SPPO: Making Million-Token LLM Training Practical on Modest GPU Clusters

qiaoling chen
Nanyang Technology University
Singapore, Singapore
qiaoling.chen@ntu.edu.sg

Shenggui Li
Nanyang Technological University
Singapore, Singapore
shenggui001@e.ntu.edu.sg

Wei Gao
Hong Kong University of Science and Technology
Hongkong, China
csgaowei@ust.hk

Peng Sun
Shanghai AI Laboratory
SHANG HAI, China
sunpeng@pjlab.org.cn

Yonggang Wen
Nanyang Technological University
Singapore, Singapore
ygwen@ntu.edu.sg

Tianwei Zhang
Nanyang Technological University
Singapore, Singapore
tianwei.zhang@ntu.edu.sg

Abstract

In recent years, Large Language Models (LLMs) have exhibited remarkable capabilities, driving advancements in real-world applications. However, training LLMs on increasingly long input sequences imposes significant challenges due to high GPU memory and computational demands. Existing solutions face two key limitations: (1) memory reduction techniques, such as activation recomputation and CPU offloading, compromise training efficiency; (2) distributed parallelism strategies require excessive GPU resources, limiting the scalability of input sequence length.

To address these gaps, we propose Adaptive Sequence Pipeline Parallel Offloading (SPPO), a novel framework that optimizes memory and computational resource efficiency for long-sequence LLM training. SPPO introduces adaptive offloading, leveraging sequence-aware offloading, and two-level activation management to reduce GPU memory consumption without degrading the training efficiency. Additionally, it develops an adaptive pipeline scheduling approach with a heuristic solver and multiplexed sequence partitioning to improve computational resource efficiency. We perform comprehensive experiments to demonstrate the superiority of SPPO: it achieves up to 3.38× throughput improvement over state-of-the-art methods like Megatron-LM, DeepSpeed and TeraPipe, realizing efficient training of a 7B LLM with sequence lengths of up to 4M tokens on only 128 A100 GPUs.

ACM Reference Format:
qiaoling chen, Shenggui Li, Wei Gao, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2026. SPPO: Making Million-Token LLM Training Practical on Modest GPU Clusters. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3797905.3800537

1 Introduction

Large Language Models (LLMs) have demonstrated exceptional capabilities, revolutionizing a multitude of domains including code generation [16, 40], image processing [30, 32], video stream analysis [41, 51], and scientific research [2, 3]. Many of these applications increasingly rely on long-range contextual understanding, such as reasoning over large codebases, processing lengthy documents or videos, and integrating heterogeneous scientific data. This trend has driven the rapid expansion of LLM context lengths, from early models supporting 4K tokens [43, 45] to more recent systems capable of handling 32K [19, 27], 128K [1, 13], and even millions of tokens [4, 9, 26, 36]. Training LLMs with such long sequences imposes significant demands on GPU memory and computational resources, as the activation computation and memory in training scales with the sequence length. For instance, training a 7B-parameter GPT-style model with a sequence length of 4 million tokens requires approximately 16,384 GB of activation memory and at least 128 NVIDIA H100 GPUs running for about 105 hours to process just 1 billion tokens, illustrating the prohibitive cost of naïvely scaling context length in current LLM training pipelines.

Therefore, many system optimizations have been proposed to improve the memory or computational resource efficiency of long-sequence LLM training. However, they are not sufficient to achieve satisfactory performance due to the following two significant gaps.

• **G1**: ***Memory reduction techniques compromise the training efficiency***. Two prominent memory reduction techniques have been widely adopted to alleviate memory overhead in LLM training: activation recomputation [6] and CPU offloading [33]. Activation recomputation reduces the memory footprint by recomputing the activations of certain layers during backward propagation instead of storing them. Despite the significant memory savings, the recomputing step can account for up to one-third of the gradient computation time. This overhead is particularly pronounced in long-sequence training, substantially prolonging the overall training time. CPU offloading relieves GPU memory pressure by transferring activations between GPU and CPU memory. Prior offloading techniques [11, 49] operate the entire sequence of activations and assume that the overhead of CPU offloading can be effectively hidden by overlapping it with GPU computation. However, as the sequence length increases, the overhead of CPU offloading grows linearly, causing substantial delays in GPU computation and markedly degrading training efficiency. Also, existing CPU offloading techniques keep the activations of at least one layer in GPU memory, which limits their scalability for longer sequences (discussed in Section 2.1). Consequently, offloading the entire sequence of activations represents a *coarse-grained* approach, rendering current CPU offloading methods inappropriate for long-sequence training.

• **G2**: ***Distributed parallelism techniques consume excessive GPU memory and resources***. Many distributed parallelism strategies [15, 28, 44, 47] have been proposed to expedite long-sequence

LLM training. However, their effectiveness relies heavily on the availability of substantial GPU memory and resources. For instance, training a GPT model with 65 billion parameters and a sequence length of 4 million results in an activation memory footprint of 80TB, necessitating **over 1,024 NVIDIA H100 GPUs** to store activations and model weights. Such enormous resource requirements significantly hinder the scalability of sequence lengths in long-sequence training.

Both gaps arise from a common reason: *granularity mismatch*. By treating a long context as an indivisible unit, offloading transfers are too coarse to hide, and pipeline schedules are too rigid, producing large bubbles and stalls. Therefore, a more promising strategy is to *perform CPU offloading and pipeline scheduling over subsequences, instead of entire sequences*. We expect such adaptations could bring two potential advantages for long-sequence LLM training: (1) Subsequence-level activation offloading benefits the overlapping of GPU computation and CPU offloading. (2) Subsequence pipeline scheduling can reduce pipeline bubbles and improve training efficiency. To our best knowledge, there are no works considering subsequence-level CPU offloading, and only one work (TeraPipe [25]) demonstrates that subsequence-level pipelining can reduce bubbles. However, TeraPipe does not resolve activation memory and transfer-time issues. Simply applying activation offloading and pipeline scheduling over subsequences faces two practical changes.

• ***Inefficient fixed offloading policies.*** Existing solutions typically rely on fixed subsequence partitioning or static offloading policies. Under a length-based policy, subsequences are partitioned into equal token lengths, which fails to account for the computational imbalance arising from attention operations, where later tokens incur higher computational cost. Alternatively, a FLOPs-based policy equalizes computation across subsequences, but at the cost of imbalanced activation memory consumption across subsequences, making CPU offloading overhead outweigh GPU computation. Furthermore, the fixed subsequence offloading strategy introduces additional tensor dependencies among subsequences, further amplifying redundant data transfers and offloading overhead.

• ***Inefficient fixed pipeline schedules.*** Existing schedule policies [12, 17, 28, 35] enforce a predetermined schedule for each subsequence, disregarding variations in memory and computational demands. More subsequences could increase kernel overhead and decrease throughput, while fewer subsequences could introduce high pipeline bubbles. Even with an optimized subsequence count N, pipeline bubbles remain inevitable. For example, with $p = 4$ and $N = 16$, the bubble ratio reaches 3/16, highlighting inefficiencies in fixed schedules, where the total computation time includes both bubble overhead and execution time.

To address these challenges, we propose Adaptive Sequence Pipeline Parallel Offloading (SPPO), a novel framework for long-sequence LLM training. It can fully exploit the potential benefits of sequence partitioning while overcoming the limitations of existing offloading policies and pipeline schedules. SPPO innovatively customizes *offloading* and *pipeline scheduling* to optimize their memory and computational resource efficiency. First, SPPO designs an **adaptive offloading approach** to efficiently overlap the offloading of the activation of a subsequence with its computation. It consists of two key components. (1) A *sequence-aware offloading* policy mitigates imbalanced memory allocation across subsequences. It adaptively computes the offload ratio to maximize the overlap between the CPU offloading for the $(i-1)^{\text{th}}$ subsequence and the computation of the i^{th} subsequence. (2) A *two-level activation management* strategy retains skeletal activations (i.e., those with high access frequency) in GPU memory, thereby preserving CPU-GPU bandwidth for activations with lower access frequency.

Second, we design an **adaptive pipeline schedule** to pipeline the computation of one subsequence in a given layer with the computation of the previous subsequence in the subsequent layer. To strike an optimal balance between GPU utilization and pipeline efficiency while explicitly considering the overlap between offloading and computation, we develop a *heuristic solver* that differs from TeraPipe in its design to determine the ideal number of subsequences. Combined with the offloading ratio in adaptive offloading, this enables us to further improve memory and training efficiency for long-sequence LLM training. Nevertheless, the *heuristic solver* is not always capable of eliminating resource bubbles in certain scenarios. To handle it, we introduce a *multiplexing sequence partitioning* to deliver a fine-grained partition of adjacent subsequences, further reducing the resource bubbles without sacrificing memory efficiency.

Our contributions are summarized as follows:

★ We present an in-depth efficiency analysis about the potential benefits and challenges of subsequence-level offloading and pipelining.

★ We design the SPPO framework to optimize LLM training at the subsequence level. By tailoring dedicated offloading techniques and optimizing the pipeline schedule, SPPO significantly improves both memory and computational efficiency for long-sequence LLM training.

★ We introduce an adaptive offloading approach, which comprises of sequence-aware offloading and two-level activation management, to maximize the overlap between CPU offloading and GPU computation.

★ We design an adaptive pipeline schedule that incorporates a heuristic solver and multiplexed sequence partitioning, enhancing training efficiency without compromising the benefits from adaptive offloading.

★ We evaluate SPPO through extensive experiments, showing up to 3.38× throughput improvement over Megatron-LM and DeepSpeed. Moreover, SPPO enables efficient training of a 7B LLM with sequence lengths of 1M, 2M, and 4M on only 32, 64, and 128 NVIDIA Ampere GPUs.

2 Background

Despite the substantial memory optimizations in attention layers achieved by FlashAttention [34, 46] and linear attention [8, 14], the linear scaling of activation memory during long-sequence LLM training still results in prohibitively high GPU memory usage. To address such overhead while maintaining computational efficiency, two primary techniques have emerged: memory reduction and distributed parallelism. However, they struggle to maintain a balance

B	batch size	PP	pipeline parallel size
H	hidden dimension	SP	sequence parallel size
S	sequence length	N	#subsequences
L	# layers	h	hidden state
A	# attention heads	α	offloading ratio
V	vocab size	s_i	subsequence i
M_a	memory of activation	M_m	memory of model

Table 1: Notations used in this paper.

between memory efficiency and computational resource efficiency simultaneously. Table 1 illustrates the notations used in this paper.

2.1 Memory Reduction Techniques

Limited GPU memory is a critical bottleneck of training LLMs with long-sequences. To address this, the LLM community has adopted activation recomputation and offloading to reduce GPU memory consumption caused by activations.

Activation recomputation [7, 20, 21], also referred to as activation checkpointing, reduces GPU memory usage by selectively discarding the activations of certain layers instead of retaining all intermediate activations. During backpropagation, the missing activations are recomputed on the fly. While activation recomputation alleviates the GPU memory consumption, the recomputation process brings additional computational overhead, degrading the overall training efficiency. For example, when we employ activation recomputation for all layers during long-sequence training, the cost of recomputing activations in backpropagation adds an extra 1/3 to the total computational time [6].

CPU Offloading [11, 39, 49] relieves the GPU memory burden by transferring activations from GPU memory to CPU and loading back to the GPU memory on demand. Unlike activation recomputation, the offloading process does not involve GPU computation. Existing offloading techniques [11, 38, 49] emphasize the ineligible cost of transferring activations between the GPU and CPU, and they attempt to maximize the overlap between the transfer process and GPU computation. However, these methods typically focus on offloading the activations of the entire input sequence for one or more layers to the CPU. During long-sequence training, the GPU memory consumption of activations grows linearly with the sequence length, quickly saturating the CPU-GPU bandwidth. As a result, the offloading process fails to effectively overlap with GPU computation, leading to training inefficiency. Worse still, these offloading techniques need to retain the activations of the entire input sequence for at least one layer in GPU memory, which significantly limits the scalability of the sequence length. For example, when training a GPT-7B LLM with a GPU equipped with 80 GB memory, retaining activations for just one layer results in a maximum sequence length of only 128K token.

2.2 Distributed Parallelism Techniques

Distributed parallelism is widely adopted to expedite LLM training. It includes several key techniques. (1) *Data Parallelism* (DP). Batches are sharded across devices, while gradients are synchronized via `all-reduce` [5, 24]. (2) *Tensor Parallelism* (TP). Within a layer, weights/activations are partitioned across devices, reducing per-GPU parameter/activation working sets at the cost of additional collectives [29]. (3) *Sequence Parallelism* (SP). The sequence dimension is partitioned across devices and synchronized via `all-gather` and `reduce-scatter` to complete attention and per-token ops [15, 18, 21, 23, 26]. (4) *Pipeline Parallelism* (PP). Layers are partitioned into stages connected by activation/gradient exchanges [17, 28]. Schedulers such as GPipe and 1F1B reduce bubbles by overlapping multiple concurrent micro-batches.

Model	L	H	A	M_m	M_a	#GPUs
GPT-7B	32	4096	32	120	16384	512+
GPT-13B	40	5120	40	234	52600	660+
GPT-65B	80	8192	64	1200	81920	1024+

Table 2: Lower bound of capacity at S=4M tokens: model (M_m) and activation (M_a) footprints in GB and minimum number of A100-80GB GPUs required for holding just activations. This shows the scale of the activation memory wall rather than endorsing any specific parallel configuration.

These techniques can accelerate LLM training but require excessive GPU resources. As shown in Table 2, training sequences with lengths of 4 million tokens can require hundreds or even thousands of GPUs. Such high GPU demand primarily arises from the massive memory required for activations. For instance, training a GPT-65B model with an input sequence length of one million tokens under SP consumes 28.8 TB of activation memory, requiring at least 288 A100 GPUs to accommodate it[1]. This underscores the critical need for an efficient approach to reduce memory consumption while maintaining training efficiency.

3 Efficiency Analysis of Subsequence-Level Offloading and Scheduling

The limitations of memory reduction and distribution parallelism techniques share a common root cause: *granularity mismatch*. Treating the long context as a single, indivisible block (for both offloading and scheduling) makes data transfers too coarse to overlap and leaves schedules too rigid to prevent idle bubbles. This naturally motivates moving both memory movement and execution scheduling to a finer subsequence granularity.

In this section, we first demonstrate that subsequence offloading and pipeline scheduling can provide potential memory and computational advantages over processing the entire sequence in long-sequence LLM training. However, realizing these advantages critically depends on the design of offloading and scheduling policies. We therefore investigate existing subsequence offloading and pipeline scheduling strategies. Unfortunately, neither of them can unlock the benefits offered by subsequence partitioning.

3.1 Potential Benefits

By partitioning a long sequence into multiple subsequences [18, 23, 25, 26], we can opportunistically employ CPU offloading and pipeline scheduling over *subsequences* to unleash the potential memory and computation advantages.

[1]This example and results in Table 2 are computed using the activation memory estimation model from [21]

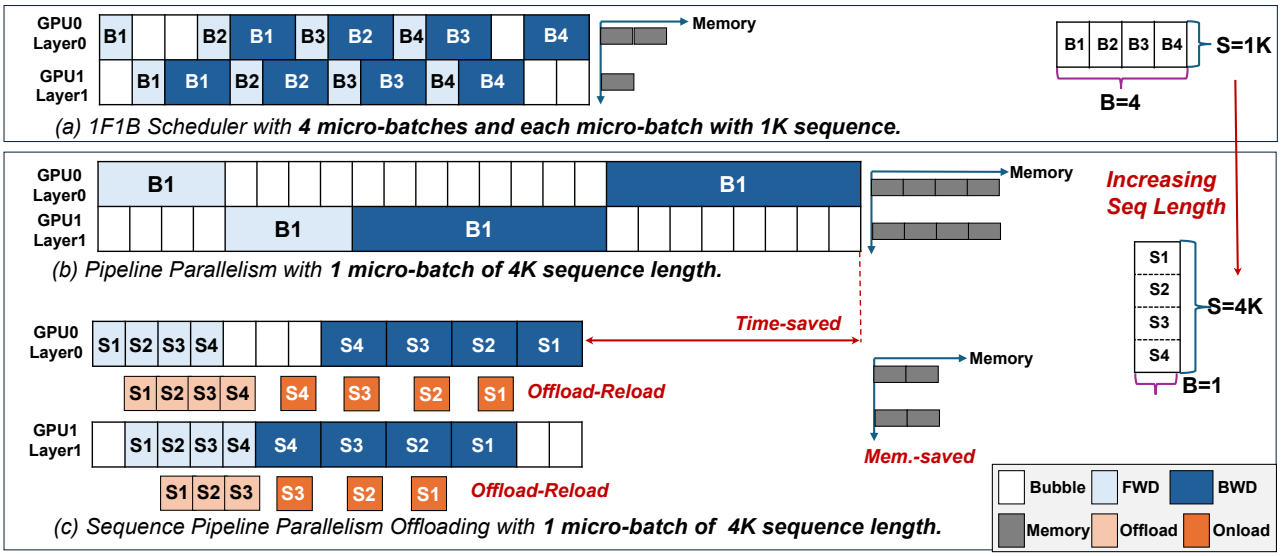

Figure 1: Illustration of the pipeline parallelism scheduling with the increasing sequence length and our sequence pipeline parallelism offloading.

• **Benefit 1**: ***Subsequence offloading***. Subsequence partitioning improves the overlap between GPU computation and CPU offloading in two aspects. First, partitioning a long sequence into subsequences reduces the granularity of both computation and data transfer, so that offloading operates on smaller units that can be more effectively pipelined with GPU execution. Second, activation memory footprint and host to device traffic scale linearly with sequence length. Under ultra-long contexts, data movement can increasingly dominate step time if transfers are not sufficiently overlapped with computation. Subsequence-level partitioning mitigates this effect by enabling finer-grained scheduling and higher effective overlap. Finally, continued improvements in host–device interfaces further increase the practical benefits of such fine-grained overlap [31, 50].

• **Benefit 2**: ***Subsequence pipeline schedule***. Incorporating subsequence partitioning with pipeline parallelism can reduce pipeline bubbles and improve training efficiency. We use a 1F1B pipeline scheduler [28] to demonstrate this in Figure 1. The 1F1B scheduler decomposes input batches into multiple micro-batches and alternates forward/backward passes as illustrated in Figure 1 (a). However, under long-sequence inputs, the micro-batch count collapses to one, thus 1F1B degenerating to the naive pipeline schedule shown in Figure 1 (b), thereby incurring larger pipeline bubbles. As illustrated in Figure 1 (c), subsequence partitioning enables the pipelining of GPU computations for one subsequence with another. This approach reduces pipeline bubbles and maximizes resource utilization, offering a more efficient pipeline schedule for long-sequence training.

3.2 Challenge 1: Inefficient Fixed Offloading

Subsequence-level CPU offloading can be achieved using some common policies from prior works: (1) in a length-based policy [18, 21, 23, 26, 48], a consistent length is fixed across partitioned subsequences; (2) in an FLOPs-based policy [25, 42], the computational FLOPs are kept consistent among the partitioned subsequences.

However, we emphasize that these *fixed* offloading strategies do not take into account the efficiency of the trade-offs between offloading and computation, as well as the access frequency patterns of the activation tensor, resulting in suboptimal overlapping between CPU offloading and GPU computation. We elaborate the reasons below.

Imbalanced GPU computation across subsequences. For the attention layer, the computational load for later token positions in a sequence is heavier than that for earlier tokens [22]. However, dividing the input sequence into multiple equal-length subsequences [21, 48] overlooks this computational load imbalance. A more efficient partitioning strategy would involve starting with longer subsequences and gradually transitioning to shorter ones.

Imbalanced memory allocation across subsequences. The FLOPs-based partitioning policy [25] ensures a consistent computational load across subsequences, as illustrated in Figure 2. However, due to the inherent computation imbalance in the attention layer, subsequence lengths vary under this policy. Since the transmission overhead caused by offloading correlates linearly with the subsequence length, the FLOPs-based offloading policy cannot overlap well between the CPU offloading and GPU computation, leading to increased GPU idle time and reduced training efficiency. As shown in Figure 3, while the computational cost remains balanced under the FLOPs-based offloading policy, the size of the activations using a fixed offloading strategy varies significantly. For 8-subsequence and 16-subsequence strategies, the activation size ranges from 10.59 GB to 2.87 GB and from 6.04 GB to 1.3 GB, respectively. Correspondingly, the transmission time ranges from 821 ms to 254 ms and from 435 ms to 90 ms, respectively. Notably, for early subsequences, the transmission time exceeds the computation time, resulting in substantial offloading overhead. This highlights the need for a more

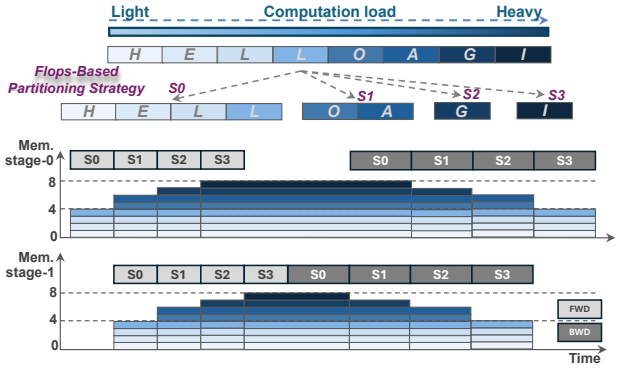

Figure 2: Background & motivation. Imbalanced computation across subsequences with its FLOPs-based offloading policy and the following memory allocation in one step.

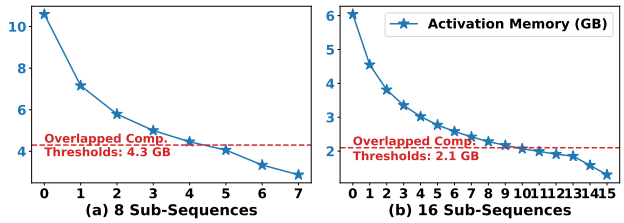

Figure 3: Activation memory allocation across subsequences when applying the FLOPs-based strategy of partitioning the sequence to 8 and 16 subsequences, respectively. The model is LLaMA-65B, and the sequence length is 128K.

fine-grained offloading policy that adapts the sequence length to maximize the overlap between computational load and activation offloading.

Unnecessary offloading overhead. Previous works have proposed to make memory offloading decisions based on the completion time of the forward pass, either at the granularity of individual layers [10, 48, 50] or batches [49]. These approaches work because the computation graph is traversed either layer-by-layer or batch-by-batch, and the intermediate activation tensors within a layer or batch are independent of those in later layers or batches. However, if we instead schedule computation at a finer granularity, such as subsequence, the tensor dependencies between subsequences increase, making offloading more complex, and the fixed offloading strategy is inefficient.

3.3 Challenge 2: Inefficient Fixed Pipeline Schedule

The fixed pipeline schedule policy, widely adopted in existing works [12, 17, 28, 35], follows a predetermined schedule for each subsequence without taking into account the varying memory and computational demands across subsequences.

Trade-off on subsequence length. Splitting a sequence into smaller subsequences introduces a trade-off between GPU utilization and pipeline efficiency. As demonstrated in Figure 4 (a), considering a single layer with a hidden dimension of 4096 and a constant sequence length of 128K, the overall forward propagation time can

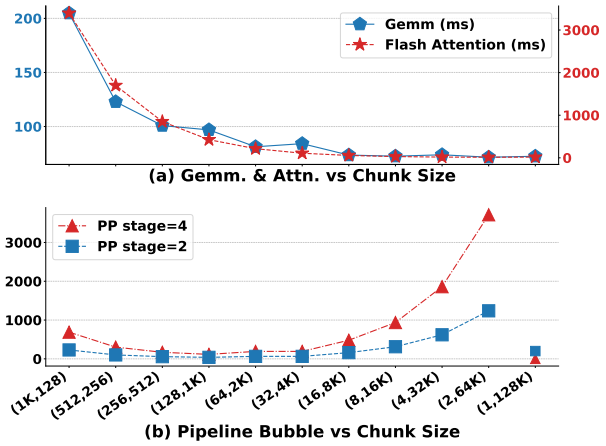

Figure 4: Relationship between subsequence length and overall forward propagation and bubble time for one transformer layer with a hidden dimension of 4096 and a fixed sequence length of 128K. x-axis (N, s) denotes a sequence is partitioned into N parts, and each part has s tokens.

be mathematically represented as the product of the number of subsequences and the time per subsequence. As the subsequence length is reduced, the total runtime escalates due to an increase in the number of subsequences and associated kernel launch overheads. Moreover, shorter subsequences lead to reduced throughput as each pipeline stage processes only a finite amount of data, thus underutilizing GPU resources. While longer subsequences are essential for higher throughput, there exists a threshold beyond which, augmenting the subsequence length ceases to enhance overall performance. This occurs as computation operators, such as attention mechanisms and generalized matrix multiplications (GEMM), stay in a compute-bound state, thereby stabilizing the total latency. On the other hand, while longer subsequences improve throughput, they introduce more pipeline bubbles, ultimately reducing overall training efficiency, as shown in Figure 4 (b). Hence, the key is to determine an *optimal* number of subsequences N to maximize the training efficiency.

Inevitable bubble overhead. Although it is possible to find an optimal balance between GPU utilization and pipeline bubbles, efficiency inevitably suffers from the presence of pipeline bubbles. To quantify this effect in this degradation pipeline parallelism schedule, we define p as the number of pipeline stages and $F(N)$ as the time to execute the forward and backward passes for all N subsequences in a sequence. The time for a single subsequence is thus $F(N)/N$. The total pipeline bubble time t_b and bubble ratio R_b are computed as:

$$t_b = (p-1) \cdot \frac{F(N)}{N}, R_b = \frac{p-1}{N}.$$

Thus, the total computation time is:

$$T = t_b + F(N) = \frac{(p-1+N)}{N} \cdot F(N).$$

To minimize the overall computation time T, we can determine an optimal value for N. However, the pipeline bubble is inevitable

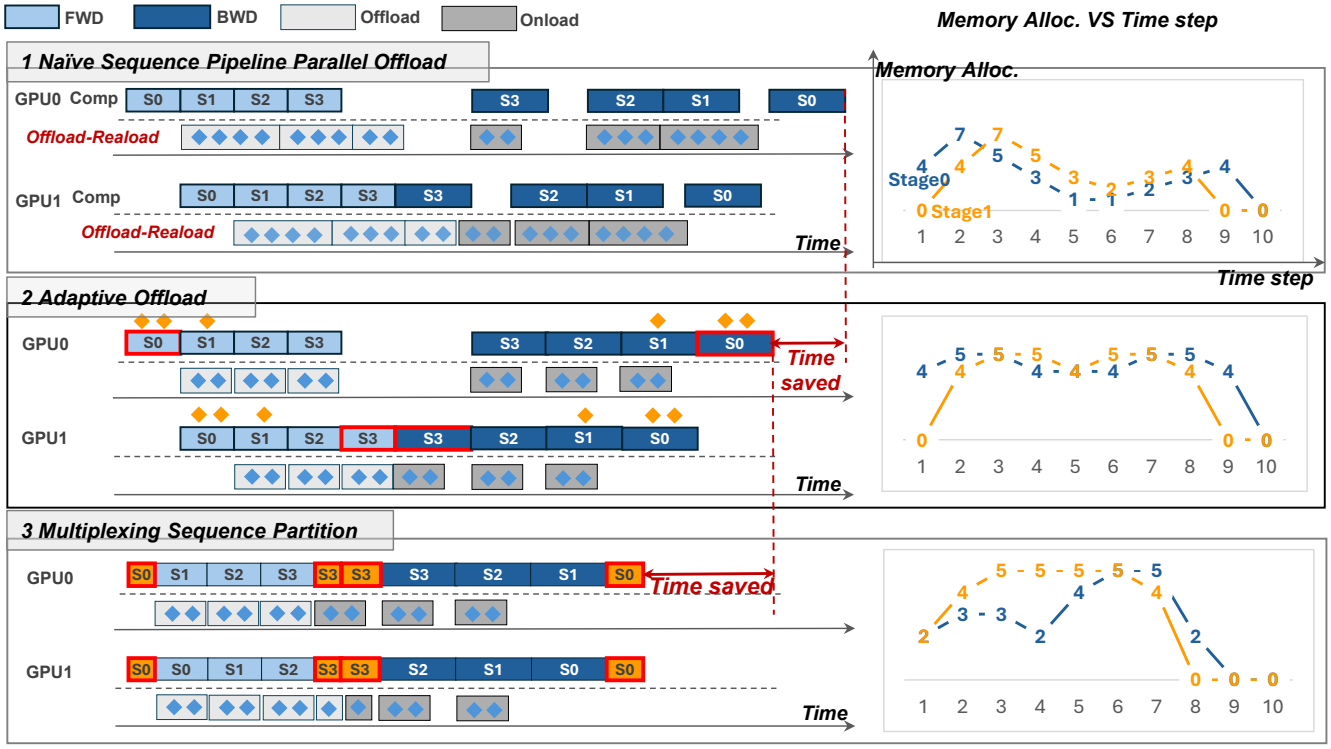

Figure 5: System Overview. The top part illustrates a minimal case of SPPO, where all subsequence activations are fully offloaded to host memory. The middle part demonstrates an adaptive offloading strategy that optimizes the minimal case, incorporating sequence-aware offloading and a two-level activation management scheme. The bottom further enhances efficiency through multiplexed sequence partitioning, achieving an almost bubble-free pipeline. The right plot visualizes the GPU activation memory allocation over time steps.

by the fixed N. For example, when $p = 4$ and $N = 16$, the bubble ratio reaches a maximum of 3/16.

4 System Overview

To enhance the memory and computational performance for long-sequence training, we propose Sequence Pipeline Parallel Offload (SPPO), a novel LLM framework that introduces innovative sequence partitioning, offloading, and pipeline scheduling customization. We introduce two approaches to address the limitations of existing offloading and pipeline solutions (Sections 3.2 and 3.3), respectively.

• ***Adaptive Offloading***. The top part of Figure 5 shows the inefficient fixed offloading (Challenge 1), which has two pipeline stages and four subsequences and employs a full offloading strategy. The offloading time of s_0 is too long, resulting in incomplete overlap with the computation. This not only delays the unloading of s_1 and increases memory consumption, but also delays the backward pass of s_0, ultimately lengthening the end-to-end latency. To address Challenge 1, we introduce an **adaptive offloading** approach that effectively overlaps the offloading of activations with computation (Section 5). It refines the offloading ratio by *sequence-aware offloading* and optimizes the selection of offloaded activation by a *two-level activation management* strategy, as depicted in the middle part of Figure 5. Each subsequence tensor is divided into multiple tensor units, categorized based on their runtime lifespan and designated for either offloading or retention. At the intra-unit level, we implement two-level activation management to handle activations of varying lifespans. At the inter-unit level, we apply sequence-aware offloading, selectively deciding whether to offload tensors to CPU or retain them in GPU memory. Adaptive offloading optimizes training efficiency and reduces peak memory consumption from 7 to 5 units.

• ***Adaptive Pipeline***. Despite improvements, pipeline bubbles remain an issue, as shown in Challenge 2 in the middle part of Figure 5. To further enhance pipeline efficiency, we introduce an **adaptive pipeline** mechanism (Section 6). It applies a *heuristic solver* to determine the ideal number of subsequences N. To reduce pipeline bubbles, this adaptive pipeline incorporates *multiplexing sequence partitioning*, which refines subsequence partitioning to minimize resource waste without increasing memory overhead, as shown in the bottom of Figure 5. By leveraging traditional SP [18], we further partition the subsequence around pipeline bubbles into smaller sub-subsequences, distributing them across GPUs for parallel computation. For instance, s_0 in stage 0 is split into two parts, allowing forward and backward passes to execute in parallel at the beginning and end of runtime, thereby reducing bubble time in stage 1. Similarly, s_3 in stage 1 is partitioned and computed in

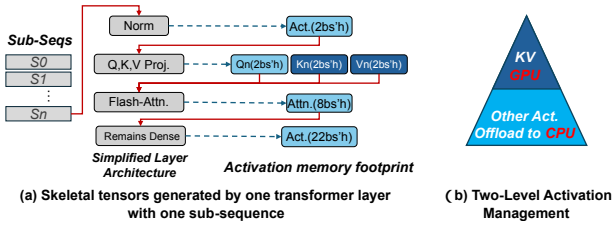

Figure 6: An example of computation of s_N in a Transformer-based model and the skeletal tensors with its sizes.

parallel, reducing bubble time in stage 0. SPPO automatically recognizes those subsequences near the bubble and re-segment them without incurring additional memory overhead.

Workflow. Given a specific sequence length and available hardware resources, SPPO operates the adaptive offloading and adaptive pipeline as follows. (1) It applies the FLOPs-based partitioning policy for balanced computation across pipeline stages instead of the length-based partition policy. Specifically, the adaptive pipeline adopts the *heuristic solver* to determine the optimal values for N and PP. (2) The adaptive offloading approach then leverages *sequence-aware offloading* to compute the offloading ratio for each subsequence and manages corresponding activations via *two-level activation management*. (3) If the pipeline scheduling still exhibits excessive bubble ratios, the adaptive pipeline applies *multiplexing sequence partitioning* to utilize these bubbles effectively, thereby enhancing training efficiency.

5 Adaptive Offloading

Traditional LLM training frameworks typically employ a uniform offloading strategy across layers or micro-batches. However, memory usage can vary significantly across different subsequences, leading to imbalanced transmission time across GPUs and CPU and, consequently, extra transmission overhead. Moreover, even within a single subsequence, different tensors often have heterogeneous lifespans and reuse patterns, making a one-size-fits-all offloading policy suboptimal. To address these challenges, SPPO enables each subsequence to independently determine its optimal offloading strategy according to its following subsequence computation.

To search for the optimal offloading strategy across different subsequences, we first analyze the inner structure of the transformer layer and decompose layer activations into finer-grained activation units. Each activation unit represents a minimal group of tensors to be offloaded or saved in GPUs. Based on this decomposition, we construct a detailed cost model and design efficient algorithms to determine the optimal offloading ratio across different subsequences.

Below, we first introduce a **two-level activation management** scheme for inner tensors within a subsequence, and then present a **sequence-aware offloading** strategy.

5.1 Two-Level Activation Management

Figure 6 (a) gives an example of how to perform the computation of subsequence s_N in a Transformer-based model, showing all skeletal tensors generated within a layer's forward pass of s_N and their sizes. Note that, for generative LLMs, due to the causal mask, after attention computation, Q_i will not be used in the forward pass anymore, while K_i and V_i need to participate in the following Q_N computation, where $i < N$. K_0 and V_0 are accessed continuously. Therefore, we cache K_i and V_i to GPU memory instead of offloading them to the CPU. Caching all (K-V) pairs only consumes $\sum_{i=0}^{n} 2Bs_iH = 2BSH$ bytes of GPU memory, while we will partially offload the remaining $36BSH$ bytes of tensors to CPU. The offloading ratio is determined by our sequence-aware offloading strategy in Section 5.2. The most important characteristic of the remaining activations is that they are needed by backward computation, so they just need to reside in GPU memory at least before the backward propagation of the subsequence begins. Figure 6 (b) shows the architecture of our activation management.

5.2 Sequence-aware Offloading

This scheduling enables overlapping data transfer operations for the $(i-1)^{th}$ subsequence with the computation of the i^{th} subsequence, thereby reducing end-to-end training time. However, incomplete offloading of the $(i-1)^{th}$ subsequence before the completion of the i^{th} subsequence computation creates dual memory demands: the GPU must simultaneously store activation A_i and maintain an offloading buffer for A_{i-1}. As demonstrated in Figure 5 (top), this memory pressure peaks at 7 activation units when S_0 offloading persists through S_2 computation.

To address the imbalanced memory utilization, we introduce an adaptive offloading ratio $\alpha \in [0, 1]$ that controls the proportion of activations transferred to host memory. Given a FLOPs-optimized subsequence partition $s = \{s_0, s_1, \ldots, s_N\}$ with $s_0 \geq s_1 \geq \cdots \geq s_N$, we assign corresponding offloading ratios $\alpha = \{\alpha_0, \alpha_1, ..., \alpha_N\}$ where $\alpha_0 \geq \alpha_1 \geq ... \geq \alpha_N$. The memory dynamics during pipeline stage i follows:

$$M_i = M_{i-1} + A_i - (\alpha_{i-1} \cdot A_{i-1})$$

where M_i is the current GPU memory consumption, A_i is the activation volume of subsequence i, and $\alpha_{i-1} \cdot A_{i-1}$ is the offloaded portion of previous activation

We optimize α to maintain $\alpha_i \cdot A_i = M_{\text{threshold}}$, ensuring the offloading time $M_{\text{threshold}}/\text{BindWidth}_{\text{D2H}}$ approximately matches the balanced computation time $T_{\text{balanced\_comp}}$ under the assumption that each GPU has exclusive access to CPU memory bandwidth, without explicitly modeling potential PCIe congestion when multiple GPUs share the same CPU socket. This assumption is also made in prior work [49]. This constraint yields the relationship $\alpha_{i-1} \cdot A_{i-1} = \alpha_i \cdot A_i$. Peak memory $M_{\max}$ occurs when $\alpha_k = 1$ for final subsequence k, resulting in:

$$M_{\max} = M_{k-1} = (1 - \alpha_{k-1}) \cdot A_{k-1}.$$

This formulation enables systematic memory optimization through our proposed scheduling algorithm.

6 Adaptive Pipeline

6.1 Heuristic Solver

Given the model, sequence length s, and the number of nodes, the objective is to search for a set of hybrid parallelism parameters (SP, PP, N) that minimize the time per iteration T. We focus on ultra-long sequence workloads where the memory footprint per sequence dominates execution, and sub-sequence parallelism (SP) is

the primary mechanism to enable scalability. To reduce the search space for performance tuning, we leverage the following heuristics:

- **Avoid cross-node SP**: TP incurs significantly larger communication overhead compared to PP within each device, making cross-node SP inefficient.
- **Allow cross-node PP**: Pipeline parallelism is explicitly allowed to span node boundaries. PP communication only occurs at stage boundaries and is significantly lighter than SP communication, making PP the preferred form of parallelism for scaling across nodes.
- **Maintain near-balanced workloads**: Empirical profiling (as shown in Figure 4) suggests that the optimal workload size per layer ranges from 2K to 16K.

Following these guidelines, we restrict N within the range of 2K to 16K per layer per device, facilitating efficient workload distribution and minimizing communication overhead. For long-sequence workloads, tensor parallelism (TP) is not considered due to its excessive activation and communication cost, while data parallelism (DP) is implicitly determined by the remaining world size after selecting SP and PP, rather than being an independent optimization variable. All sequences are assumed to be long and of uniform length; dynamic or mixed-length batching is outside the scope of this work.

6.2 Multiplexing Sequence Partition

To address significant pipeline bubbles caused by reduced values of N, we propose a latency optimization strategy termed *multiplexing sequence partitioning*. Building on SPPO's adaptive offloading, our method introduces finer-grained partitioning of bubble-adjacent subsequences across distributed GPUs using the traditional Megatron sequence parallelism (Section 2.2). This approach effectively minimizes idle time in pipeline stages while avoiding additional memory overhead.

Formally, for a PP-stage pipeline processing N subsequences, the forward pass of each stage consists of three computation phases: (1) **Left-SP**: initial parallel computation phase leveraging distributed subsequences. (2) **Steady**: central computation phase optimized through adaptive offloading. (3) **Right-SP**: final parallel computation phase, continuing sequence parallelism.

Each phase is characterized by three properties: (1) execution paradigm, (2) subsequence identifier mapping, and (3) communication scope. The backward pass follows a similar yet asymmetric structure. While Left-SP and Right-SP extend SPPO with secondary sequence partitioning, Steady maintains the core adaptive offloading optimization.

For pipeline stage $i \in \{0, 1, \ldots, PP-1\}$, its phase characteristics are defined as follows:

Definition 6.1 (Subsequence Identification Mapping). Given partition size $PP \in \mathbb{N}$, number of subsequences $N \in \mathbb{N}$, and stage index i, the phase-specific subsequence IDs $\mathcal{I}(i)$ are:

$$\mathcal{I}(i) = \begin{cases} \{x \in \mathbb{N}_0 \mid 0 \le x \le PP-1-i\} & \text{(Left-SP)} \\ \{x \in \mathbb{N}_0 \mid PP-1-i \le x \le N-i\} & \text{(Steady)} \\ \{x \in \mathbb{N}_0 \mid N-i \le x \le N-1\} & \text{(Right-SP)} \end{cases} \quad (1)$$

where $\mathbb{N}_0$ denotes non-negative integers.

Component	Stage 0	Stage 1	Stage 2	Stage 3
Left Seq. IDs	{0, 1, 2}	{0, 1}	{0}	∅
Steady Seq. IDs	{3, 4, 5, 6, 7}	{2, 3, 4, 5, 6}	{1, 2, 3, 4, 5}	{0, 1, 2, 3, 4}
Right Seq. IDs	∅	{7}	{6, 7}	{5, 6, 7}
Left SP Range	{0, 1, 2, 3}	{1, 2, 3}	{2, 3}	∅
Right SP Range	∅	{0, 1}	{0, 1, 2}	{0, 1, 2, 3}

Table 3: Illustration of multiplexed sequence partitioning for $PP = 4$ stages and $N = 8$ subsequences. SP ranges denote GPU allocations for parallel computation.

Definition 6.2 (Inter-Stage Communication Scope). The communication range $C(i)$ for stage i is:

$$C(i) = \begin{cases} \{x \in \mathbb{N}_0 \mid i \le x \le PP-1\} & \text{(Left-SP)} \\ \{x \in \mathbb{N}_0 \mid 0 \le x \le PP-1\} & \text{(Steady)} \\ \{x \in \mathbb{N}_0 \mid 0 \le x \le i\} & \text{(Right-SP)} \end{cases} \quad (2)$$

Figure 5 illustrates this mechanism: Subsequence s_0 in Stage 0 undergoes partitioning, enabling parallel execution of forward and backward passes during the initial and final phases. Simultaneously, s_3 in Stage 1 is partitioned similarly, facilitating distributed computation and reducing bubble durations in adjacent stages. Our approach dynamically identifies critical subsequences near pipeline bubbles and performs memory-efficient repartitioning without reallocation.

Table 3 illustrates this partitioning scheme for $PP = 4$ pipeline stages and $N = 8$ subsequences. The Left-SP phase processes edge subsequences near pipeline boundaries, while the Right-SP phase manages terminal computations. The Steady phase handles central subsequences using standard pipeline parallelism. Sequence parallelism ranges exhibit complementary patterns across stages, ensuring efficient resource utilization and minimizing cross-stage dependencies.

7 Evaluation

Implementation. We implement SPPO upon Megatron-LM using Python and CUDA. It consists of around 4000 lines of code (LOC): 2845 LOC for adaptive offloading and 1155 LOC for multiplexing sequence partitioning. To achieve the highest bandwidth between GPU and host, we bind the Non-Uniform Memory Access node for each process and use page-locked memory for CPU buffers.

Additional baseline implementations. To enable a fair component-wise comparison, we implement a simple offloading variant for TeraPipe (which does not include offloading in its original design). This represents the "full CPU offloading" baseline in our breakdown experiments.

Testbed. SPPO undergoes a thorough evaluation in the training of Transformer-based models with the GPT architecture. The size of the models range from 7 billion to 65 billion, and parallelism configurations (SP, PP, N) are summarized in Table 4. The training process occurs on a physical cluster comprising 16 GPU servers. Each server is equipped with 8 GPUs and 128 CPU cores, a total of 128 NVIDIA Ampere GPUs. Each GPU boasts 80GB of memory. The GPUs are interconnected through NVLink and NVSwitch, while inter-node communication is facilitated by four NVIDIA Mellanox 200 Gbps HDR InfiniBand. Each node has 2 TB of CPU memory, and the GPU-CPU communication bandwidth is 32 GB/s.

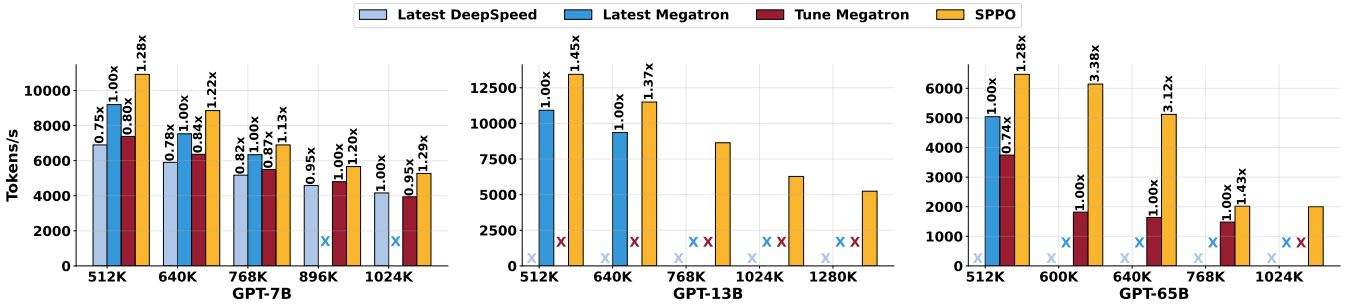

Figure 7: End-to-end evaluation results of training models of different sizes and sequence lengths.

Model	#GPUs	S	Mega. & SPPO		SPPO	Tune Mega.		DS
			SP	PP	N	SP	PP	SP
GPT-7B	32	512K	8	4	32	32	1	32
		640K			64			
		768K			80			
		896K			128			
		1024K			160			
GPT-13B	64	512K	8	8	32	8	8	-
		640K			48			
		768K			64			
		1024K			80			
		1280K			160			
GPT-65B	128	512K	16	8	32	64	2	-
		600K			64			
		640K			80			
		768K			96			
		1024K			128			

Table 4: Configurations for various GPT models.

Baselines and metrics. We compare against DeepSpeed-Ulysses [18], Megatron-LM [21], and TeraPipe [25].Megatron-Tuned represents the best-effort Megatron configuration for long sequences (pipeline depth, micro-batch size, and checkpointing ratios tuned per workload). Training –throughput is measured in tokens per GPU per second (TGS). All baselines use activation recomputation to support longer sequences.DeepSpeed-Ulysses is enhanced with ZeRO [37] and FPDT [48] for activation offloading. TeraPipe uses full offloading for comparison. Megatron-LM is tuned for longer sequences according to Nvidia's recommended configurations. Further Megatron tuning can narrow gaps at moderate sequence lengths but does not remove the granularity mismatch at extreme lengths, where tuned Megatron still hits memory limits or pipeline inefficiency while SPPO remains feasible with higher utilization.

7.1 End-to-End Evaluation

Figure 7 illustrates the training throughput across various LLMs and sequence lengths. For fairness, we optimize the parallelism strategy for each system, as reported in Table 4.

Throughput and scalability. SPPO consistently outperforms DeepSpeed-Ulysses and Megatron-LM across various settings. Its speedups range from 1.13× to 3.38×. A key advantage of SPPO is its ability to handle ultra-long sequences without out-of-memory (OOM) issues. In contrast, Megatron-LM struggles with sequences exceeding 896K for GPT-7B; DeepSpeed-Ulysses fails to support sequence lengths beyond 512K for GPT-13B and GPT-65B.

Memory efficiency. SPPO mitigates the memory pressure through nearly zero-overhead offloading and pipeline parallelism. It avoids costly activation recomputation, a limitation that significantly impacts Megatron-LM under the same parallelism strategy. For GPT-65B, SPPO supports sequence lengths of up to 1024K, whereas DeepSpeed-Ulysses is limited to 512K and Megatron-LM to 768K. This demonstrates SPPO's superior memory management capabilities.

Model-specific performance. For GPT-7B, under the same parallelism strategy as the latest Megatron-LM, SPPO achieves a 1.13× to 1.29× speedup. At sequence lengths beyond 896K, Megatron-LM encounters OOM issues, while SPPO continues to deliver superior throughput. For GPT-13B, SPPO supports sequence lengths of up to 1280K, whereas Megatron-LM is limited to 768K. DeepSpeed-Ulysses cannot train GPT-13B at all due to model architectural constraints. For GPT-65B, SPPO achieves remarkable speedups of 3.38× and 3.12× over Megatron-LM at sequence lengths of 600K and 640K, respectively. DeepSpeed-Ulysses, on the other hand, is unable to scale effectively for GPT-65B beyond 512K.

Limitations. While SPPO demonstrates superior performance overall, it is important to acknowledge that the baseline systems (DeepSpeed-Ulysses and Megatron-LM) have their own strengths in specific scenarios. For instance, Megatron-LM performs well for relatively shorter sequence lengths (e.g., a GPT-7B model with a sequence length of 768K) despite employing activation recomputation. In such cases, SPPO offers only marginal performance gains, as the computational workload of the GPT-7B model is insufficient to fully exploit the offloading overlap. This highlights that the benefits of SPPO are most pronounced in large-scale, long-sequence training scenarios where memory pressure and communication overhead become dominant bottlenecks.

7.2 Speedup Breakdown

To gain deeper insights into the key optimizations underlying the performance gains of SPPO, we conduct a detailed performance breakdown analysis to quantify the contribution of each core technique. Figure 8 illustrates the normalized speedup of SPPO relative to Megatron-LM. From this figure, we derive three key observations, as discussed below.

First, full CPU offloading alone does not always improve performance. While CPU offloading can optimize GPU memory efficiency,

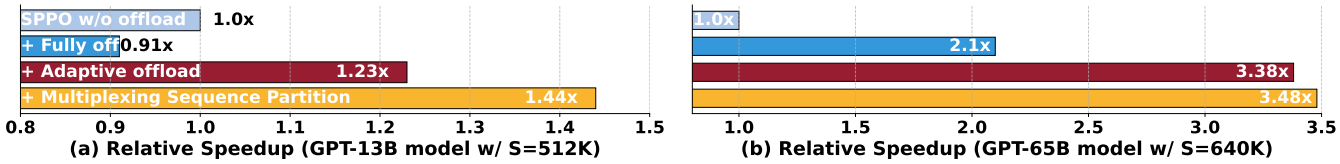

Figure 8: Breakdown analysis

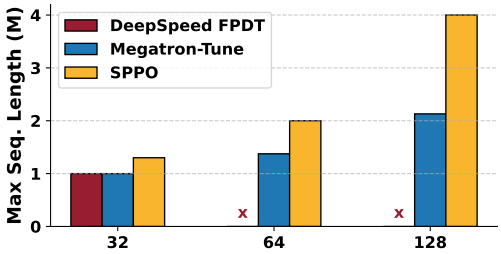

Figure 9: Scaling maximum sequence length with different numbers of GPUs in training GPT-7B model.

activation fully offloading overhead is non-negligible in certain scenarios, as discussed in Section 5 and shown in Figure 8(a). SPPO only attains a relative speedup of 0.91 compared to SPPO w/o offload in a GPT-13B model with a sequence length of 512K. However, full CPU offloading benefits the parallelism strategies that exhibit high communication efficiency but poor memory efficiency, as demonstrated in Figure 8(b). It achieves a relative speedup of 2.1 × in the GPT-65B model with a sequence length of 640K.

Second, adaptive offloading consistently improves both training performance and memory efficiency across diverse settings. Empirically, it delivers relative speedups of 1.23 × and 3.38 × for GPT-13B and GPT-65B, respectively, demonstrating its effectiveness in accommodating varying model scales and sequence lengths.

Third, multiplexing sequence partitioning significantly enhances training efficiency. Its benefits are particularly pronounced when the bubble ratio is high, which is determined by the number of subsequences (N) and pipeline stages (P) used during training. MSP achieves a relative speedup of 1.44 × for the GPT-13B model and a remarkable speedup of 3.48 × for the GPT-65B model.

Overall, each optimization technique plays a pivotal role in SPPO, collectively contributing to the significant speedup demonstrated by our empirical results.

7.3 Sequence Length Scalability

To evaluate the sequence-length scalability of SPPO under fixed GPU resources, we progressively increase the per-GPU sequence length from 1K to 10K tokens until out-of-memory (OOM) errors occur. For a fair comparison, we enable activation checkpointing at each layer for both DeepSpeed and Megatron-LM. As shown in Figure 9, SPPO consistently supports longer sequence lengths and exhibits clear advantages in scalability compared to the two baseline systems.

Specifically, DeepSpeed-Ulysses partitions computation along with attention heads, making it less effective for models with a limited number of heads. For instance, the restricted number of heads in GPT-7B becomes a scalability bottleneck for DeepSpeed-Ulysses. This explains why DeepSpeed-Ulysses gets a sharp decline in the maximum supported sequence length, dropping from 1K at 32 GPUs to 0 at 64 GPUs. Megatron-Tuned exhibits partial scalability (1×, 1.375×, 2.13× at 32/64/128 GPUs). The activation recomputation hinders the linear scaling in sequence length.

In contrast, SPPO is not constrained by the number of attention heads, achieving near-linear scalability. It can support sequence lengths of 1.3×, 2×, and 4× the baseline at 32, 64, and 128 GPUs, respectively. At 128 GPUs, SPPO outperforms Megatron-Tuned by 88% (4× vs. 2.13×). Moreover, SPPO achieves higher speedup at more GPUs (4× at 128 GPUs vs. 2× at 64 GPUs), indicating its ability to reduce communication overhead and enhance memory efficiency.

In sum, our empirical analysis of sequence length scalability highlights the critical role of parallelism strategies in long-sequence training. By decoupling from head-based partitioning, SPPO can flexibly adjust the configurations for parallelism strategies, enabling the training of sequences exceeding multi-million tokens. In contrast, existing LLM training systems face significant challenges in handling such extreme-scale scenarios.

8 Conclusion

In this work, we introduce Adaptive Sequence Pipeline Parallel Offloading (SPPO), a novel framework that significantly improves the efficiency of long-sequence LLM training by jointly addressing memory pressure and computational bottlenecks. By leveraging adaptive subsequence-aware offloading and optimized pipeline scheduling, SPPO effectively balances the GPU memory usage and training speed, overcoming the inefficiencies of existing methods. Our comprehensive experimental results demonstrate substantial performance improvements of SPPO, achieving up to 3.38× higher throughput compared to state-of-the-art frameworks while simultaneously reducing GPU resource requirements. These results highlight SPPO as a practical and scalable solution for training LLMs with extremely long input sequences, paving the way for broader deployment in both academic research and industrial applications.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Microsoft Research AI4Science and Microsoft Azure Quantum. 2023. The impact of large language models on scientific discovery: a preliminary study using gpt-4. *arXiv preprint arXiv:2311.07361* (2023).

[3] Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu, and Qi Tian. 2023. Accurate medium-range global weather forecasting with 3D neural networks. *Nature* 619, 7970 (2023), 533–538.

[4] Qiaoling Chen, Diandian Gu, Guoteng Wang, Xun Chen, YingTong Xiong, Ting Huang, Qinghao Hu, Xin Jin, Yonggang Wen, Tianwei Zhang, et al. 2024. Internevo: Efficient long-sequence large language model training via hybrid parallelism and redundant sharding. *arXiv preprint arXiv:2401.09149* (2024).

[5] Qiaoling Chen, Qinghao Hu, Zhisheng Ye, Guoteng Wang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. AMSP: Super-Scaling LLM Training via Advanced Model States Partitioning. *CoRR* abs/2311.00257 (2023).

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[8] Tri Dao and Albert Gu. 2024. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060* (2024).

[9] Kimi Developers. 2024. kimi. https://kimi.moonshot.cn/

[10] NVIDIA Developers. 2023. NVIDIA Transformer Engine Offloading strategy. https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/api/pytorch.html#transformer_engine.pytorch.get_cpu_offload_context.

[11] NVIDIA Developers. 2023. TransformerEngine. https://github.com/NVIDIA/TransformerEngine

[12] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[13] Yao Fu, Rameswar Panda, Xinyao Niu, Xiang Yue, Hannaneh Hajishirzi, Yoon Kim, and Hao Peng. 2024. Data engineering for scaling language models to 128k context. *arXiv preprint arXiv:2402.10171* (2024).

[14] Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2023).

[15] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, et al. 2024. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485* (2024).

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS'19)*. Curran Associates Inc.

[18] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *CoRR* abs/2309.14509 (2023).

[19] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[20] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).

[21] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[22] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Xuezhe Ma, Ion Stoica, Joseph E Gonzalez, and Hao Zhang. 2023. Distflashattn: Distributed memory-efficient attention for long-context llms training. *arXiv preprint arXiv:2310.03294* (2023).

[23] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2021. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120* (2021).

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *CoRR* abs/2006.15704 (2020).

[25] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. *CoRR* abs/2102.07988 (2021).

[26] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. *CoRR* abs/2310.01889 (2023).

[27] LLaMA2-7B-32K. 2022. LLaMA2-7B-32K. https://huggingface.co/togethercomputer/LLaMA-2-7B-32K

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery.

[29] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *CoRR* abs/2104.04473 (2021).

[30] Houlsby Neil and Weissenborn Dirk. 2020. Transformers for Image Recognition at Scale. *Online: https://ai. googleblog. com/2020/12/transformers-for-image-recognitionat. html* (2020).

[31] PCIeGen5. 2024. PCIeGen5. https://www.intel.com/content/www/us/en/docs/programmable/683501/23-1-9-0-0/features.html

[32] William Peebles and Saining Xie. 2023. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 4195–4205.

[33] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905.

[34] PyTorch. 2023. Accelerating Large Language Models with Accelerated Transformers. https://pytorch.org/blog/accelerating-large-language-models/.

[35] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2023. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241* (2023).

[36] qwen Developers. 2024. qwen. https://tongyi.aliyun.com/qianwen/

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press.

[38] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[39] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[40] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[41] Ludan Ruan and Qin Jin. 2022. Survey: Transformer based video-language pre-training. *AI Open* 3 (2022), 1–13.

[42] Ao Sun, Weilin Zhao, Xu Han, Cheng Yang, Xinrong Zhang, Zhiyuan Liu, Chuan Shi, and Maosong Sun. 2024. Seq1f1b: Efficient sequence-level pipeline parallelism for large language model training. *arXiv preprint arXiv:2406.03488* (2024).

[43] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3, 6 (2023), 7.

[44] Jakub M. Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021-12-06. Piper: Multidimensional Planner for DNN Parallelization. In *Advances in Neural Information Processing Systems (NeurIPS '21)*.

[45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[46] Patrick von Platen. 2023. Optimizing your LLM in production. https://huggingface.co/blog/optimize-llm.

[47] Fuzhao Xue, Yukang Chen, Dacheng Li, Qinghao Hu, Ligeng Zhu, Xiuyu Li, Yunhao Fang, Haotian Tang, Shang Yang, Zhijian Liu, et al. 2024. Longvila: Scaling long-context visual language models for long videos. *arXiv preprint arXiv:2408.10188* (2024).

[48] Jinghan Yao, Sam Ade Jacobs, Masahiro Tanaka, Olatunji Ruwase, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2024. Training Ultra Long Context Language Model with Fully Pipelined Distributed Transformer. *arXiv preprint arXiv:2408.16978* (2024).

[49] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. 2024. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 545–561.

[50] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, Fang Yang, Yuanbo Peng, Dian Jiao, Shuaipeng Li, Jinbao Xue, et al. 2024. Efficiently Training 7B LLM with 1 Million Sequence Length on 8 GPUs. *arXiv preprint arXiv:2407.12117* (2024).

[51] Zangwei Zheng, Xiangyu Peng, Tianji Yang, Chenhui Shen, Shenggui Li, Hongxin Liu, Yukun Zhou, Tianyi Li, and Yang You. 2024. Open-sora: Democratizing efficient video production for all. *arXiv preprint arXiv:2412.20404* (2024).