ZeRO: Zero Redundancy Optimizer

Charlie Magri *, Ricardo Quadras *, and Sinead Dubois *
* University of Guelph, Guelph, Ontario, Canada.

***Abstract*—The paper "ZeRO: Memory Optimizations Toward Training Trillion Parameter Model" goes over the need for more efficient GPU training algorithms. The algorithm developed, ZeRO showed faster throughput results and more efficient memory allocation with their improved optimizer states and gradient handling. This will be tested and replicated on A100 chips to observe the effects on training. Results showed that during linear training ZeRO still produces faster training times and more precise models. These results are in line with the papers that show amplified results when using increasing cluster sizes. [1]**

I. Introduction

As models trend towards billions to even trillion parameters in size, it becomes difficult to train models of very large parameter sizes while retaining optimal communication, computation and training speeds across hundreds to thousands of training devices. To combat this, researchers at Microsoft who were passionate about optimizations analyzed the full spectrum of memory consumed by models in training to develop "ZeRO". ZeRO is a training method that can be used to optimize the memory and communication across devices used in training in order to improve training speed, reduce memory requirements and scale models to hardware availability [1]. This allows AI models to be trained with high computational granularity while retaining low communication volumes, without requiring communication heavy model parallelism.

This paper outlines the steps taken to replicate the results behind ZeRO on a smaller scale that is appropriate for the level of hardware available, as the ZeRO research team tested their results from 64 to 400 GPUs to demonstrate linear scalability. Comparably, the results of this paper will be replicated on a single A100 to demonstrate the impact of this ZeRO's novel form of data-parallelism and residual memory reduction on smaller sized models with which training optimizations can have considerable impacts on smaller companies. This paper also dives into the research and usage following ZeRO, and how this work has changed the way researchers shard the memory use in training workloads.

Finally, this paper presents challenges, assumptions and difficulties encountered with replicating this research as well as workable code that can be used to replicate our findings. This can be paired with the readme file attached, to replicate our research. This research is important to improving the accessibility of training larger models, as well as reducing the barrier to entry for the field of AI.

II. Related Work

A. DeepSpeed Library for PyTorch

Formed in 2018, Microsoft's AI at Scale initiative has focused on improving efficiency and scalability of deep learning models. A major contribution towards their goal, the DeepSpeed library was released in 2020, providing easy access to a variety of open-source tools for optimizing various types of deep learning models. Most notably, ZeRO is one of the tools included in the library, enabling simplified implementation through PyTorch. [2]

B. Mesh-TensorFlow

As a precursor to the development of ZeRO, Mesh-TensorFlow (M-TF) was developed to simplify the implementation of simultaneous data and model parallelism. Acknowledging the limitations of data-parallelism and the complexity of model-parallelism, the authors introduce their solution; a language which divides models into multidimensional tensors. Data parallelism for a Single Program Multiple Data (SPMD) system forms a 1-dimensional tensor array, dividing batches across a processor. Model parallelism is performed by creating a multi-dimensional set of tensors (data subsets) and a similar multi-dimensional mesh representation of the target hardware. The final 'computation layout' is created by mapping portions of the tensor-dimensions onto the mesh-dimensions. The main limitation of this development appears to be the manually configured nature of its mapping, resulting in potentially sub-optimal configurations, and depending on the user to prevent redundancy in datasets. [3]

C. Megatron-LM

With a focus on leveraging and modifying existing transformer architectures in PyTorch, Megatron-LM implements distributed tensor computation (similar to that of M-TF) without the need for standalone frameworks or compilers. Megatron-LM introduces model parallelism to the transformer networks by adding synchronization primitives to their components. A transformer layer is comprised of two blocks performing separate algorithms – a self attention block and a two-layer, multi-layer perceptron (MLP). An MLP consists of a general matrix to matrix multiplication (GEMM) followed by a gaussian error linear unit (GeLU) activation, then followed by another GEMM. To introduce parallelism while minimizing overhead, the first GEMM is split along the second matrix's columns (reducing complexity from GeLU's inherent non-linearity) while the second GEMM is split along the second matrix's rows. These splits functionally occur by moving the calculation of those portions of the matrix multiplications to separate GPUs. The self attention block receives a similar division of its matrix multiplications, though with substantially less complexity due to the lack of non-linear functions [4]. Ultimately, this technique is easy to implement in PyTorch and does provide notable improvements

to accuracy and throughput, though its scalability gains fail to reach the performance of ZeRO on exceptionally large models (ie. Turing-NLG[approximately 17 billion parameters]). [1]

D. Striped Attention

Following ZeRO's development and its proven efficacy for sharding of entire models, innovators have begun to focus on finer elements within models to further optimize the distribution of workloads. Striped Attention (and its predecessor Ring Attention) enable the distribution and scheduling of self attention computations across multiple accelerators in a ring topology to drastically increase the size of transformer networks – computation structures which are heavily leveraged in the field of generative language models [5]. Following the discovery of workload imbalances within its predecessor, Striped Attention was developed to further optimize multi-device attention computations. Striped Attention works by reordering the permutations received by each device, such that the attention computations are performed cascading across devices, rather than merely as subsets of the original sequence. This workload distribution is optimized similarly to the coarser partitioning utilized in ZeRO, and provides a substantial speedup (up to 1.65x) compared to its predecessor. [6]

III. Contribution

ZeRO has two main sets of optimizations employed to achieve efficiencies across devices. ZeRO-DP which targets the largest consumer of memory while training, the sharing of "model states" across devices, and ZeRO-R which targets the residual memory consumption of models in training.

With ZeRO-DP, there exists 3 stages of optimizations that can be employed to progressively eliminate memory redundancy in data-parallel training of large datasets. Stage 1 focuses on partitioning optimizer states instead of replicating them across GPUs, reducing memory by O(N) where N is the number of data-parallel processes. Stage 2 extends stage 1 by also partitioning gradients to eliminate gradient redundancy. Stage 3, which is the most aggressive, partitions the model parameters themselves such that these parameters are sharded across GPUs with communication used to gather parameters when needed. This does have a noticeable communication overhead but is compensated by the highest memory efficiency. Before these stages, optimizer states, gradients and model parameters were replicated instead of partitioned, heavily intensifying redundancies at high levels of GPU resource sharing.

With these improvements to model training and memory sharing, researchers can reliably train models up to 13B parameters without employing model parallelism or pipeline parallelism, allowing more freedom to train and use larger models.

Following on this trend, while the main paper talks about how the data parallelism employed by ZeRO-DP retains a higher scaling efficiency than model parallelism at larger numbers of GPUs, especially across node clusters, our research aims to find the results of these optimizations across a single high-end GPU.

To showcase this, we employ ZeRO-Stage 3 training techniques on a single high-end GPU, with an IMDB dataset set to an 80-20 train-test split to perform binary sentiment analysis with the help of the 'bert-large-uncased' model (336M parameters) to identify training and memory optimizations on a smaller scale than that of the original research paper. This follows in line with the paper's original ideology of reducing the training costs and allowing "democratization of large model training" but from a much smaller scale.

IV. Methodology

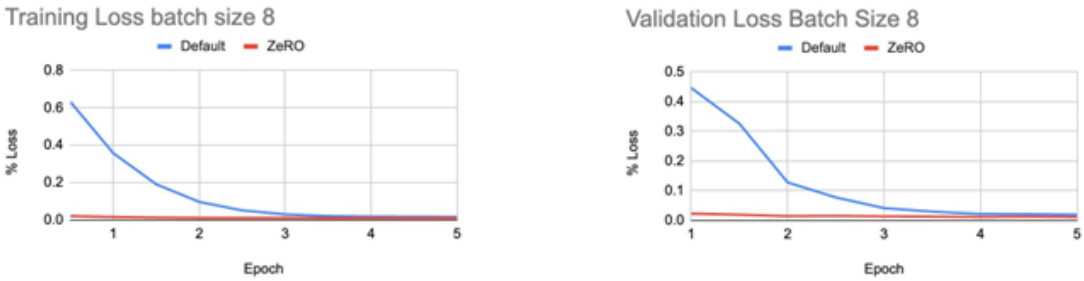
In the base paper training was observed on 64, 128, 256, and 400 GPU clusters. This allowed for extreme levels of parallelism, increasing maximum model size and greatly reducing training time. In the recreation, due to financial and physical limitations, training on GPU clusters was not an option, meaning speedups would need to be observed on a single GPU. Training was decided to be run virtually to allow for state-of-the-art A100 chips to be used. This will show results on a GPU designed for AI training compared with otherwise local training that would run-on general-purpose processors.

Now that a platform had been decided, A program for ZeRO had to be designed to show metrics of training. Python was the language of choice as it has a high variety of support for AI training and thus higher model diversity. Since publication of the paper, ZeRO has been implemented directly within the DeepSpeed library. This allowed for easy integration into a training script. The script uses PyTorch, a common training library to simplify the training process. Additionally, it was critical that the seed used would remain the same across tests to ensure fair comparison between training methods.

During the design process many different levels of epochs, model sizes and batch sizes were chosen to see the effects on the results. Different data sets had to be used as models were scaled. Wikitext was used as a simple dataset to train the model for decoder models and an IMDB dataset was used for encoder models. Additionally, models were only trained on portions of the dataset to allow for quick training, reducing hours of training time to minutes. The metrics recorded were "Training loss, validation loss, average and peak GPU usage, and training time". These will be compared with the base papers results in the next section.

V. Results

The paper showed that ZeRO outperformed any current method of training and that it improved throughput and model capacity. With the tests, we've shown it increases throughput, validation and training losses. Average GPU memory usage was inconsistent, due to the single GPU training not providing as much memory reduction when on a single device. However, during tests the program would produce OOP errors when attempting to run heavy models on higher batch sizes without using ZeRO. ZeROs more effective gradient and memory management showed that the overhead was needed when the GPU was heavily strained. The first result to show is the



(a) Training Loss **(b)** Validation Loss

Fig. 1: Training and Validation loss

training and validation losses with and without ZeRO. Fig. 1 shows the respective graphs.

These graphs show the immediate effects of using ZeRO, showing they have great losses in both regards. ZeRO shows that it trains on the data and retains the data much better than without using ZeRO. This is mostly due to the more effective gradient techniques it uses that also reduces noise. To add, the final value at epoch 5 was 1.22% faster for validation loss and 1.18% faster for training loss showing that even after converging there remains a notable difference.

Speed difference showed to be apparent while using all model sizes and different epoch or batch sizes. Below shows the difference between using ZeRO and Default settings as model size increases

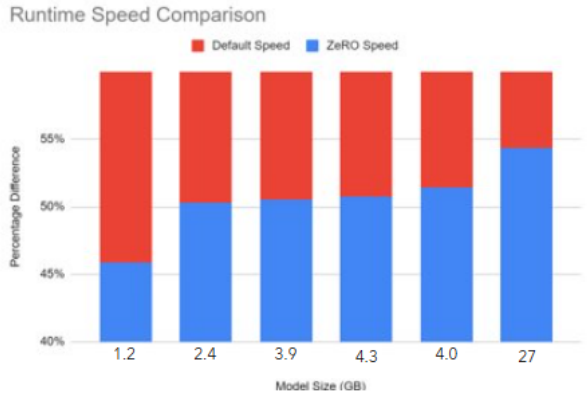


Fig. 2: Runtime Speed comparison of models without optimization and models optimized with ZeRO

This shows that as the model size increases, so does the speed-up provided by ZeRO. Notice that the first small-scale model has a negative speedup of almost 5%. This is due to ZeROs overhead costing more in setup time than it provides speed up. This is no longer the case when crossing the 2GB barrier and is always increasing. This shows that under increased GPU strain, ZeRO produces better results. This is similar to the paper finding; however, they noticed a retraction of the increase after reaching a certain barrier.

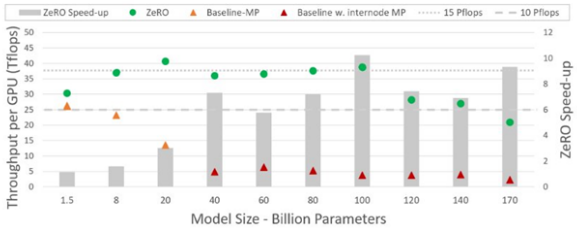


Fig. 3: Throughput and Speedup relative to previous State of The Art (SOTA) baseline

This shows an increasing trend until 20 billion parameters, stagnant results from 20 to 100 billion and then clear decreasing trend until concluding at 170 billion. This trend could not be seen in our results as the largest model trained was the GPT 6B parameter model which still falls in the increasing performance category.

VI. Conclusion

This study showcases that the benefits of ZeRO extend beyond massive-scale GPU clusters originally tested in Microsoft's research, showing significant improvements even in single-GPU environments that are more accessible to smaller to medium-sized organizations. While the original paper focuses on the scalability of training trillion-parameter models across hundreds of GPUs, this paper showcases that ZeRO's memory optimizations and improved training efficiency is valuable regardless of computing scale.

The successful replication of ZeRO's advantages onto a single A100 GPU (hosted on Google Cloud) has important implications in the democratization of large model training. Smaller to medium sized companies, which typically cannot afford large GPU clusters are able to benefit from the training memory and optimization techniques, allowing faster training throughput and model capacity even in a scaled-down environment. This showcases that ZeRO's architecture is flexible enough to benefit different scales of operation.

Practically, this means that one can employ ZeRO even with limited hardware resources to reduce training costs in terms of computational costs and energy consumption. This paired with the accessibility of DeepSpeed's library allows straightforward implementation for varying levels of technical expertise, reducing hardware barriers to entry. Even though

ZeRO was designed for very large models, its benefits are valuable to all forms of AI development, from individual researchers to large corporations.

REFERENCES

[1] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," 2020. [Online]. Available: https://arxiv.org/abs/1910.02054

[2] Microsoft, "Deepspeed: Accelerating large-scale model training," https://www.deepspeed.ai, 2020, accessed: 2025-04-12.

[3] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-tensorflow: Deep learning for supercomputers," 2018. [Online]. Available: https://arxiv.org/abs/1811.02084

[4] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020. [Online]. Available: https://arxiv.org/abs/1909.08053

[5] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," 2023. [Online]. Available: https://arxiv.org/abs/2310.01889

[6] W. Brandon, A. Nrusimha, K. Qian, Z. Ankner, T. Jin, Z. Song, and J. Ragan-Kelley, "Striped attention: Faster ring attention for causal transformers," 2023. [Online]. Available: https://arxiv.org/abs/2311.09431

APPENDIX A
PYTHON CODE USED TO IMPLEMENT ZERO

```
!pip install --upgrade transformers datasets accelerate deepspeed huggingface_hub

import torch
import time
import importlib.metadata
from transformers import AutoModelForSequenceClassification, AutoTokenizer, Trainer, TrainingArguments
from datasets import load_dataset, DatasetDict


# Verifing package versions
print("huggingface_hub version:", importlib.metadata.version("huggingface_hub"))
print("transformers version:", importlib.metadata.version("transformers"))


# Load dataset subject to change if switching model
full_dataset = load_dataset("imdb")
train_test_split = full_dataset["train"].train_test_split(test_size=0.2, seed=42)
dataset = DatasetDict({"train": train_test_split["train"], "test": train_test_split["test"]})


# Load model and tokenizer
model_name = "bert-large-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)


# The tokenization function will change for different models
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True, max_length=128)


# Tokenize datasets
tokenized_datasets = dataset.map(tokenize_function, batched=True)
train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(800))
eval_dataset = tokenized_datasets["test"].shuffle(seed=42).select(range(200))

# For quantifying output
def get_peak_memory():
    torch.cuda.synchronize()
    return torch.cuda.max_memory_allocated() / 1024**2  # Convert to MB

#can change epochs steps or any part of process
#to fit your training tests
def train_model(with_zero=False, batch_size=8):
    torch.cuda.empty_cache()

    training_args = TrainingArguments(
        output_dir=f"./results_bs{batch_size}_{'zero' if with_zero else 'std'}",
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        num_train_epochs=1,
        eval_strategy="steps",  # Fixed deprecation warning
        logging_strategy="steps",
        logging_steps=10,
        eval_steps=50,
        save_total_limit=1,
        report_to="none"
    )
```

```
    if with_zero:
        training_args.deepspeed = {
            "zero_optimization": {
                "stage": 3,
                "offload_optimizer": {"device": "cpu"}
            }
        }


    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
    )

    print(f"\n=== Training {'w/ZeRO' if with_zero else 'standard'} BS={batch_size} ===")
    start_time = time.time()
    peak_mem_before = get_peak_memory()

    trainer.train()

    training_time = time.time() - start_time
    peak_mem_after = get_peak_memory()

    print(f"Peak GPU Memory: {peak_mem_after:.2f} MB")
    print(f"Training Time: {training_time:.2f}s")
    return training_time, peak_mem_after

# Run benchmarks, or add more of you own
print("\n=== Starting Tests ===")
std_time, std_mem = train_model(with_zero=False, batch_size=8)
zero_time, zero_mem = train_model(with_zero=True, batch_size=8)
zero_large_time, zero_large_mem = train_model(with_zero=True, batch_size=16)

# Results uncommented
print("\n=== Final Results ===")
print(f"| Configuration   | Time (s) | Memory (MB) |")
print(f"|-----------------|----------|-------------|")
print(f"| Standard BS=8   | {std_time:.1f}     | {std_mem:.2f}       |")
#print(f"| ZeRO Stage2 BS=8| {zero_time:.1f}     | {zero_mem:.2f}     |")
print(f"| ZeRO Stage2 BS=16| {zero_large_time:.1f}|{zero_large_mem:.2f}       |")
```