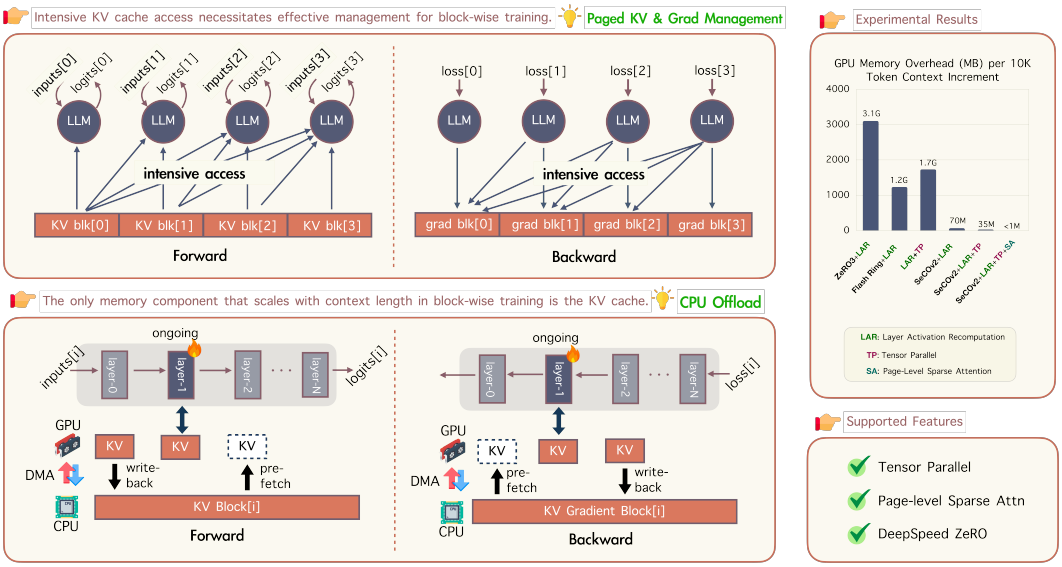


Figure 1: **Overview of the SeCOv2 Training Framework.** (*Top*) SeCOv2 processes long sequences in blocks, maintaining a constant activation memory footprint regardless of context length. (*Bottom*) This exposes the KV cache as the primary memory bottleneck in block-wise training. To solve this, we offload the cache to CPU memory and apply sparse attention, achieving near-constant GPU memory usage. (*Right*) The combined approach yields a >1000× reduction in memory consumption for every additional 10K context compared to strong baselines.

are unreliable indicators of true long-range reasoning abilities. Indeed, [13, 4] emphasizes that achieving commercial-grade performance requires dedicated fine-tuning with a meticulously curated data pipeline.

To this end, this paper directly confronts the central problem: how to train longer-context models with fewer resources, without compromising performance. We introduce a block-recurrent training framework that processes sequences in segments, analogous to an RNN. During the forward pass, each block's activations are computed and then immediately discarded. When needed for the backward pass, they are recomputed on-the-fly. This strategy maintains a constant activation memory footprint equivalent to a single block, regardless of the total sequence length.

However, a critical distinction from standard RNNs complicates this picture: Transformer blocks are globally coupled via the KV cache. Processing any single block necessitates access to the KV caches from all preceding blocks. This global dependency introduces two primary bottlenecks: (1) **Memory Bottleneck**: The KV cache and its associated gradients must be retained throughout the training step, scaling linearly with the context length. (2) **Computational Overhead**: The forward and backward passes require frequent access to the entire KV cache and its gradient, introducing significant latency.

To overcome these challenges, we have engineered a highly integrated system centered on efficient KV cache management, synergistically combines several co-designed optimizations:

1. **Paged KV Cache and Gradient Management.** We introduce a paged memory manager (inspired by [21]) for both the KV cache and its gradients. This design eliminates expensive `concatenation`, `reallocation` and `copy` operations and mitigates memory fragmentation when appending new key-value pairs.
2. **Specialized Op and Kernels.** We execute all operations on the KV cache and its gradients directly within the Op, making them opaque to PyTorch's autograd system. This avoids storing the KV cache as an activation and allows for direct gradient accumulation via CUDA `atomic_add`, improving both storage and computational efficiency.
3. **Asynchronous KV Cache CPU Offloading.** Since the KV cache and its gradients are the only components whose memory grows with context length, we designed an asynchronous mechanism to preemptively offload them to CPU memory. Experiments confirm that the resulting data transfer latency is effectively masked by the attention computation.

Table 1: **Comparison of Long-Context Training Methods.** SeCO series uniquely achieve $\mathcal{O}(1)$ activation memory complexity, where N denotes the context length. This key advantage allows SeCOv2, when combined with other techniques, to theoretically enable single-GPU training on contexts exceeding 10M tokens.

Method	GPUs	Act Memory	Exact Attn	Max Ctx Len.
ZeRO3 Offload [33]	8×H20	$\mathcal{O}(N)$	✔	64K
Ring Flash Attention [27]	8×H20	$\mathcal{O}(N/8)$	✔	256K
Tensor Parallel [19, 34]	8×H20	$\mathcal{O}(N/8)$	✔	256K
SeCO [23]	1×H20	$\mathcal{O}(1)$	✔	256K
SeCOv2	1×H20	$\mathcal{O}(1)$	✔	4M
SeCOv2 + Sparse Attn	1×H20	$\mathcal{O}(1)$	✗	10M+

The synergy of these techniques yields exceptional GPU memory efficiency. Our empirical results show that for every additional 10K tokens of context, the end-to-end training memory overhead increases by a mere 160MB for LLaMA3-8B [15] and 70MB for Qwen2.5-7B [40]. This stable and highly reproducible incremental cost is attributable solely to the paged KV cache and its gradients.

While context parallelism (CP) methods [27, 25, 34] are also highly memory-efficient, they do not extend the maximum training sequence length on a single GPU. Consequently, training a Qwen2.5-7B [40] model with a 4M context length requires a large cluster of 256 H100s [39]. In contrast, our approach enables training Qwen2.5-7B [40] with 4M-token context on a single 96GB GPU, representing a substantial advance in resource efficiency. To showcase its flexibility, we also successfully integrated our framework with tensor parallel [34, 19] and page-level sparse attention [29, 42].

2 Related Work

Efficient Long-Context LLM Training. A core challenge in extending language models to long contexts is their failure to generalize to token distances and positional encodings not seen during pre-training [18]. While continued training can resolve these issues, the associated GPU memory costs are often prohibitive. This has motivated the development of training-free context extension methods, which typically modify positional encodings to accommodate longer sequences [7, 18, 20, 3]. These approaches, however, introduce significant trade-offs. Methods based on interpolation or extrapolation of positional encodings [7] often degrade performance on short-text tasks by reducing the resolution of positional information [13]. Others that reuse position indices [18, 20, 3] achieve high scores on perplexity benchmarks but have been shown to fail on tasks requiring deep contextual understanding [13], indicating a superficial grasp of the extended context.

Given these limitations, state-of-the-art commercial and open-source models [15, 40, 26] still rely on fine-tuning to expand their context windows. However, the prevailing trend has shifted from algorithmic efficiency [8] towards overcoming memory barriers with massive computational resources [27, 22, 19, 34]. For instance, recent efforts [39] have required 256×H100 GPU cluster to extend LLaMA3.1 [15] from 128K to 4M context.

Serial vs. Parallel Training Paradigms. Language model training architectures can be broadly classified as parallel (Transformers [37], CNNs [31, 12], SSMs [11, 16]) or serial (RNNs [36]). Parallel training processes an entire sequence in a single forward and backward pass, maximizing GPU utilization and enabling scalability. However, this paradigm's memory footprint, which scales linearly with sequence length, creates 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, not raw throughput, is the primary constraint [30, 13]. This motivates a hybrid approach that processes sequences in blocks—parallel within each block and serial between them. This block-wise strategy is already standard in SOTA LLM inference engines (vLLM [21], FlashInfer [41]), where it incurs negligible latency. However, its application to training remains underdeveloped. An early exploration,

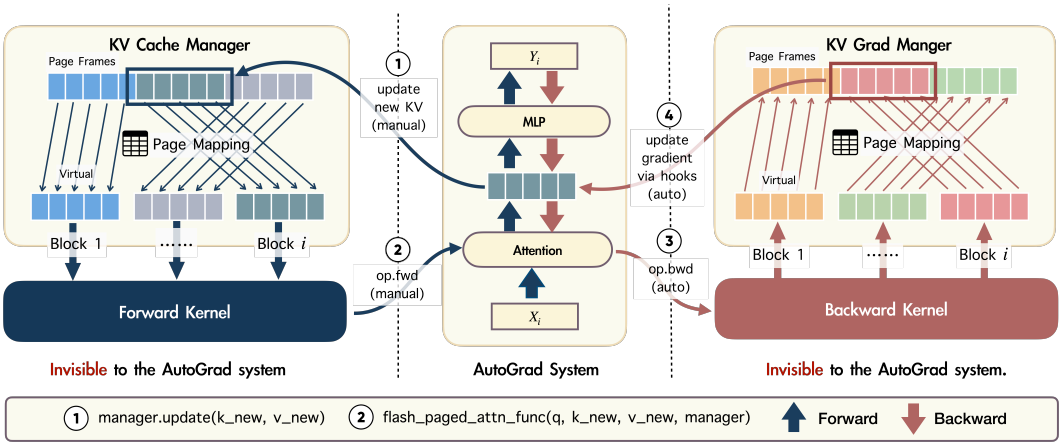


Figure 2: **An Overview of the Efficient KV Cache Management Framework.** Our framework optimizes frequent KV cache operations via two key strategies: (1) We bypass the autograd engine and manually manage KV cache values and their gradients. This ensures the portion of the computation graph handled by autograd has a constant memory footprint, resulting in $\mathcal{O}(1)$ activation memory. (2) We employ a paged memory scheme for the cache, which improves efficiency, reduces fragmentation, and natively supports page-level sparse attention [42, 29].

SeCO [23], introduced block-wise training but lacked critical optimizations at the operator and memory management levels. As a result, it could only train a 16K context model on a single RTX 3090 GPU. Under identical conditions, our fully-optimized system extends this capability by more than $10\times$, demonstrating a dramatic improvement in efficiency.

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 [27] and implemented in frameworks like Megatron-LM [34], DeepSpeed Ulysses [19] and TorchTitan [24], distributes a single sequence across multiple GPUs. This design introduces substantial inter-GPU communication overhead that scales quadratically with GPU count. In contrast, our serial approach's overhead is limited to repeated KV cache accesses and the latency from CPU offloading.

The trade-off is clear: CP excels with large GPU clusters but requires a high GPU count (e.g., at least $128\times$80GB GPUs for a 4M context) and specialized, high-bandwidth networks to manage its communication demands. Our method, however, maintains moderate and predictable overhead, eliminating the need for large-scale GPU clusters and specialized interconnects.

3 Preliminary: SeCOv1

Activation recomputation is a well-established technique for reducing the memory footprint of training RNNs. In a standard RNN, the forward pass iteratively computes an output y_i and a new hidden state m_i at each timestep i based on the input x_i and the previous state m_{i-1}:

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

where $\mathbf{a_i}$ represents the intermediate activations required for the backward pass:

$$\mathrm{d}\Theta,\ \mathrm{d}m_{i-1} \leftarrow \text{backward}(\mathrm{d}y_i, \mathbf{a_i}) \tag{2}$$

For long sequences, caching the activations $\mathbf{a_i}$ for all timesteps results in a memory footprint that scales linearly with sequence length, quickly becoming a bottleneck. Activation recomputation [35, 5] circumvents this by discarding $\mathbf{a_i}$ during the forward pass and only storing the much smaller hidden states m_i. The required activations are then regenerated on-the-fly just before their use in the backward pass. This trades a modest amount of recomputation for a substantial reduction in memory consumption.

Hybrid Serial-Parallel Training of Transformers. While Transformers are typically trained in a fully parallel fashion, the same principle of serial processing can be adapted to make their training

memory-efficient. We partition the input sequence into S blocks, $\{X_1, X_2, ..., X_S\}$, and process them serially, akin to the timesteps of an RNN. The forward pass for block i is defined as:

$$Y_i, M_i, \mathbf{A_i} = \mathcal{F}_{\text{Transformer}}(X_i, \{M_1, M_2, ..., M_{i-1}\}; \Theta) \tag{3}$$

Here, the model attends to the KV caches $\{M_1, ..., M_{i-1}\}$ from preceding blocks, which serve a role analogous to the RNN's hidden state. It produces the output Y_i, its own KV cache M_i, and the intermediate activations $\mathbf{A_i}$. The backward pass is likewise performed block-by-block:

$$\mathrm{d}\Theta, \{\mathrm{d}M_1, \mathrm{d}M_2, ..., \mathrm{d}M_{i-1}\} \leftarrow \text{backward}(\mathrm{d}Y_i, \mathrm{d}M_i, \mathbf{A_i}) \tag{4}$$

Following the recomputation strategy, the large activation tensor $\mathbf{A_i}$ for each block is discarded immediately after the forward pass and recomputed just before its corresponding backward pass.

This block-wise recomputation fundamentally differs from the layer-wise activation checkpointing commonly used in existing training frameworks [17, 33, 19]. While layer-wise checkpointing reduces memory, the total activation memory still scales linearly with the sequence length. In contrast, our block-wise approach ensures that the activation memory footprint remains constant, regardless of the sequence length, making it uniquely suited for training on extremely long contexts.

4 SeCOv2

4.1 Efficient KV Cache Management

Existing attention operators require the KV cache and its gradients to be stored in contiguous memory. This constraint poses no performance penalty in standard parallel training, where the KV cache is computed once per iteration and remains static. Consequently, most mainstream training frameworks lack a dedicated memory management mechanism for the KV cache.

In our block-wise training setting, however, the KV cache grows dynamically with each forward step. Enforcing memory contiguity under these conditions severely degrades system efficiency and memory utilization. Maintaining a contiguous KV cache requires expensive memory operations with each new block, such as `concatenate`, `copy`, and `reallocate`. Furthermore, allocating large, contiguous tensors for long contexts leads to memory fragmentation, preventing smaller, unused memory segments from being repurposed. These combined issues result in a substantial waste of both memory and computational resources.

Although pre-allocating a single large, contiguous buffer might seem like a solution, this approach is incompatible with dynamic memory-saving techniques like sequence-level activation recomputation or KV cache offloading. In our system, the KV cache is highly dynamic; its constituent blocks may be discarded, recomputed, or offloaded to CPU memory. A large, static pre-allocation would unnecessarily reserve memory that could otherwise be used by other components.

To quantify the memory inefficiency in a system lacking a dynamic KV cache manager, we benchmarked SeCOv1 [23] with a block-wise regime. As shown in Figure 3, even when using Flash Attention [9] and KV cache offloading, the system's memory consumption exhibits a sharp inflection point beyond a 50K token context. Past this point, the memory footprint for each additional 10K tokens increases at a rate nearly three times higher than theoretically expected. After eliminating other potential causes, including the attention operator itself, we concluded that this anomalous memory growth is a direct consequence of the lack of a dynamic memory management mechanism.

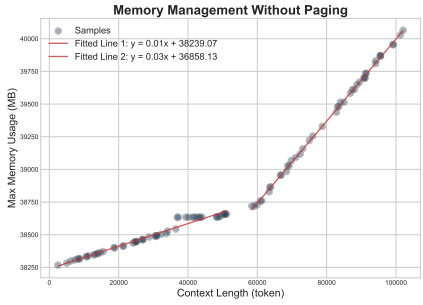


Figure 3: Without a paging mechanism, peak memory allocation for the long-context KV cache can be 3× higher than anticipated.

Paged Management for the KV Cache in Training. To address these challenges and maximize trainable context length, we introduce a paged memory management system for both the KV cache and its gradients, tightly integrated with our block-wise training framework. Existing paged attention implementations [21, 41] are designed for inference and lack support for backpropagation. We extend

this approach by developing a paged management system for the KV cache gradients. Specifically, as depicted in Figure 2, our mechanism partitions the KV cache tensors M_i and their corresponding gradients $\mathrm{d}M_i$ (from Eq. 4) into fixed-size, non-contiguous pages. These pages are stored in a pre-allocated memory pool, and a page table tracks their physical addresses.

Optimizing for L2 Cache Locality. A potential drawback of paged memory is reduced data locality, which can lower the L2 cache hit rate. To overcome this, our memory allocator attempts to assign pages from the same logical block to physically contiguous page frames whenever possible. This simple heuristic yields a measurable improvement in system throughput.

Decoupling the KV Cache from the Autograd System. Managing the KV cache via PyTorch's autograd system, while straightforward, introduces significant overhead. Standard autograd execution requires allocating intermediate buffers for gradients and performing separate read-add-write operations on the cache's `.grad` tensor. This process is inefficient for the large and frequently accessed KV cache and complicates advanced memory management like offloading. To circumvent this, we bypass autograd for KV cache gradients (see Figure 2). Our CUDA Op computes the gradient and uses a single `atomic_add` operation to directly update the gradient tensor in place. This approach eliminates intermediate memory allocation and redundant data movement, increasing system throughput especially in large-context scenarios where the KV cache consumes tens of gigabytes.

Eliminating Redundant KV Computation in Activation Recomputation. Activation recomputation saves memory by re-calculating intermediate results during the backward pass. However, a naive implementation recomputes all activations, including the KV cache. This is unnecessary, as the KV cache is intentionally preserved across forward passes and remains available for the backward pass. We therefore implement a specialized recomputation pipeline that selectively regenerates only the required intermediate results, entirely skipping the redundant KV projection. This targeted approach avoids wasteful computation and improves system throughput.

4.2 Scaling Context Length via KV Cache Offloading

The KV cache and its gradient (which are isolated from the autograd system) are the sole components whose memory demands scale linearly with the context length. This overhead presents a significant bottleneck.

To overcome this limitation, we leverage the temporal locality inherent in the Transformer's sequential, layer-by-layer execution. We introduce a strategy of asynchronous CPU offloading and pre-fetching that minimizes peak GPU memory by retaining only the KV cache for the currently active layer.

During computation for layer i, dedicated CUDA streams concurrently pre-fetch the KV cache for layer $i+1$ from the CPU and offload the cache for layer $i-1$ to the CPU. This data flow is reversed during the backward pass, as illustrated in Figure 1 (Bottom). When layer computations are sufficiently large (*i.e.*, using a large block size), the latency from these asynchronous transfers is effectively masked by the main computation.

Experiments on a single A100 GPU[1] show that for 4096 block size, our method achieves a GPU-CPU transfer throughput approximately double that of FlashAttention-2.5.8 [9] (Table 2). This result indicates that, in theory, the overhead from CPU offloading can be entirely masked by computation.

Table 2: Our CPU offload mechanism's throughput is significantly higher than that of FlashAttention-2.5.8, ensuring the communication overhead is fully masked by computation.

Context	KV Size (GB)	Latency (s)		Throughput (GB/s)	
		flash attn	pre-fetch	flash attn	pre-fetch
100K	0.4	0.0324	0.0172	12.32	23.25
1M	4.0	0.3285	0.1668	12.17	24.17

4.3 Overcoming Training Time with Sparse Attention

With memory efficiency achieved, excessive training time becomes the primary bottleneck. To address this, we incorporate Native Sparse Attention (NSA) [42], which holds CPU offloading communication overhead constant and scales attention computation nearly linearly with context length. This capability is crucial for processing extremely long contexts. Extensive experiments confirm that training with NSA on ultra-long texts leads to significant efficiency gains.

[1]The A100 features faster computation but a lower bandwidth compared to the H20.

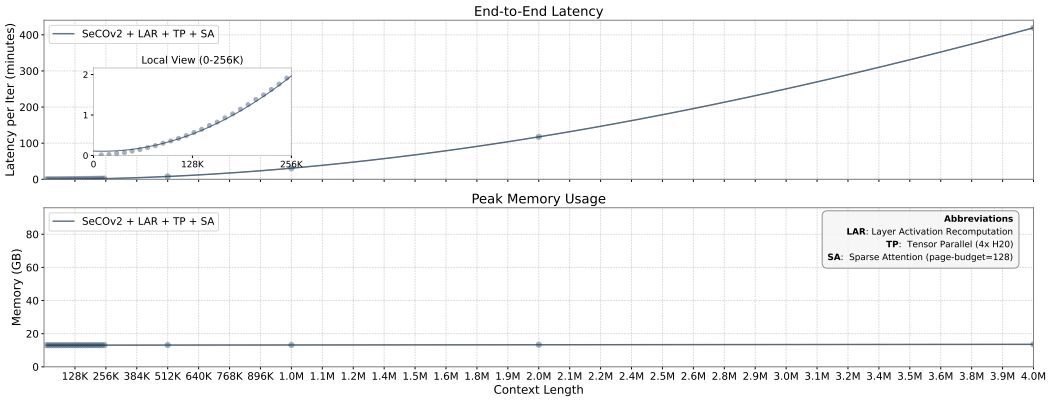


Figure 4: **Scalability of SeCOv2 for Training 7B Models.** SeCOv2, combined with tTensor parallelism (TP) and sparse attention (SA), achieves a nearly constant memory footprint while maintaining manageable latency growth. The limited gains from sparse attention are due to Triton kernel implementation details, with significant improvements anticipated in future CUDA releases.

5 Experiments

SeCOv2 is a general purpose framework for long context training, designed to be agnostic to specific training recipes and data. Consequently, our experiments (Section 5.1 and 5.2) benchmark the intrinsic performance of the framework itself, with a primary focus on its time and memory efficiency.

5.1 Main Results

Experimental Setup. To evaluate time and memory efficiency, we use the Qwen2.5-7B [40] model on the PG19 [32] dataset. We duplicated and concatenated the dataset as necessary to achieve the desired sequence lengths. All experiments were conducted using `bf16` precision with the AdamW optimizer. Since our focus is on efficiency rather than model convergence, all other hyperparameters were kept at their default values.

All baseline methods utilize a GQA-compatible version of FlashAttention [9] with a per GPU batch size of 1. All experiments involve full-parameter fine-tuning. For SeCOv2, we always set the block size to 4096 and the page size to 128. For experiments involving sparse attention, we consistently use a page budget of 128. We measure peak memory usage using `max_memory_allocated()` and iteration latency using CUDA events. All reported results are the minimum from 3 runs.

By default, methods do not employ acceleration techniques such as tensor parallelism (TP), layer activation recomputation (LAR), or sparse attention (SA) unless explicitly indicated by a suffix.

Efficiency and Scalability up to 4M Tokens. We benchmark the per-iteration training time and memory of SeCOv2 on sequence lengths ranging from 2K to 4M tokens, comparing full and sparse attention implementations. As shown in Figure 4, SeCOv2 achieves high memory efficiency with both attention mechanisms. With sparse attention, the training time exhibits a more favorable growth.

Comparison with ZeRO3-Offload. Since both SeCOv2 and ZeRO3-Offload [33] utilize CPU offloading, we conduct a direct comparison to evaluate their relative time and memory efficiency. We evaluate three DeepSpeed ZeRO3 configurations: (1) pure ZeRO3, (2) ZeRO3 with parameter offloading, and (3) ZeRO3 with both parameter offloading and layer activation recomputation. The ZeRO3 experiments are run on 8$\times$H20 GPUs with a per device batch size of 1, and we report statistics from GPU 0 (see Figure 5). SeCOv2 is run on a single H20 GPU for this comparison.

Comparison with Context Parallelism Methods. We benchmark SeCOv2 against CP methods, which also partition input sequences into blocks. We selected Ring Flash Attention (RFA) [25], a heavily optimized and representative open-source implementation of Ring Attention [27], as our primary baseline. Both methods were benchmarked for per device training throughput and memory

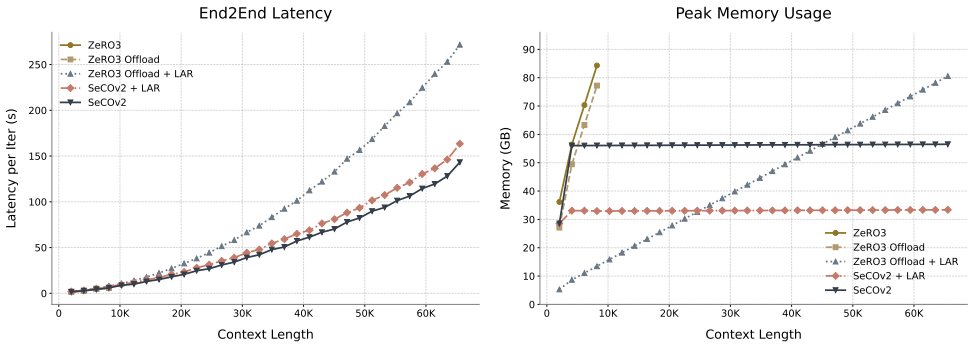


Figure 5: Compared to the widely-recognized DeepSpeed ZeRO3 [33], SeCOv2 demonstrates substantially higher per-GPU efficiency with respect to both computation time and memory usage.

Table 3: Throughput comparison with context parallel methods. We compared the per-GPU training throughput for 128K and 256K context lengths. Results shows that the SOTA implementation, Ring Flash Attention (RFA) [25], achieves a throughput comparable to that of SeCOv2. (The results highlighted in red are from [6].)

Context	**Ring Attention** [27]		**RFA** [25]	**SeCOv2+TP**	
	8×A100	16×TPUv4	8×H20	1×H20	4×H20
128K	-	-	232 tok/s	240 tok/s	229 tok/s
256K	50 tok/s	49 tok/s	125 tok/s	139 tok/s	121 tok/s

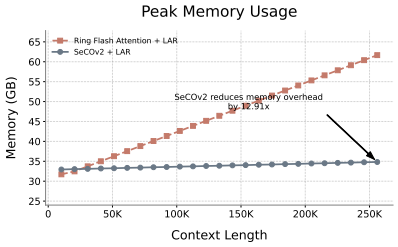


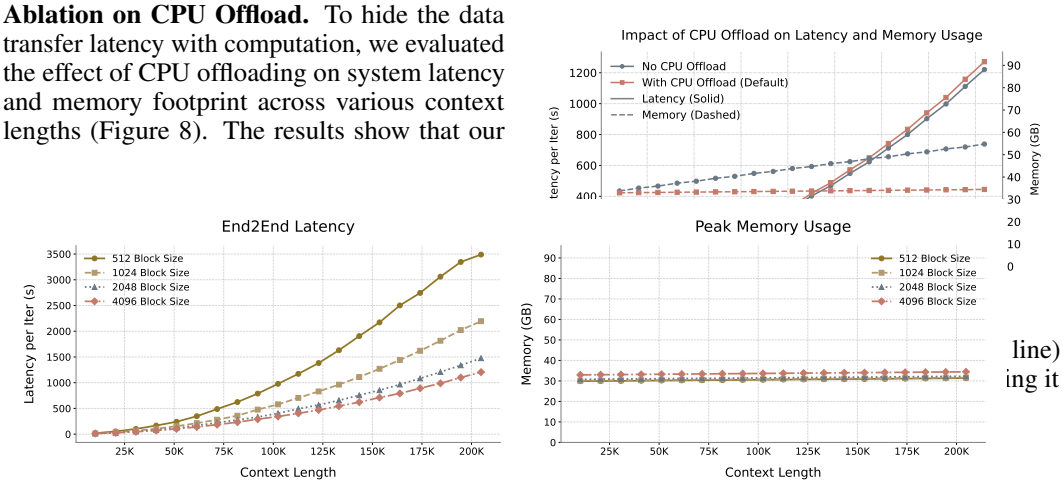
Figure 6: Memory usage compared to context parallel methods.

usage. The results in Table 3 and Figure 6 demonstrate that SeCOv2 delivers throughput comparable to RFA but with substantially improved memory efficiency.

5.2 Ablation Results

Ablation on Block Size. We ablate the block size, doubling it from 512 to 4096. As shown in Figure 7, computation time saturates with increasing block size, yielding diminishing returns beyond 4096. Meanwhile, memory consumption shows minimal difference. Notably, as the block size approaches infinity, our method degenerates to standard parallel training.

Ablation on CPU Offload. To hide the data transfer latency with computation, we evaluated the effect of CPU offloading on system latency and memory footprint across various context lengths (Figure 8). The results show that our



line) ing it

Figure 7: Increasing the block size from 512 to 4096 yields diminishing marginal returns, with performance eventually converging to the parallel training baseline (corresponds to an ∞ block size).

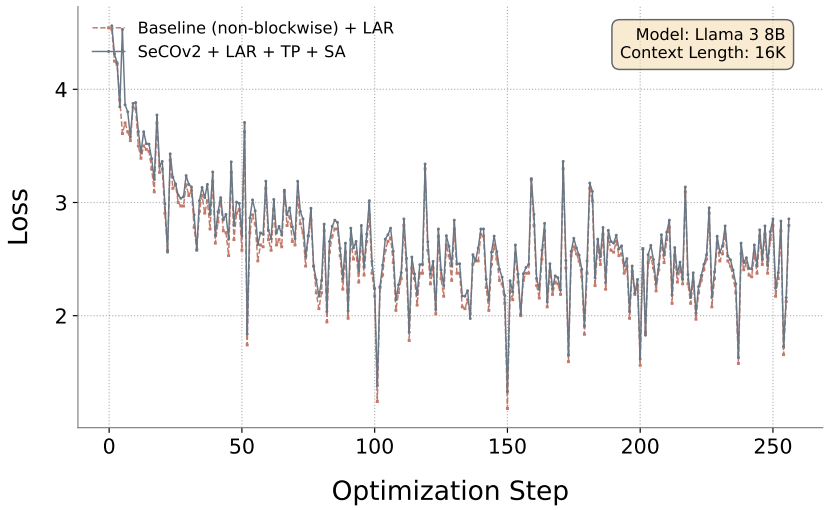


Figure 9: **Accuracy Validation.** Our SeCOv2 implementation, using 25% sparse attention and 4-GPU tensor parallelism, achieves nearly identical final accuracy to a baseline employing FlashAttention-2 (v2.5.8). This result confirms the correctness and reliability of our system.

method introduces a minor latency overhead of 5%, primarily due to data transfer initiation. In return, it substantially reduces the memory footprint, which remains nearly constant for contexts up to 200K. These findings confirm that the KV cache is the primary memory bottleneck and demonstrate that offloading is an effective mitigation strategy.

5.3 Accuracy Validation

To evaluate SeCOv2, we quantify discrepancies arising from its custom sparse attention, tensor parallelism, and block-wise optimizations. We compare the loss reduction of SeCOv2 against a baseline using parallel training with FlashAttention (v2.5.8) [9]. As shown in Figure 9, at a 25% sparsity level (achieved with a page size and sparse page budget of 64), the final error after 256 training steps is approximately 0.03.

6 Conclusion and Limitations

Conclusion. In this work, we introduced SeCOv2, a highly memory-efficient training framework that significantly lowers the resource barrier for training LLMs on million-token contexts. By combining a block-recurrent training strategy with on-the-fly activation recomputation, we achieve a constant memory footprint for activations, irrespective of sequence length. We address the subsequent KV cache bottleneck through a suite of co-designed optimizations: a paged memory manager for the cache and its gradients, autograd-decoupled cache operations, and an asynchronous CPU offloading mechanism. The synergy of these techniques enables training a 4-million-token context Qwen2.5-7B model on a single 96GB GPU, a task that previously required large-scale clusters. Our framework is designed to be a plug-and-play wrapper, compatible with existing training systems, to help democratize research and development in ultra-long-context LLMs.

Limitations. The primary limitation of SeCOv2 is the trade-off between memory efficiency and computational time. The serial, block-wise processing and activation recomputation introduce latency overhead compared to standard parallel training. While we demonstrate that this overhead is manageable and can be partially mitigated by larger block sizes and integration with tensor parallelism or sparse attention, it remains an inherent characteristic of the design. Furthermore, our asynchronous

CPU offloading mechanism relies on the computational workload of a block being sufficient to hide data transfer latency, which may not hold for smaller models or block sizes. Finally, this work focuses on the systems-level efficiency of the training framework itself; we do not propose new long-context learning recipes or data curation strategies, which are orthogonal but essential for achieving optimal model performance.

References

[1] L. AI. Litgpt. `https://github.com/Lightning-AI/litgpt`, 2023.

[2] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, et al. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *EMNLP*, 2023.

[3] C. An, F. Huang, J. Zhang, S. Gong, X. Qiu, et al. Training-free long-context scaling of large language models. In *ICML*, 2024.

[4] Y. Bai, X. Lv, J. Zhang, Y. He, J. Qi, L. Hou, et al. LongAlign: A recipe for long context alignment of large language models. In *EMNLP*, 2024.

[5] W. Bencheikh, J. Finkbeiner, and E. Neftci. Optimal gradient checkpointing for sparse and recurrent architectures using off-chip memory. *arXiv preprint arXiv:2412.11810*, 2024.

[6] W. Brandon, A. Nrusimha, K. Qian, Z. Ankner, T. Jin, et al. Striped attention: Faster ring attention for causal transformers. 2023.

[7] S. Chen, S. Wong, L. Chen, and Y. Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[8] Y. Chen, S. Qian, H. Tang, X. Lai, Z. Liu, et al. Longlora: Efficient fine-tuning of long-context large language models. In *ICLR*, 2024.

[9] T. Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *ICLR*, 2024.

[10] Y. Ding, L. L. Zhang, C. Zhang, Y. Xu, N. Shang, et al. Longrope: extending llm context window beyond 2 million tokens. In *ICML*, 2024.

[11] D. Y. Fu, T. Dao, K. K. Saab, A. W. Thomas, A. Rudra, et al. Hungry hungry hippos: Towards language modeling with state space models. In *ICLR*, 2023.

[12] D. Y. Fu, E. L. Epstein, E. Nguyen, A. W. Thomas, M. Zhang, T. Dao, A. Rudra, and C. Re. Simple hardware-efficient long convolutions for sequence modeling. In *ICLR*, 2023.

[13] T. Gao, A. Wettig, H. Yen, and D. Chen. How to train long-context language models (effectively). *arXiv preprint arXiv:2410.02660*, 2024.

[14] T. Gao, A. Wettig, H. Yen, and D. Chen. How to train long-context language models (effectively). In *ICLR*, 2025.

[15] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[16] A. Gu, K. Goel, and C. Ré. Efficiently modeling long sequences with structured state spaces. In *ICLR*, 2022.

[17] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, et al. Accelerate: Training and inference at scale made simple, efficient and adaptable. `https://github.com/huggingface/accelerate`, 2022.

[18] C. Han, Q. Wang, H. Peng, W. Xiong, Y. Chen, et al. Lm-infinite: Zero-shot extreme length generalization for large language models. In *NAACL*, 2024.

[19] S. A. Jacobs, M. Tanaka, C. Zhang, M. Zhang, R. Y. Aminadabi, et al. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. In *PODC*, 2024.

[20] H. Jin, X. Han, J. Yang, Z. Jiang, Z. Liu, et al. Llm maybe longlm: Selfextend llm context window without tuning. In *ICML*, 2024.

[21] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023.

[22] D. Li, R. Shao, A. Xie, E. P. Xing, X. Ma, et al. DISTFLASHATTN: Distributed memory-efficient attention for long-context LLMs training. In *CoLM*, 2024.

[23] W. Li, Y. Zhang, G. Luo, D. Yu, and R. Ji. Training long-context llms efficiently via chunk-wise optimization. In *ACL*, 2025.

[24] W. Liang, T. Liu, L. Wright, W. Constable, A. Gu, et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2025.

[25] Z. Lin. Ring flash attention: Ring attention implementation with flash attention. `https://github.com/zhuzilin/ring-flash-attention`, 2025.

[26] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2025.

[27] H. Liu, M. Zaharia, and P. Abbeel. Ringattention with blockwise transformers for near-infinite context. In *ICLR*, 2024.

[28] J. Liu, D. Zhu, Z. Bai, Y. He, H. Liao, H. Que, Z. Wang, C. Zhang, G. Zhang, J. Zhang, et al. A comprehensive survey on long context language modeling. *arXiv preprint arXiv:2503.17407*, 2025.

[29] E. Lu, Z. Jiang, J. Liu, Y. Du, T. Jiang, et al. Moba: Mixture of block attention for long-context llms. *arXiv preprint arXiv:2502.13189*, 2025.

[30] B. Peng, J. Quesnelle, H. Fan, and E. Shippole. YaRN: Efficient context window extension of large language models. In *ICLR*, 2024.

[31] M. Poli, S. Massaroli, E. Nguyen, D. Y. Fu, T. Dao, et al. Hyena hierarchy: towards larger convolutional language models. In *ICML*, 2023.

[32] J. W. Rae, A. Potapenko, S. M. Jayakumar, C. Hillier, and T. P. Lillicrap. Compressive transformers for long-range sequence modelling. In *ICLR*, 2020.

[33] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: memory optimizations toward training trillion parameter models. In *SC*, 2020.

[34] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2020.

[35] N. S. Sohoni, C. R. Aberger, M. Leszczynski, J. Zhang, and C. Ré. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2022.

[36] R. C. Staudemeyer and E. R. Morris. Understanding lstm – a tutorial into long short-term memory recurrent neural networks. *arXiv prepring arXiv:1909.09586*, 2019.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, et al. Attention is all you need. In *NeurIPS*, 2017.

[38] X. Wang, M. Salmani, P. Omidi, X. Ren, M. Rezagholizadeh, et al. Beyond the limits: a survey of techniques to extend the context length in large language models. In *IJCAI*, 2024.

[39] C. Xu, W. Ping, P. Xu, Z. Liu, B. Wang, et al. From 128k to 4m: Efficient training of ultra-long context large language models. *arXiv preprint arXiv:2504.06214*, 2025.

[40] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

[41] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[42] J. Yuan, H. Gao, D. Dai, J. Luo, L. Zhao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. 2025.