Enabling Resource-Efficient On-Device Fine-Tuning of LLMs Using Only Inference Engines

Lei Gao [*] **Amir Ziashahabi** [*] **Yue Niu** **Salman Avestimehr** **Murali Annavaram**
University of Southern California
{leig, ziashaha, yueniu, avestime, annavara}@usc.edu

Abstract

Large Language Models (LLMs) have demonstrated exceptional performance in automating various tasks, such as text generation and summarization. Currently LLMs are trained and fine-tuned on large cloud server. Deploying and fine-tuning these models on resource-constrained edge devices remains a significant challenge due to their substantial memory and computational requirements. This paper introduces a resource-efficient zeroth-order optimization approach that lowers the barriers for fine-tuning LLMs in such constrained environments. Our method features a parallelized randomized gradient estimation (P-RGE) technique, which performs gradient estimation with high parallel efficiency. P-RGE leverages outer-loop and inner-loop parallelization to perform multiple function queries and forward passes in parallel, reducing the wall-clock end-to-end training time. By integrating this technique with parameter-efficient fine-tuning methods (e.g., LoRA) and on-device inference engines (e.g., ExecuTorch), we demonstrate efficient fine-tuning of LLMs on both server-side and edge devices. Experiments show that P-RGE achieves significant runtime speedups and memory savings while maintaining fine-tuning accuracy, which paves the way for more practical deployment of LLMs in real-time, on-device applications.

1 Introduction

Large Language Models (LLMs) have recently achieved remarkable success across various domains, including chatbot assistants [27, 8], image and video generation [33, 14], and healthcare applications [30, 31]. As the field continues to progress, there is a growing demand to deploy LLMs with billions of parameters directly on resource-constrained edge devices, such as smartphones, wearables, and other IoT devices [48, 43, 28]. In these scenarios, the ability to fine-tune models on users' data for personalized experiences, while preserving privacy, becomes crucial [5, 17, 42]. However, fine-tuning LLMs requires significant memory to store model weights, activations, and optimizer states [36], which poses substantial challenges for edge devices.

Resource-efficient fine-tuning methods, such as parameter-efficient fine-tuning (PEFT) [16, 15, 21, 20] and memory-efficient fine-tuning [9, 23, 18, 24], have been explored to reduce the memory footprint. However, even these methods struggle with the memory overhead of storing intermediate activations during backpropagation. For instance, fine-tuning Llama-7B can require 45.6 GB of memory for activations alone [23], making it impractical for many edge devices.

Zeroth-order (ZO) optimization has gained traction in resource-constrained setups, as it estimates gradients using only forward passes, reducing the memory required for activations. This approach, particularly the randomized gradient estimator (RGE) [10, 26], approximates gradients by computing function differences along randomly chosen directions, eliminating the need for backpropagation.

[*]equal contribution

However, RGE improves accuracy by increasing the number of function queries (i.e., estimating the gradient multiple times for a given input), which causes runtime to scale linearly [47, 12, 40]. Although ZO fine-tuning for LLMs holds promise for edge devices, practical implementations on non-CUDA platforms, such as Android, remain underexplored.

In this work, we propose the parallelized randomized gradient estimator (P-RGE), which introduces both outer-loop and inner-loop parallelization to efficiently handle multiple function queries and forward passes in parallel. P-RGE enhances the real-time performance while maintaining minimal memory overhead and integrates seamlessly with quantization methods for further optimization. Moreover, we demonstrate the deployment of P-RGE on edge devices using ExecuTorch and PyTorch, showcasing its real-world feasibility for fine-tuning LLMs on resource-constrained devices. In summary, our key contributions are:

- We introduce the parallelized randomized gradient estimator (P-RGE), a novel optimization framework that combines zeroth-order optimization with two key innovations: outer-loop parallelization and inner-loop parallelization. This combination enables efficient fine-tuning of LLMs on resource-constrained devices.
- Our outer-loop parallelization innovation allows multiple function queries to be executed simultaneously during each training step. By trading off a larger batch size for ZO fine-tuning with more function queries and a smaller batch size, we show that multi-query RGE with outer-loop parallelization improves performance without increasing computational cost or runtime compared to single-query RGE.
- Our second innovation proposes inner-loop parallelization, which enables parallel execution of two forward passes (with positive and negative perturbations), achieving up to $2\times$ speedup per training step when combined with outer-loop parallelization.
- We benchmark and demonstrate on-device fine-tuning of LLMs, deploying P-RGE on both Android smartphones (with NPU backend via ExecuTorch) and Jetson Nano (with CUDA backend via PyTorch), showcasing the feasibility of deploying large models on edge devices.

2 Background and Related Work

Low-Rank Adaptation. To reduce the resource demands of fine-tuning LLMs while maintaining comparable performance, parameter-efficient fine-tuning methods like LoRA [16] have been introduced. LoRA is designed to update only a small fraction of the model's parameters by leveraging the observation that weight changes in LLMs during fine-tuning exhibit a low-rank structure. In a linear layer, LoRA freezes the pre-trained weights $\mathbf{W} \in \mathbb{R}^{k_{in} \times k_{out}}$ and injects trainable low-rank matrices $\mathbf{A} \in \mathbb{R}^{k_{in} \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times k_{out}}$. Given that the rank r is much smaller than $\min(k_{in}, k_{out})$, the total number of trainable parameters is significantly reduced. The forward pass is then modified as $\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{x}\mathbf{A}\mathbf{B}$, where the input is $\mathbf{x} \in \mathbb{R}^{k_{in}}$ and the output is $\mathbf{y} \in \mathbb{R}^{k_{out}}$. The matrix $\mathbf{A}$ is initialized from a random Gaussian distribution, while $\mathbf{B}$ is initialized to zero to ensure the output $\mathbf{y}$ remains the same as the original layer at the beginning of training.

LoRA-FA [44], a variation of LoRA, further reduces the number of trainable parameters by freezing the randomly initialized matrix $\mathbf{A}$ and only updating the matrix $\mathbf{B}$, while still maintaining performance. QLoRA [9] further reduces the memory footprint of weight storage by quantizing the non-trainable weight matrices, excluding $\mathbf{A}$ and $\mathbf{B}$, into 4-bit integers. Since these matrices are not updated during training, their quantization scale remains unchanged. During forward and backward propagation, these parameters are dequantized back to BF16 precision for computation.

Zeroth-Order Optimization. ZO optimization methods have been widely applied across various machine learning applications [32, 39, 22]. Unlike FO optimization methods, which rely on direct gradient calculations to find optimal solutions, ZO optimization methods are gradient-free alternatives. They approximate FO gradients using function value-based estimates, referred to as ZO gradient estimates. ZO methods typically follow the algorithmic structure of their FO counterparts but replace the FO gradient with the ZO gradient estimate. Among ZO gradient estimators, the randomized gradient estimator (RGE) is particularly effective, especially for fine-tuning LLMs [24].

Given a labeled dataset $\mathcal{D}$ and a model with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, let the loss function on a minibatch $\boldsymbol{\mathcal{B}} \subset \mathcal{D}$ of size B be denoted as $\mathcal{L}(\boldsymbol{\theta}; \boldsymbol{\mathcal{B}})$. The RGE estimates the gradient of the loss $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$ on a minibatch $\boldsymbol{\mathcal{B}}$ using the following approximation:

$$\hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\boldsymbol{\mathcal{B}}) = \frac{1}{q}\sum_{i=1}^{q}\left[\frac{\mathcal{L}(\boldsymbol{\theta}+\epsilon\boldsymbol{z}_i;\boldsymbol{\mathcal{B}}) - \mathcal{L}(\boldsymbol{\theta}-\epsilon\boldsymbol{z}_i;\boldsymbol{\mathcal{B}})}{2\epsilon}\boldsymbol{z}_i\right], \tag{1}$$

where $\boldsymbol{z}_i \in \mathbb{R}^d$ is a random vector drawn from $\sim \mathcal{N}(0, \boldsymbol{I}_d)$, q is the number of function queries, and $\epsilon > 0$ is the perturbation scale.

RGE requires only two forward passes through the model to compute a single gradient estimate. As a result, there is no need to implement automatic differentiation to perform backpropagation. The choice of q balances the variance of the ZO gradient estimate and the computational cost. According to [47], the variance of the RGE is approximately $O(d/q)$.

ZO-SGD is an optimizer similar to standard SGD, but it replaces first-order gradients with ZO gradient estimates for updates, defined as $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\boldsymbol{\mathcal{B}}_t)$, where η is the learning rate and $\hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\boldsymbol{\mathcal{B}}_t)$ represents the ZO gradient estimate at training step t.

ZO LLM Fine-Tuning. Conventional ZO methods using RGE require twice the inference memory due to the need to cache the random noise $\boldsymbol{z}$. Memory-efficient ZO (MeZO) [24] is a memory-efficient variant of ZO that addresses this issue by storing the random seed instead of the random noise. During the forward pass, it resamples the same random noise $\boldsymbol{z}$ using the stored seed, thereby reducing the training memory cost to nearly match the inference memory cost.

MeZO sets $q = 1$ to minimize computational costs in each training step, although this comes with a performance trade-off. While MeZO eliminates the need for backpropagation, it still requires sequential operations to apply the perturbation tensor on the model weights, leading to longer runtime per training step. For LLMs, this sequential process can be especially time-consuming, potentially offsetting the advantages gained from bypassing backpropagation. A detailed description of the MeZO algorithm and its limitations is provided in Appendix A.1.

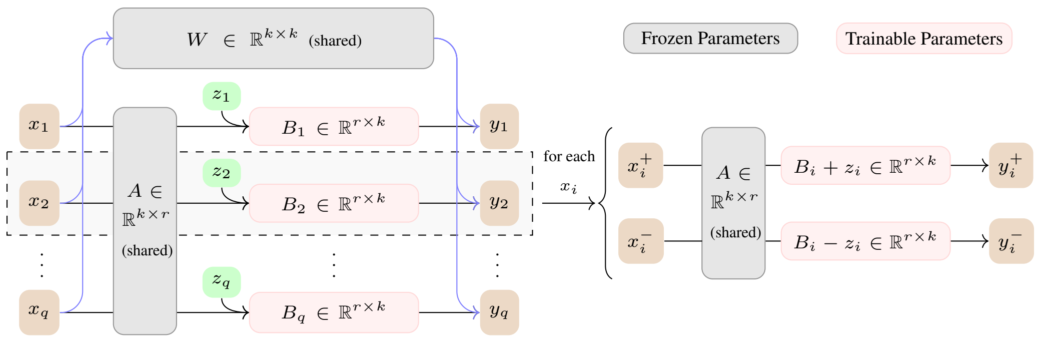
Extreme-sparse-MeZO [13] proposed integrating first-order Fisher information-based sparse training with the MeZO method. This approach significantly reduces the number of trainable parameters and outlines a potential workflow for on-device training, but it still lacks real-world validation. MeZO-SVRG [12] incorporates the first-order SVRG approach into MeZO. While this method demonstrates strong performance, it occasionally requires estimating gradients on the entire dataset, resulting in substantial computational overhead. DeepZero [6] proposed to use coordinate-wise gradient estimation to pre-train DNNs from scratch, which is not applicable to LLM fine-tuning. AdaZeta [40] introduced an adaptive query scheduling strategy to address persistent divergence issues in ZO fine-tuning. However, it still relies on sequentially execution of multiple function queries per training step, which slows down overall training time.

3 Methods

To address the challenges in ZO fine-tuning of LLMs, we propose parallelized randomized gradient estimation (P-RGE). P-RGE introduces a series of optimizations aimed at improving runtime speed and memory efficiency while still harnessing the performance gains from multi-query RGE. It is built on two key innovations: outer-loop parallelization and inner-loop parallelization for fast gradient estimation. We further present an implementation of P-RGE in PyTorch to facilitate the deployment of on-device training via inference engines.

3.1 Outer-loop Parallelization

Previous work [47] has shown that increasing the query budget improves the accuracy of RGE but comes at the cost of linearly increased computational overhead resulting in slower execution. To overcome this limitation, we propose parallel execution of multiple queries per training step by duplicating the model inputs and trainable parameters, then performing forward passes across different queries in parallel. To maintain the same computational cost as single-query RGE, we proportionally reduce the input batch size. To mitigate memory overhead associated with model parameters and reduce slowdown caused by sequential operations on parameters, we adopt PEFT methods to reduce the number of trainable parameters. Our preliminary experiments (see Appendix A.2 for more details) show that combining ZO with LoRA-FA outperforms other PEFT methods, such



$W \in \mathbb{R}^{k\times k}$ (shared)
Frozen Parameters
Trainable Parameters
x_1
x_2
x_q
$A \in \mathbb{R}^{k\times r}$ (shared)
z_1
z_2
z_q
$B_1 \in \mathbb{R}^{r\times k}$
$B_2 \in \mathbb{R}^{r\times k}$
$B_q \in \mathbb{R}^{r\times k}$
y_1
y_2
y_q
for each x_i
x_i^+
x_i^-
$A \in \mathbb{R}^{k\times r}$ (shared)
$B_i + z_i \in \mathbb{R}^{r\times k}$
$B_i - z_i \in \mathbb{R}^{r\times k}$
y_i^+
y_i^-

(a) Outer-loop parallelization (b) Inner-loop parallelization

Figure 1: Proposed P-RGE method.

as DoRA [41] and VeRA [19], making LoRA-FA our default setup. Although our implementation is based on LoRA-FA, this approach is adaptable to other PEFT methods.

As illustrated in Figure 1 (a), we first duplicate the model input batch (x) q times. We keep W and LoRA-$\mathbf{A}$ matrix frozen. Then, we duplicate the LoRA-$\mathbf{B}$ matrix q times. Each LoRA-$\mathbf{B}$ matrix is perturbed with different random noise during the forward pass, and each input batch is multiplied by its corresponding LoRA-$\mathbf{B}$ matrix using batched matrix multiplication. The layer inputs $x_1, x_2, \ldots, x_q$ originate from the same model input batch of arbitrary size B, but their values diverge after passing through the first LoRA module. Since the model input batch is duplicated across queries, the *effective batch size* becomes $E = q \cdot B$. To keep the computational cost and memory usage constant across different values of q, we proportionally reduce the batch size B, ensuring that E remains consistent. As we demonstrate later this trade-off of reducing B while increasing q leads to better model accuracy.

By adopting outer-loop parallelization, we increase parallelism across queries while enhancing data locality for model parameters. Specifically, the required weights are loaded once and reused across different queries, significantly reducing external memory access. With minimal overhead from batched matrix multiplication, P-RGE achieves comparable runtime per gradient estimation as single-query RGE but with the advantage of multiple-query gradient estimation.

3.2 Inner-loop Parallelization

While outer-loop parallelization increases parallelism across multiple queries, gradient estimation still requires two forward passes per query: one with positive perturbation and one with negative perturbation, traditionally executed sequentially in the RGE algorithm.

To further accelerate gradient estimation, we propose inner-loop parallelization, which performs both forward passes simultaneously. As illustrated in Figure 1 (b), each input batch and LoRA-$\mathbf{B}$ matrix is duplicated once more. One copy of the LoRA-$\mathbf{B}$ matrix is perturbed with positive noise, and the other with negative noise. By computing the loss difference in parallel, we can estimate the gradient using a single combined forward pass. This approach reduces external memory access for loading model parameters, alleviating the memory bandwidth burden, particularly for LLMs. As a result, P-RGE achieves a faster wall-clock time per training step compared to the sequential two step forward-pass execution in conventional RGE.

With inner-loop parallelization, the activations at each layer are doubled, as it doubles the effective batch size. However, this does not result in significant memory overhead. Unlike first-order methods, ZO methods allow activations from previous layers to be discarded during forward passes, preventing accumulation across layers. This property, as noted in [47], enables ZO methods to scale more efficiently with long sequence lengths and large batch sizes compared to FO methods. To minimize memory costs for storing LoRA-$\mathbf{B}$ weight matrices, it is possible to keep a master copy of LoRA-$\mathbf{B}$ and generate perturbed copies dynamically during the forward pass. At each LoRA layer, only the master copy is updated with the gradient and learning rate. Perturbed copies of LoRA-$\mathbf{B}$ are then generated and discarded once the output is computed, ensuring that the number of additional trainable parameters remains the same as in the standard LoRA-FA method.

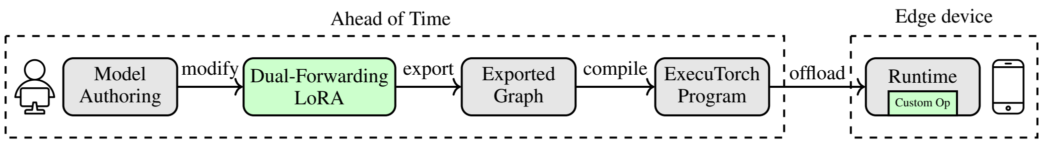


Figure 2: On-device training workflow via ExecuTorch. The green box represents additional procedure in addition to standard steps for inference deployment on edge devices.

3.3 On-Device Training Workflow

For the on-device implementation of our proposed methods, we have selected ExecuTorch [1] as the inference engine. ExecuTorch, the successor to PyTorch Mobile [3], allows developers to perform model inference across various platforms with different backends (e.g., CPUs, NPUs, and DSPs) using the same toolchains and SDKs provided by PyTorch.

Deploying a PyTorch model (`nn.Module`) on edge devices for inference via ExecuTorch generally involves two main steps. First, we convert the PyTorch model into an ExecuTorch program, which is essentially a computation graph of the model with its parameters. This process generates a binary file that contains ExecuTorch instructions, which the ExecuTorch runtime can interpret and execute. Second, we offload both the binary file and the runtime library to the target platform. The runtime, written in C++ and OS-independent, includes an operator library tailored for the hardware backend of the device and is responsible for executing the ExecuTorch program.

However, the MeZO implementation (Algorithm 2) is not natively supported in the ExecuTorch workflow as it requires major modifications on the device side (e.g., resetting the random seed, generating noise, extracting weights, applying gradients, etc.). Extracting weights from the binary file is particularly challenging, as ExecuTorch runtime does not provide an API for this purpose. To simplify the deployment process, we leverage inner-loop parallelization and propose a dual-forwarding LoRA module implementation in PyTorch. In this approach, the training procedure for P-RGE is defined within the model's `forward` function and fully exportable to an ExecuTorch program. This enables us to generate and offload the ExecuTorch program with minimal server-side modifications, allowing for training without any changes to the ExecuTorch runtime on edge devices.

In our dual-forwarding LoRA module as shown in Algorithm 1, the first step is to compute the difference between the perturbed weights $\mathbf{B}[0]$ (positive perturbation) and $\mathbf{B}[1]$ (negative perturbation), which is the same random noise scaled by ϵ from the previous step. Since resetting the random seed is not an exportable operation when converting to an ExecuTorch program (line 14 and 22 in Algorithm 2), we retain all copies of matrix $\mathbf{B}$ in memory rather than maintaining a single master copy. This allows us to recover the random noise from the previous step without regenerating it using a seed. Lines 4 and 5 in the algorithm restore the original value of matrix $\mathbf{B}$ from the previous step, update the weights with gradients, and then apply new random noise. The output is subsequently computed by combining the original linear transformation $\mathbf{xW}$ with a batched matrix multiplication between matrices $\mathbf{xA}$ and $\mathbf{B}$. This method can also be extended to handle larger input batch sizes and incorporate with the outer-loop parallelization technique.

Figure 2 illustrates the workflow for enabling on-device training using dual-forwarding through ExecuTorch. Starting with a pre-trained PyTorch model, we inject the dual-forwarding LoRA module and direct the projected gradient g to each LoRA module. Following the standard ExecuTorch work-

Algorithm 1 Dual-forwarding LoRA-FA module definition

```
Require: $\mathbf{x} \in \mathbb{R}^{2\times \text{seq\_len} \times \text{k}}$, $\mathbf{A} \in \mathbb{R}^{k\times r}$, $\mathbf{B} \in \mathbb{R}^{2\times r\times k}$, $\mathbf{W}^{k\times k}$, learning rate $\eta$, perturbation scale $\epsilon$,
    projected gradient $g$
1: diff $\leftarrow \frac{\mathbf{B}[0]-\mathbf{B}[1]}{2}$
2: update $\leftarrow \eta \cdot g \cdot \frac{\text{diff}}{\epsilon}$
3: $\mathbf{z} \leftarrow \epsilon \cdot \text{randn\_like}(\mathbf{B}[0])$
4: $\mathbf{B}[0] \leftarrow \mathbf{B}[0] - \text{diff} - \text{update} + \mathbf{z}$
5: $\mathbf{B}[1] \leftarrow \mathbf{B}[1] + \text{diff} - \text{update} - \mathbf{z}$
6: return output $\leftarrow \mathbf{xW} + \text{bmm}(\mathbf{xA}, \mathbf{B})$
```

flow, we export, compile, and offload the model to the edge device. On the device, the ExecuTorch runtime executes the binary file. For random noise generation, a customized operator can be easily integrated into the existing runtime library via the ExecuTorch API [2].

4 Experiments

We conduct comprehensive experiments on the TinyLlama-1.1B [45] and Llama2-7B [35] models across different systems to evaluate both model performance and system efficiency. The experiments begin by fine-tuning the models on server-side systems to establish their baseline accuracy.

4.1 Fine-Tuning Performance

We compare two sets of baselines: the first uses an FO-Adam optimizer in both the full and LoRA-FA parameter spaces, and the second employs a ZO-SGD optimizer with $q = 1$ in the same parameter spaces. To ensure equivalent computation per training step while varying q, we set the effective batch size $E = 16$ for all ZO methods. Zero-shot performance is also reported. For ZO-LoRA-FA with $q > 1$ (multi-query RGE), it is implemented by outer-loop parallelization to improve computational efficiency. Additional experimental details, including the training procedure and hyperparameters, are provided in Appendix A.3 and A.4.

For the smaller-scale TinyLlama-1.1B model, we evaluate its performance on the GLUE dataset [37]. The results in Table 1 show that increasing the number of queries while decreasing the batch size outperforms the baseline (i.e., $q = 1, B = 16$) by $1.4 - 8.5\%$ accuracy.

Table 1: Performance of fine-tuning TinyLlama-1.1B on different tasks with different optimizers.

Task	**SST-2**	**RTE**	**MRPC**	**QQP**	**QNLI**	**WNLI**
Zero-shot	55.3	52.3	68.3	32.8	52.7	43.6
FO-Full	91.9	72.5	77.4	82.4	80.8	56.3
FO-LoRA-FA	94.2	82.6	82.3	84.4	86.5	56.3
ZO-Full ($q = 1$)	91.5	66.4	71.3	71.4	73.9	63.4
ZO-LoRA-FA ($q = 1$)	88.8	70.8	74.8	76.0	74.0	57.7
ZO-LoRA-FA ($q > 1$)	91.6	72.9	77.7	78.0	79.4	66.2

For the larger Llama2-7B model, we evaluate its performance on SST-2 [37], WinoGrande [29], and the SuperGLUE [38] dataset using the same experimental setup. As shown in Table 2, multi-query RGE consistently improves performance over the baseline in the LoRA-FA parameter space. Although ZO in the full parameter space with $q = 1$ achieves higher accuracy on some tasks, it is computationally inefficient, as detailed in Appendix A.1.

Table 2: Performance of fine-tuning Llama2-7B on different tasks with different optimizers. ZO-LoRA-FA ($q > 1$) is implemented by outer-loop parallelization.

Task	**SST-2**	**RTE**	**BoolQ**	**WSC**	**WiC**	**MultiRC**	**COPA**	**WinoGrande**	**SQuAD**
Zero-shot	58.0	59.2	71.9	51.9	50.0	54.6	79.0	62.7	23.8
FO-Full	92.5	78.7	80.6	63.4	67.2	71.7	81.0	68.2	79.2
FO-LoRA-FA	95.5	87.7	86.5	76.9	75.3	82.8	87.0	70.0	76.5
ZO-Full ($q = 1$)	94.7	77.3	81.9	70.2	58.8	72.7	82.0	65.0	76.9
ZO-LoRA-FA ($q = 1$)	89.3	65.0	80.2	65.4	59.2	68.2	88.0	64.3	78.3
ZO-LoRA-FA ($q > 1$)	92.2	76.2	81.5	66.3	63.5	72.9	89.0	64.6	82.1

4.2 Server-Side System Performance

We further evaluate the server-side performance of P-RGE by measuring the runtime per training step and memory usage across different sequence lengths and batch sizes. The ZO-SGD optimizer performs the forward pass in FP16 precision to maximize computational efficiency, due to its tolerance for low-precision gradient estimation [47]. For a fair comparison, the FO optimizer uses standard SGD without momentum, combined with FP16 mixed-precision training.

Runtime Speedup. Figure 3 shows the runtime per training step for vanilla MeZO-Full, ZO-LoRA-FA ($q = 1$), and ZO-LoRA-FA ($q = 1$) with inner-loop parallelization, measured at different sequence lengths and effective batch sizes. ZO-LoRA-FA achieves faster runtime per training step than vanilla ZO-Full, as PEFT methods reduce the overhead associated with sequential process of model parameters. We observe that as long as the effective batch size E remains constant, ZO-LoRA-FA ($q > 1$) implemented by outer-loop parallelization introduces no runtime overhead compared to the ZO-LoRA-FA ($q = 1$) baseline as shown in Appendix A.6, meaning measuring ZO-LoRA-FA ($q = 1$) is sufficient for runtime analysis.

More importantly, when inner-loop parallelization is enabled, the training runtime is further improved, up to $1.79\times$ for Llama2-7B when the sequence length is 64 and the batch size is 1. This speedup is achieved by reusing model weights across two forward passes, reducing cache access and increasing arithmetic intensity. As a result, system performance improves, particularly when the system is memory-bound. In addition, when inner-loop parallelization is combined with NF4 quantization, runtime speedup improves further, reaching up to $1.97\times$. Detailed experiments and discussion on this are provided in Appendix A.5.

In practice, outer-loop and inner-loop parallelization can yield even greater speedups. Unlike the benchmarks reported above, which use unpadded batches, realistic datasets often include tokenized sequences of varying lengths, which are padded to ensure uniform batch sizes. Larger batch sizes can result in more padding tokens, leading to wasted computation. However, using smaller and duplicated batch sizes in the P-RGE optimizer reduces the number of padding tokens during the forward pass, resulting in faster runtime per step. The wall-clock end-to-end training time for actual tasks are provided in Appendix A.6.

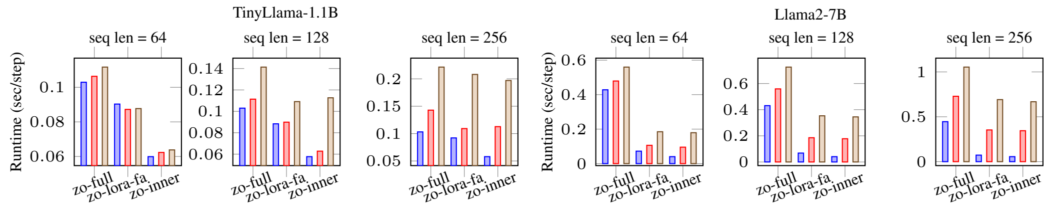


Figure 3: Runtime per training step of TinyLlama-1.1B and Llama2-7B for different sequence lengths and batch sizes (blue/red/brown represent $B = 1/8/16$ respectively).

Memory Efficiency. We evaluate the memory overhead of inner-loop parallelization versus sequential execution for each function query. We measure peak memory usage, subtracting memory for model weights, which varies with the quantization method. The measured footprint includes CUDA kernels, activations, gradients, and other implementation-specific details.

As shown in Figure 4, memory usage is significant for the FO optimizer due to the need to store activations from all intermediate layers, along with a master copy of the model weights in FP32 during mixed-precision training [25]. In contrast, for the ZO optimizer, inner-loop parallelization roughly doubles memory usage by increasing the size of the largest output tensor during the forward pass. However, it still requires significantly less memory than the FO optimizer. For example, in Llama2-7B, when the sequence length is 256 and the batch size is 16, memory usage increases from 0.99GB to 1.97GB, while FO requires over 30GB of memory. The detailed memory usage during training actual tasks are provided in Appendix A.7.

4.3 On-Device Training Experiments

For on-device training experiments, we begin by performing a sanity check, verifying the loss values per step on two edge platforms: the NVIDIA Jetson Nano Orin (8GB) GPU and the OnePlus 12 smartphone (12GB) NPU. We ensure that both platforms produce the same values as those observed on the server side. Detailed edge system specifications are provided in Appendix A.9. After verification, we measure and report the runtime per step of P-RGE.

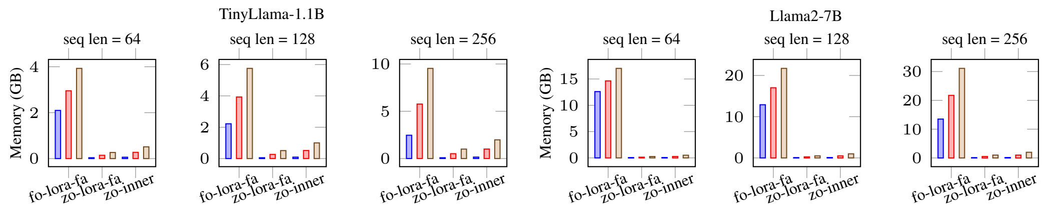


Figure 4: Peak memory usage (excluding model weights) of TinyLlama-1.1B and Llama2-7B for different sequence lengths and batch sizes (blue/red/brown represent $B = 1/8/16$ respectively).

Since the Jetson platform runs a Linux system, we utilize the PyTorch library for the model's forward pass and the bitsandbytes library [11] for quantization. Table 3 presents the speedup of inner-loop parallelization under NF4 quantization, achieving up to $1.83\times$ faster performance. The observed trend is similar to that on the server side, where speedup diminishes as the system becomes more computationally bound.

Table 3: Runtime (sec/step) and speedup ratio of inner-loop parallelization on Jetson GPU backend for TinyLlama-1.1B and Llama2-7B with NF4 quantization.

Model	Sequence length	64				128			
	Batch size	1	2	4	8	1	2	4	8
TinyLlama-1.1B	ZO-LoRA-FA	0.69	0.71	0.89	1.28	0.70	0.88	1.27	2.18
	ZO-inner	0.43	0.49	0.69	1.15	0.49	0.69	1.13	2.00
	Speedup ratio	1.62	1.45	1.29	1.12	1.42	1.29	1.12	1.09
Llama2-7B	ZO-LoRA-FA	3.10	3.37	4.44	6.46	3.37	4.44	6.47	10.83
	ZO-inner	1.69	2.22	3.22	5.38	2.22	3.22	5.37	8.60
	Speedup ratio	1.83	1.52	1.38	1.20	1.52	1.38	1.21	1.26

On the smartphone platform, which runs Android OS, PyTorch is not supported, so we follow the ExecuTorch workflow. However, since Executorch currently lacks support for weight-only quantization, we run TinyLlama-1.1B in FP16 mode. We also observe that Executorch does not handle batch sizes greater than 1 efficiently, as it is primarily designed for chat LLMs, where a single user input sequence is processed at a time, as shown in Table 4. Exploring alternative backends on ARM SoCs, such as Vulkan backend on GPU, is left as future work.

Table 4: Runtime (sec/step) of dual-forwarding implementation on Android NPU backend for TinyLlama-1.1B without quantization.

Sequence length	64				128			
Effective batch size	2	4	8	16	2	4	8	16
Dual-Forwarding	1.04	2.34	4.70	10.43	2.49	4.83	10.36	15.73

5 Conclusion

This work introduces parallelized randomized gradient estimation (P-RGE) to address the computational and memory challenges of fine-tuning large language models (LLMs) in resource-constrained environments. P-RGE leverages outer-loop and inner-loop parallelization to enable efficient multi-query gradient estimation, improving accuracy without adding computational overhead. Our experiments demonstrate that P-RGE significantly enhances wall-clock training time efficiency and reduces memory usage on both server and edge platforms, enabling real-time on-device fine-tuning. By integrating P-RGE with inference engines like ExecuTorch, we showcase the practical applicability of our method across diverse hardware backends, including Android NPUs and Jetson GPUs. Future work will explore extending P-RGE to other ZO optimization methods and refining quantization techniques to support broader deployment scenarios.

Acknowledgment

We sincerely thank all the reviewers for their time and constructive comments. This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001120C0088, NSF award number 2224319, REAL@USC-Meta center, and VMware gift. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We would also like to extend our thanks to the ExecuTorch team from Meta and Yanzhi Wang from Northeastern University for their valuable discussions and advice.

References

[1] Meta AI. Executorch, 2024. URL `https://pytorch.org/executorch-overview`.

[2] Meta AI. Executorch kernel registration, 2024. URL `https://pytorch.org/executorch/stable/kernel-library-custom-aten-kernel`.

[3] Meta AI. Pytorch mobile, 2024. URL `https://pytorch.org/mobile/home`.

[4] Stephen H. Bach and et al. Promptsource: An integrated development environment and repository for natural language prompts, 2022. URL `https://arxiv.org/abs/2202.01279`.

[5] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[6] 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.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016. URL `https://arxiv.org/abs/1604.06174`.

[8] Aakanksha Chowdhery and et al. Palm: Scaling language modeling with pathways, 2022. URL `https://arxiv.org/abs/2204.02311`.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[10] 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. doi: 10.1109/TIT.2015.2409256.

[11] BytesAndBytes Foundation. Bytesandbytes, 2024. URL `https://github.com/bitsandbytes-foundation/bitsandbytes`.

[12] Tanmay Gautam, Youngsuk Park, Hao Zhou, Parameswaran Raman, and Wooseok Ha. Variance-reduced zeroth-order methods for fine-tuning language models. In *5th Workshop on practical ML for limited/low resource settings*, 2024.

[13] Wentao Guo, Jikai Long, Yimeng Zeng, Zirui Liu, Xinyu Yang, Yide Ran, Jacob R. Gardner, Osbert Bastani, Christopher De Sa, Xiaodong Yu, Beidi Chen, and Zhaozhuo Xu. Zeroth-order fine-tuning of llms with extreme sparsity, 2024. URL `https://arxiv.org/abs/2406.02913`.

[14] Wenyi Hong, Ming Ding, Wendi Zheng, Xinghan Liu, and Jie Tang. Cogvideo: Large-scale pretraining for text-to-video generation via transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

[15] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[16] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[17] Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, Philip S. Yu, and Xuyun Zhang. Membership inference attacks on machine learning: A survey. *ACM Comput. Surv.*, 2022.

[18] Jeonghoon Kim, Jung Hyun Lee, Sungdong Kim, Joonsuk Park, Kang Min Yoo, Se Jung Kwon, and Dongsoo Lee. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2024.

[19] Dawid Jan Kopiczko, Tijmen Blankevoort, and Yuki M Asano. VeRA: Vector-based random matrix adaptation. In *The Twelfth International Conference on Learning Representations*, 2024.

[20] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021.

[21] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, 2021.

[22] Z Liu, J Lou, W Bao, Y Hu, B Li, Z Qin, and K Ren. Differentially private zeroth-order methods for scalable large language model finetuning, 2024. URL `https://arxiv.org/abs/2402.07818`.

[23] Kai Lv, Yuqing Yang, Tengxiao Liu, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2024.

[24] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[25] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[26] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 2017.

[27] OpenAI, Josh Achiam, and et al. Gpt-4 technical report, 2024. URL `https://arxiv.org/abs/2303.08774`.

[28] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zheng Lin, Xianhao Chen, and Kaibin Huang. Mobile edge intelligence for large language models: A contemporary survey, 2024. URL `https://arxiv.org/abs/2407.18921`.

[29] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 2021.

[30] Karan Singhal and et al. Large language models encode clinical knowledge, 2022. URL `https://arxiv.org/abs/2212.13138`.

[31] Karan Singhal and et al. Towards expert-level medical question answering with large language models, 2023. URL `https://arxiv.org/abs/2305.09617`.

[32] Tianxiang Sun, Zhengfu He, Hong Qian, Yunhua Zhou, Xuanjing 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*, 2022.

[33] Gemini Team, Rohan Anil, and et al. Gemini: A family of highly capable multimodal models, 2024. URL `https://arxiv.org/abs/2312.11805`.

[34] Gemma Team, Morgane Riviere, and et al. Gemma 2: Improving open language models at a practical size, 2024. URL `https://arxiv.org/abs/2408.00118`.

[35] Hugo Touvron and et al. Llama 2: Open foundation and fine-tuned chat models, 2023. URL `https://arxiv.org/abs/2307.09288`.

[36] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. Efficient large language models: A survey. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. Survey Certification.

[37] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2019.

[38] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems, 2020. URL `https://arxiv.org/abs/1905.00537`.

[39] Xiaoxing Wang, Wenxuan Guo, Jianlin Su, Xiaokang Yang, and Junchi Yan. ZARTS: On zero-order optimization for neural architecture search. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[40] Yifan Yang, Kai Zhen, Ershad Banijamal, Athanasios Mouchtaris, and Zheng Zhang. Adazeta: Adaptive zeroth-order tensor-train adaption for memory-efficient large language models fine-tuning, 2024. URL `https://arxiv.org/abs/2406.18060`.

[41] Shih yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. DoRA: Weight-decomposed low-rank adaptation. In *Forty-first International Conference on Machine Learning*, 2024.

[42] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, 2024.

[43] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. Llm as a system service on mobile devices, 2024. URL `https://arxiv.org/abs/2403.11805`.

[44] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning, 2023. URL `https://arxiv.org/abs/2308.03303`.

[45] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model, 2024. URL `https://arxiv.org/abs/2401.02385`.

[46] Susan Zhang and et al. Opt: Open pre-trained transformer language models, 2022. URL `https://arxiv.org/abs/2205.01068`.

[47] Yihua Zhang, Pingzhi Li, Junyuan Hong, Jiaxiang Li, Yimeng Zhang, Wenqing Zheng, Pin-Yu Chen, Jason D. Lee, Wotao Yin, Mingyi Hong, Zhangyang Wang, Sijia Liu, and Tianlong Chen. Revisiting zeroth-order optimization for memory-efficient LLM fine-tuning: A benchmark. In *Forty-first International Conference on Machine Learning*, 2024.

[48] Ligeng Zhu, Lanxiang Hu, Ji Lin, Wei-Chen Wang, Wei-Ming Chen, and Song Han. Pock-engine: Sparse and efficient fine-tuning in a pocket. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

A Appendix

A.1 MeZO Algorithm and Its Limitation

We evaluate the runtime efficiency of the MeZO optimizer, restated from the original work in Algorithm 2. MeZO uses a random seed trick to eliminate the need for storing random noise, trading off computational efficiency and memory usage.

MeZO operates over four distinct loops within each iteration:

1. The first loop adds positive noise to the trainable parameters.
2. The second loop perturbs the weights in the opposite direction using the same noise.
3. The third loop restores the weights to their original state before the update.
4. The fourth loop applies the computed gradients to update the weights.

This method reduces memory overhead from $O(d)$ to $O(1)$ by avoiding the storage of random noise. However, the computation cost escalates from $O(1)$ to $O(d)$ because each parameter update requires individual processing, which cannot be efficiently parallelized. In practical settings, especially with LLMs, iterating over the full parameter set four times per update can significantly slow down the training process, thus negating the benefits of eliminating backpropagation.

In contrast, PyTorch's FO optimizers utilize a *foreach* implementation by default. This method aggregates all layer weights into a single tensor during parameter updates, which speeds up the computation. However, this approach also increases the memory usage by $O(d)$, as it requires maintaining a copy of the weights for the update process.

Algorithm 2 MeZO with $q > 1$.

```
Require: parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, loss $L : \mathbb{R}^d \to \mathbb{R}$, step budget $T$, function query budget $q$, perturbation scale $\epsilon$, learning rate $\eta$
1:  for $t = 1, \dots, T$ do
2:      for $i = 1, \dots, q$ do
3:          seeds, projected_grads $\leftarrow$ []
4:          Sample batch $\mathcal{B} \subset \mathcal{D}$ and random seed $s$
5:          $\boldsymbol{\theta} \leftarrow$ PerturbParameters($\boldsymbol{\theta}, \epsilon, s$)
6:          $\ell_+ \leftarrow L(\boldsymbol{\theta}; \mathcal{B})$
7:          $\boldsymbol{\theta} \leftarrow$ PerturbParameters($\boldsymbol{\theta}, -2\epsilon, s$)
8:          $\ell_- \leftarrow L(\boldsymbol{\theta}; \mathcal{B})$
9:          $\boldsymbol{\theta} \leftarrow$ PerturbParameters($\boldsymbol{\theta}, \epsilon, s$)
10:         projected_grads[i] $\leftarrow (\ell_+ - \ell_-)/(2\epsilon)$
11:         seeds[i] $\leftarrow s$
12:     end for
13:     for $i = 1, \dots, q$ do
14:         Reset random number generator with seed seeds[i]
15:         for $\boldsymbol{\theta}_j \in \boldsymbol{\theta}$ do
16:             $z \sim \mathcal{N}(0, 1)$
17:             $\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j - (\eta_t/q) \times$ projected_grads[i] $\times z$
18:         end for
19:     end for
20: end for

21: function PerturbParameters($\boldsymbol{\theta}, \epsilon, s$)
22:     Reset random number generator with seed $s$
23:     for $\boldsymbol{\theta}_j \in \boldsymbol{\theta}$ do
24:         $z \sim \mathcal{N}(0, 1)$
25:         $\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j + \epsilon z$
26:     end for
27:     return $\boldsymbol{\theta}$
28: end function
```

Table 5 compares the runtime of the Llama2-7B model using both FO-SGD and MeZO-SGD optimizers ($q = 1$) over the full parameter space across various batch sizes and sequence lengths. The FO optimizer is run with FP16 mixed-precision training, while MeZO uses pure FP16 to maximize computational speed. To avoid out-of-memory errors, we utilize two NVIDIA A100 (40GB) GPUs for the FO optimizer, which incurs additional GPU communication time in a distributed environment.

When both the batch size and sequence length are small, MeZO exhibits significantly higher runtime due to the overhead of sequential operations required to apply perturbations and gradients. However, as the batch size and sequence length increase, where forward and backward passes, as well as GPU communication, dominate the runtime, the MeZO optimizer demonstrates improved performance. This behavior highlights the importance of applying PEFT methods with MeZO to mitigate the computation overhead caused by the sequential processing of model parameters.

Table 5: Runtime (sec/step) of Llama2-7B using FO and MeZO optimizers over full parameter space.

Sequence length	64			128			256		
Batch size	1	4	8	1	4	8	1	4	8
FO-SGD	0.17	0.21	0.34	0.19	0.33	0.49	0.18	0.49	0.90
MeZO-SGD ($q = 1$)	0.43	0.48	0.56	0.43	0.56	0.73	0.45	0.73	1.05

A.2 Preliminary Experiment of ZO with Different PEFT Methods

We conducted a preliminary experiment by fine-tuning the OPT-1.3B model [46] for 10,000 iterations on the SST2 dataset [37] using ZO-SGD optimizer with different PEFT methods. We use hyperparameter grid search with learning rate $\in \{5e-6, 5e-5, 5e-4, 5e-3\}$ and $\epsilon \in \{1e-3, 1e-2\}$. LoRA [16], LoRA-FA [44], and DoRA [41] are configured with $r = 16$, and VeRA [19] uses $r = 1024$. The results in Table 6 indicate that the LoRA-FA method outperforms other PEFT methods in terms of accuracy.

Table 6: ZO accuracy of OPT-1.3B on SST2 dataset using different PEFT methods.

PEFT Methods	LoRA	LoRA-FA	DoRA	VeRA
Accuracy	90.9	92.0	90.9	91.4

A.3 Experimental Setup

We evaluate the performance of the TinyLlama-1.1B model on six tasks from the GLUE dataset [37]: sentiment analysis (SST2), paraphrase (MRPC and QQP), and natural language inference (QNLI, RTE, and WNLI).

For the larger Llama2-7B model, evaluations were performed on two tasks from the GLUE dataset: SST2 and RTE. Additionally, the model was tested on six tasks from the SuperGLUE dataset [38], categorized as follows: text classification (BoolQ, WSC, WIC, and MultiRC), multiple-choice (COPA), and question-and-answering (SQuAD). We include one additional multiple-choice task from WinoGrande [29] dataset. For question-and-answering tasks, we utilize the F1 score as a metric, while accuracy metrics are used for the rest.

We achieve text classification, multiple-choice and question-and-answering tasks through next-word prediction, adopting the same prompt template from MeZO [24] and PromptSource[4], as shown in Table 7. We compute the loss value of prediction over the entire vocabulary space instead of only the vocabulary space of the ground true.

For these tests, we also adopt a low-volume data condition, limiting our samples to 1,000 for training, 500 for validation, and 1,000 for testing, as proposed in the original MeZO work [24].

The runtime and memory usage measurements of ZO-SGD optimizer are conducted on a single NVIDIA A100 (40GB) GPU. The memory usage measurements of FO-SGD optimizer are conducted on two NVIDIA A100 (40GB) GPUs to avoid out-of-memory error.

Table 7: The prompt template of the datasets we used in our experiments.

Dataset	**Type**	**Prompt**
SST-2	cls.	`<text>` It was terrible/great
RTE	cls.	`<premise>` Does this mean that "`<hypothesis>`" is true? Yes or No? Yes/No
MRPC	cls.	Do the following two sentences mean the same thing? Yes or No? Sentence 1: `<sentence1>` Sentence 2: `<sentence2>` Yes/No
QQP	cls.	Are these two questions asking the same thing? Yes or No? Question 1: `<question1>` Question 2: `<question2>` Yes/No
QNLI	cls.	Does this sentence answer the question? Yes or No? Sentence 1: `<sentence1>` Sentence 2: `<sentence2>` Yes/No
WNLI	cls.	Given the first sentence, is the second sentence true? Yes or No? Sentence 1: `<sentence1>` Sentence 2: `<sentence2>` Yes/No
BoolQ	cls.	`<passage>` `<question>` `<answer>`? Yes/No
WSC	cls.	`<text>` In the previous sentence, does the pronoun "`<span2>`" refer to `<span1>`? Yes/No
WIC	cls.	Does the word "`<word>`" have the same meaning in these two sentences? `<sent1>` `<sent2>` Yes, No?
MultiRC	cls.	`<paragraph>` Question: `<question>` I found this answer "`<answer>`". Is that correct? Yes or No?
COPA	mch.	`<premise>` so/because `<candidate>`
WinoGrande	mch.	`<context>` `<subject>` `<object>`
SQuAD	QA	Title: `<title>` Context: `<context>` Question: `<question>` Answer:

A.4 Hyperparameters

We report the hyperparameters searching grids in Table 8. For LoRA hyperparameters, we choose the LoRA rank to be 16 and alpha to be 16. Note that for the multi-query RGE with outer-loop parallelization, we do not search for the case where $E \neq 16$.

Table 8: Hyperparameters for Llama2-7B and TinyLlama-1.1B experiments. FO experiments use Adam optimizer and are trained for 1,000 iterations. ZO experiments are trained for 20,000 iterations. Performance on the test dataset is evaluated every 100 steps for all tasks except SQuAD, which is evaluated every 500 steps.

	Llama2-7B	
Experiment	Hyperparameters	Values
FO-Full	Batch size	{8}
	Learning rate	{1e-5, 5e-5, 8e-5} or {1e-7, 5e-7, 8e-7} for SQuAD
FO-LoRA-FA	Batch size	{8}
	Learning rate	{1e-4, 3e-4, 5e-4}
ZO-Full	Batch size	{16}
	Learning rate	{1e-7, 5e-7, 1e-6}
	ϵ	1e-3
ZO-LoRA-FA	Effective batch size	{16}
	q	{1, 2, 4, 8, 16}
	Learning rate	{1e-4, 5e-4, 1e-3, 5e-3} or {1e-5, 5e-5, 1e-4, 5e-4} for RTE, MultiRC and WinoGrande
	ϵ	1e-2
	TinyLlama-1.1B	
FO-Full	Batch size	{8}
	Learning rate	{1e-5, 5e-5, 8e-5}
FO-LoRA-FA	Batch size	{8}
	Learning rate	{1e-4, 3e-4, 5e-4}
ZO-Full	Batch size	{16}
	Learning rate	{1e-7, 5e-7, 1e-6}
	ϵ	1e-3
ZO-LoRA-FA	Effective batch size	{16}
	q	{1, 2, 4, 8, 16}
	Learning rate	{1e-5, 5e-5, 1e-4, 5e-4} or {1e-4, 5e-4, 1e-3, 5e-3} for SST2
	ϵ	1e-2

A.5 Runtime Speedup of Inner-loop Parallelization With Quantization

We report the memory usage for storing model weights under various weight-only quantization schemes, which is independent of the optimizer used. Therefore, the peak memory usage is the sum of the values reported in Figure 4 and the memory required to store the model weights, as shown in Table 9.

Table 9: Weight-only quantization memory usage (GB).

Quantization	FP32	FP16	INT8	NF4
TinyLlama-1.1b	4.10	2.05	1.15	0.70
Llama2-7b	25.10	12.56	6.52	3.50

We also evaluate the speedup achieved by inner-loop parallelization under weight-only INT8 and NF4 quantization. As shown in Figure 5, inner-loop parallelization yields the highest speedup when combined with NF4 quantization, achieving up to a 1.97× improvement compared to the sequential execution of the two forward passes. Since NF4 dequantization requires more operations than INT8 during the forward pass, the efficiency of inner-loop parallelization comes from dequantizing the weights only once per training step. This reduces the overhead from repeated dequantization, making the process more computation-efficient overall.

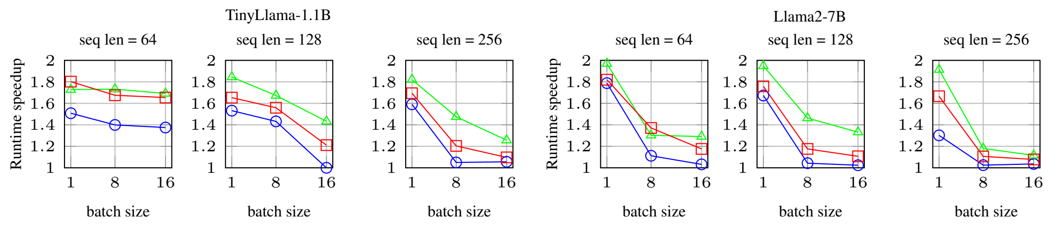


Figure 5: Runtime speedup per training step of TinyLlama-1.1B and Llama2-7B for different quantization methods, sequence lengths and batch sizes (blue/red/green represent FP16/INT8/NF4 quantization respectively).

A.6 End-to-end Training Runtime Efficiency

The results presented in Tables 10 and 11 provide a detailed comparison of the wall-clock end-to-end time for fine-tuning the TinyLlama-1.1B and Llama2-7B models across various tasks using different configurations of the ZO optimizer. Vanilla implementation represents the sequential execution of two forward pass per function query. For $q > 1$, we enable outer-loop parallelization.

This further confirms that outer-loop parallelization does not incur extra runtime overhead as long as E remains constant. ZO-LoRA-FA with both outer-loop and inner-loop parallelization demonstrates the most significant reduction in runtime, achieving up to a 2× speedup compared to the vanilla ZO-LoRA-FA ($q = 1$) when fine-tuning Llama2-7B on RTE task. These results highlight the advantage of inner-loop and outer-loop parallelization, especially in tasks requiring larger models and longer training times.

A.7 End-to-end Training Memory Efficiency

The results presented in Tables 12 and 13 compare the peak memory usage for fine-tuning the TinyLlama-1.1B and Llama2-7B models across different tasks using various configurations of the ZO optimizer. For both models, the ZO-LoRA-FA method with inner-loop parallelization incurs higher memory usage compared to vanilla implementations due to the simultaneous computation of two forward passes. Specifically, for Llama2-7B, tasks like SQuAD and MultiRC see an increase in memory usage of up to 30% when using inner-loop parallelization due to larger sequence length. Despite this increase, the memory efficiency remains within acceptable bounds.

Table 10: Runtime (min/task) of fine-tuning TinyLlama-1.1B for the ZO runs corresponding to the results reported in Table 1.

Task	**SST-2**	**RTE**	**MRPC**	**QQP**	**QNLI**	**WNLI**
ZO-Full ($q = 1$)	38.31	61.51	45.71	40.76	46.30	43.57
ZO-LoRA-FA ($q = 1$)						
Vanilla	34.66	55.53	35.45	35.00	37.44	34.40
Inner-loop	23.55	54.07	35.72	28.76	36.59	33.22
ZO-LoRA-FA ($q > 1$)						
Vanilla	35.57	38.18	37.29	36.19	35.86	36.81
Inner-loop	24.77	31.98	35.41	25.83	27.43	27.39

Table 11: Runtime (min/task) of fine-tuning Llama2-7B for the ZO runs corresponding to the results reported in Table 2.

Task	**SST-2**	**RTE**	**BoolQ**	**WSC**	**WiC**	**MultiRC**	**COPA**	**WinoGrande**	**SQuAD**
ZO-Full ($q = 1$)	159.44	288.1	384.07	209.72	173.01	526.49	146.4	154.74	480.90
ZO-LoRA-FA ($q = 1$)									
Vanilla	43.79	185.73	285.88	99.04	62.21	371.23	30.76	39.40	398.50
Inner-loop	42.40	181.51	281.62	95.95	58.94	368.95	26.42	37.14	388.74
ZO-LoRA-FA ($q > 1$)									
Vanilla	34.25	94.17	217.63	86.15	46.45	426.87	31.40	37.03	326.38
Inner-loop	27.97	90.46	210.80	85.08	45.24	416.91	26.86	34.13	321.20

Table 12: Peak memory usage (GB) of fine-tuning TinyLlama-1.1B for the ZO runs corresponding to the results reported in Table 1.

Task	**SST-2**	**RTE**	**MRPC**	**QQP**	**QNLI**	**WNLI**
ZO-Full ($q = 1$)	2.56	3.38	2.74	2.74	2.74	2.77
ZO-LoRA-FA ($q = 1$)						
Vanilla	2.35	3.27	2.63	2.63	3.06	2.66
Inner-loop	2.63	4.46	3.18	3.18	4.04	3.24
ZO-LoRA-FA ($q > 1$)						
Vanilla	2.44	3.18	2.64	2.65	3.14	2.70
Inner-loop	2.81	4.28	3.19	3.22	4.22	3.33

Table 13: Peak memory usage (GB) of fine-tuning Llama2-7B for the ZO runs corresponding to the results reported in Table 2.

Task	**SST-2**	**RTE**	**BoolQ**	**WSC**	**WiC**	**MultiRC**	**COPA**	**WinoGrande**	**SQuAD**
ZO-Full ($q = 1$)	13.64	16.23	18.39	14.51	13.82	18.39	13.60	13.60	18.39
ZO-LoRA-FA ($q = 1$)									
Vanilla	13.41	16.00	18.16	14.27	13.58	18.22	12.98	13.15	18.16
Inner-loop	14.19	19.37	23.69	15.92	14.53	23.81	13.33	13.67	23.69
ZO-LoRA-FA ($q > 1$)									
Vanilla	13.44	15.54	18.17	14.27	13.64	18.18	12.98	13.16	18.17
Inner-loop	14.24	18.45	23.71	15.92	14.65	23.74	13.33	13.69	23.71

A.8 Additional Memory Efficiency Analysis

For LLMs with extensive vocabulary sizes, the final output tensor from the classifier, namely the logits, can be significantly large. For example, the Gemma model [34] has a vocabulary size of 256,000, compared to Llama2's 32,000. This results in an $8\times$ increase in memory consumption during the forward pass. To address this issue, a potential solution is to implement a tiling operation, which computes the loss in smaller chunks. This approach reduces memory usage of output activations by eliminating the need to calculate the entire logits tensor at once.

Several techniques have been developed to reduce the memory costs associated with intermediate activations during backpropagation. For example, gradient checkpointing [7] discards selected activations during the forward pass and recomputes them during backpropagation, trading additional computation for reduced memory usage. Gradient accumulation proposes accumulating gradients over multiple smaller batches and updating the model weights once, instead of after each batch, reducing memory overhead. Mixed-precision training [25] stores activations in FP16 while maintaining a FP32 master copy of the weights and applying gradients in FP32, so the memory savings from activations can be offset when the model has a large number of parameters. PockEngine [48] introduces an approach that limits backpropagation to a subset of layers and updates only the weights of those layers, thereby eliminating the need to store activations for layers closer to the input. This method can also be interpreted as a form of gradient estimation.

More importantly, these methods depend on training frameworks that support automatic differentiation and backpropagation, while also being flexible enough to support state-of-the-art LLM architectures and integrate aforementioned memory-saving techniques. This greatly increases the engineering complexity of developing training frameworks on the device, particularly when ensuring compatibility with various hardware backends from different vendors. In contrast, our approach leverages highly optimized on-device inference engines, which are developed and maintained by the industry or research community, reducing the need for extensive engineering effort.

A.9 Edge Devices Used in Experiments

Table 14 presents the specifications of the edge computing devices used in the experiments, detailing the CPU, memory, and accelerator components.

Table 14: Edge devices used in the experiments.

Device	CPU	Memory	Accelerator
NVIDIA Jetson Orin Nano	6× 1.5GHz Cortex-A78AE	8GB LPDDR5 68GB/s	1024-core Ampere GPU 625MHz
OnePlus 12	1× 3.3GHz Cortex-X4 3× 3.2GHz Cortex-A720 2× 3.0GHz Cortex-A720 2× 2.3GHz Cortex-A520	12GB LPDDR5 77GB/s	Hexagon NPU