Benchmarking Distributed Training Framework for Large Scale Models

Dev Mehrotra, Sai Krishna Ponnam, Rachael Lang

08 May 2025

1 Introduction

Deep neural networks (DNNs) and large language models (LLMs) have achieved remarkable success across a wide range of applications, driving a push toward increasingly larger models trained on vast datasets. Training these models demands immense computational resources, often requiring several ExaFLOPs of processing power and the handling of petabytes of data. However, as model sizes and complexities continue to grow, these models face significant computational and memory bottlenecks that challenge both efficient training and practical deployment. Traditional single-node training quickly becomes impractical due to memory limitations and excessive training durations. Consequently, ensuring scalable and efficient training has emerged as a critical priority, necessitating the development of advanced parallelization strategies.

Recent technological advancements have markedly advanced efforts to address the inherent challenges of large-scale model training, primarily through the development of diverse parallelization strategies [6]. Distributed training frameworks have emerged as indispensable solutions, facilitating scalable and efficient computation across GPUs, TPUs, and multi-node environments. Prominent frameworks—including TensorFlow, PyTorch, Horovod, DeepSpeed, and Ray—are purpose-built to accelerate training processes by distributing computational loads across multiple hardware units. These systems implement a spectrum of parallelization techniques, such as data parallelism, model parallelism, pipeline parallelism, and hybrid strategies, each tailored to optimize performance under varying computational demands. While data parallelism is widely adopted due to its simplicity and effectiveness, it is often hindered by communication overhead during gradient synchronization. In contrast, model parallelism enables the deployment of expansive models across several devices but introduces complexities in maintaining balanced and efficient workload distribution.

Understanding the trade-offs inherent in various distributed training approaches and frameworks is essential for optimizing performance when training large-scale models. Identifying system bottlenecks enables the mitigation of inefficiencies in distributed training, particularly in the context of training cutting-edge architectures such as Vision Transformers (ViTs) on benchmark datasets like CIFAR. By evaluating these strategies through the lens of ViT model training, this project seeks to derive actionable insights that promote the efficient utilization of distributed training frameworks for large model development.

This project undertakes a comparative analysis of prominent distributed training strategies, with a particular emphasis on Fully Sharded Data Parallel (FSDP) and DeepSpeed frameworks. The primary objective is to determine the most effective methodology for training large-scale models like ViT. Central to this investigation is a detailed examination of critical performance metrics — including training duration, memory consumption, throughput, and network utilization — to assess the impact of different parallelization techniques on training efficiency. Through rigorous empirical evaluation, the project endeavors to identify the strategy that offers the most favorable trade-off between computational speed and resource efficiency. Ultimately, this comprehensive analysis aims to elucidate the scalability and performance characteristics of FSDP and DeepSpeed, thereby informing the selection of optimal distributed training paradigms for contemporary deep learning applications.

The structure of this project report is organized as follows: The Literature Review section examines a range of techniques employed in training large-scale models, with a focus on their underlying architectures and the algorithms used to implement them. This is followed by the Methodology section, which outlines the framework designed for evaluating these training techniques. The Results section presents a comparative analysis based on the evaluation metrics derived from the methodology. This is followed by conclusion and future scope.

2 Background

In the ongoing effort to push the boundaries of artificial intelligence through large scale models like LLMs and ViTs, the research community has increasingly concentrated on devising and optimizing parallelization strategies to address the substantial computational demands associated with training such large-scale models. Among the most promising solutions are FSDP and DeepSpeed, both of which offer innovative approaches to model training. These frameworks employ parameter sharding techniques that distribute model parameters across multiple GPUs, thereby substantially reducing memory usage on individual devices. This reduction not only facilitates more efficient scaling of training workloads but also enhances the feasibility of deploying and executing large models within distributed computing environments.

2.1 Non-Distributed Training

Single-GPU training is distinguished by its straightforward implementation, wherein both the model and dataset are entirely allocated to a single GPU. This approach eliminates the need for complex inter-device communication and synchronization mechanisms inherent to distributed training paradigms. Within this setup, all computational processes—including both the forward and backward propagation—are confined to a single GPU, which simplifies the training workflow. However, this simplicity comes with inherent limitations, particularly in terms of memory capacity and computational throughput. These constraints can restrict the size of models and batch inputs that can be effectively handled, potentially resulting in extended training durations when compared to distributed alternatives. Nevertheless, for smaller-scale models or scenarios constrained by limited hardware resources, single-GPU training remains a practical and efficient solution for model development and experimentation.

2.2 Distributed Data Parallel

Distributed Data Parallel (DDP) has become a widely adopted technique for accelerating deep learning model training by leveraging parallelism across multiple GPUs, either within a single node or distributed across multiple nodes in a high-performance computing environment. In this approach, each GPU maintains an identical replica of the model. The training dataset is partitioned into mini-batches, which are then distributed across the available GPUs for concurrent processing.

During the forward pass, each GPU independently computes the activations and corresponding loss for its assigned mini-batch. These locally computed losses are then aggregated across all GPUs to derive a global loss, which serves as the basis for gradient computation in the backward pass. DDP ensures synchronization of gradients across all model replicas using collective communication operations such as AllReduce. Subsequently, model parameters are updated in a consistent and synchronized manner. This process is effectively implemented in frameworks such as PyTorch, which encapsulates the DDP mechanism through a well-structured pseudo-algorithm. [5]

However, DDP may encounter significant challenges when applied to large-scale models or extensive datasets. A primary limitation stems from the requirement that each GPU must maintain a complete replica of the model, which can impose substantial memory overhead. Furthermore, the performance of DDP is susceptible to degradation due to communication bandwidth limitations and network latency, particularly in distributed computing environments. It is also noteworthy that DDP does not natively support advanced memory optimization techniques such as parameter sharding or activation checkpointing, which constrains its scalability and efficiency when dealing with extremely deep or memory-intensive neural architectures.

2.3 Fully Sharded Data Parallel

Fully Sharded Data Parallel (FSDP) has been developed as an advanced parallel training strategy to mitigate memory constraints and enhance the scalability of deep learning models across multiple GPUs. Traditional parallelization techniques, such as DDP, often encounter significant limitations when applied to large-scale models and datasets, primarily due to the restricted memory capacity of individual GPUs. FSDP addresses these challenges by partitioning model parameters, gradients, and optimizer states across all available GPUs. This sharding mechanism enables a more balanced distribution of both computational workload and memory footprint. The practical implementation of FSDP can be observed in the FSDP class provided by PyTorch [11]. The architectural design of FSDP is illustrated in Figure 1.

Model Initialisation: Training large-scale models typically demands substantial memory resources, even during the model initialization phase. FSDP mitigates this challenge through the use of deferred initialization. In this approach, model parameters are initially allocated on a placeholder or fake device, enabling model construction without occupying actual GPU memory. During this stage, all parameter-related operations are recorded. Once the model is wrapped with FSDP, these recorded operations are replayed incrementally on physical GPUs, allowing for safe and memory-efficient parameter initialization. This mechanism enables FSDP to scale to models that would otherwise be too large to instantiate using conventional PyTorch workflows.

Forward Pass: During the forward pass, FSDP ensures that only a single shard of the model's parameters resides in memory at any given time. Prior to executing each layer, FSDP performs an **AllGather** operation to reconstruct full parameters from distributed shards. Computation proceeds on these unsharded parameters, after which the temporary parameter copies are promptly released. This design bounds memory usage to just one FSDP unit at a time. Furthermore, FSDP supports forward prefetching, which proactively initiates AllGather operations for subsequent layers, thereby overlapping computation and communication to reduce latency.

Backward Pass: The backward pass adheres to a similar strategy, with the addition of gradient reduction. Required parameter shards are again collected via AllGather before gradient computations commence. Following gradient computation, a **ReduceScatter** operation is employed to distribute gradient shards back across devices, ensuring that each GPU retains only its respective partition. To further optimize performance, FSDP incorporates techniques such as backward prefetching, which anticipates upcoming parameter needs, and CUDA stream overlap, which helps to conceal communication delays along the critical path.

FSDP mainly supports 2 types of Sharding Mechanism:

- Full Sharding: Each GPU holds a disjoint shard of parameters with no redundancy, minimizing memory use at the cost of increased communication. However, this comes at the expense of increased communication overhead due to the need for synchronization across devices.

- Hybrid Sharding: This method represents a compromise between full replication and full sharding. It leverages hardware topology—such as differences in intra-node versus inter-node bandwidth—to optimize performance and reduce internode communication costs.

Together, these features allow FSDP to scale large model training effectively and efficiently. By controlling memory usage during initialization and training, overlapping communication and computation, and supporting flexible sharding strategies, FSDP makes large-scale distributed training more accessible and performant.

2.4 DeepSpeed

DeepSpeed is an open-source deep learning optimization library developed by Microsoft, designed to facilitate the training of large-scale models with enhanced efficiency, scalability, and performance. A core innovation in DeepSpeed is the ZeRO (Zero Redundancy Optimizer) [7], which comes in three stages—ZeRO-1, ZeRO-2, and ZeRO-3. ZeRO-1 partitions optimizer states across data parallel processes, ZeRO-2 further partitions gradients, and ZeRO-3 adds partitioning of model parameters themselves. This drastically reduces memory redundancy, achieving up to 10x memory savings compared to traditional data parallel training. ZeRO-Infinity extends this further by integrating CPU and NVMe offloading, enabling training of models with trillions of parameters even on hardware with limited GPU memory. This allows DeepSpeed to support training of models that would otherwise not fit into the memory of a single or even multiple GPUs, thus enabling massive model scalability.

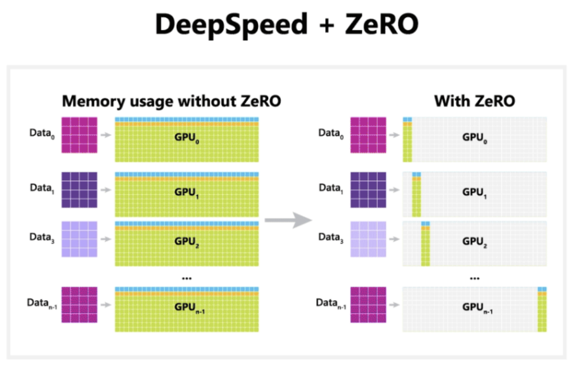


Figure 2: Memory Usage with and without ZeRO Optimization in DeepSpeed

DeepSpeed also supports mixed-precision training via integration native PyTorch AMP (Automatic Mixed Precision), which reduces memory usage and improves throughput by leveraging float16 computations without sacrificing model accuracy. Additionally, it incorporates gradient accumulation, activation checkpointing, and asynchronous I/O operations to further optimize performance and reduce memory load. These features allow users to train large models without significantly increasing the GPU count or requiring specialized hardware. Additionally, it offers support for efficient inference, allowing large models to be deployed with low latency and high throughput. These optimizations make DeepSpeed particularly well-suited for both training and serving large-scale models used in natural language processing and other AI domains.

3 Related Work

Scalability challenges in distributed training have long been a research focus. Keuper and Preundt in their paper [4] provide an in-depth analysis of the limitations of data-parallel stochastic gradient descent (SGD) in distributed

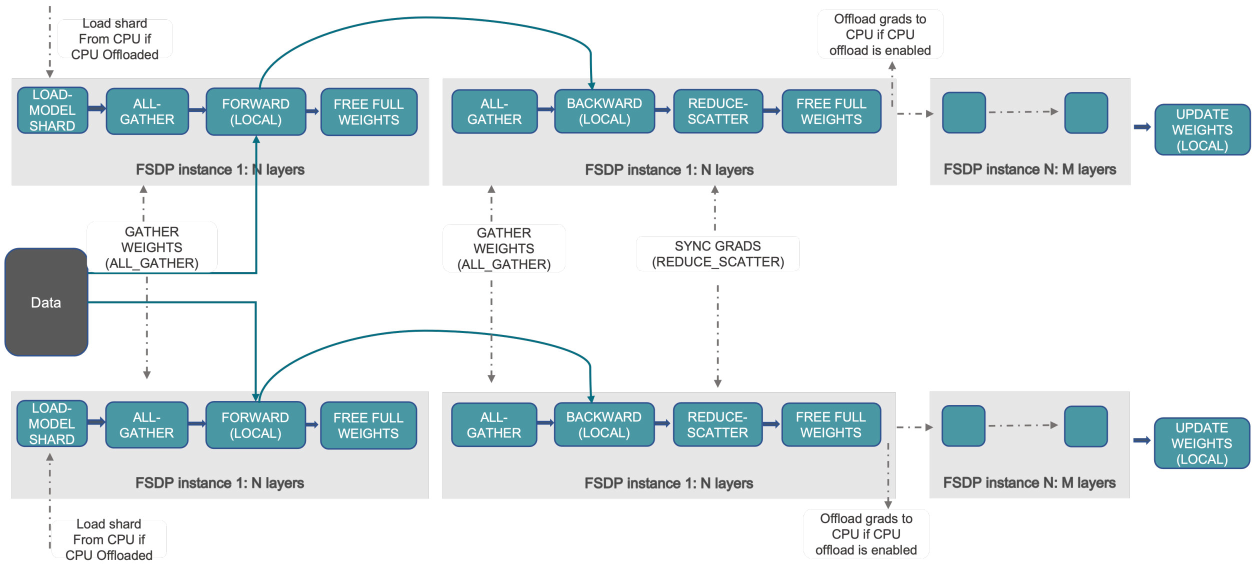

Load shard From CPU if CPU Offloaded
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
Offload grads to CPU if CPU offload is enabled
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)
GATHER WEIGHTS (ALL_GATHER)
GATHER WEIGHTS (ALL_GATHER)
SYNC GRADS (REDUCE_SCATTER)
Data
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)
Load shard From CPU if CPU Offloaded
Offload grads to CPU if CPU offload is enabled

Figure 1: Architecture of PyTorch's FSDP [1]

settings. They demonstrate that beyond a certain number of nodes, training scalability is severely impacted by three specific bottlenecks - the communication overhead, parallelization of matrix operations and training data distribution, rendering the process inefficient. The paper highlights that even with optimizations like communication reduction techniques, distributed training remains bottlenecked by synchronization delays and data movement costs.

Several benchmarking studies have assessed the effects of network bandwidth, gradient synchronization, and checkpointing strategies on distributed training performance. The paper Optimizing Distributed Training of Large Language Models [2] explores the trade-offs between various parallelization approaches, highlighting the significance of overlapping computation and communication to enhance efficiency. The implementation of activation checkpointing, mixed precision training, and tensor offloading has played a key role in reducing memory overhead and boosting throughput.

Distributed training has been an active area of research, with multiple frameworks emerging to address the scalability challenges posed by large-scale deep learning models. Traditional data-parallel approaches, such as Distributed Data Parallel (DDP) in PyTorch and Horovod, synchronize gradients across multiple GPUs to ensure model consistency. However, these methods suffer from high memory overhead due to redundant storage of model parameters on each GPU. Recent advancements, such as Fully Sharded Data Parallel (FSDP) and DeepSpeed's Zero Redundancy Optimizer (ZeRO), aim to combine the benefits of both approaches while minimizing memory and communication overhead. But they remain underexplored in terms of their full configuration potential for training pipelines. Their advanced features (e.g., ZeRO, sharded optimizers) are rarely benchmarked in end-to-end training scenarios, leaving users without clear guidelines for configuring them to maximize performance [9].

4 Methodology

For evaluating our project we plan to do the following: We focus on **PyTorch FSDP** and **Microsoft DeepSpeed**, leveraging their cutting-edge configurations to optimize distributed training for ViT on CIFAR dataset. Our approach tests out multiple different configurations as follows

PyTorch FSDP Configurations

- **Sharding Strategies**:
  - `FULL_SHARD`: Shards model parameters, gradients, and optimizer states across GPUs.
  - `HYBRID_SHARD`: Combines data parallelism with selective sharding for large layers.

Microsoft DeepSpeed Configurations

- **ZeRO Stages**:
  - **ZeRO-1**: Partitions optimizer states.
  - **ZeRO-2**: Adds gradient partitioning.

- **ZeRO-3**: Extends to parameter partitioning for extreme-scale models.

- **Activation Checkpointing**:
  - Selective recomputation of activations to minimize memory footprint.

Our approach eliminates manual tuning by leveraging these frameworks' built-in coordination mechanisms, enabling seamless scaling from single-node to multi-node setups. Unlike traditional tools, FSDP and DeepSpeed unify data sharding, memory optimization, and communication efficiency into a single workflow, directly addressing training bottlenecks [9, 10].

We will benchmark **FSDP** and **DeepSpeed** configurations using **CIFAR** and **ViT-Base/32** [3], focusing on training efficiency:

1. **Scalability**: Training time and memory usage across 1–8 GPUs.
2. **ZeRO vs. Sharding**: Compare FSDP's sharding with DeepSpeed's ZeRO.
3. **Communication Overhead**: Gradient accumulation vs. pipeline parallelism.

Experiments will measure throughput (images/sec), GPU memory utilization, and final model accuracy to identify optimal configurations for distributed training.

4.1 Experimental Setup

Our experimental evaluation was conducted using CloudLab, utilizing a cluster of two compute nodes provisioned with high-performance hardware suitable for large-scale distributed training. Each node was equipped with four NVIDIA A100-SXM4 GPUs, each featuring 40GB of memory, providing ample computational power and memory bandwidth necessary for training large models efficiently. The nodes were interconnected using dual-port Mellanox ConnectX-6 DX 200Gb/s network interface cards (NICs), enabling high-throughput, low-latency communication essential for distributed deep learning workloads. The nodes were configured with CloudLab's small-lan profile. Each node also housed two AMD EPYC 7413 processors, totaling 48 physical cores per machine, running at 2.65GHz.

Our software stack included PyTorch as the primary deep learning framework for FSDP and deepspeed module for training using DeepSpeed. The distributed training strategies were implemented using torch.distributed, which facilitated various parallelism techniques. . To track and visualize training performance and resource utilization, we integrated Weights & Biases (wandb) into our training pipelines. This setup allowed for comprehensive monitoring and reproducibility across our experimental runs.

4.2 Metrics

To assess the effectiveness of the FSDP and DeepSpeed training strategies, the following metrics were recorded:

1. GPU Memory Allocation: We tracked the amount of GPU memory allocated during training to understand the memory footprint of different model configurations and parallelism strategies.
2. Network Traffic: Inter-node and intra-node communication was closely analyzed by measuring network traffic through the NICs. This metric was critical for understanding communication overhead introduced by distributed training. High network utilization often indicated bottlenecks due to gradient synchronization or parameter sharding.
3. CPU Usage: We measured CPU utilization to assess data preprocessing overhead, I/O performance, and orchestration efficiency. Monitoring CPU usage ensured that compute resources outside the GPU pipeline were not becoming a limiting factor in end-to-end training performance.

While Wandb tracks nearly 25 metrics by default, our analysis focused primarily on the most critical ones relevant to understanding performance trade-offs between FSDP and DeepSpeed.

5 Results

5.1 Resource Utilization Comparison Across Configurations

We evaluated the performance of various FSDP and DeepSpeed configurations based on three system-level metrics: GPU memory usage, network traffic, and CPU utilization. The following subsections analyze these metrics over time, comparing the effectiveness of sharding strategies and optimization stages.

5.1.1 GPU Memory Allocation

Figure 3 shows that **FSDP-Full** maintained the highest memory usage (around 85%), as it performs full parameter sharding and gathers full tensors before each computation step. In contrast, **DeepSpeed-R2** and **DeepSpeed-R1** showed significantly lower memory usage, stabilizing near 25%. **FSDP-Hybrid** and **DeepSpeed-R0** provided intermediate memory profiles. These results reaf-

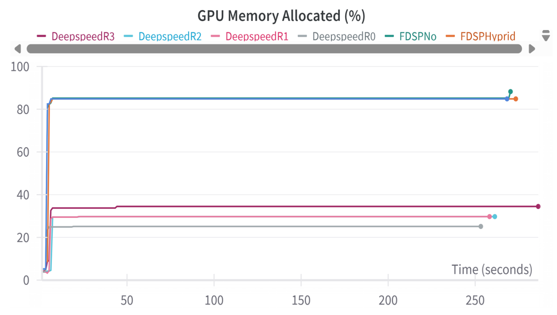


Figure 3: GPU Memory Allocated (%) over time for various configurations

firm the memory efficiency of DeepSpeed for larger-scale distributed training, particularly when using ZeRO-2 and ZeRo-3.

5.1.2 Network Traffic

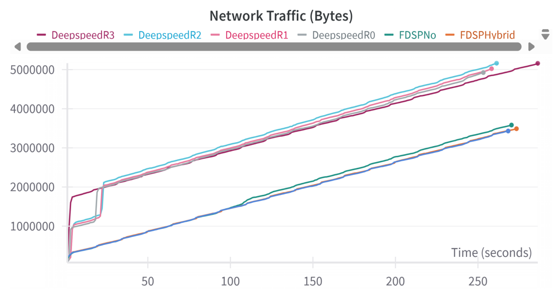


Figure 4: Network Traffic (Bytes) over time across configurations

As shown in Figure 4, **DeepSpeed** configurations incurred consistently higher network traffic over time, especially **ZeRO-2**, due to more aggressive partitioning and synchronization requirements. In contrast, **FSDP-Full** and **FSDP-No Shard** exhibited the lowest cumulative data transfer, suggesting reduced inter-device communication at the expense of higher on-device memory usage. This trade-off highlights DeepSpeed's reliance on bandwidth for memory reduction.

5.1.3 CPU Usage

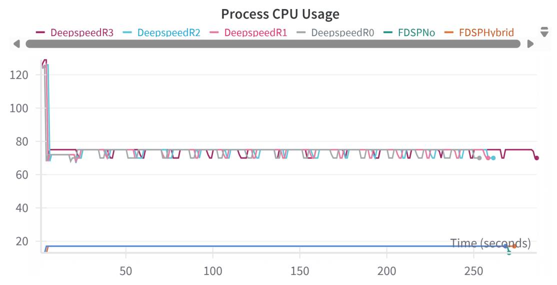


Figure 5: CPU Utilization (%) over time for all training setups

Figure 5 indicates that most **DeepSpeed** configurations maintained high and relatively stable CPU utilization (peaking near 75–80%), with some periodic drops possibly due to I/O waits or synchronization delays. On the other hand, **FSDP-Full** showed significantly lower CPU usage, remaining around 20–25%. This reflects its reliance on GPU-side parallelism and indicates lower host-side compute overhead.

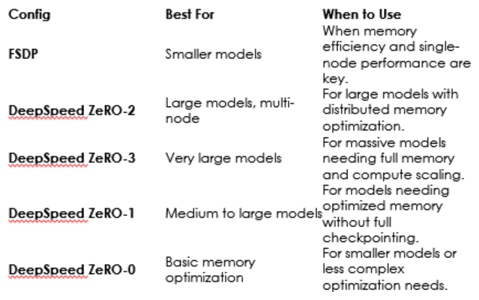
FSDP offerws better performance for small to medium-sized models on a single node, while DeepSpeed scales more effectively across multiple nodes

Verification: We conducted experiments in both single-node and multi-node environments. On a *single node*, **FSDP** achieved an average of **13% faster training** across all batch sizes for ViT-Base, primarily due to reduced communication overhead and efficient CUDA stream overlap. This makes it well-suited for setups where all GPUs reside on the same machine.

In contrast, in a *multi-node* setting, **DeepSpeed** demonstrated superior scalability and memory efficiency. By leveraging ZeRO-2 and ZeRO-3 optimizations, DeepSpeed used up to **30% less memory** than FSDP, enabling training with larger batch sizes. However, this efficiency comes at the cost of increased network bandwidth usage.

Setup	**Best Framework**	**Speed Gain**	**Memory Savings**	**Trade-off**
Single Node	FSDP	+13%	—	Less scalable across nodes
Multi Node	DeepSpeed	—	–30%	Higher network bandwidth usage

Table 1: Comparison of FSDP and DeepSpeed across single-node and multi-node environments.



Config	Best For	When to Use
FSDP	Smaller models	When memory efficiency and single-node performance are key.
DeepSpeed ZeRO-2	Large models, multi-node	For large models with distributed memory optimization.
DeepSpeed ZeRO-3	Very large models	For massive models needing full memory and compute scaling.
DeepSpeed ZeRO-1	Medium to large models	For models needing optimized memory without full checkpointing.
DeepSpeed ZeRO-0	Basic memory optimization	For smaller models or less complex optimization needs.

Figure 6: Table showing best use cases for FSDP and Deepspeed

6 Conclusion

By rigorously benchmarking **FSDP** and **DeepSpeed** configurations for distributed training of ViT on CIFAR, this work provides a systematic analysis of their performance under varying sharding, precision, and communication strategies. Our results will clarify how to best configure these frameworks for large-scale training, enabling researchers and engineers to reduce costs, accelerate convergence, and scale models beyond current limits. This research fills a critical gap, as no prior studies have explored the interplay between FSDP/DeepSpeed configurations and end-to-end training efficiency

7 Future Scope

Future work could involve benchmarking FSDP and DeepSpeed on significantly larger clusters (e.g., 16 to 64 GPUs), particularly to understand performance ceilings and scalability bottlenecks. While this work focused on resource utilization and training throughput, future studies should monitor convergence behavior over long training runs to assess if memory-optimized strategies inadvertently affect learning dynamics. Evaluating the performance of FSDP and DeepSpeed on even larger models is essential to understand the true capabilities and limits of each framework. Such experiments would reveal framework behaviors under extreme memory and compute stress.

8 Contribution

Implementation and experiment logs are available at: https://github.com/Sai-Krishna-Ponnam/CS744-Project

- Dev Mehrotra: Initiated the research idea and led the problem formulation. Designed and implemented DeepSpeed training pipelines with different ZeRO configurations. Conducted experimental validation on single-node environments and analyzed GPU memory usage. Set up Wandb for tracking metrics. Contributed to writing the Introduction, Related work, and Results sections.
- Sai Krishna Ponnam: Setup the CloudLab and handled resource provisioning. Developed scripts to install Nvidia drivers and install Python environment. Developed and executed the benchmarking scripts for PyTorch FSDP. Contributed to writing the Introduction, Background, and Methodology sections.
- Rachael Lang: Set up and managed the ImageNet dataset and ViT model preprocessing. Contributed to writing the Introduction, Background, and Conclusion sections. Ran experiments with varying batch sizes to analyze model performance under different configurations.

References

[1] Fsdp tutorial: https://docs.pytorch.org /tutorials/intermediate/fsdp_tutorial.html.

[2] S. Dash, I. Lyngaas, J. Yin, X. Wang, R. Egele, G. Cong, F. Wang, and P. Balaprakash. Optimizing distributed training on frontier for large language models, 2023.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[4] J. Keuper and F.-J. Preundt. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 19–26, 2016.

[5] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[7] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[8] R. Ramkeesoon. Benchmarking data parallel distributed training of deep learning models on pytorch and tensorflow.

[9] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. Fully sharded data parallel: Faster ai training with pytorch. *Microsoft Research Blog*, 2020.

[10] Z. Zhang, Z. Gan, S. Li, et al. Deepspeed: System optimizations enable training models with over 100 billion parameters. *arXiv preprint arXiv:2103.16307*, 2021.

[11] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, P. Damania, B. Nguyen, G. Chauhan, Y. Hao, A. Mathews, and S. Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.