ZO-Offloading: Fine-Tuning LLMs with 100 Billion Parameters on a Single GPU

Liangyu Wang[1] **Jie Ren**[1] **Hang Xu**[1,2] **Junxiao Wang**[1,3]
David E. Keyes[1] **Di Wang**[1]
[1]King Abdullah University of Science and Technology,
[2]HPC-AI Technology Inc., [3]Guangzhou University
{liangyu.wang, jie.ren, hang.xu, junxiao.wang}@kaust.edu.sa
junxiao.wang@gzhu.edu.cn
{david.keyes, di.wang}@kaust.edu.sa

Abstract

Fine-tuning pre-trained LLMs typically requires a vast amount of GPU memory. Standard first-order optimizers like SGD face a significant challenge due to the large memory overhead from back-propagation as the size of LLMs increases, which necessitates caching activations during the forward pass and gradients during the backward pass. In contrast, zeroth-order (ZO) methods can estimate gradients with only two forward passes and without the need for activation caching. Additionally, CPU resources can be aggregated and offloaded to extend the memory and computational capacity of a single GPU. To enable efficient fine-tuning of LLMs on a single GPU, we introduce ZO-Offloading, a framework that strategically utilizes both CPU and GPU resources for ZO. ZO-Offloading dynamically offloads model parameters to the CPU and retrieves them to the GPU as needed, ensuring continuous and efficient computation by reducing idle times and maximizing GPU utilization. Parameter updates are integrated with ZO's dual forward passes to minimize redundant data transfers, thereby improving the overall efficiency of the fine-tuning process. With ZO-Offloading, for the first time, it becomes possible to fine-tune extremely large models, such as the OPT-175B with over 175 billion parameters, on a single GPU with just 24GB of memory—a feat previously unattainable with conventional methods. Moreover, our framework operates without any additional time cost compared to standard ZO methodologies.

1 Introduction

As Large Language Models (LLMs) grow to hundreds of billions of parameters, exemplified by OPT-175B [1] and Llama 3.1 405B [2], efficient GPU memory management becomes critical. This challenge arises from the need to balance model performance with current hardware constraints. CPU offloading has emerged as a key technique to address this issue, transferring less frequently accessed data from GPU to the typically larger and more cost-effective CPU memory. While widely applied in inference tasks such as KV cache offloading [3, 4] and Mixture of Experts (MoE) offloading [5, 6], CPU offloading's application in training, particularly fine-tuning, remains underexplored. This paper examines the potential of CPU offloading in LLM fine-tuning, addressing the growing need for efficient memory utilization in large-scale model development.

Recently, some works [7, 8] have tried to introduce CPU offloading into LLM training. However, they are typically constrained by the capabilities of first-order optimizers such as SGD and Adaptive Moment Estimation (AdamW) [9], and limited GPU memory, restricting large-scale model scalability on single GPU systems. In detail, using first-order optimizers introduces two major inefficiencies in

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

CPU offloading: **(1) Multiple communication operations**: During the training of LLMs, parameters are used not only for computing the loss during the forward pass but also for gradient computation in the backward pass. This necessitates offloading the same data (parameter) twice—once for each pass (see Appendix Figure 3a for an illustration). Such redundancy not only doubles the communication volume between the CPU and GPU but also introduces significant latency and inefficiency due to repetitive data transfers. **(2) Huge data transfer volume per communication operation**: Furthermore, both parameters and activations (hidden states) are required in the backward pass to complete gradient computations. This means that parameters and activation values must be offloaded during each forward pass and re-uploaded to the GPU for the backward pass. The result is a significant increase in the volume of data transferred, which severely impacts training throughput and efficiency.

On the other hand, compared to first-order optimization methods, zeroth-order (ZO) methods offer a novel approach to fine-tuning LLMs [10, 11, 12]. These methods utilize dual forward passes to estimate parameter gradients and subsequently update parameters, as illustrated in Figure 3b. This approach eliminates the traditional reliance on backward passes, thereby streamlining the training process by significantly reducing the number of computational steps required.

Based on the above observations, we conjecture that ZO's architecture is particularly well-suited for CPU offloading strategies. Intuitively, by eliminating backward passes and the need to store activation values, it can significantly reduce GPU memory demands through efficient parameter offloading. However, despite these advantages, ZO training via CPU offloading introduces new challenges, particularly in the realm of CPU-to-GPU communication. Transferring parameters between the CPU and GPU, which is crucial for maintaining gradient computation and model updates, becomes a critical bottleneck due to inherent communication delays. Although ZO methods inherently extend computation times because of the dual forward passes, potentially allowing for better overlap between computation and communication (Section 2.2), there remain significant inefficiencies. The necessity to upload parameters to the GPU for upcoming computations introduces a large volume of communications.

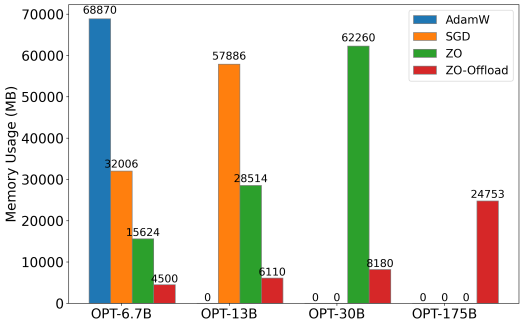


Figure 1: Memory usage comparison for training LLMs across different optimizers (AdamW, SGD, ZO, and ZO-Offloading) and model sizes (OPT-6.7B, OPT-13B, OPT-30B, OPT-175B). The '0' values indicate that training was not feasible due to excessive memory demand.

To tackle the inefficiencies highlighted, we introduce ZO-Offloading, a novel framework specifically designed for ZO fine-tuning in LLMs with CPU offloading. These innovations make it feasible to fine-tune extremely large models, such as the OPT-175B [1] with **over 175 billion parameters, on a single GPU equipped with just 24GB of memory** (Figure 1). Our contributions can be summarized as follows:

Innovative use of CPU-offloading for ZO methods. We pioneer the application of CPU offloading in the context of ZO optimization methods to dramatically reduce GPU memory requirements. This method allows for the efficient handling of model parameters by dynamically transferring inactive data between the CPU and GPU, significantly extending the capacity to train large models like OPT-175B on a single GPU.

Low memory but a high-throughput framework. We introduce a series of optimized features that substantially reduce GPU memory use while maintaining high throughput. Our dynamic scheduler improves GPU utilization by optimizing computation and communication overlaps. Reusable memory blocks minimize overhead and stabilize memory use, while efficient parameter updating synchronizes updates with dual forward passes to reduce data transfers.

Empirical validation and experimentation. Our experiments demonstrate that ZO-Offloading can efficiently fine-tune the OPT-175B model, with over 175 billion parameters, on a single 24GB GPU—previously impossible with traditional methods. Crucially, this is achieved with no additional

time cost and decreases in accuracy, showcasing the framework's effectiveness and efficiency for large-scale model training.

2 Method

In this section, we first provide some related work and preliminaries in Appendix B and C, and then present our ZO-Offloading framework.

2.1 Framework Overview

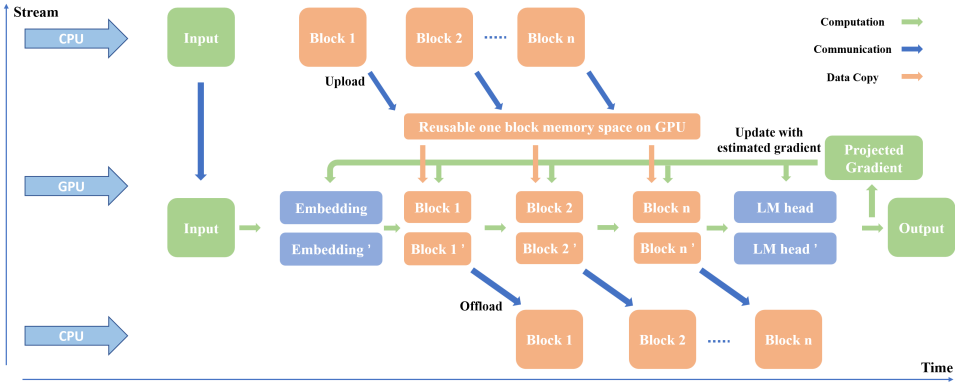


Figure 2: Workflow of the ZO-Offloading framework for fine-tuning LLMs.

ZO optimization procedure. To better illustrate this idea, we first describe the original version of the computational workflow of the ZO optimizer for fine-tuning LLMs. Initially, input data is loaded from the disk into the CPU and subsequently transferred to the GPU. Within the GPU, each module—including the embedding layer, transformer blocks, and the language model (LM) head—executes dual forward computations to estimate the projected gradient and update parameters. From the system perspective, traditional deep learning frameworks like PyTorch [13] typically manage both communication (via interconnections, e.g., PCIe) and computation tasks with a single CUDA stream[1], leading to significant inefficiencies.

Vanilla CPU-offloading and its limitations. Specifically, for ZO optimization, the i-th transformer block is uploaded from the CPU to the GPU (the GPU is designated for computation-intensive tasks using its CUDA and Tensor Cores, and the CPU memory is used for parameter storage), undergoes dual forward computation, and then is offloaded back to the CPU. The $i+1$-th block must wait for the offloading of the i-th block to finish before its uploading, leading to idle CUDA and Tensor Cores during communication while the interconnection remains idle during computation. See Figure 4 in Appendix for an illustration.

How ZO-Offloading is different. Central to our ZO-Offloading framework is the strategic utilization of CPU and GPU resources (Section 2.2). This approach involves dynamically offloading model parameters to the CPU and uploading them back to the GPU as needed for computation. Specifically, for the transformer model structure, each transformer block is individually uploaded for processing and subsequently offloaded post-computation, thus balancing communication and computation across blocks. As illustrated in Figure 2, while the i-th transformer block is being computed, the $i+1$-th block is pre-uploaded, and the $i-1$-th block is offloaded simultaneously. This strategic overlapping ensures continuous and efficient computation, reducing idle times and maximizing GPU utilization. In the uploading phase of ZO-Offloading, transformer blocks are transferred into a reusable memory space on the GPU, eliminating the extra time typically required for CUDA memory allocation (Section D). Moreover, parameter updates are ingeniously fused with the dual forward passes to minimize redundant data transfers, thereby enhancing the overall efficiency of the model training process (Section E).

2.2 Dynamic Scheduler Design for Efficient Overlap

Asynchronous execution on CUDA streams. To overlap the data loading and computation process, we propose a dynamic scheduler, utilizing the asynchronous execution on different CUDA streams.

[1] https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Algorithm 1 ZO-Offloading Dynamic Scheduler

Require: Transformer blocks $\{W_i\}_{i=1}^{N}$ with number of transformer blocks N, embedding parameters $Embedding$, and LM head $LMhead$.

Initialize a dynamic scheduler $S\{\cdot\}$ to control dual forward computation $C(\cdot)$, uploading $U(\cdot)$, and offloading $O(\cdot)$ operations.
Asynchronously launch $S\{U(W_1), C(Embedding)\}$.
for $i = 1$ to $N - 1$ **do**
    Synchronously wait until $U(W_i)$ finished.
    if $i = 1$ **then**
        Asynchronously launch $S\{U(W_{i+1}), C(W_i)\}$.
    else
        Synchronously wait until $C(W_{i-1})$ finished.
        Asynchronously launch $S\{U(W_{i+1}), C(W_i), O(W_{i-1})\}$.
    end if
end for
Synchronously wait until $U(W_N)$ and $C(W_{N-1})$ finished.
Asynchronously launch $S\{C(W_N), O(W_{N-1})\}$.
Synchronously wait until $C(W_N)$ finished.
Asynchronously launch $S\{C(LMhead), O(W_N)\}$.

Specifically, our scheduler includes three CUDA streams (Figure 2), which are utilized to control the i-th transformer block's computation, the $i + 1$-th block's uploading, and the $i - 1$-th block's offloading can occur concurrently. This design minimizes data transfer conflicts and maximizes GPU utilization by keeping computational and communication channels active.

Locking mechanism. However, designing this dynamic scheduler presents challenges when communication tasks outlast computation tasks, leading to potential errors. For example, if the upload of the i-th block is incomplete when its computation begins, this can lead to errors, as the GPU computes with an incomplete set of parameters. Similarly, if the computation of the i-th block is still ongoing when its offloading begins, it can also result in errors because the computation is disrupted by the removal of necessary data. To address this, our scheduler implements a locking mechanism for each block's computation task, ensuring it only starts once its corresponding upload is confirmed complete. While this solution mitigates the issue of incomplete parameters, it can still potentially create bottlenecks if communication tasks consistently outlast computation tasks. Surprisingly, our evaluations show that with ZO's unique dual forward passes, which extend computation times, communication delays are no longer the primary bottleneck in most scenarios. The detailed scheduler design to apply ZO-Offloading on LLMs is shown in Algorithm 1.

3 Experiment

The experimental evaluation of our framework was conducted using the PyTorch deep learning library, integrated with NVIDIA CUDA streams to optimize parallel computation tasks. We selected the Open Pre-trained Transformer (OPT) [1] model family as the subject of our experiments due to its open-source availability, widespread adoption in the research community, and diverse range of model sizes, ranging from 125 million to 175 billion parameters, which allows for a comprehensive assessment of our framework's performance across varying scales of model complexity. In our evaluation, MeZO [11] serves as the baseline method, as it is the most memory-throughput efficient ZO method currently. Our framework builds upon MeZO, reducing GPU memory usage while maintaining throughput and precision. All performance evaluation experiments are done with dataset SST-2 ([14]). Additional experimental settings and experiments are included in Appendix F and G.

The performance results of our experiments are presented in Table 1, where we compare the GPU memory usage and throughput of the MeZO and ZO-Offloading frameworks, employing both FP32 and FP16 data formats. The results demonstrate a consistent advantage of ZO-Offloading in terms of GPU memory utilization across all model sizes, highlighting significant efficiency improvements, especially in large-scale models like **OPT-175B**. This efficiency is attributed to ZO-Offloading's design, which strategically utilizes GPU memory to temporarily store only a limited number of transformer blocks for computation rather than the entire model. Notably, the memory savings

become more pronounced as the model size increases. For smaller models, the GPU memory savings are less pronounced due to the significant proportion of memory allocated for input data, which diminishes the relative impact of the memory optimization.

Table 1: **Main results of ZO-Offloading performance for various model configurations and both FP32 and FP16 modes.** Instances of '-' in the table indicate scenarios where the corresponding method failed to execute due to memory constraints. The values in parentheses (x) represent the ratio of each measurement compared to the baseline MeZO (first column) configuration.

Model	GPU Memory Usage (MB) ↓				Throughput (tokens/sec) ↑			
	MeZO(32)	ZO-Offload(32)	MeZO(16)	ZO-Offload(16)	MeZO(32)	ZO-Offload(32)	MeZO(16)	ZO-Offload(16)
OPT-125M	3091	2941(x0.95)	1801(x0.58)	**1661(x0.54)**	14889	13074(x0.89)	**31058(x2.09)**	**31058(x2.09)**
OPT-350M	4219	3393(x0.81)	2389(x0.57)	**1643(x0.39)**	5274	5099(x0.97)	**13508(x2.56)**	12284(x2.32)
OPT-1.3B	9117	4413(x0.48)	4887(x0.54)	**2651(x0.29)**	1954	1954(x1.00)	**6788(x3.47)**	**6788(x3.47)**
OPT-2.7B	15277	5261(x0.34)	7933(x0.52)	**3111(x0.20)**	1087	1087(x1.00)	**4227(x3.89)**	**4227(x3.89)**
OPT-6.7B	32083	8329(x0.26)	16311(x0.51)	**4539(x0.14)**	499	499(x1.00)	**2455(x4.92)**	**2455(x4.92)**
OPT-13B	58251	12113(x0.21)	29411(x0.50)	**6445(x0.11)**	270	270(x1.00)	**1406(x5.21)**	1340(x4.96)
OPT-30B	-	18879	63953	**10369**	-	122	**651**	597
OPT-66B	-	29937	-	**14143**	-	40	-	**273**
OPT-175B	-	49203	-	**24667**	-	14	-	**37**

In terms of throughput, ZO-Offloading maintains a performance comparable to MeZO in most tested scenarios without any additional time overhead. The instances where ZO-Offloading exhibits a decrease in throughput, such as with the OPT-125M model in FP32 format, can be primarily attributed to the dynamics of computation and communication. In these cases, the computation of each transformer block's dual forward passes completes quicker than their corresponding communication tasks, leading to idle times as the dynamic scheduler (discussed in Section 2.2) synchronizes and waits for these communication tasks to conclude. It is important to note that our results do not show a consistent pattern where either smaller or larger models benefit more significantly from the computation-communication overlap, indicating that the effectiveness of this overlap does not linearly correlate with model size.

4 Conclusion

In this paper, we presented ZO-Offloading, an efficient framework that enables the training of extremely large language models, such as the OPT-175B, on a single 24GB GPU—a capability previously unattainable with traditional methods. By effectively integrating CPU offloading, high-performance dynamic scheduler, efficient memory management, and efficient parameter updating, our framework reduces GPU memory demands while maintaining high throughput without additional time costs. These innovations not only lower the bar for teams with limited hardware resources and advance the democratization of large models, but also open new avenues for advancing AI technology more efficiently. Moving forward, we plan to further enhance ZO-Offloading, exploring synergies with emerging hardware and optimization techniques to keep pace with the evolving demands of AI model training.

References

[1] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[2] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[3] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.

[4] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[5] Artyom Eliseev and Denis Mazur. Fast inference of mixture-of-experts language models with offloading. *arXiv preprint arXiv:2312.17238*, 2023.

[6] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. Moe-infinity: Activation-aware expert offloading for efficient moe serving. *arXiv preprint arXiv:2401.14361*, 2024.

[7] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

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

[9] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[10] Yihua Zhang, Pingzhi Li, Junyuan Hong, Jiaxiang Li, Yimeng Zhang, Wenqing Zheng, Pin-Yu Chen, Jason D Lee, Wotao Yin, Mingyi Hong, et al. Revisiting zeroth-order optimization for memory-efficient llm fine-tuning: A benchmark. *arXiv preprint arXiv:2402.11592*, 2024.

[11] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems*, 36:53038–53075, 2023.

[12] Tanmay Gautam, Youngsuk Park, Hao Zhou, Parameswaran Raman, and Wooseok Ha. Variance-reduced zeroth-order methods for fine-tuning language models. *arXiv preprint arXiv:2404.08080*, 2024.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[14] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[15] Tianxiang Sun, Zhengfu He, Hong Qian, Yunhua Zhou, Xuan-Jing Huang, and Xipeng Qiu. Bbtv2: Towards a gradient-free future with large language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3916–3930, 2022.

[16] Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. Black-box tuning for language-model-as-a-service. In *International Conference on Machine Learning*, pages 20841–20855. PMLR, 2022.

[17] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, 2020.

[18] Shuyu Cheng, Guoqiang Wu, and Jun Zhu. On the convergence of prior-guided zeroth-order optimization algorithms. *Advances in Neural Information Processing Systems*, 34:14620–14631, 2021.

[19] Utkarsh Singhal, Brian Cheung, Kartik Chandra, Jonathan Ragan-Kelley, Joshua B Tenenbaum, Tomaso A Poggio, and Stella X Yu. How to guess a gradient. *arXiv preprint arXiv:2312.04709*, 2023.

[20] Aochuan Chen, Yimeng Zhang, Jinghan Jia, James Diffenderfer, Konstantinos Parasyris, Jiancheng Liu, Yihua Zhang, Zheng Zhang, Bhavya Kailkhura, and Sijia Liu. Deepzero: Scaling up zeroth-order optimization for deep model training. In *The Twelfth International Conference on Learning Representations*, 2024.

[21] HanQin Cai, Daniel McKenzie, Wotao Yin, and Zhenliang Zhang. Zeroth-order regularized optimization (zoro): Approximately sparse gradients and adaptive sampling. *SIAM Journal on Optimization*, 32(2):687–714, 2022.

[22] HanQin Cai, Yuchen Lou, Daniel McKenzie, and Wotao Yin. A zeroth-order block coordinate descent algorithm for huge-scale black-box optimization. In *International Conference on Machine Learning*, pages 1193–1203. PMLR, 2021.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

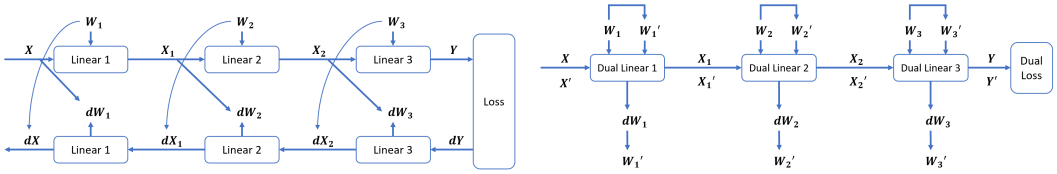
[24] Georgi Gerganov. llama.cpp, 2023.

[25] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

[26] John C Duchi, Michael I Jordan, Martin J Wainwright, and Andre Wibisono. Optimal rates for zero-order convex optimization: The power of two function evaluations. *IEEE Transactions on Information Theory*, 61(5):2788–2806, 2015.

Appendix

A Additional Details on Motivations and Previous Approaches



(a) Model using first-order optimizer with forward-backward passes workflow

(b) Model using zero-order optimizer with only forward passes workflow

Figure 3: **Motivation**. Comparison of model workflows using first-order and zeroth-order optimizers. (a) depicts a traditional first-order optimizer workflow with forward and backward passes, while (b) shows a zeroth-order optimizer workflow utilizing only forward passes.

Why ZO is Suitable for CPU Offloading Figure 3 illustrates the distinct operational differences between first-order and zeroth-order optimization methods applied to model training. Figure 3(a) demonstrates a traditional first-order optimizer setup, where the model employs a forward-backward pass sequence to update weights. Here, the input X progresses through several linear transformations (Linear 1, 2, 3), generating intermediate activations (X_1, X_2) and the final output Y, which is used to compute the loss. Subsequent backward passes calculate gradients (dW_1, dW_2, dW_3) for each weight and derivatives for each activation (dX, dX_1, dX_2), necessary for parameter updates through gradient descent.

In contrast, Figure 3(b) presents the zeroth-order optimizer's workflow, which simplifies the training process by eliminating the backward passes. This setup involves dual forward passes through slightly perturbed versions of the model weights (W_1', W_2', W_3') at each layer (Dual Linear 1, 2, 3). The resulting outputs from each layer (X', X_1', X_2') and the final output Y' are used to compute a dual loss.

This dual loss approximates the gradient required for updating the original weights, relying solely on forward computations. This approach significantly reduces the computational overhead and memory requirements by avoiding the need to store activations for backpropagation, making it particularly advantageous for training large models on limited hardware.

This comparison highlights how zeroth-order optimization offers a more memory-efficient alternative by leveraging only forward passes, thereby facilitating the training of large-scale models in constrained environments.

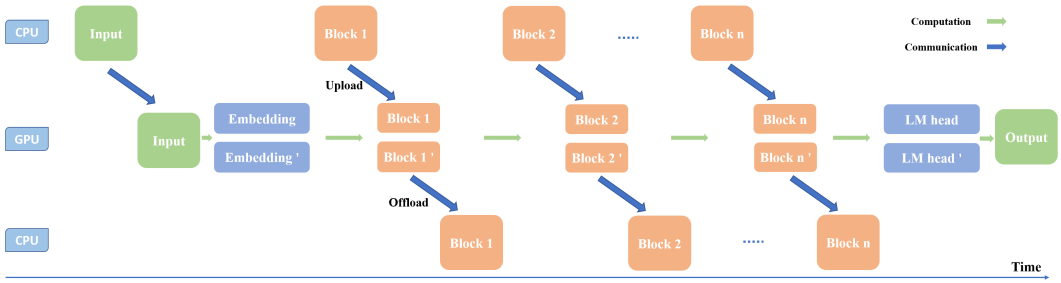


Figure 4: Workflow of the naive and non-overlap ZO-Offloading framework with only dual forward passes. This diagram demonstrates the sequential process without communication and computation overlap, using the pure PyTorch framework.

Why the Dynamic Scheduler and Overlap Matter Figure 4 provides a visual depiction of the workflow in the naive ZO-Offloading framework, specifically illustrating the naive, non-overlapping approach to dual forward passes. In this workflow, data is initially loaded from the CPU to the GPU, starting with the input processed through the embedding layer. Each transformer block (from Block 1 to Block n) is then sequentially processed: first uploaded to the GPU, where dual forward computations occur, and then offloaded back to the CPU after computation is complete.

This step-by-step process highlights a significant inefficiency in the current implementation: the GPU must wait for each block to be offloaded back to the CPU before the next block can be uploaded and processed. This results in substantial idle times for the GPU during offloads, and the CPU during uploads, as each unit must wait for the other to complete its task before proceeding. Such lack of overlap between computation (green arrows) and communication (blue arrows) tasks demonstrates a critical area for improvement, underlining the necessity for an overlapped or asynchronous approach to enhance overall system efficiency and throughput. By addressing this inefficiency, we can significantly reduce the training time and increase the utilization of both CPU and GPU resources.

B Related Work

Zeroth-Order (ZO) Optimization. ZO optimization offers a gradient-free alternative to first-order (FO) optimization by approximating gradients through function value-based estimates. These estimates theoretically require only two forward passes but are believed to be prohibitively slow for optimizing large models. Despite this limitation, ZO methods have been utilized in deep learning to generate adversarial examples or adjust input embeddings [15, 16], though they have not been widely

adopted for direct optimization of large-scale models [17]. Several acceleration techniques have been proposed to address the scaling challenges of ZO optimization and some of them have been used for LLM fine-tuning. These include using historical data to improve gradient estimators [18], exploiting gradient structures [19] or sparsity to reduce the dependence of ZO methods on the size of the problem [20, 21, 22], and reusing intermediate features [20] and random perturbation vectors [11] during the optimization process. These advancements suggest that ZO optimization could increasingly be applied to more complex and large-scale ML problems. While previous ZO optimization efforts have primarily targeted algorithmic improvements for GPU memory efficiency, our approach extends these optimizations to the system level, enabling more robust memory management and enhanced performance for large-scale machine learning applications.

CPU Offloading for LLMs. With recent advancements in LLMs, several approaches have emerged to offload data to CPU memory, mitigating GPU memory limitations. One such method is vLLM [23], which utilizes PagedAttention to dynamically manage the key-value (KV) cache at a granular block level. Portions of the KV cache can be temporarily swapped out of GPU memory to accommodate new requests. Llama.cpp [24] addresses oversized LLMs by using static layer partitioning. It stores certain contiguous layers in CPU memory while keeping others in GPU memory. During computation, the CPU handles the layers in its memory, followed by the GPU computing its assigned layers. FlexGen [4], a GPU-centric inter-layer pipeline method, seeks to improve throughput by pinning some model weights in GPU memory for each layer. During computation, it overlaps GPU processing of the current layer with data loading for the next. DeepSpeed [7] introduces a technique to offload the first-order optimizer state to the CPU, significantly reducing GPU memory requirements during training. Zero-offload [8] extends the DeepSpeed approach by not only offloading data to the CPU but also engaging the CPU in computational tasks. Despite these advancements, the predominant focus of previous research has been on optimizing LLM inference or first-order optimization through strategic CPU-GPU data transfers. Our work, in contrast, introduces a novel approach by implementing CPU offloading specifically for zeroth-order optimization and fine-tuning of LLMs.

C Preliminaries on ZO-SGD

ZO optimization offers a gradient-free alternative to first-order (FO) optimization by approximating gradients through function value-based estimates. There are different ZO optimizers for estimating the gradient. To better illustrate our framework, in this paper, we focus on the randomized gradient estimator (RGE) proposed by [25], which approximates the FO gradient using finite differences of function values along randomly chosen direction vectors and has been used widely in the ZO optimization literature. Our idea can be applied to other ZO optimizers.

Given a scalar-valued function $f(\cdot)$ and a model x, the RGE employed by [11], referred to as $\hat{\nabla} f(x)$, is to approximate $\nabla f(x)$ and is expressed using central difference:

$$\hat{\nabla} f(x) = \frac{f(x + \epsilon z) - f(x - \epsilon z)}{2\epsilon} z, \tag{1}$$

where z is a random direction vector drawn from the standard Gaussian distribution $\mathcal{N}(0, \mathrm{I})$, and $\epsilon > 0$ is a small perturbation step size, also known as the smoothing parameter. The rationale behind RGE stems from the concept of the directional derivative [26]. As ϵ approaches 0, the directional derivative provides us an unbiased gradient estimator of $\nabla f(x)$. Thus, the RGE $\hat{\nabla} f(x)$ can be interpreted as an approximation of the FO gradient $\nabla f(x)$ using the directional derivative [10]. Zeroth-order stochastic gradient descent (ZO-SGD) follows a similar algorithmic framework to its first-order counterpart, SGD, but replaces the gradient with an estimated gradient via zeroth order (function value) information for the descent direction.

Fine-tuning pre-trained LLMs typically demands substantial GPU memory. Previous first-order SGD-based methods encounter major challenges as LLM sizes grow, primarily due to the significant memory overhead required for backpropagation, which involves storing activations during the forward pass and gradients during the backward pass. In contrast, ZO-SGD can estimate gradients with only forward passes, eliminating the need for activation caching. [11] utilized the classical ZO-SGD algorithm (based on RGE), named MeZO, to fine-tune pre-trained LLMs with up to 30 billion parameters on a single GPU. They capitalized on the memory-efficient nature of ZO optimization, which eliminates the need for backpropagation and reduces memory costs. Since CPU resources can be combined and offloaded to expand the memory and computational capacity of a single GPU. To facilitate efficient fine-tuning of LLMs on a single GPU, we introduce ZO-Offloading, a framework that strategically leverages both CPU and GPU resources for ZO-SGD.

D Efficient Memory Management via Reusable One Block Space on GPU

We can further optimize memory management by initially pre-allocating a reusable transformer block of memory on the GPU. This strategy is implemented to circumvent the substantial time overhead associated with repeated CUDA memory allocations (malloc) and frees, which are typically required each time when data is transferred between the CPU and the GPU. By establishing a dedicated memory space initially and reusing it for each transformer block, we avoid the need for multiple malloc and free operations overhead the training process.

This reusable memory space is dynamically assigned to accommodate the parameters of each transformer block sequentially. Once a block's computation is complete and its data is offloaded back to the CPU, the same GPU memory space is immediately prepared to receive the next block's parameters from the CPU. This approach not only expedites the data transfer process but also stabilizes the GPU's memory usage, preventing fluctuations that could otherwise impact computational efficiency and performance.

E Efficient Parameter Update Strategy

In the ZO-Offloading framework, the parameter update strategy is meticulously designed to precede the dual forward computations of each transformer block. Traditionally, each transformer block is subjected to two distinct data transfer phases: one

for the dual forward computations and another for applying gradient updates. This requirement stems from the fact that the (approximated) gradients are obtained only after completing the dual forward computations for the entire model. Consequently, parameters must be uploaded for the computation phase, offloaded upon completion, and then re-uploaded and offloaded again for the gradient update phase. This iterative process effectively doubles the communication load and extends the duration of training.

By implementing preemptive parameter updates, the framework significantly curtails the number of data transfers required per iteration. With this strategy, once blocks are updated with the last iteration's gradients, only a single upload and offload cycle is necessary for each block. This adjustment not only halves the usage of interconnection bandwidth but also enhances the efficiency of the training process, thereby streamlining operations and reducing overhead.

F Experiment Settings

1. Model Specifications:

- **Model Family:** We used the Open Pre-trained Transformer (OPT) [1] model family for our experiments, ranging from 125 million to 175 billion parameters, to assess our framework's scalability and performance across different complexities.
- **Baseline Model:** The MeZO (Memory-efficient Zeroth-Order) serves as the baseline for comparison, known for its efficiency in memory throughput among Zeroth-Order offloading methods.

2. Dataset:

- **Dataset Used:** All performance evaluation experiments were conducted using the Stanford Sentiment Treebank (SST-2) dataset, a standard benchmark for evaluating natural language processing models.

3. Hyperparameters:

- **Learning Rate:** 1×10^{-7}
- **Steps:** 100
- **Batch Size:** 1
- **Sequence Length:** 2048

4. Computational Resources:

- **GPU:** NVIDIA A100 with 80GB of memory.
- **CPU:** AMD Milan.
- **Software:** Experiments were conducted using PyTorch version 3.11, integrated with CUDA version 12.1.

5. Evaluation Metrics:

- **GPU Memory Usage:** Measured in gigabytes (GiB).

- **Throughput:** Evaluated as tokens per second to assess the efficiency of the model training under various configurations.

G Ablation Study of Scheduler, Reusable Memory, and Efficient Updating

Table 2: **Throughput (token/sec) results to validate proposed features.**

Model	MeZO	ZO-Offloading (no scheduler overlap)	ZO-Offloading (no reusable memory)	ZO-Offloading (no efficient update)	ZO-Offloading
OPT-1.3B	1954	1109 (x0.57)	735 (x0.38)	1567 (x0.80)	1954 (x1.00)
OPT-2.7B	1087	573 (x0.52)	422 (x0.39)	849 (x0.78)	1087 (x1.00)
OPT-6.7B	499	225 (x0.45)	184 (x0.37)	373 (x0.74)	499 (x1.00)

In order to discern the individual contributions of key features within the ZO-Offloading framework to its overall performance, an ablation study was conducted focusing on three critical components: the dynamic scheduler (Sec. 2.2), reusable memory (Sec. D), and efficient parameter updating (Sec. E). This study mainly focused on throughput because the primary objective of the three features under investigation was to enhance throughput without impacting ZO-Offloading's inherent capability to reduce GPU memory usage. The main results, as presented earlier, clearly demonstrated that ZO-Offloading effectively decreases GPU memory consumption. Therefore, an ablation study on memory usage was deemed unnecessary, as the CPU-offloading mechanism inherently manages to reduce memory demands without the need for additional features aimed specifically at memory reduction. Given the tightly integrated nature of our system, traditional ablation methodologies that add one feature at a time to a baseline are impractical. Instead, we adopted a reverse ablation approach where each feature was individually disabled. This allowed us to observe the decrement in throughput relative to the fully operational framework, thereby highlighting the significance of each component. We mainly use OPT-1.3B, OPT-2.7B, and OPT-6.7B in the ablation study.

The results, presented in Table 2, provide a clear illustration of how the absence of each feature impacts the system's throughput: (1) **Horizontal Comparison.** Across all models, the removal of reusable memory results in the most substantial decrease in throughput, followed by the dynamic scheduler, and finally, the efficient parameter updating. This order of impact suggests that while all three features are pivotal, the overhead introduced by CUDA malloc operations, which are eliminated by reusable memory, significantly outweighs the communication delays between the CPU and GPU, managed by the dynamic scheduler and efficient parameter updating. For instance, when reusable memory is not employed, the throughput drops to 37% of the fully optimized framework for the OPT-6.7B model, highlighting its critical role in enhancing performance. (2) **Vertical Comparison.** As the model size increases, the relative importance of the dynamic scheduler and efficient parameter updating grows more pronounced. This trend is observable from the throughput: for larger models like OPT-6.7B, the reduction in throughput when the scheduler and efficient update features are disabled is relatively larger than in small models. This indicates that as models become larger, the complexities and overheads associated with managing and optimizing communications between CPU and GPU become more critical to maintaining performance. Conversely, the impact of reusable memory remains

relatively constant across different model sizes, reinforcing the idea that while CUDA malloc operations are significant, their relative burden does not scale in the same way as communication overheads.