Training Long-Context LLMs Efficiently via Chunk-wise Optimization

Anonymous ACL submission

Abstract

While long-context large language models (LLMs) exhibit remarkable document processing capabilities, their prohibitively high training costs often hinder customized applications. To mitigate this issue, we propose *Sequential Chunk-wise Optimization* (SeCO), a memory-efficient training paradigm that partitions lengthy inputs into manageable chunks. Each chunk independently constructs its computational graph and performs localized backpropagation, ensuring that only one chunk's forward activations are stored in memory. Building on SeCO, we further introduce *Sparse Chunk-wise Optimization* (SpaCO), which reduces computational overhead by selectively propagating gradients to specific chunks and incorporates a carefully designed compensation factor to ensure unbiased gradient estimation. SpaCO decouples the computational cost of backpropagation from the context length, enabling training time to gradually converge to inference time as sequences become longer. Implemented as lightweight training wrappers, both SeCO and SpaCO offer substantial practical benefits. For example, when fine-tuning an 8B model with LoRA on a single RTX 3090 GPU, SeCO expands maximum sequence length from 1K to 16K tokens, while SpaCO demonstrates accelerated training speed—achieving up to 3× faster than SeCO under the same experimental setup. These innovations provide new insights into optimizing long-context models, making them more accessible for practical applications. We have open-sourced the code at Anonymous/SeCO.

1 Introduction

Recent advancements in long-context LLMs (Chen et al., 2024; An et al., 2024; Peng et al., 2024) have demonstrated unprecedented capabilities in processing lengthy documents, offering superior retrieval quality compared to retrieval-augmented generation (RAG) approaches (Liu, 2022), making them particularly valuable for commercial applications requiring nuanced document understanding.

However, fine-tuning these models faces significant resource challenges: *(i) Time Overhead:* The quadratic scaling of attention mechanisms leads to prohibitive training time (de Vries, 2023). *(ii) Memory Constraints:* Despite optimizations like FlashAttention (Dao, 2024), the storage requirements for forward activations still increases linearly with sequence length, quickly depleting GPU memory. As a result, fine-tuning 8B models (Meta-AI, 2024) with LoRA (Hu et al., 2022) on a single RTX 3090 GPU is limited to sequences of only 1K tokens.

Existing architectural modifications, exemplified by LongLoRA's S^2-attention (Chen et al., 2024), aim to alleviate these issues by reducing computational overhead to sub-quadratic through attention approximation. However, these methods incur gradient accuracy compromises while offering limited resource savings,[1] motivating our exploration of alternative efficiency improvements strategies.

We introduce *Sequential Chunk-wise Optimization* (SeCO), a novel training method that preserves exact gradients while dramatically reducing memory consumption. The key innovation of SeCO is the application of gradient checkpointing (Bulatov, 2018; Chen et al., 2016) along the sequence dimension using chunk-level checkpoints. This approach represents a fundamental departure from traditional gradient checkpointing, which typically employs a fixed number of checkpoints for static layer-wise or block-wise partitioning of the computational graph. Unlike these conventional techniques, where memory requirements for forward activations scale linearly with sequence length, SeCO maintains a con-

[1] As shown in Figure 1 (*Mid*) of the LongLoRA paper, the proposed method exhibits memory scaling patterns similar to those of full fine-tuning baseline, achieving only about a 2-fold extension in sequence length.

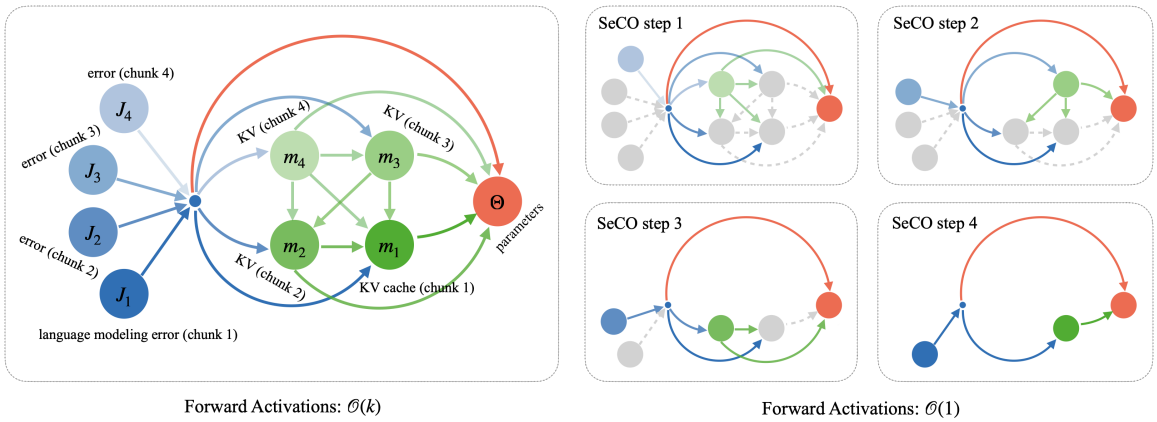


Figure 1: (*Left*) Computational graph for chunk-wise optimization with $k = 4$ chunks. The dense connections among KV caches (green arrows) complicate memory management, leading popular training frameworks (MicroSoft, 2021; Gugger et al., 2022) to rely on end-to-end parallel training. (*Right*) By analyzing the topology of this graph, we propose SeCO, a bootstrapping method leveraging gradient checkpointing along the sequence dimension. SeCO ensures that only the computational graph of a single chunk is stored at any time.

stant overhead for forward activations, regardless of sequence length. This innovation achieves order-of-magnitude memory reduction while maintaining manageable training time overhead, establishing it as an efficient solution for fine-tuning long-context LLMs under resource-constrained conditions.

While SeCO successfully mitigates memory constraints in long-context LLM fine-tuning, it maintains computational overhead comparable to naive parallel training. This computational burden significantly undermines its applicability for processing extended sequences, thereby limiting its practical utility. To address this limitation, we propose *Sparse Chunk-wise Optimization* (SpaCO), an enhanced variant of SeCO that achieves substantial computational savings. SpaCO preserves the integrity of forward propagation while implementing selective backpropagation through a fixed subset of chunks. This modification decouples computational cost from sequence length during gradient computation. Our theoretical framework reveals a crucial architectural insight: The gradient chain length between key-value (KV) cache chunks exhibits inherent boundedness determined by model depth (as also noted in Dai et al., 2019). This fundamental property enables SpaCO to employ randomized chunk sampling while preserving unbiased gradient estimation, achieving significant computational reduction without compromising theoretical guarantees (for more detailed explanation, please refer to Section 5).

Empirical evaluations highlight the substantial practical advantages of SeCO and SpaCO:

- **Scalability**: The memory overhead for SeCO and SpaCO scales minimally with increasing sequence length, as the only contributing factor is the storage of the KV cache. Moreover, SpaCO's training time converges to inference time as the sequence length expands, demonstrating efficient computational scaling.
- **Performance**: Although SpaCO does not compute exact gradients like SeCO, it incurs only a small performance gap. Specifically, at a sparsity ratio of $1/8$, the language modeling error increases by less than 0.1 compared to exact gradient training.

Our contributions are threefold:

- A memory efficient training paradigm (SeCO) that enables long-context fine-tuning through sequence dimensional gradient checkpointing.
- A computation-efficient extension (SpaCO) leveraging sparsification, with theoretical guarantees of unbiased gradient estimation.
- Open-source implementations that achieve up to an order of magnitude training sequence length improvements on consumer hardware.

2 Related Works

Long-Context LLMs. Efforts to extend the context window of LLMs primarily rely on augmenting positional embeddings and applying limited post-training to adapt models pre-trained on shorter contexts (Chen et al., 2024; Peng et al., 2024). While

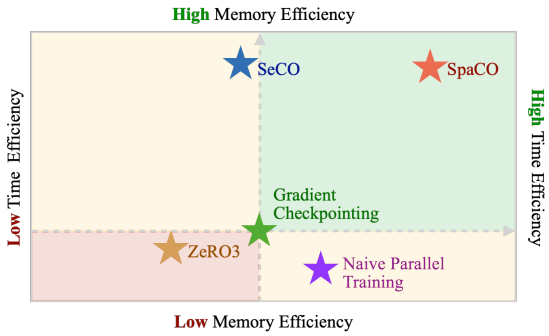


Figure 2: Qualitative comparison of different methods. *(i)* SeCO achieves significant memory reduction while maintaining time efficiency comparable to layer-level gradient checkpointing. *(ii)* Building upon SeCO, SpaCO significantly reduces computational overhead, making the training time converges to inference time as the sequence length expands.

these methods are effective, the inherent quadratic computational complexity of attention mechanisms renders long-context training prohibitively expensive (de Vries, 2023). Recent works, such as LongLoRA (Chen et al., 2024), address this issue using S^2-attention, which achieves linear computation scalability. However, its architectural modification introduces biased gradient computation.

Gradient Checkpointing. Gradient checkpointing techniques (Chen et al., 2016; Bulatov, 2018) optimize memory consumption by recomputing activations during backpropagation rather of storing them. In Transformer (Vaswani et al., 2017) architectures, conventional layer-level checkpointing offers limited benefits for long sequences due to the static partitioning of the computational graph. Gradient checkpointing applied along the sequence dimension enables maintaining a constant memory footprint for storing forward activations, presenting substantial advantages. Nevertheless, current implementations in mainstream deep learning frameworks (Chintala et al., 2016; Google, 2015) are primarily designed for checkpointing within individual forward pass, lacking the capability to handle concatenated computational graphs that emerge across multiple iterative processes.

Gradient Estimation. The concept of approximate gradient predates modern deep learning, exemplified by stochastic gradient descent's use of mini-batch (Bottou and Bousquet, 2007). SpaCO introduces a novel paradigm that aligns with the philosophy of SGD: by utilizing a limited number of gradient propagation pathways to estimate the underlying true gradient, it significantly reduces computational overhead while maintaining acceptable performance.

3 Preliminary

When processing large amounts of data all at once, GPU threads can become saturated, causing parallel processing time to scale linearly with data size, offering no advantage over sequential processing while increasing memory usage. To address this, efficient LLM serving frameworks such as vLLM (Kwon et al., 2023) and FlashInfer (Ye et al., 2025) adopt a chunk pre-filling strategy, splitting long contexts into smaller chunks and processing them sequentially. We extend this idea from LLM inference to LLM training.

Computational Graph. For an input sequence X partitioned into k chunks $\{x_j\}_{j=1}^{k}$, let m_j denote the KV cache and J_j the error component for chunk j. The model f with parameter Θ processes chunks sequentially:

$$(J_j, m_j) = f(x_j; m_1, m_2, ..., m_{j-1}; \Theta). \quad (1)$$

The parameter gradient combines direct and indirect contributions through KV cache:

$$\nabla_\Theta J_j = \underbrace{\frac{\partial J_j}{\partial \Theta}}_{\text{Direct term}} + \sum_{i=1}^{j} \underbrace{\frac{\mathrm{d}J_j}{\mathrm{d}m_i}\frac{\partial m_i}{\partial \Theta}}_{\text{Indirect contributions}} . \quad (2)$$

Due to the iterative nature of $f(\cdot)$, the computation of $\mathrm{d}J_j/\mathrm{d}m_i$ involves nested dependencies:

$$\begin{aligned}\frac{\mathrm{d}J_j}{\mathrm{d}m_i}\frac{\partial m_i}{\partial \Theta} = &\frac{\partial J_j}{\partial m_i}\frac{\partial m_i}{\partial \Theta} \\ &+ \sum_{i<t_1\leq j} \frac{\partial J_j}{\partial m_{t_1}}\frac{\partial m_{t_1}}{\partial m_i}\frac{\partial m_i}{\partial \Theta} \\ &+ \sum_{i<t_1<t_2\leq j} \frac{\partial J_j}{\partial m_{t_2}}\frac{\partial m_{t_2}}{\partial m_{t_1}}\frac{\partial m_{t_1}}{\partial m_i}\frac{\partial m_i}{\partial \Theta} \\ &+ \dots .\end{aligned} \quad (3)$$

Although Eq. (3) appears complex, it fundamentally demonstrates that gradients propagate through all possible multi-hop paths among $\{m_t\}_{t=i}^{j}$. To visualize this process, we present partial derivatives $\partial\triangle/\partial\bigcirc$ as directed edges from $\triangle$ to $\bigcirc$, forming the complete computational graph of gradient propagation from $\{J_j\}_{j=1}^{k}$ to Θ as illustrated in Figure 1. This graph explicitly captures the computational dependencies for chunk-wise optimization.

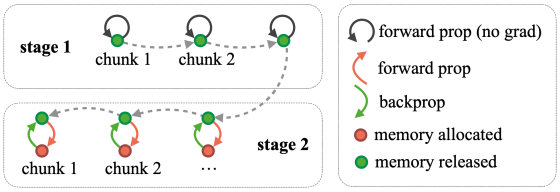


Figure 3: A visualized illustration of Algorithm 1. *Stage 1* corresponds to the first for-loop, generating KV caches for all data chunks through inference-mode, serving as checkpoints. *Stage 2* corresponds to the second for-loop, where the computational graph is constructed and localized backpropagation is performed.

Gradient Checkpointing. Gradient checkpointing trades computational time for reduced memory usage. The fundamental principle guiding checkpoint placement requires that the complete subsequent computational graph must be reconstructible using only these designed checkpoints and existing leaf nodes. As formalized in Eq. (1), the output of chunk-j is uniquely determined by two components: *(i)* the preceding KV cache sequence $\{m_i\}_{i=1}^{j-1}$ and *(ii)* the model parameters Θ (leaf nodes). By storing these KV caches, we enable the complete reconstruction of any subsequent chunk's output during backpropagation, making them ideal checkpoint candidates. During the forward propagation, only the checkpointed KV caches need to be computed and stored, eliminating the need to retain intermediate activations.

4 Sequential Chunk-wise Optimization

SeCO is a plain version of this chunk-wise optimization that does not save any computation and obtains exact gradients (verified in Appendix D).

4.1 Methodology

During forward propagation, we compute all chunks sequentially in inference mode to generate corresponding KV caches $\{m'_1, m'_2, \ldots, m'_k\}$, where prime notation distinguishes inference-generated caches from training-phase counterparts.

For backpropagation, the computational graph topology in Figure 1 dictates a sequential reverse-order reconstruction strategy. For chunk j:

1. Reconstruct computational graph using Eq. (1) to compute error J_j and KV cache m_j.
2. Transfer gradients from m'_j to m_j.
3. Backpropagate J_j and m_j to accumulate gradients for model parameters and preceding checkpoints $m'_1, \ldots, m'_{j-1}$.

After processing all chunks, accumulated parameter gradients match those from naive parallel training modulo numerical precision. Implementation details follow Algorithm 1, and the corresponding visualization is presented in Figure 3.

4.2 Efficiency

We analyze the theoretical efficiency of SeCO, in terms of computation and storage.

Memory Savings. By reconstructing at most one chunk's computational graph at any given time, SeCO effectively prevents forward activations from scaling linearly with sequence length. This design reduces the memory requirements for storing forward activations by a factor of k. However, it is important to note that SeCO does not optimize fixed memory components such as optimizer states and model parameters, nor does it alleviate the memory overhead of the KV cache.

Computational Overhead. SeCO introduces two primary sources of computational overhead: *(i)* additional recomputation during backpropagation, and *(ii)* frequent kernel launches for small-scale tensor operations.

For the first component, since backpropagation typically requires approximately twice the FLOPs of forward propagation (DeepSpeed, 2021; Kaplan, 2019), the subgraph reconstruction introduces an estimated 33% computational overhead.

Regarding the second component, modern GPUs like the RTX 3090 contain fixed computational resources (82 streaming multiprocessors with 128 cores each). When using sufficiently large chunk sizes, these resources can achieve near-saturation utilization. Experimental results demonstrate that increasing chunk sizes beyond 128 yields diminishing returns, with only marginal reductions in computational time observed.

Algorithm 1 Sequential Chunk-wise Optimization

Require: Model f, data $X = \{x_1, x_2, \ldots, x_k\}$, parameters Θ
Ensure: ∇_Θ

```
for i = 1 to k do
    m'_i ← f(x_i; {m'_j}_{j=1}^{i-1}; Θ)
end for
for i = k to 1 do
    J_i, m_i ← f(x_i; {m'_j}_{j=1}^{i-1}; Θ)
    m_i.grad ← m'_i.grad
    backprop(J_i)
end for
```

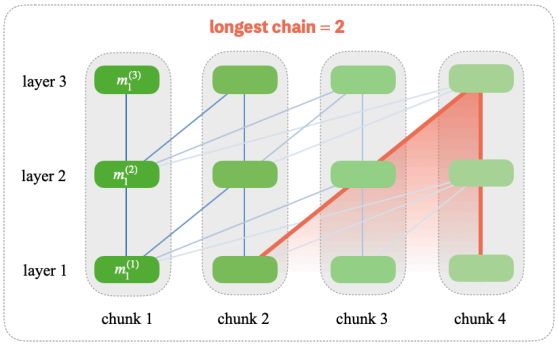


Figure 4: In the Transformer architecture, the gradient flow traverses through at most a number of KV caches equal to the layer depth. This observation was also highlighted in Transformer-XL (Dai et al., 2019).

5 Sparse Chunk-wise Optimization

While SeCO reduces memory consumption, it introduces additional computational overhead, further prolonging the already time-consuming training process. This limitation hinders its ability to handle ultra-long sequences efficiently. To alleviate this issue, we propose SpaCO, an improvement over SeCO. By introducing sparsification in backpropagation, SpaCO significantly accelerates training while maintaining memory efficiency.

The key insight stems from the observation that the checkpoints $\{m'_1, \ldots, m'_k\}$ enable independent computational graph construction for individual chunk through Eq. (1). Capitalizing on this, SpaCO implements a stochastic backpropagation scheme that randomly selects a subset of chunks for gradient computation during each training iteration.

This sparsification, however, poses the risk of biased gradient estimation. In extreme cases where only one chunk is selected, the gradient flow between chunks is disrupted—akin to non-overlapping chunked attention mechanisms, which constrain the model to local dependencies.

One might hypothesize that dense gradient propagation is essential for learning global patterns, given that the longest gradient chain spans all KV cache chunks. However, our theoretical analysis reveals that this is not necessary.

The Longest Gradient Chain. In the computational graph shown in Figure 1, the longest gradient chain is:

$$\frac{\partial J_4}{\partial m_4} \cdot \frac{\partial m_4}{\partial m_3} \cdot \frac{\partial m_3}{\partial m_2} \cdot \frac{\partial m_2}{\partial m_1} \cdot \frac{\partial m_1}{\partial \Theta}, \tag{4}$$

which spans all chunks. Omitting any chunk would break this chain, leading to biased gradient estimation. However, in the Transformer (Vaswani et al., 2017) architecture, KV cache chunks within the same layer are independent and computed in parallel. As a result, errors propagate from one KV cache chunk to another only between adjacent layers (Dai et al., 2019), as shown in Figure 4. Thus, the maximum gradient chain length is bounded by the number of layers. Theoretically, unbiased gradient estimation is achievable if the number of selected chunks meets the number of layers, ensuring sufficient coverage.

Challenges in Unbiased Estimation. While bounded gradient chain length suggests theoretical feasibility, practical implementation faces significant hurdles. Consider a DAG with n nodes and $n(n-1)/2$ edges, as shown in Figure 5 (*Left*). The number of p-length paths follows combinatorial principles:

$$d_p = \binom{n}{p+1} = \frac{n!}{(p+1)!(n-p-1)!}. \tag{5}$$

Let superscripts d and s denote dense (k chunks) versus sparse (t chunks) configurations respectively. The path count ratio between these two configurations exhibits:

$$\frac{d_p^{\mathsf{d}}}{d_p^{\mathsf{s}}} = \frac{k(k-1)(k-2)\cdots(k-p)}{t(t-1)(t-2)\cdots(t-p)}, \tag{6}$$

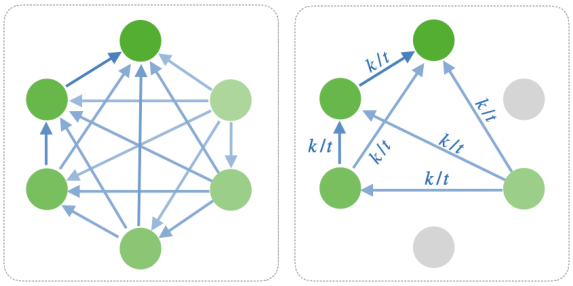
for $t \gg p$, this simplifies to:

$$\frac{d_p^{\mathsf{d}}}{d_p^{\mathsf{s}}} \approx \left(\frac{k}{t}\right)^{p+1}. \tag{7}$$

This exponentially decaying pattern highlights a crucial insight: graph sparsification disproportionately weakens longer gradient chains. Since removing a node multiplicatively affects all paths passing through it, longer chains suffer a greater cumulative impact. Consequently, naive sparsification introduces systematic bias.

To ensure unbiased estimation, we must incorporate compensation mechanisms that counteracts this attenuation. A viable solution is to strategically apply scaling factors to the preserved paths, effectively rebalancing gradient contributions across different chain lengths.

Compensation Factor. To analyze gradient propagation under sparsification, we consider all gradient



(a) Dense graph (k chunks). (b) Sparse graph (t chunks).

Figure 5: SpaCO sparsifies the gradient flow among KV caches. (*Left*) The original graph. $k = 6$. (*Right*) Only the gradient flow between $t = 4$ chunks is retained. By adding a factor k/t to each path, the gradient computed from this sparse graph remains an unbiased estimate.

Algorithm 2 Sparse Chunk-wise Optimization

Require: Model f, data $X = \{x_1, x_2, \ldots, x_k\}$, parameters Θ, fixed budget t
Ensure: ∇_Θ

```
for i = 1 to k do
    m'_i ← f(x_i; {m'_j}_{j=1}^{i-1}; Θ)
end for
Randomly select t distinct indices from {1, ..., k}, denoted as I.
for i in I do
    J_i, m_i ← f(x_i; {m'_j}_{j=1}^{i-1}; Θ)
    m_i.grad ← (k/t) · m'_i.grad·
    backprop(J_i)
end for
```

chains of length p in Eq. (3), denoted by $\mathbf{z}_p$:

$$\mathbf{z}_p = \sum_{i<t_1<\ldots<t_{p-1}\leq j} \frac{\partial J_j}{\partial m_{t_{p-1}}} \cdot \frac{\partial m_{t_{p-1}}}{\partial m_{t_{p-2}}} \cdot \ldots \cdot \frac{\partial m_i}{\partial \Theta}. \tag{8}$$

A crucial observation is that any gradient chain in $\mathbf{z}_p$ necessitates the sampling of all its constituent chunks. The survival probability of such a chain under t-out-of-k sparse sampling can be derived as $(t/k)^p$ based on the following reasoning:

- The initial term $\partial J_j / \partial m_{t_{p-1}}$ survives with probability t/k.
- Given that the previous term survives, each subsequent term (except the last) also survives with probability t/k.

Using this survival probability, we can express the expected value of $\mathbf{z}_p$ after sparsification, denoted as $\bar{\mathbf{z}}_p$:

$$\bar{\mathbf{z}}_p = \left(\frac{t}{k}\right)^p \mathbf{z}_p + \left(1 - \frac{t^p}{k^p}\right) 0 = \left(\frac{t}{k}\right)^p \mathbf{z}_p. \tag{9}$$

Given the complete gradient $\mathbf{Z} = \sum_{p=1}^{\infty} \mathbf{z}_p$ from Eq. (3), its expectation after sparsification becomes:

$$\bar{\mathbf{Z}} = \frac{t}{k}\mathbf{z}_1 + \left(\frac{t}{k}\right)^2 \mathbf{z}_2 + \left(\frac{t}{k}\right)^3 \mathbf{z}_3 + \ldots \tag{10}$$

To achieve unbiased gradient estimation ($\bar{\mathbf{Z}} = \mathbf{Z}$), each gradient chain $\mathbf{z}_p$ requires compensation by factor $(k/t)^p$. This multiplicative scaling counteracts the exponential decay induced by sparsity.

Implementation. The compensation factor is implemented through modifying backpropagation, which occurs during gradient computation: When calculating $\partial m_i / \partial \{m'_j\}_{j=1}^{i-1}$, we scale the gradient by k/t. Through the nested structure of $f(\cdot)$, this creates compound compensation where each p-length chain automatically accumulates $(k/t)^p$ scaling through successive operations.

Figure 5 (*Right*) illustrates this dynamic scaling mechanism, with full implementation details provided in Algorithm 2. This approach enhances computational efficiency while maintaining the statistical accuracy of full backpropagation.

6 Experiment

We designed experiments to address two key questions:

- How do SeCO and SpaCO compare to mainstream training methods in time and memory efficiency?
- Can SpaCO provide reliable gradient estimation and maintain competitive performance compared to exact gradient training?

The following sections present our comprehensive analysis. Additional experiments, including the verification of gradient computation accuracy for SeCO, are presented in Appendix D.

6.1 Experimental Setup

Our experiments utilize the LLaMA3-8B (Meta-AI, 2024) as the base model, implementing LoRA (Hu et al., 2022) fine-tuning with hyperparameters $r = 8$ and $\alpha = 16$. The training datasets comprises 1,000 instances sampled from the PG19 training split (Rae et al., 2018), with sequence lengths truncated to 16K tokens.

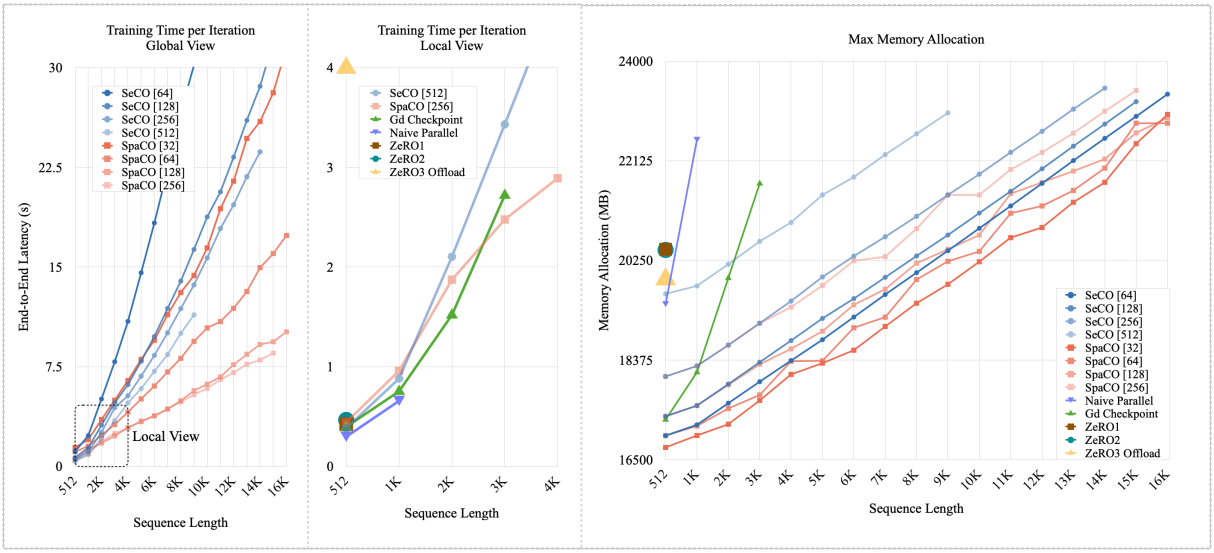


Figure 6: (*Left*) Compared to SeCO, SpaCO achieves lower training time with more favorable linear-like scaling as the sequence length increases. (*Middle*) A zoomed-in view of the left panel shows that SeCO only incurs ∼30% additional time overhead compared to naive parallel training, significantly outperforming ZeRO3 offload which suffers from GPU-CPU communication bottlenecks and demonstrates approximately 10× slower training speed. (*Right*) SeCO and SpaCO exhibit superior memory efficiency, achieving more than 4× memory reduction compared to standard gradient checkpointing and an order-of-magnitude improvement over naive parallel training.

6.2 Baseline Methods

We compare our methods against three mainstream training paradigms: DeepSpeed (MicroSoft, 2021), conventional layer-level gradient checkpointing, and naive parallel training, all of which leveraging FlashAttention (Dao, 2024).

DeepSpeed. DeepSpeed (MicroSoft, 2021) is a high-performance distributed training framework based on *Fully Sharded Data Parallel* (FSDP). It provides three optimization levels, ZeRO1/2/3, which progressively reduce memory consumption by distributing optimizer states, gradients, and parameter across multiple GPUs. ZeRO3 offload further extends ZeRO3 by offloading these components to CPU for additional memory savings. We evaluate DeepSpeed on 8 RTX 3090 GPUs with default ZeRO1/2 configurations and a custom ZeRO3 setup (Appendix C.2). Notably, FSDP offers limited benefits in parameter-efficient scenarios, where its advantages may not fully emerge.

Gradient Checkpointing. A standard gradient checkpointing implementation on a single RTX 3090 GPU, placing checkpoints at each LLM layer's input hidden states to reduce memory usage.

Naive Parallel Training. A baseline implementation on a single RTX 3090 GPU, utilizing no memory-efficient techniques except for FlashAttention. This serves as a reference for time efficiency.

6.3 Efficiency Analysis

We evaluate time and memory efficiency of SeCO and SpaCO implemented on a single RTX 3090 GPU with varying sequence lengths.

- **Configuration:** For SeCO, we evaluate chunk sizes $\{64, 128, 256, 512\}$. For SpaCO, we use a chunk budget of $t = 8$ and test with chunk sizes $\{32, 64, 128, 256\}$.[2]
- **Measurement Protocol:** Peak memory usage recorded via PyTorch's memory profiler, and the minimum end-to-end iteration time among the first 10 iterations is reported.

Our findings are concluded in Figure 6.

Practical Guidance. Based on our findings, we recommend maximizing the chunk size within the available memory limits. This accelerates training while maintains the same memory scalability.

6.4 Effectiveness Analysis

While SpaCO theoretically ensures unbiased gradient estimation through compensation factors, its increased gradient estimation variance raises practical effectiveness concerns. To investigate this, we evaluate its performance using a common training recipe for context window extension: As outlined

[2]We observe that $t = 8$ already achieves satisfactory performance in practice.

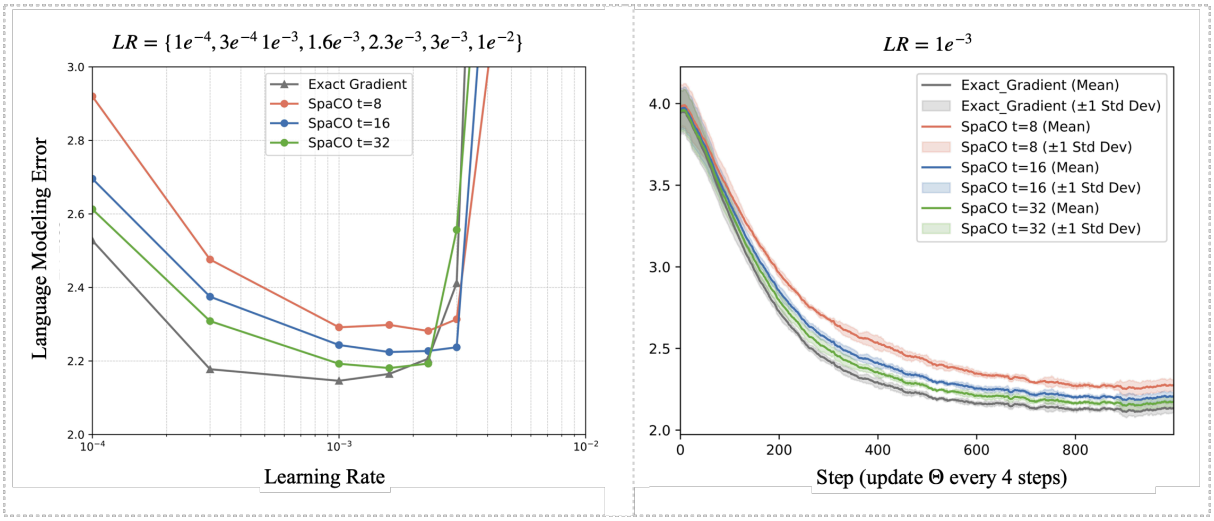


Figure 7: (*Left*) Performance comparison of SpaCO across $t = \{8, 16, 32\}$ under varying learning rates, using the same training setup and random seed. (*Right*) EMA-smoothed learning curves (α=0.95) for SpaCO and exact gradient method, both trained with the optimal learning rate of 1e-3.

in the experimental setup, we extend LLaMA3-8B's original 8K window to 16K via LoRA.

For SpaCO, we fix the chunk size to 128 and conduct experiments with budgets of $t = \{8, 16, 32\}$, corresponding to sparsity ratios of $1/16$, $1/8$ and $1/4$ respectively. The training results using model parallelism combined with gradient checkpointing serves as the baseline reference. All training runs use a batch size of 4, allowing for a total of 250 parameter updates. To mitigate numerical instability and gradient vanishing or explosion, we limit the compensation factor to a maximum value of 2.[3]

Performance Under Varying Learning Rates. SpaCO introduces additional noises to the training process, necessitating independent tuning of the learning rate. To this end, we perform grid search over seven learning rates {1e-4,3e-4,1e-3,1.6e-3,2.3e-3,3e-3,1e-2} to identify the optimal values. Results are shown in Figure 7 (*Left*).

Learning Curves. We further record the learning curves for each configuration using the optimal LR of 1e-3, as determined from Figure 7 (*Left*). Training is conducted across four random seeds (controlling dataset shuffling), and we record the mean trajectories along with ±1 standard deviation bands. Results are presented in Figure 7 (*Right*). The results demonstrate that with proper hyperparameter tuning, SpaCO achieves comparable performance to exact gradient training.

[3]Even in the absence of the compensation factor, excessively long gradient chains often result in vanishing or exploding gradients, diminishing their overall impact.

Practical Guidance. We recommend setting an upper bound (*e.g.*, 2) on the compensation factor to reduce gradient estimation variance. While omitting this constraint does not compromise training stability, it may lead to sub-optimal results. Furthermore, although our experiments used fixed batch size and training iterations for fair comparisons, we suggest using larger batch size and more training epochs than exact gradient training. This adjustment can promote better results in practical applications.

7 Conclusion

To address the critical challenge of efficiency in long-context LLM training, we introduce two training paradigms: SeCO and its enhanced variant SpaCO. By partitioning the input sequence into smaller, manageable chunks and performing localized backpropagation for each chunk, SeCO achieves substantial memory savings. Building upon this foundation, SpaCO introduces a carefully designed sparsification mechanism that randomly selects few chunks for backpropagation, reducing computational overhead. The integration of a mathematically-grounded compensation factor ensures unbiased gradient estimation. Our methods achieve impressive memory efficiency, enabling the fine-tuning of 8B models with 16K tokens on a single RTX 3090 GPU. This represents a 16× memory reduction compared to naive parallel training. SeCO and SpaCO significantly lower the barrier for practitioners working with long-context LLMs.

Limitations

SeCO and SpaCO each present unique advantages but also have exhibit distinct limitations. SeCO achieves accurate gradient computation and efficient memory usage but suffers from a quadratic increase in computation with sequence length, making it impractical for training on ultra-long sequences. In contrast, SpaCO significantly reduces computational cost and maintains comparable memory efficiency but sacrifices gradient accuracy, introducing substantial randomness that complicates convergence. Ultimately, no single training strategy perfectly balances the trade-offs in all training scenarios. A practical approach requires identifying an optimal balance among the "impossible triangle" of computation, memory efficiency, and gradient accuracy.

Ethics Statement

By optimizing memory consumption and computational efficiency, our approach not only lowers the financial barriers to training such models but also reduces energy consumption, contributing to more sustainable AI practices.

However, as with any significant technological advancement, ethical concerns must be considered. Lowering the cost and resource requirements for training long-context models may inadvertently enable the misuse of these models, including the creation of harmful or malicious language systems. It is essential to address these risks through responsible research practices and the development of robust safeguards.

References

Chenxin An, Fei Huang, Jun Zhang, Shansan Gong, Xipeng Qiu, Chang Zhou, and Lingpeng Kong. 2024. Training-free long-context scaling of large language models. In *ICML*.

Léon Bottou and Olivier Bousquet. 2007. The tradeoffs of large scale learning. In *NeurIPS*.

Yaroslav Bulatov. 2018. Fitting larger networks into memory. Medium.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv*.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2024. LongloRA: Efficient fine-tuning of long-context large language models. In *ICLR*.

Soumith Chintala, Gregory Chanan, Dmytro Dzhulgakov, Edward Yang, and Nikita Shulga. 2016. pytorch/pytorch. Github.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In *ACL*.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *ICLR*.

Harm de Vries. 2023. In the long (context) run. Personal website.

DeepSpeed. 2021. Deepspeed's flops profiler. Deepspeed documentation.

Google. 2015. tensorflow/tensorflow. Github.

Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. 2022. huggingface/accelerate. Github.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *ICLR*.

Jared Kaplan. 2019. Notes on contemporary machine learning for physicists. Semantic Scholar.

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

Jerry Liu. 2022. run-llama/llama_index. Github.

Meta-AI. 2024. The llama 3 herd of models. Technical report.

MicroSoft. 2021. microsoft/deepspeed. Github.

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

Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. 2018. google-deepmind/pg19. Github.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv*.

Table 1: Arguments for DeepSpeed ZeRO3 offload

Argument	Value
overlap_comm	true
contiguous_gradients	true
reduce_bucket_size	5e8
stage3_max_live_parameters	1e9
stage3_max_reuse_distance	1e9
stage3_prefetch_bucket_size	5e8

A Experimental Datasets

PG19 Dataset. The PG19 corpus, an open-source long-text dataset released by DeepMind, is derived from books in the Project Gutenberg repository published prior to 1919. This collection is supplemented with metadata containing book titles and publication dates. For model training, we randomly selected 1,000 samples from the PG19 training partition. To ensure consistent sequence lengths, text samples exceeding 16K tokens were truncated to this threshold. The PG19 dataset is publicly available under the Apache License 2.0.

B Language Models

LLaMA3-8B. The LLaMA3-8B model, an open-source large language model developed by Meta AI, serves as the foundational model in our experiments. This selection is motivated by its widespread adoption within the research community. The licensing terms for the LLaMA3 series models are governed by the Meta Llama 3 Community License Agreement, which notably permits academic and commercial use with specific attribution requirements.

C Implementation Details

C.1 Pseudocode

The workflows of SeCO and SpaCO primarily manage the KV cache, focusing on its updates and the relay of gradients during backpropagation. These operations require overriding the default backpropagation mechanism in deep learning frameworks, which poses implementation challenges. To clarify this process, we provide pseudocode below.

C.2 ZeRO3 Offload

Detailed configurations are provided in Table 1.

Table 2: Training results of SeCO vs. Model Parallelism (Baseline) across different learning rates.

Method	LR		
	1e-4	3e-4	1e-3
Baseline	2.52	2.16	2.13
SeCO	2.53	2.18	2.15

D Additional Results

Direct Validation of Gradient Accuracy. To assess the accuracy of the computed gradients, we conducted experiments using Qwen2.5-0.5B with `float64` precision. Gradients were obtained for sequences of 512 tokens using both naive parallel training and SeCO (with a chunk size of 64) and then compared element-wise. The results show that the gradients computed with SeCO achieve a precision exceeding 12 decimal places. The test code for this experiment is publicly available in our repository under the `test_estimate` directory.

Indirect Validation of Gradient Accuracy. To evaluate SeCO's performance in real training scenarios, we follow the experimental setup described in the main text. We compare SeCO's training results with those obtained using model parallelism and gradient checkpointing. The results are summarized in Table 2.

The minor performance gap may be attributed to numerical issues arising from the increased number of operations in SeCO. For example, FlashAttention introduces randomness during backpropagation due to the use of atomic additions (see Github issue). Since SeCO involves tens of times more such operations than parallel training, it exhibits greater numerical instability.

```
def update_kv_cache(kv_cache, keys, vals):
    try:
        return concat(kv_cache.keys, keys), concat(kv_cache.vals,
                vals)
    finally:
        if is_gradient_enabled():
            kv_cache.keys.append(keys)
            kv_cache.vals.append(vals)
        else:
            k_detach, v_detach = keys.detach(), vals.detach()
            k_detach.requires_grad_(), v_detach.requires_grad_()
            kv_cache.keys.append(k_detach)
            kv_cache.vals.append(v_detach)


def grad_hook(grad, base, scaler=1):
    return grad + base * scaler


def copy_grad(a, b):
    for ak, av, bk, bv in zip(a.keys, a.vals, b.keys, b.vals):
        bk.register_hook(partial(grad_hook, base=ak.grad))
        bv.register_hook(partial(grad_hook, base=av.grad))
```