RADIUS: RANGE-BASED GRADIENT SPARSITY FOR LARGE FOUNDATION MODEL PRE-TRAINING

Mingkai Zheng [1] **Zhao Zhang** [1]

ABSTRACT

We present Radius, a gradient sparsity algorithm and system to accelerate large foundation model (FM) training while preserving downstream task performance. Radius leverages two key insights in large FM pre-training: 1) only a small portion of gradients contribute to the model updates in each iteration, and 2) the spatial distribution of the gradients with large magnitude is stable over time. Radius overcomes the scaling problem of existing top-k sparsity methods, as it maintains the structure of sparse gradients thus avoids dense communication. We examine the convergence and speed of Radius on pre-training GPT models (355M and 2.0B) in data-parallel and compare it with the baseline top-k sparsification methods. Our results show that using the existing top-k method with AdamW optimizer fails to converge, and the training speed improvement with sparse communication is marginal. In contrast, Radius with 40% sparsity reduces per-step training time by 21% (19% for overall training time) across 64 NVIDIA A100 GPUs that are connected by the Slingshot 11 interconnect while preserving the downstream task performance.

1 INTRODUCTION

Large Foundation Models (FMs), such as GPT-like Large Language Models (LLMs), have shown unparalleled performance in various fields, including Natural Language Processing (NLP), digital agriculture (Bran et al., 2023), biology (Ahdritz et al., 2022), economics (Lee et al., 2024), and mathematics (Microsoft Research et al., 2023). Training a large FM model is expensive and time-consuming. According to the results reported by Meta AI team, the pre-training of LLaMA-7B and LLaMA-65B on a cluster of NVIDIA A100-80GB GPUs consume 82,432 and 1,022,362 GPU hours, respectively (Touvron et al., 2023). Similarly, pre-training a GPT-8.3B model using Megatron-LM on 512 NVIDIA V100 GPUs takes 2.1 days to finish an epoch (Shoeybi et al., 2020). In addition to the intrinsic computation complexity of LLMs, such long pre-training time is attributed to the frequent and high-volume communication between GPUs. Optimus-CC (Song et al., 2023) reports that communication accounts for 40% of the overall training time of GPT-8.3B pre-training across 128 NVIDIA A100 GPUs. Data Parallelism (DP) (Li et al., 2014) is a commonly used training technique for accelerating the training process. By replicating the initialized model and distributing the training dataset to multiple workers, each worker performs forward and backward computation and then uses the averaged gradients to update the model. Large FMs usually require model parallelism techniques, such as Pipeline Parallelism (PP) (Harlap et al., 2018; Huang et al., 2019; Narayanan et al., 2021) and Tensor Parallelism (TP) (Shoeybi et al., 2020), so that the aggregated High Bandwidth Memorys (HBMs) can provide sufficient space for both the model and the training dataset. PP distributes decoder layers of an LLM onto multiple GPUs, whereas TP splits the multi-head attention operation and matrix-matrix multiplication operations onto multiple GPUs. Taking the GPT model family as an example, PP distributes the decoder layers over multiple stages, where each stage is placed on one or several GPUs. On the tensor dimension, TP splits the multi-head attention operation and matrix-matrix multiplication operations onto multiple GPUs. The combination of these three scaling techniques forms 3D parallelism. It brings opportunities to train increasingly larger LLMs but injects high communication overhead into the training process.

Many gradient compression methods have been proposed to reduce the communication overhead introduced by DP, including low-rank approximation (Vogels et al., 2020; Bian et al., 2024) and sparsification (Shi et al., 2019; Wang et al., 2021; Li & Hoefler, 2022). These existing methods either assume a very low-bandwidth network, a naive sparsity strategy, or a fixed optimizer such as momentum SGD. With the rapid evolution of communication hardware and optimizers, the assumptions of these methods are no longer valid for

[1] Department of Electrical and Computer Engineering, Rutgers University, New Brunswick, New Jersey, USA. Correspondence to: Mingkai Zheng <mz687@rutgers.edu>, Zhao Zhang <zhao.zhang@rutgers.edu>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

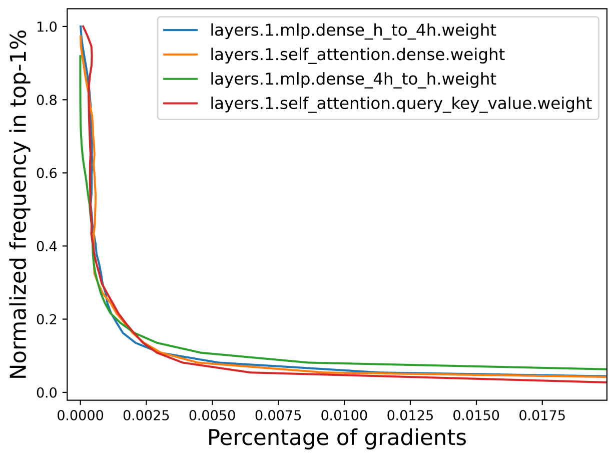


Figure 1. The distribution of gradients with top-1% greatest magnitude values for pre-traing GPT-355M from step 80,000 to step 84,000

LLM pre-training. For instance, DGC (Lin et al., 2018) requires a high sparsity, such as 1%, to justify its communication overhead of exchanging top-k values on a 1 Gbps Ethernet. In contrast, modern GPU clusters, such as the Perlmutter supercomputer, are equipped with high-bandwidth interconnects such as the 400 Gbps and 800 Gbps Infiniband (De Sensi et al., 2020). When measuring the speedup of the state-of-the-art gradient compression method (Song et al., 2023) on Perlmutter, it did not provide an observable improvement. From the optimizer perspective, AdamW (Zhao et al., 2024) (Adam with weight-decay) has become the default optimizer for pre-training LLMs. It has been theoretically proven by DGC (Lin et al., 2018) that directly applying the top-k sparsification method to optimizers, such as Adam (Kingma & Ba, 2017) and AdamW (Loshchilov & Hutter, 2019), can harm the convergence. This is because these optimizers perform non-linear operations on the gradients before updating the model parameters. Low-rank approximation methods, such as PowerSGD (Vogels et al., 2020), maintain the gradient in a matrix or vector structure, but when the model size is sufficiently large, the time complexity of computing the approximation of the gradient can dominate communication. Thus, we need to revisit the top-k sparsification method on its correctness and effectiveness in reducing communication overhead.

To address the above challenges, we introduce **Radius**, a range-based top-k gradient sparsification method for pre-training large FMs. Radius is designed based on the following two insights:

- After 15-20% of the total training steps, the indices of gradients that have top-1% greatest magnitudes exhibit temporal locality: their indices in a flattened gradient vector are relatively stable across the remaining pre-training steps. Figure 1 illustrates this observation in an MLP layer and an Attention layer in GPT-355M pertaining from step 80,000 to step 84,000, where the horizontal axis shows the percentage of indices, and the vertical axis is the normalized frequency gradients falling within the top-1% range. We can see that for both layers, a small percentage of gradients have normalized frequency close to 1, suggesting that these indices and their corresponding gradients are selected as top-1% in almost every step.
- Infrequently performing top-k selection operation on gradient does not cause a serious staleness effect with Adam and AdamW.

Based on these two insights, Radius can leverage *allreduce* to communicate sparse gradients in a structure, which effectively avoids the scalability bottleneck of *allgather* in existing top-k methods.

We develop Radius based on Megatron-LM framework (Shoeybi et al., 2020), and then verify the convergence of Radius on GPT-355 and GPT-2.0B models trained using OpenWebText dataset (Gokaslan & Cohen, 2019) on the NERSC Perlmutter supercomputer. By applying a sparsity of 40%, Radius achieves 19.03% speedup for GPT-2.0B pre-training, without affecting their scores on the downstream tasks evaluation, including GLUE and SuperGLUE.

2 Background

In this section, we review the complex model parallelism in LLM training and existing gradient compression methods.

2.1 3D parallelism

In this section, we describe 3D parallelism for training LLMs, including data parallelism, pipeline parallelism, and tensor parallelism.

2.1.1 Data parallelism

Data parallelism (DP) partitions a large dataset into smaller chunks and distributes them to multiple workers, where each worker has a replication of the initialized model (Li et al., 2014; 2020). During the training process, to update the model parameters, each worker computes the forward pass to obtain the loss of the Neural Network (NN) and then performs a backward pass to compute the gradients using the training data and the activation values:

$$\text{Loss} = \frac{1}{N}\sum_{n=1}^{N} f(\theta, x_n), \tag{1}$$

$$G = \frac{1}{N}\sum_{n=1}^{N} \nabla f(\theta, x_n), \tag{2}$$

where θ is the model parameters, f is the loss function, x_i is a mini-batch of training data, and G is the gradient. As the size of the dataset increases, the total computation time also increases. By fixing the global batch size, the size of the partitioned dataset decreases proportionally to the number of workers. Nevertheless, this does not yield the same factor of speedup on the total training process, because of the existing communication and I/O overhead.

Many DP architectures have been proposed, including parameter server architecture (Li et al., 2014) and distributed data-parallel (DDP) (Li et al., 2020). In the parameter-server architecture, there is one server and multiple workers. Initially, the server distributes the partitioned dataset chunks and the up-to-date model to the workers. Then, after all the workers finish computing the gradients, they transmit their gradients to the server, and the server updates its model parameters using these gradients. Finally, the server distributes another set of dataset chunks and the model to the workers to perform the next step of parameter updates. The major disadvantage of this approach is that when all workers transmit the gradients, each of which has the same size as the model, communication congestion might occur, dramatically slowing down the whole training process. Instead, in the DPP architecture, each worker maintains and updates its own model parameters. To exchange the gradients, ring $allreduce$ is usually adopted, and its communication cost can be estimated as

$$C_{\text{ring-}allreduce} = 2\alpha(n-1) + 2D\beta\frac{n-1}{n}, \tag{3}$$

where n is the number of workers, D is the size of the gradient, α is the communication latency, and β is the inverse of the communication bandwidth. When the number of workers n is large enough, $C_{\text{ring-}allreduce}$ is dominated by the size of the gradient and the communication bandwidth. Moreover, when a model and the partitioned dataset are too large to fit into the HBM of a GPU, DP cannot be used.

2.1.2 Pipeline parallelism

Pipeline parallelism (PP) (Harlap et al., 2018; Huang et al., 2019; Narayanan et al., 2021) is one of the approaches to use the aggregated HBM of multiple GPUs to hold the model and the dataset. Since the modern Deep Learning (DL) models have many layers of NNs (e.g., GPT-series models are composed of multiple decoder layers), PP partitions them onto multiple GPUs, and during the forward- and backward-pass, each worker communicates to its adjacent workers to transmit the activation values and intermediate gradients. To improve the pipeline utilization rate, one mini-batch is split into multiple micro-batches, and the model updates its parameters after the forward- and backward-pass of all micro-batches have been finished. When two adjacent pipeline stages are partitioned onto GPUs in different nodes, the corresponding inter-node point-to-point communication cost is non-negligible.

2.1.3 Tensor parallelism

Tensor parallelism (TP) (Shoeybi et al., 2020) is another model parallelism method specially designed for transformer models (Vaswani et al., 2023). Since the multi-head attention operation can naturally be performed independently and the matrix-matrix multiplication operations in Multi-Layer Perceptron (MLP) layers can also be parallelized, by splitting these operations across workers, TP effectively reduces the memory burden on each worker. The main overhead introduced by TP is that the input x_n and the results need to be broadcasted to all workers that belong to the same operation group in the forward pass, and the corresponding $allreduce$ operations are required in the backward pass, which can cause substantial communication overhead if these workers spread across nodes. Therefore, the common practice is to constrain the TP operations into workers in the same node, where workers are usually connected with each other through high communication bandwidth, such as NVLink, which has bandwidth as high as 600 GB/s for NVIDIA Amphere architecture GPUs.

2.2 Gradient compression

Gradient compression is one of the methods for reducing the communication overhead mentioned above. From Equation (3), we see that $C_{\text{ring}-allreduce}$ is dominated by the gradient size D when the number of workers n is large enough. Thus, reducing D is a very effective method to reduce $C_{\text{ring}-allreduce}$. More details can be found in Section 3.

3 Related Work

Many gradient compression methods have been proposed to reduce communication overhead, which can be summarized into three main categories, namely sparsification (Lin et al., 2018; Shi et al., 2019; Wang et al., 2021; Li & Hoefler, 2022), low-rank approximation (Vogels et al., 2020; Song et al., 2023), and quantization (Wen et al., 2017; Bernstein et al., 2018; Tang et al., 2021).

Gradient sparsification selects and exchanges the gradients that have top-k greatest magnitudes. However, such naive top-k-based method has two main issues. First, it has low scalability. Since each worker has different top-k greatest gradients with different indices in the gradient tensor, $Allgather$ method needs to be performed to collect such magnitudes and indices across all workers. The cost of ring-based $allgather$ operation can be formulated as

$$C_{allgather} = \alpha n + D\beta\frac{(n-1)}{n}. \tag{4}$$

Given there are n workers in a *allgather* communication group, each worker hence needs to prepare n buffers, where each buffer has the same size as the selected top-k gradients, and the total communication volume is proportional to n. Therefore, it eventually surpasses dense *allreduce* when n is sufficiently large, and it introduces high memory overhead, which limits its scalability. Second, the top-k gradient selection operation is expensive. As the size of the gradient increases, its time cost can exceed time cost by performing *allreduce* on the sparsified gradient tensor. Many research works have been proposed to solve these two issues of top-k-based gradient sparsification method. For instance, DGC (Lin et al., 2018) uses a randomly selected subset of gradient tensor to reduce the top-k selection time. Ok-Top$k$ (Li & Hoefler, 2022) and gtop-k (Shi et al., 2019) propose new methods to estimate the threshold of the top-k magnitude. Moreover, Ok-Top$k$ (Li & Hoefler, 2022) splits the gradient tensor into many regions, and each worker is only responsible for the reduction of its assigned region, which reduces the memory overhead caused by *allgather* operation. Researchers also propose to use Fast-Furious Transform (FFT) for top-k selection on the gradients (Wang et al., 2021).

Low-rank approximation uses a tensor with a lower rank to approximate the gradient tensor. Singular Value Decomposition (SVD) is one of the commonly used methods for constructing such low-rank matrix. Because of the significantly high computation overhead of SVD, it is not often incorporated into the training process. PowerSGD (Vogels et al., 2020) uses an iterative approach to approximate the results of SVD, which reduces such computation overhead. Optimus-CC (Song et al., 2023) applies PowerSGD to reduce the communication overhead in data parallelism and pipeline parallelism and achieves good performance.

Quantization uses fewer number of bits to represent each value in the gradient tensor. SignSGD (Bernstein et al., 2018) and 1-bit Adam (Tang et al., 2021) use 1-bit binary value to represent the gradients. TernGrad (Wen et al., 2017) quantize the gradients from either full precision or half precision to {-1, 0, 1} represented by 2 bits.

4 System Design

In this section, we describe the design of Radius in detail. The pseudo-code of Radius is shown in Algorithm 1.

4.1 *Allreduce*-based top-k sparsification

The only communication collective in Radius is *allreduce*. It is designed so to avoid the aforementioned high bandwidth and high memory requirement of the existing *allgather*-based top-k sparsification method (Equation (4)). To avoid exchanging the top-k gradient values with their corresponding indices, we leverage our observation that the indices of top-k gradients are relatively stable after 15%-20% of the total number of training steps. From Figure 1, we can see that the distributions of the top-1% gradients in those layers are stable across many training steps.

This suggests the top-k indices selected in one training step can be reused in the following steps, and frequently performing the top-k selection operation is unnecessary. However, reusing the same top-k indices for too many steps can also lead to divergence (Li & Hoefler, 2022). Therefore, in addition to the compression density d, we introduce another hyper-parameter T, the top-k indices resampling interval. When t % T = 0, dense *allreduce* is preformed, and then top-k indices selection is performed. Since all workers have

Algorithm 1 Radius

```
Require: Loss function: $f$
Require: Learning rate: $\alpha$
Require: Model parameters on worker $n$ at step $t$: $w_n^t$
Require: Dataset partition on worker $n$ at step $t$: $x_n^t$
Require: Gradients on worker $n$ at step $t$: $G_n^t$
Require: Compression density: $D \in (0, 1]$
Require: Top-$k$ indices at step $t$: top-$k$_idx$^t$
Require: Resampling interval top-$k$ indices: $T$
1: for $t = 1, 2, 3 \dots T$ do
2:   $G_n^t \leftarrow \nabla f(w_n^t, x_n^t)$
3:   if $t <$ threshold then
4:     All-reduce($G_n^t$): $G^t \leftarrow \sum_{n=1}^{N} G_n^t / N$
5:     $G_n^t \leftarrow G^t$
6:   else
7:     if $t$ % $T$ == 0 then
8:       $G_n^t \leftarrow$ residual $+ G_n^t$
9:       All-reduce($G_n^t$): $G^t \leftarrow \sum_{n=1}^{N} G_n^t / N$
10:      residual $\leftarrow 0$
11:      $G_n^t \leftarrow G^t$
12:    else
13:      $G_{n,\text{top}-k}^t \leftarrow G_n^t$[top-$k$_idx$^t$]
14:      All-reduce($G_{n,\text{top-}k}^t$):
                 $G_{\text{top-}k}^t \leftarrow \sum_{n=1}^{N} G_{n,\text{top-}k}^t / N$
15:      $G_n^t$[top-$k$_idx$^t$] $\leftarrow G_{\text{top}-k}^t$
16:      residual$\leftarrow$residual$+G_n^t$[$\sim$ top-$k$_idx$^t$]
17:      $G_n^t$[$\sim$ top-$k$_idx$^t$] $\leftarrow 0$
18:    end if
19:  end if
20:  $G_n^t \leftarrow$ gradient_clip($G_n^t$)
21:  $G_n^t \leftarrow$ gradient_correction($G_n^t$)
22:  if $t \geq$ threshold and $t$ % $T$ == 0 then
23:    top-$k$_idx$^t \leftarrow$ top-$k$_selection($|G^t|, D$)
24:  end if
25:  $w_{n+1}^t \leftarrow w_n^t - \alpha G_n^t$
26: end for
```

the same aggregated gradient results after the $allreduce$ operation, every worker is guaranteed to have the same top-k indices at the end of this step. When t % $T \neq 0$, the previously obtained top-k indices are reused to compress the gradient.

As the top-k indices are identical on all workers, sparse representation, such as coordinate list (COO), compressed sparse row (CSR), and compressed sparse column (CSC), is not needed. Only a dense matrix, which is $1/d$ times smaller than the original gradient matrix, needs to be transmitted, saving the communication bandwidth by $1/d$ times.

Algorithm 2 Gradient correction

```
Require: Betas: $\beta_1, \beta_2$
Require: Weight decay: $\lambda$
Require: Model parameters at step $t$: $w_t$
Require: Gradients at step $t$: $G_t$
Require: The first-order momentum at step $t$: $m_t$
Require: The second-order momentum at step $t$: $v_t$
1: $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)G_t$
2: $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)G_t^2$
3: $\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$
4: $\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$
5: $G_t \leftarrow \hat{m}_t/(\sqrt{\hat{v}_t}+\epsilon) + \lambda w_t$
```

4.2 Gradient correction

It is important to realize that small gradients do not necessarily lead to small changes in the model parameters. For advanced optimizers, the value used to update a model parameter is a combination of nonlinear operations on optimizer states, gradients, and model parameters. Inspired by DGC (Lin et al., 2018), we perform the gradient correction operations, shown in Algorithm 2, before selecting the indices of top-k gradients. Gradient correction here is the same as the operations performed within the AdamW optimizer (Loshchilov & Hutter, 2019), one of the most commonly used optimizers for training LLMs. Performing the top-k selection on the corrected gradients gives a more accurate selection of the indices with major model parameter changes in the following T steps. Gradient clipping is performed before the gradient correction to maintain the same execution order without using Radius.

4.3 Error feedback

Gradient sparsification approximates the gradient information, and those gradient values not selected by the top-k operation are the approximation errors. At every training step, each worker in a DP group has its own small batch of training data and computes its own gradient value. With the same top-k indices on all workers, each worker thus has its own approximation errors. Similar to PowerSGD (Vogels et al., 2020), we use a residual buffer to accumulate the non-top-k gradient values when t % $T \neq 0$. We compensate the gradient with the accumulated errors and then compute the average of the results for all workers when t % $T = 0$. After the error compensation operation, we set the residual buffer to 0 to prepare it for the next T steps. This method has been proven effective in many different types of gradient sparsification methods, and it also prevents Radius from divergence.

5 IMPLEMENTATION

We implemented Radius based on Megatron-LM (Shoeybi et al., 2020). The top-k gradient selection is performed by `torch.topk` function. As the operations in gradient correction are identical to AdamW optimizer, they are implemented by reusing the fused implementation of AdamW in `Apex` (NVIDIA, 2018) to reduce the computation overhead.

To achieve minimal memory overhead, instead of storing the indices in `Int64` format, we used a buffer of binary values to represent whether a gradient value is selected as top-k or not. Furthermore, we used SGD optimizer to perform model parameter updates, but its momentum buffer was discarded to further reduce Radius's memory overhead. It is worth mentioning that PyTorch prevents the slicing operation from using a binary buffer of size greater than `INT32_MAX`. Therefore, to support models with parameters greater than 2.1B, each layer has its own indices buffer instead of using one unified buffer to store all the indices.

Table 1. Model architecture

MODEL	n_{LAYERS}	n_{HEADS}	d_{MODEL}	d_{HEAD}
GPT-355M	24	16	1024	64
GPT-2.0B	24	32	2560	80

6 RESULTS

We evaluated the convergence and scalability of Radius by pre-training GPT-355M and GPT-2.0B. In this section, we will first describe the experiment setup. Then, we will present the experiment results in detail. Finally, we will provide some discussions on the results.

6.1 Experiment setup

In this section, we describe our experiment environment, the model architectures, the pre-training dataset, and the hyperparameter settings used to pre-train the GPT models.

6.1.1 Experiment environment

Our experiment used 16 compute nodes of the Perlmutter supercomputer, where each node has 4 NVIDIA A100-80GB

Table 2. Train, validation perplexity, wall-clock time, and overall training speedup.

		Baseline	Radius				
			$d = 0.9$, $T = 200$	$d = 0.5$, $T = 200$	$d = 0.4$, $T = 200$	$d = 0.4$, $T = 400$	$d = 0.1$, $T = 200$
GPT-355M	Train PPL	**14.64**	—	14.65	—	—	—
	Val. PPL	14.32	—	**14.31**	—	—	—
GPT-2.0B	Train PPL	**10.86**	10.87	10.89	10.94	10.94	11.85
	Val. PPL	11.42	**11.37**	11.39	11.41	11.46	11.98
	Wall-clock time (days)	6.44	6.82	5.73	5.43	5.41	4.78
	Speedup	—	-5.55%	12.40%	18.55%	19.03%	34.76%

GPUs connected by NVLink3, providing a total intra-node communication bandwidth of 600 GB/s. The inter-node communication network is HPE Slingshot 11, and each GPU within a node has a NIC of 25 GB/s, which provides an aggregated bandwidth of 100 GB/s.

6.1.2 *Models architecture and training dataset*

We pre-trained both GPT model with 355M parameters (GPT-355M) and GPT model with 2.0B parameters (GPT-2.0B). GPT-355M model has 24 decoder layers, where within each decoder layer, it has 16 attention heads and a hidden dimension of 1024, resulting in each attention head of dimension 64. GPT-2.0B model has the same number of decoder layers, but each decoder layer has 32 attention heads and a hidden dimension of 2560, which results in each attention head of dimension 80. The details of the model architectures are summarized in Table 1.

We used OpenWebText Corpus (Gokaslan & Cohen, 2019) as our pre-training dataset, an open-source version of the WebText dataset, which can be accessed through Hugging Face.

We used mixed precision training. We store the model parameters in half-precision format (`bfloat16`), and to guarantee the training stability, we use the full precision format (`float32`) for the gradient.

We first verify the convergence of Radius with $d = 0.5$ and $T = 200$ by pre-training the GPT-355M model, then we use Radius to pre-train the GPT-2.0B model. We only provide the wall-clock training time, the average per-step speedups, and the overall speedups for different density d and top-k indices resampling interval T for pre-training the GPT-2.0B model because performing compression on the gradients of the GPT-355M model cannot provide observable speedup as the size of its gradients is much lower than Perlmutter's inter-node communication bandwidth.

6.1.3 *Hyper-parameter selection and training strategy*

We followed the training scripts provided by Megatron-LM to set the hyper-parameters, including the learning rate, the learning rate scheduler, the minimum learning rate, the gradient clipping, and the weight decay. We pre-trained GPT-355M and GPT-2.0B for 500,000 iterations and 300,000 iterations, and we switched from using dense $allreduce$ to Radius at 80,000 iteration and 60,000 iteration, respectively. We used DP = 64, PP = 1, and TP = 1 to pre-train both models. We used global batch size 512 and micro-batch size 8 as the default setting if not otherwise mentioned in the following sections.

6.2 Training efficiency

The full training perplexity (PPL) curves for GPT-2.0B model using the baseline method (dense $allreduce$), and Radius with different compression rates d's and resampling intervals T's are shown in Figure 2, and the speedup for each method is shown in Table 2.

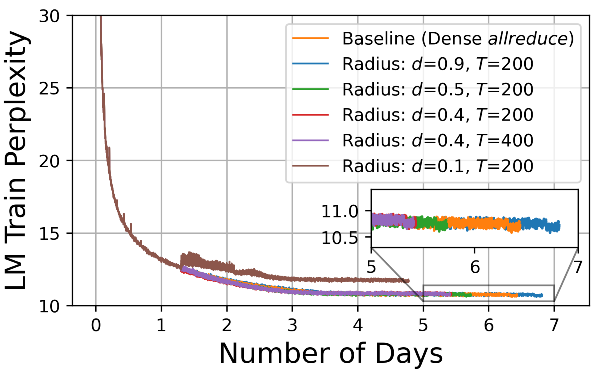


Figure 2. Training PPL vs wall-clock time for training GPT-2.0B.

The average computation and communication time breakdown per training step are provided in Figure 3, from which we can see that compressing gradients by half can reduce the communication cost by 58.76%. If we continue compressing the gradients to $d = 0.4$, the communication cost can be reduced by 89.69%. $d = 0.1$ can provide the highest speedup, but as we will see in the following section, its final training and validation PPL values are much higher than the baseline.

Since Radius is designed based on the observation that top-k gradient values exhibit temporal stability after 15-20% of the total steps trained by using dense $allreduce$, Radius can only speed up the rest, resulting in a lower speedup for the overall training process. We pre-trained GPT-355M using Radius with d = 0.4 and T = 200 starting from 5% of training steps, but its loss on downstream evaluation tasks is significant.

Radius adds computation overhead to the training process from two sources: the top-k indices sampling and gradient correction. Because the operation of resampling top-k indices is performed every T steps, this overhead can be amortized by T to every training step. We used the fused implementation of AdamW to implement gradient correction so that its computation overhead is not more expensive than calling a `optimizer.step()`. When the speedup achieved by exchanging compressed gradients cannot cover the computation overhead, we can see a slowdown in the wall-clock training time, such as the case when d = 0.9 and T = 200 in Figure 3.

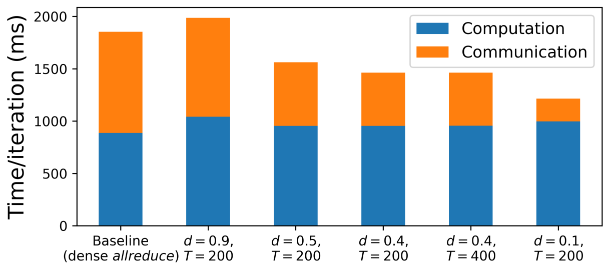


Figure 3. Average per-step time breakdown for pre-training GPT-2.0B model on 64 A100 GPUs with DP = 64, TP = 1, and PP = 1, using dense $allreduce$ and Radius. Radius with d = 0.9 and T = 200, d = 0.5 and T = 200, d = 0.4 and T = 200, d = 0.4 and T = 400, d = 0.1 and T = 200. The averaged communication time per step is compressed from 964.99 ms to 944.71 ms (**1.02**×), 2299 607.82 ms (**1.59**×), 508.69 ms (**1.90**×), and 508.15 ms (**1.90**×), 216.39 (**4.46**×), respectively.

We compared Radius to PowerSGD (Vogels et al., 2020), a low-rank approximation-based communication compressing method, implemented by Optimus-CC (Song et al., 2023), using rank = 128. We measured the breakdown of communication and computation overhead for Radius and PowerSGD on 64 A100 GPUs on Perlmutter. The detailed results are shown in Table 3. We found that the computation overhead introduced by PowerSGD (mainly due to the orthogonalization operations) outweighs its speedup gained through compressed gradient communication. Radius with d = 0.4 is 5.14% faster than PowerSGD with rank = 128, and the final validation PPL is much lower than it reported by Optimus-CC.

Table 3. Breakdown of the per-step time for the baseline, PowerSGD, and Radius measured on 64 A100 GPUs.

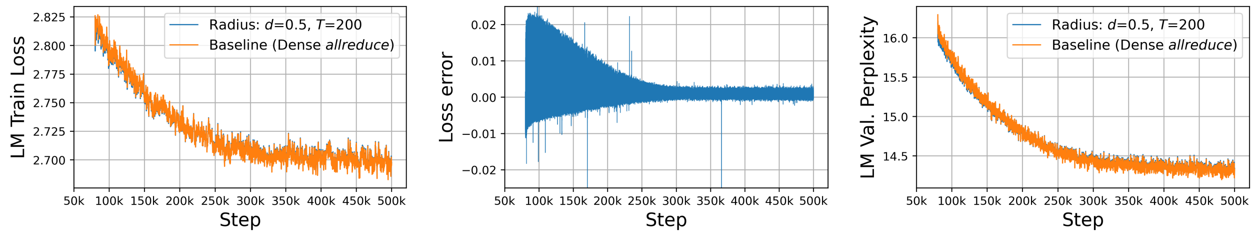
	Time per step (ms)	Allreduce (ms)	Computation overhead (ms)
Baseline	1852.94	964.99	0.00
PowerSGD (rank=128)	1538.43	75.79	574.70
Radius (d=0.5, T=200)	1562.75	607.82	66.98
Radius (d=0.4, T=200)	1463.18	508.69	66.54
Radius (d=0.1, T=200)	1214.66	216.39	110.32

6.3 Training error analysis

The training loss, per-step training loss errors between baseline (dense $allreduce$) and different Radius schemes, and the validation PPL curves for GPT-355M and GPT-2.0B are shown in Figure 4 and Figure 5, respectively.

From Figure 5b, we can see that for compression rate d = 0.5, the training loss error between baseline and Radius oscillates, and as the training process continues, the error range and the expected error become smaller. When d = 0.1, the training error is significant, especially after the first 100,000 iterations of the switch. By comparing those five Radius schemes, we can see that a higher compression rate leads to higher training loss error. A similar trend can be observed from the validation PPL curves in Figure 5c.

The oscillation on the training loss error and the degradation on the final validation PPL are caused by the aforementioned error feedback mechanism. When $t \% T \neq 0$, the error between the sparsified gradient and the original gradient is accumulated into a residual buffer, and when $t \% T$ = 0, the accumulated errors are added to the gradient and then all workers in a DP group exchange their gradients. Because of the non-linear nature of operations in AdamW optimizer, unlike the case in DGC (Lin et al., 2018), which uses the SGD optimizer, accumulating gradients (no matter whether they are corrected or not) cannot recover the original optimizer state. Although adding the errors back to the gradient causes the optimization direction to change, if we consider the training process as walking through the error space in high dimension, it maintains approximately the same optimization direction as if the dense $allreduce$ is utilized because approximately half of the gradient information is maintained for every training step. However, for very high compression rates, such as 10%, the sparsified gradient cannot approximate the correct descent direction and thus cannot maintain the final validation PPL close to the baseline. This can be observed from the plots for Radius with d = 0.1 and T = 200 in Figure 5.



(a) Training loss curve (b) Training loss error (baseline − Radius) (c) Validation PPL curve

Figure 4. Curves of pre-training GPT-355M with baseline (dense $allreduce$) and Radius

Table 4. Evaluation results on zero-shot downstream tasks, including LAMBADA, RACE, MathQA, PIQA, and WinoGrande for the pre-trained GPT-355M and GPT-2.0B models

		LAMBADA OpenAI (ACC / PPL)	LAMBADA Standard (ACC / PPL)	RACE (ACC)	MathQA (ACC)	PIQA (ACC)	WinoGrande (ACC)
GPT-355M	Baseline	**43.12 / 16.00**	32.54 / 44.22	31.01	**23.05**	65.18	**52.41**
	d = 0.5, T = 200	42.73 / 16.24	**32.72 / 44.18**	**31.20**	22.58	**65.34**	51.62
GPT-2.0B	Baseline	50.15 / 9.73	41.49 / 19.17	33.01	22.98	70.02	**57.77**
	d = 0.9, T = 200	50.44 / 9.73	41.80 / 18.91	32.82	**23.35**	69.70	57.62
	d = 0.5, T = 200	**50.86 / 9.49**	**42.05 / 18.65**	32.54	23.18	**70.46**	57.62
	d = 0.4, T = 200	50.57 / 9.52	41.68 / 18.97	32.82	23.05	70.18	56.83
	d = 0.4, T = 400	50.26 / 9.52	41.22 / 19.53	33.01	22.88	69.91	57.14
	d = 0.1, T = 200	48.71 / 10.63	40.00 / 22.53	**33.30**	22.71	69.21	55.41

6.4 Evaluation on downstream tasks

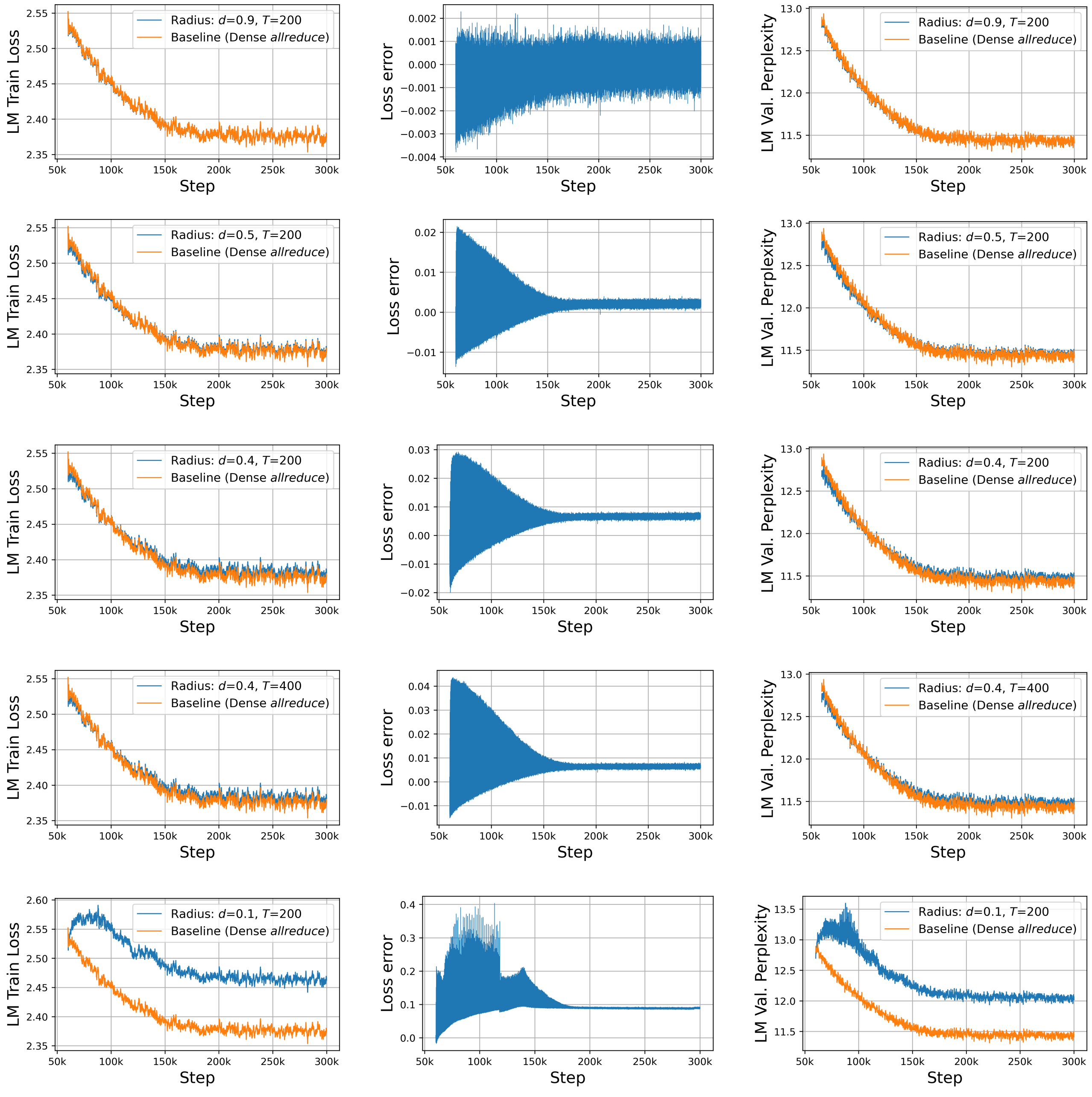
We evaluated the pre-trained GPT-355M and GPT-2.0B models without fine-tuning on several zero-shot downstream tasks using the lm-evaluation-harness framework (Gao et al., 2024), including LAMBADA (Paperno et al., 2016), GLUE (Wang et al., 2018), SuperGLUE (Wang et al., 2019), RACE (Lai et al., 2017), MathQA (Amini et al., 2019), PIQA (Bisk et al., 2020), and WinoGrande (Sakaguchi et al., 2019). The evaluation scores for GPT-355M model and GPT-2.0B model trained with different schemes are shown in Table 4, Table 5, and Table 6.

We report both the scores of LAMBADA OpenAI, the LAMBADA dataset with plain text, and LAMBADA Standard, the LAMBADA dataset with post-processing. The LAMBADA Standard splits contractions like "don't" into "do n't". These post-processed texts cannot be properly handled by GPT2 tokenizer. Thus, models' evaluation scores for LAMBADA Standard are lower than those for LAMBADA OpenAI. Since we used OpenWebText dataset from Hugging Face instead of following instructions provided by Megatron-LM to build WebText dataset, many evaluation scores of our baseline models are lower than those reported by Megatron-LM, especially the LAMBADA scores.

The evaluation scores generally align well with the training and validation accuracy and PPL in Table 2. We can also see that compared to the baseline results, Radius with $d \geq 0.4$ can achieve higher evaluation scores in many GLUE and SuperGLUE downstream tasks. For some specific tasks, such as SST-2 and BoolQ, the accuracy degradation is noticeable. The evaluation scores in Table 4, Table 5, and Table 6 are not the average scores and contain variance. Due to the constrained computation budget, we could only perform a one-time pre-training with Radius using the same random seed as the baseline.

6.5 Comparison between naive top-k and Radius

To highlight the convergence property of Radius, we evaluated Radius and naive top-k approach by pre-training GPT-355M on the Vista supercomputer, an HPC system composed of NVIDIA GH200-96GB superchips connected by NVIDIA Quantum-2 MQM9790 NDR switch having 64 ports of 400 Gb/s InfiniBand per port, and provide their training loss curves and validation PPL curves in Figure 6. Here, the naive top-k refers to the top-k sparsification method that first re-computes the top-k gradients in every step and then uses $allgather$ to exchange top-k values and indices between all workers. It also uses error feedback to store each worker's non-top-k values locally and then adds them to the gradient in the next step before the top-k



(a) Training loss curves (b) Training loss error curves (c) Validation PPL curves

Figure 5. Curves of pre-training GPT-2.0B with baseline (dense $allreduce$) and Radius

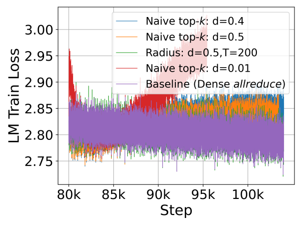
Table 5. Evaluation results on GLUE downstream tasks for pre-trained GPT-355M and GPT-2.0B models

		GLUE					
		MNLI (M / MM)	QQP (ACC / F1)	QNLI (ACC)	SST-2 (ACC)	MRPC (ACC / F1)	RTE (ACC)
GPT-355M	BASELINE	34.80 / 34.94	**36.82 / 53.80**	**50.41**	**58.03**	**68.38 / 81.22**	53.43
	$d = 0.5$, $T = 200$	**35.11 / 35.62**	36.81 / **53.80**	49.79	52.64	**68.38 / 81.22**	**54.51**
GPT-2.0B	BASELINE	34.36 / 34.76	37.28 / 50.75	**51.20**	**51.03**	59.80 / 72.94	51.99
	$d = 0.9$, $T = 200$	34.95 / **35.45**	36.11 / 51.47	50.63	50.00	59.56 / 72.82	53.43
	$d = 0.5$, $T = 200$	34.48 / 35.12	36.74 / **53.50**	50.32	49.31	66.42 / 79.52	50.90
	$d = 0.4$, $T = 200$	35.24 / 35.33	36.71 / 52.41	50.49	49.77	61.27 / 74.60	49.46
	$d = 0.4$, $T = 400$	**35.31** / 35.12	**37.34** / 52.27	50.74	49.31	56.86 / 70.47	53.07
	$d = 0.1$, $T = 200$	34.90 / 34.53	36.93 / 52.06	50.01	49.89	**67.65 / 80.47**	**56.32**

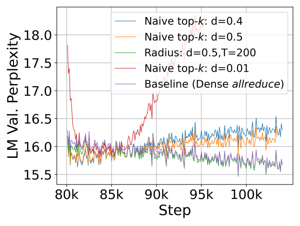
Table 6. Evaluation results on SuperGLUE downstream tasks for the pre-trained GPT-355M and GPT-2.0B models

		SUPER GLUE							
		BOOLQ (ACC)	CB (ACC / F1)	COPA (ACC)	MULTIRC (ACC)	RECORD (F1 / EM)	RTE (ACC)	WIC (ACC)	WSC (ACC)
GPT-355M	BASELINE	57.31	**28.57 / 15.69**	**71.00**	54.48	**80.43 / 79.64**	53.43	50.00	**40.39**
	$d = 0.5$, $T = 200$	**58.07**	26.79 / 18.28	69.00	**54.87**	80.23 / 79.44	**54.51**	**50.16**	39.42
GPT-2.0B	BASELINE	59.39	33.93 / **31.05**	77.00	48.80	84.7 / 83.94	51.99	50.16	38.46
	$d = 0.9$, $T = 200$	58.41	39.29 / 28.75	77.00	50.33	84.01 / 84.78	53.79	49.84	36.54
	$d = 0.5$, $T = 200$	57.77	30.36 / 28.88	76.00	48.82	**85.27** / 84.46	50.54	**50.78**	**42.31**
	$d = 0.4$, $T = 200$	58.62	30.36 / 27.90	76.00	48.37	85.26 / **84.50**	49.46	49.84	38.46
	$d = 0.4$, $T = 400$	58.93	**37.50** / 29.84	**78.00**	51.46	85.02 / 84.21	53.07	49.69	38.46
	$d = 0.1$, $T = 200$	**60.03**	21.43 / 19.52	76.00	**52.95**	84.62 / 83.86	**56.32**	50.16	36.54

selection operation.



(a) Training loss errors



(b) Validation PPL errors

Figure 6. Comparison between using naive top-k and Radius for pre-training GPT-355M. After switching from the baseline method (dense $allreduce$), naive top-k quickly exhibits the trend of divergence, whereas Radius can follow the baseline method and continue reducing the training loss and the validation PPL.

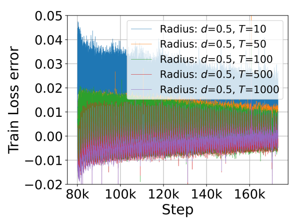
As mentioned above, the low scalability of naive top-k limits its applicability in LLM pre-training. Because of $allgather$, naive top-k with $d = 0.5$ causes more than 10× slowdown on the communication process for aggregating gradients between all workers in a group of DP = 8. When testing naive top-k with $d = 0.5$ and $d = 0.4$, we could only manage to use DP = 8 to pre-train GPT-355M, due to its extremely high requirement on HBM size.

As we can see from both Figure 6a and Figure 6b, after switching from the baseline method (dense $allreduce$) to the naive top-k method at step 80,000, they fail to follow the convergence trend. Specifically, the naive top-k method with $d = 0.01$ rapidly diverges in the following 10,000 steps. Though naive top-k with $d = 0.4$ and $d = 0.5$ seem to keep making progress at step 90,000, as we continue the pre-training, the training loss and validation PPL gradually stop reducing, and both curves eventually flatten.

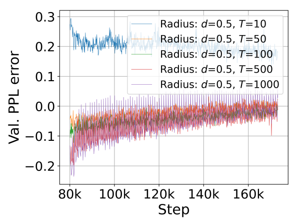
6.6 Exploration on top-k resampling interval T

In addition to the compression rate d, Radius has a unique hyper-parameter: top-k resampling interval T. In Figure 7, we fix the sparsity to 50% and plot the training loss error curves for Radius with different T's.

One counterintuitive but interesting phenomenon is that when T = 10, the training loss error is larger than that of those with larger T's. Our explanation is that when T is too small, such as T = 10, the magnitudes of the non-top-k gradient values accumulated through T training steps



(a) Training loss errors



(b) Validation PPL errors

Figure 7. Comparison between using different top-k indices resampling intervals T's with fixed compression rate d = 0.5 to pre-train GPT-355M

are still much smaller than those top-k values. Therefore, in the following T training steps, the roughly same top-k indices are used again. As this process continues, the effect of completely losing those small gradient values starts to reflect on the training loss and validation PPL curves. When T is large enough, those small gradient values accumulated through T steps may have magnitudes large enough to be selected as the top-k values. Thus, in the next T steps, the corresponding model parameters of these small gradients can be updated, and the staleness effect can be alleviated. Note that this does not suggest that T can be randomly large. When T = 1000, we can see that the validation PPL error curve oscillates vigorously. Through extensive experiments, we found that T = 200 performs well for GPT models with different numbers of parameters and can be used as the default value.

6.7 Strong scaling efficiency

We analyzed Radius's strong scaling efficiency ranging from 8 to 128 GPUs. We set PP and TP to 1, and thus each GPU represents one worker in the DP group. The results are shown in Table 7. Compared to the baseline, Radius achieved much better scaling efficiency. When 128 GPUs are used (i.e., DP = 128), Radius with d = 0.4 and d = 0.1 can sustain the scaling efficiency at 42.5% and 55.5%, respectively, whereas the baseline can only achieve 32.8%.

Table 7. Strong scaling results for Radius.

Number of GPUs	8	16	32	64	128
Baseline	1	0.896	0.715	0.522	0.328
d = 0.9, T = 200	1	0.862	0.688	0.478	0.304
d = 0.5, T = 200	1	0.921	0.779	0.598	0.398
d = 0.4, T = 200	1	0.929	0.785	0.632	0.425
d = 0.1, T = 200	1	0.959	0.869	0.738	0.555

7 Conclusion

In this work, we proposed Radius, a top-k-based gradient compression method for reducing the communication volume in training Large FMs, without causing downstream task performance degradation. Radius leverages the temporal stability of gradient values in training LLMs and solely uses $allreduce$ communication for higher scalability of the top-k sparsification methods. Our experiments show that with compression density d = 0.4, Radius reduces the GPT-2.0B pretraining time by 19% without loss in downstream task performance. With a more aggressive compression density d = 0.1, Radius achieves a speedup of 35% in time. Although the downstream task performance is comparable to that of higher compression densities, we observe instability in the training loss. In future work, we plan to apply Radius to 3D parallelism FM training and reduce communication overhead in pipeline and tensor parallelism.

8 Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award DDR-ERCAP0032037. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing computational resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu. Mingkai Zheng and Zhao Zhang are partially supported by the NSF ICICLE AI Institute (OAC-2112606) and OAC-2340011.

References

Ahdritz, G., Bouatta, N., Floristean, C., Kadyan, S., Xia, Q., Gerecke, W., O'Donnell, T. J., Berenberg, D., Fisk, I., Zanichelli, N., Zhang, B., Nowaczynski, A., Wang, B., Stepniewska-Dziubinska, M. M., Zhang, S., Ojewole, A., Guney, M. E., Biderman, S., Watkins, A. M., Ra, S., Lorenzo, P. R., Nivon, L., Weitzner, B., Ban, Y.-E. A., Sorger, P. K., Mostaque, E., Zhang, Z., Bonneau, R., and AlQuraishi, M. OpenFold: Retraining AlphaFold2 yields new insights into its learning mechanisms and capacity for generalization. *bioRxiv*, 2022. doi: 10.1101/2022.11.20.517210. URL `https://www.biorxiv.org/content/10.1101/2022.11.20.517210`.

Amini, A., Gabriel, S., Lin, P., Koncel-Kedziorski, R., Choi, Y., and Hajishirzi, H. MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms, 2019.

Bernstein, J., Wang, Y.-X., Azizzadenesheli, K., and Anandkumar, A. signSGD: Compressed Optimisation for Non-Convex Problems, 2018. URL https://arxiv.org/abs/1802.04434.

Bian, S., Li, D., Wang, H., Xing, E., and Venkataraman, S. Does Compressing Activations Help Model Parallel Training? In Gibbons, P., Pekhimenko, G., and Sa, C. D. (eds.), *Proceedings of Machine Learning and Systems*, volume 6, pp. 239–252, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/file/71381211d0abef73ed1887b83c4547b1-Paper-Conference.pdf.

Bisk, Y., Zellers, R., Bras, R. L., Gao, J., and Choi, Y. PIQA: Reasoning about Physical Commonsense in Natural Language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

Bran, A. M., Cox, S., Schilter, O., Baldassari, C., White, A. D., and Schwaller, P. ChemCrow: Augmenting large-language models with chemistry tools, 2023. URL https://arxiv.org/abs/2304.05376.

De Sensi, D., Di Girolamo, S., McMahon, K. H., Roweth, D., and Hoefler, T. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2020. doi: 10.1109/SC41405.2020.00039.

Gao, L., Tow, J., Abbasi, B., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., Le Noac'h, A., Li, H., McDonell, K., Muennighoff, N., Ociepa, C., Phang, J., Reynolds, L., Schoelkopf, H., Skowron, A., Sutawika, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. A framework for few-shot language model evaluation, 07 2024. URL https://zenodo.org/records/12608602.

Gokaslan, A. and Cohen, V. OpenWebText Corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. PipeDream: Fast and Efficient Pipeline Parallel DNN Training, 2018.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, 2019. URL https://arxiv.org/abs/1811.06965.

Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Lai, G., Xie, Q., Liu, H., Yang, Y., and Hovy, E. RACE: Large-scale ReAding Comprehension Dataset From Examinations. In Palmer, M., Hwa, R., and Riedel, S. (eds.), *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 785–794, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1082. URL https://aclanthology.org/D17-1082.

Lee, J., Stevens, N., Han, S. C., and Song, M. A Survey of Large Language Models in Finance (FinLLMs), 2024. URL https://arxiv.org/abs/2402.02315.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

Li, S. and Hoefler, T. Near-optimal sparse allreduce for distributed deep learning. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pp. 135–149, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392044. doi: 10.1145/3503221.3508399. URL https://doi.org/10.1145/3503221.3508399.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020. URL https://arxiv.org/abs/2006.15704.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep Gradient Compression: Reducing the communication bandwidth for distributed training. In *The International Conference on Learning Representations*, 2018.

Loshchilov, I. and Hutter, F. Decoupled Weight Decay Regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Microsoft Research, AI4Science, and Microsoft Azure Quantum. The Impact of Large Language Models on Scientific Discovery: a Preliminary Study using GPT-4, 2023. URL https://arxiv.org/abs/2311.07361.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, 2021. URL https://arxiv.org/abs/2104.04473.

NVIDIA. Apex. https://github.com/NVIDIA/apex, 2018.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context, 2016. URL https://arxiv.org/abs/1606.06031.

Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. WinoGrande: An Adversarial Winograd Schema Challenge at Scale, 2019. URL https://arxiv.org/abs/1907.10641.

Shi, S., Wang, Q., Zhao, K., Tang, Z., Wang, Y., Huang, X., and Chu, X. A Distributed Synchronous SGD Algorithm with Global Top-k Sparsification for Low Bandwidth Networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2238–2247, Los Alamitos, CA, USA, jul 2019. IEEE Computer Society. doi: 10.1109/ICDCS.2019.00220. URL https://doi.ieeecomputersociety.org/10.1109/ICDCS.2019.00220.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020.

Song, J., Yim, J., Jung, J., Jang, H., Kim, H.-J., Kim, Y., and Lee, J. Optimus-CC: Efficient Large NLP Model Training with 3D Parallelism Aware Communication Compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pp. 560–573, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575712. URL https://doi.org/10.1145/3575693.3575712.

Tang, H., Gan, S., Awan, A. A., Rajbhandari, S., Li, C., Lian, X., Liu, J., Zhang, C., and He, Y. 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed, 2021. URL https://arxiv.org/abs/2102.02888.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. LLaMA: Open and Efficient Foundation Language Models, 2023. URL https://arxiv.org/abs/2302.13971.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need, 2023. URL https://arxiv.org/abs/1706.03762.

Vogels, T., Karimireddy, S. P., and Jaggi, M. PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization, 2020. URL https://arxiv.org/abs/1905.13727.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pp. 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-5446. URL https://aclanthology.org/W18-5446.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/4496bf24afe7fab6f046bf4923da8de6-Paper.pdf.

Wang, L., Wu, W., Zhang, J., Liu, H., Bosilca, G., Herlihy, M., and Fonseca, R. SuperNeurons: FFT-based Gradient Sparsification in the Distributed Training of Deep Neural Networks, 2021. URL https://arxiv.org/abs/1811.08596.

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning, 2017. URL https://arxiv.org/abs/1705.07878.

Zhao, R., Morwani, D., Brandfonbrener, D., Vyas, N., and Kakade, S. Deconstructing What Makes a Good Optimizer for Language Models, 2024. URL https://arxiv.org/abs/2407.07972.