Distributed Training and Efficient Inference for Large Language Models: A Survey

Manoj Myneni
Department of Computer Science
University of Illinois Chicago
Chicago, Illinois, USA
Email: mmyneni@uic.edu

Shubham Khatri
Department of Computer Science
University of Illinois Chicago
Chicago, Illinois, USA
Email: skhat25@uic.edu

***Abstract*—Large Language Models like GPT-3 and MT-NLG 530B set the new bar for natural language understanding and generation tasks, powering everything from chatbots to advanced text analytics. However, their training involves enormous computation and memory challenges that are well beyond single-GPU limits. In this scenario, distributed training supported by innovative system-level optimizations has emerged as a critical solution to scale these models up to hundreds of billions of parameters with feasible training times and costs. This survey represents an overview of the key research directions and milestones related to large-scale transformer-based LLM training and efficient inference. Key strategies including data, model, pipeline, and expert parallelism will be reviewed; memories optimally enhanced with techniques like DeepSpeed's ZeRO for trillion-parameter model training are discussed. We also discuss kernel-level advances—FLASHATTENTION—that makes attention mechanisms more memory-efficient with a small overhead in additional computational, and frameworks like Megatron-LM and DeepSpeed reduce the complexity of scaling efforts. Furthermore, we discuss how latency-hiding communication optimization techniques are tuned to handle massive models with mostly modest GPU clusters. Also, we discuss techniques reducing latency and leveraging heterogeneous memory, e.g., CPU and NVMe, among others, for handling massive model training on modest GPU clusters. Our survey synthesizes insights from a broad spectrum of works, outlines their relative merits and limitations, and identifies open challenges, including balancing parallelism schemes, addressing communication overheads, and extending beyond simple training efficiency metrics toward energy use and fault tolerance. To reinforce these literature-derived insights, we present a small experimental setup where we train a DistilBERT model for text classification using PyTorch Distributed Data Parallel and DeepSpeed's ZeRO. The initial results indicate better convergence speed and scalability, which is consistent with the results surveyed. This is a literature survey that pinpoints an effective road map for those practitioners and researchers who are trying to adopt an effective distributed strategy for building and deploying a megascaling LLM.**

***Index Terms*—Distributed Training, Large Language Models, Transformer-Based Models, DeepSpeed, ZeRO, Parallelism, Memory Optimization, HPC for NLP**

I. INTRODUCTION AND MOTIVATION

Large Language Models (LLMs) have fundamentally reshaped the landscape of Natural Language Processing (NLP). In recent years, models such as BERT, GPT-2, GPT-3, Megatron-LM, and MT-NLG 530B have demonstrated their ability to generate coherent, contextually aware, and human-like text, as well as perform a wide range of downstream tasks with state-of-the-art accuracy. These capabilities are largely attributed to the unprecedented size of these models, which can consist of tens or hundreds of billions of parameters, and in some cases even approach the trillion-parameter scale. This drastic increase in model size and complexity has been motivated by empirical evidence showing that larger models tend to generalize better, show emergent reasoning abilities, and adapt more readily to new tasks with minimal finetuning. However, along with these advances in performance and generality come formidable computational and memory-related challenges that single-GPU or single-machine solutions cannot adequately address.

The key challenge lies in the interaction between model size, data complexity, and available hardware resources. As model parameters scale to the order of billions and beyond, the memory required to store weights, gradients, optimizer states, and intermediate activations during training grows proportionally. Traditional single-device training pipelines quickly run out of memory when attempting to train large models, forcing practitioners to resort to smaller batch sizes, gradient accumulation steps, or smaller models, all of which can degrade model quality or training efficiency. Furthermore, massive models require an extensive computational effort. Running a forward and backward pass through these models on billions of tokens is computationally expensive. Even with specialized accelerator hardware, such as NVIDIA A100 or H100 GPUs, training can span weeks or months, making the training process both time-consuming and cost-prohibitive.

It is here that distributed training emerges as a key enabler. By leveraging multiple GPUs—often spanning multiple nodes and sometimes entire clusters—training can be parallelized. Distributed strategies aim to partition the workload in such a way that both memory and compute resources are balanced across devices, significantly accelerating the training process. Various forms of parallelism have been explored in the literature, including data parallelism, model parallelism (tensor and pipeline parallelism), and more recently, expert parallelism for Mixture-of-Experts (MoE) models. Each strategy comes with its own trade-offs, communication overheads, and implementation complexities, requiring sophisticated runtime systems to achieve efficient scaling.

A. Memory and Computational Bottlenecks

The memory requirement for training extremely large models often surpasses the capacity of a single GPU. For instance, loading a large LLM like MT-NLG 530B or LLaMA-70B model exceeds hundreds of gigabytes of memory, and that is just for the parameters, not counting gradients, optimizer states, and activation memory. Storing and updating these states presents a significant barrier to using a single-device setup. Data parallelism, the simplest approach, replicates full model states across all devices, worsening the memory burden linearly with the number of GPUs. Although data parallelism scales out to more GPUs, it does not solve the per-device memory usage problem. Conversely, model parallelism—by slicing parameters or layers across multiple devices—can alleviate memory issues, but introduces extra communication costs and often complicates scaling beyond a single node due to bandwidth limitations and latency overheads. Pipeline parallelism splits the model vertically into stages and can reduce per-GPU memory load, but it may create pipeline bubbles and necessitate careful orchestration of micro-batches to maintain efficiency [9].

B. ZeRO and Memory Efficiency

A crucial development in this area is the Zero Redundancy Optimizer (ZeRO), a series of techniques that partition optimizer states, gradients, and parameters across data-parallel processes, minimizing memory redundancy [3]. By eliminating replicated states, ZeRO enables training models with trillions of parameters that were previously considered untrainable due to memory constraints. ZeRO's different stages—ranging from only partitioning optimizer states (Stage 1) to progressively partitioning gradients (Stage 2) and parameters (Stage 3)—provide a spectrum of memory-saving strategies. These optimizations have been integrated into frameworks like DeepSpeed, which simplify distributed training and allow fine-grained control over memory usage and communication patterns.

C. Performance and Communication Overheads

Scaling large models across multiple devices is not just about memory; the interconnect and communication overhead become critical bottlenecks. Consider a scenario where a single training step involves multiple all-reduce operations and parameter-sharding synchronizations. The effective training speed can degrade substantially if communication operations are frequent, large in volume, or poorly synchronized [5]. Therefore, frameworks must carefully balance communication with computation and implement efficient collective communication patterns. NCCL, Gloo, MPI, and custom communication libraries are leveraged to reduce latency. Efforts have been made to encode communication schedules that overlap communication with computation, as well as use advanced scheduling to minimize straggler effects.

D. Complexity of Implementing Distributed Training

The complexity of distributed training, including configuring parallelisms, partitioning data, scheduling pipeline stages, and choosing appropriate memory optimization strategies, poses a steep learning curve for developers. Although frameworks like DeepSpeed, Megatron-, and others have lowered the barrier, they still require a careful setup of configuration files, code integration, and tuning of hyperparameters to achieve optimal performance. The literature highlights a lack of comprehensive comparisons and guidelines for choosing among parallelization strategies and how to tune them for different architectures and workloads. Without these, trial-and-error remains common practice, consuming developer time and computational resources.

E. Heterogeneous Memory and Inference Challenges

While this literature primarily focuses on training, inference poses another angle of complexity. Once trained, serving large models at scale necessitates efficient memory management. ZeRO-Inference and CPU/NVMe offloading strategies emerge as solutions. These methods enable massive models to be served on systems with limited GPU memory by fetching weights layer-by-layer from CPU or NVMe to GPU memory as needed. Such techniques provide ways to leverage heterogeneous memory hierarchies to reduce inference latency and improve cost-efficiency. Furthermore, the design of these methods can benefit from insights gained during training. For instance, if a method for partitioning parameters at training time enables efficient load balancing and minimal communication overhead, similar logic can apply at inference time. With the increase in the size of models and frequency of queries in production systems, scaling inference across multiple GPUs and nodes becomes essential for reducing latency and meeting user demands [8]. The literature indicates that a comprehensive understanding of distributed training principles also informs scalable inference solutions.

F. Sparse Models and Expert Parallelism

An emerging class of architectures uses sparse Mixture-of-Experts (MoE) layers to scale models more efficiently. These add complexity to the parallelization logic, as experts must be dynamically selected and tokens dispatched to the appropriate expert nodes [2], [11]. Expert parallelism distributes these experts across devices, and the challenge is to keep load balanced and communication overhead low. This requires advanced parallelization techniques that coordinate with data and tensor parallelism for both training and inference. Such architectures show promise in improving scaling efficiency and model quality, but their complexity heightens the importance of well-designed distributed training frameworks.

G. Energy Efficiency and Environmental Impact

In addition to the conventional metrics of training speed and memory usage, there is a growing interest in energy efficiency and sustainability. The massive scale of LLMs leads to high energy consumption, raising environmental concerns.

Distributed training on specialized, efficient hardware and intelligent parallelization strategies can reduce energy per training run. By accelerating model convergence and decreasing total compute cycles, distributed training can indirectly lower the carbon footprint of AI research and deployment. Although the literature does not always emphasize this, the motivation to improve energy efficiency is implicit and increasingly important.

H. Gaps and Future Directions

Despite the considerable progress in distributed training methods and frameworks, gaps persist. Communication often emerges as a bottleneck at scale, and efforts to reduce it must be balanced with memory optimization and ease of use. The field also needs more systematic evaluations and benchmarks that compare different configurations, frameworks, and parallelization strategies across a variety of model sizes and architectures [5], [6]. A unified set of metrics—spanning memory usage, TFLOPS utilization, scaling efficiency, energy consumption, and even reliability under node failures—would provide richer insights. Moreover, the complexity of distributed training demands user-friendly interfaces and abstractions that hide low-level details, enabling practitioners with limited systems expertise to efficiently train large models. Recent libraries strive to simplify these challenges, but additional research and development are needed to achieve a plug-and-play experience. Innovative techniques, such as dynamic parallelism or autotuning systems that adapt parallelization strategies at runtime based on performance feedback, could offer breakthroughs. Finally, as transformer models continue to evolve—introducing new forms of attention, multi-modal capabilities, or even more intricate architectures—the distributed training methodologies must adapt. Future research may concentrate on specialized kernels, communication-avoiding algorithms, or even integration of more advanced hardware topologies to keep pace with model innovation.

I. Summary

The motivation behind distributed training of large language models is driven by the interplay of model scale, performance demands, and resource constraints. Without distributed strategies, the promise of LLMs to handle unprecedented volumes of data, learn richer representations, and tackle increasingly complex tasks would remain unrealized. By distributing computations, memory footprints, and communication loads across multiple GPUs and nodes, practitioners can train trillion-parameter models, accelerate experimentation, and ultimately push the boundaries of what LLMs can achieve. This report delves into the literature that explores these solutions—DeepSpeed's ZeRO optimization, parallelization schemes, memory offloading strategies, and efficient kernel implementations—offering a comprehensive understanding of how to navigate the complexity of distributed training. Through these efforts, researchers and engineers aim to create a future where massive LLMs are accessible and efficient to train, fueling the next wave of breakthroughs in NLP and beyond.

II. BACKGROUND

The exponential growth in the size and complexity of Large Language Models (LLMs) has made distributed training a cornerstone of their development, enabling efficient and scalable solutions for these computationally demanding models. To fully understand the challenges and advancements in this area, it is critical to explore the underlying technologies, frameworks, and hardware that have facilitated the state-of-the-art in training and inference for large-scale transformer models. This section delves into the foundational techniques and trends driving progress in this field.

A. Transformer Architectures and Scaling Trends

The introduction of the transformer architecture marked a paradigm shift in natural language processing. The design, centered around attention mechanisms, allows the model to process long-range dependencies in sequences with unprecedented efficiency. Key features such as multi-head self-attention layers and feed-forward networks stack together to create powerful, deep representations that have become the backbone of modern LLMs. As research demonstrated that scaling up model size directly correlated with improvements in accuracy and capability, the NLP community embraced a "bigger is better" philosophy. This led to groundbreaking models, from BERT and GPT-2, to GPT-3, Megatron-LM , and MT-NLG 530B. These larger models have showcased exceptional performance in a wide range of tasks, including few-shot and zero-shot learning, while introducing challenges in terms of memory, computation, and energy requirements. Scaling these models involves increasing their depth (number of layers), width (hidden dimensions and attention heads), and sequence length. The quadratic scaling of attention mechanisms with sequence length poses a significant memory bottleneck. Additionally, emerging architectures like Mixture-of-Experts (MoE) aim to scale parameters more efficiently but introduce complex conditional computations and routing mechanisms [2], [11].

B. Basic Parallelization Techniques

Distributed training strategies rely on three core parallelization techniques: data parallelism, model parallelism, and pipeline parallelism, each addressing different aspects of the scalability challenge.

1) Data Parallelism: Data parallelism divides the training dataset across multiple GPUs, with each processing a fraction of the data independently. After computing gradients, GPUs communicate via all-reduce operations to synchronize updates across devices. While conceptually simple and supported by frameworks like PyTorch's Distributed Data Parallel (DDP), this approach replicates the model states across all devices, leading to excessive memory usage that becomes prohibitive for extremely large models.

2) Model Parallelism: Model parallelism partitions the model itself, with two primary variants:

- Tensor Parallelism (Horizontal Splitting): This technique slices weight matrices across multiple GPUs, reducing the per-device memory footprint. GPUs perform partial computations, requiring communication during each forward and backward pass to synchronize results. Although effective in single-node systems with fast interconnects like NVLink, tensor parallelism struggles with inter-node bandwidth constraints [5], [7].
- Pipeline Parallelism (Vertical Splitting): Pipeline parallelism divides the model's layers into stages, assigning each stage to a different GPU. Micro-batches are processed sequentially through the pipeline, with multiple batches running concurrently to maximize throughput. Advanced scheduling strategies like 1F1B minimize idle times. However, challenges include managing activation checkpoints and ensuring balanced workloads.

3) Expert Parallelism for Sparse MoE Models: Expert parallelism distributes specialized subnetworks ("experts") across GPUs, activating only a subset for each input. While this approach optimizes parameter utilization, dynamic routing and all-to-all communication introduce significant overheads [8], [11].

C. Memory Optimization Techniques

As LLMs scale into the trillion-parameter range, memory optimization becomes a necessity to manage the storage and computation of weights, gradients, and activations.

1) ZeRO (Zero Redundancy Optimizer): ZeRO partitions optimizer states, gradients, and parameters across GPUs, minimizing memory redundancy. Its three stages progressively optimize memory usage: Stage 1 partitions optimizer states, Stage 2 includes gradients, and Stage 3 partitions parameters. By distributing these components, ZeRO enables the training of models that would otherwise exceed the memory capacity of GPUs [3].

2) Activation Checkpointing: Also known as recomputation, activation checkpointing reduces memory usage by discarding intermediate activations during the forward pass and recomputing them during the backward pass. This trade-off increases computational overhead but allows for training larger models or using larger batch sizes [5].

3) Fused Kernels and FLASHATTENTION: Fused kernels improve efficiency by combining operations into a single kernel, reducing memory access overhead. FLASHATTENTION enhances this further by implementing attention mechanisms in an IO-aware kernel, improving memory bandwidth utilization and overall training speed.

D. Communication Aspects

Communication overhead is a critical challenge in distributed training, especially at scale.

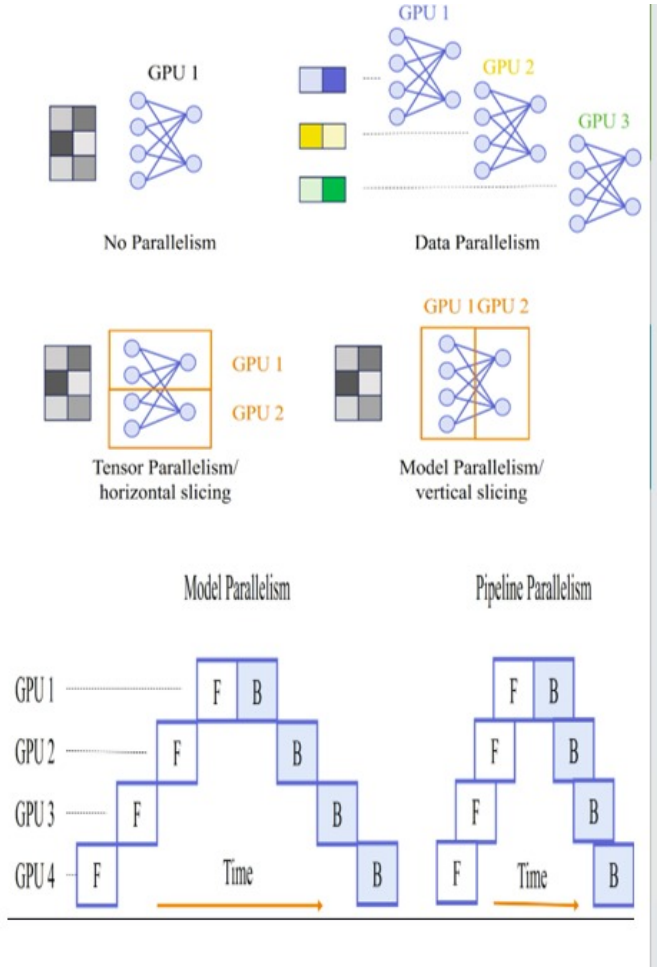

Fig. 1. Comparison of Parallelism Techniques for Distributed Model Training

1) Collective Communication Libraries: Tools such as NVIDIA NCCL, Gloo, and MPI optimize synchronization for gradient updates. Techniques such as overlapping communication with computation help mitigate latency and improve performance [4], [6].

2) Parallelism Coordinated Communication (PCC): Advanced communication strategies for MoE models optimize token expert routing, restructuring all-to-all operations into collective patterns that reduce latency and enhance scalability [11].

E. Frameworks and Tooling

Several frameworks simplify the complexities of distributed training:

- DeepSpeed: DeepSpeed integrates ZeRO and other optimizations, offering a user-friendly library for distributed training configurations, memory-saving strategies, and kernel optimizations.
- Megatron-LM: This framework focuses on tensor parallelism, providing tools and fused kernels for large-scale training, often complementing DeepSpeed's features [7].

- Other Systems: Platforms like Colossal-AI, Alpa, and TVM explore automated scheduling and parallelization, further streamlining distributed training workflows [5].

F. Emerging Trends

- Fine-Grained Parallelism: Partitioning within individual heads or layers offers new avenues for scalability and efficiency [2].
- Dynamic Systems: Auto-tuning frameworks dynamically adjust parallelization strategies based on performance feedback, accommodating hardware variations and workload shifts [6].
- Inference Scalability: Techniques like CPU-offloading and NVMe storage adapt memory optimization strategies from training to inference, enabling efficient model serving [8].

G. Summary

This section highlights the foundational strategies and innovations that drive distributed training for LLMs. Techniques like ZeRO, FLASHATTENTION, and activation checkpointing address key memory and computation challenges, while frameworks like DeepSpeed and Megatron-LM streamline deployment. These advances provide a roadmap for training and serving trillion-parameter models efficiently, setting the stage for breakthroughs in NLP and beyond.

III. DISTRIBUTED TRAINING: SURVEYS AND CASE STUDIES

The increasing complexity and scale of large language models (LLMs) have led to a surge of research and engineering efforts focusing on distributed training as a foundational technique. While single-device setups struggle under the memory and computational load of multi-billion- or trillion-parameter models, distributed training offers a pathway to efficiently leverage hundreds or even thousands of GPUs. In this section, we survey key research works and case studies that have advanced distributed training methods for LLMs. We highlight the evolution of techniques like data parallelism, model parallelism, pipeline parallelism, and more nuanced approaches, as well as their practical implementations and performance characterizations.

A. Data Parallelism and Baseline Distributed Techniques

Early works in training large-scale models predominantly relied on data parallelism to increase throughput [1]. By splitting the input dataset across multiple devices, data parallelism provides a straightforward scaling mechanism. However, as noted in multiple surveys [1], [2], [6], data parallelism alone falls short for extremely large models due to memory constraints—duplicating entire model states on each GPU is neither memory- nor cost-efficient. The seminal PyTorch Distributed Data Parallel (DDP) approach [1] simplifies gradient synchronization, allowing linear scaling up to tens or hundreds of GPUs. Although widely adopted, DDP by itself cannot address the memory footprint of models like GPT-3 (175B parameters) or MT-NLG 530B [5].

B. Model Parallelism: Tensor and Pipeline Approaches

To tackle models too large to fit into a single GPU's memory, model parallelism emerged as a critical technique. Works like Megatron-LM [7] introduced tensor slicing, partitioning individual layers' parameters across GPUs. This approach reduces per-GPU memory usage but requires careful synchronization. Another dimension is pipeline parallelism, which splits models by layer-stacks into pipeline stages, allowing a micro-batch scheduling pattern that attempts to hide pipeline bubbles. While pipeline parallelism does not reduce memory footprints as effectively as tensor slicing, it helps in balancing workloads and can achieve near-linear speedups when combined with data parallelism [9].

Megatron-LM's 3D parallelism—combining data, tensor, and pipeline parallelism—proved integral for scaling to multi-trillion parameter models, as described in Hagemann et al. [5] and other follow-up works [7]. These studies highlight the importance of communication optimization at scale. On top of these foundational parallelization methods, frameworks like Microsoft's DeepSpeed have offered more accessible building blocks, such as ZeRO-1, ZeRO-2, and ZeRO-3 optimization stages, to partition optimizer states, gradients, and parameters across data-parallel ranks. The result is a significant memory footprint reduction that scales to trillion-parameter models [2].

C. Memory-Efficient Strategies: ZeRO and Beyond

ZeRO's staged approach to memory reduction has garnered attention as a key technique. Initially introduced by Rajbhandari et al. [3], ZeRO eliminates redundancy by ensuring that each data-parallel GPU stores only a partition of the optimizer states and gradients. Subsequent improvements, including full-parameter partitioning (ZeRO-3) and hybrid offloading strategies to CPU and NVMe storage, push memory boundaries even further. This shift allows training models that were once out of reach, using modest GPU clusters. Studies have shown that ZeRO can even support inference scenarios where model parameters are too large for GPU memory [8].

Beyond ZeRO, FLASHATTENTION and improved fused kernels reduce both memory and time complexity, making it feasible to handle extremely long sequences without overwhelming GPU memory. Expert parallelism, as demonstrated by mixtures-of-experts models [2], [11], adds another dimension: by distributing different "experts" across devices, one can scale model capacity without linearly increasing compute requirements for every input token.

D. Communication and Networking Considerations

The performance of distributed training often hinges on the efficiency of communications. Works focusing on the communication characteristics of distributed training [4] emphasize that scaling training to multiple nodes introduces latency and bandwidth bottlenecks. Studies like Wenxue Li et al. [4] carefully analyze point-to-point and collective operations, shedding light on when to apply ring-based allreduce or hierarchical

topologies. Techniques such as NCCL optimizations, mixed-precision communication, and overlapping communication with computation are widely discussed in the literature.

Hagemann et al. [5] present systematic ways of deciding parallelization layouts—how to pick the correct ratio of data, tensor, and pipeline parallelism—to achieve near-optimal resource utilization and performance efficiency. Their studies highlight that even with the best memory optimizations, choosing poor parallelization strategies can degrade performance. Their "efficient parallelization layouts" guidance helps practitioners avoid trial-and-error and pick configurations that align with their cluster's GPU, memory, and bandwidth characteristics.

E. Scaling to Trillion Parameters and Performance Analysis

Recent breakthroughs, such as training models beyond the hundred-billion parameter scale [5], would not be possible without distributed training optimizations. Duan et al.'s survey [6] consolidates many of these cutting-edge techniques, illustrating how memory partitioning strategies (e.g., ZeRO and FSDP), communication libraries (NCCL, MPI), and compiler-level optimizations converge to produce record-scale models. They also highlight the synergy of automated parallelization frameworks and sophisticated runtime systems that can balance load, reduce pipeline stalls, and adaptively choose data parallel splits at runtime.

Moreover, Bagus Hanindhito et al. [10] specifically measure the bandwidth characteristics of DeepSpeed-based training, showing that sustaining high throughput requires not just raw GPU counts but also a well-matched networking infrastructure. Larger batch sizes, longer sequence lengths, and complex attention patterns can stress both memory and network bandwidth, and the authors' experimental work underscores how critical all these components are in practice.

F. Inference Considerations and Heterogeneous Resources

While most literature emphasizes training efficiency, inference at scale is also a central concern. Aminabadi et al. [8] propose DeepSpeed Inference, enabling efficient inference of enormous LLMs by leveraging CPU and NVMe memory in addition to GPU memory. This heterogeneous approach ensures that production scenarios, where models must handle real-time queries, can also benefit from the distributed strategies, albeit with inference-centric optimizations. Although inference is not always identical to training, the underlying principles—such as pipeline scheduling, expert parallelism, and memory partitioning—remain applicable.

G. Our Small Experiment: A Validation Exercise

Inspired by these findings, we conducted a small-scale experiment using DistilBERT, a significantly smaller LLM, trained via distributed data parallelism on a multi-GPU setup. While not aiming to achieve state-of-the-art scaling, the experiment helps validate the fundamental claims from the literature: that even a modest distributed configuration can reduce training times and better utilize hardware compared to a single-GPU baseline. Such a practical check, though limited in scale, offers hands-on insight into the complexity and benefits of distributed training strategies.

H. Summary of Key Lessons from the Literature

The surveyed literature and case studies collectively emphasize that no single approach suffices for all scenarios. Instead, a combination of parallelization techniques (data, tensor, pipeline), memory optimization frameworks like ZeRO, and kernel-level improvements like FLASHATTENTION or fused GeMM kernels is crucial. Expert parallelism adds another dimension, enabling massive sparse models to run efficiently without linearly expanding computation costs.

Practical frameworks like DeepSpeed simplify these integrations, while community-driven best practices and automated layout selection tools reduce guesswork. Communication remains a critical bottleneck at scale, necessitating careful tuning of all-reduce operations and network topologies.

Overall, this survey reveals a rich ecosystem of proven strategies and active research directions, all geared toward enabling efficient distributed training and inference for unprecedented-scale models. It sets the stage for our subsequent empirical observations, guided by lessons learned from pioneering works.

IV. MEMORY AND COMMUNICATION OPTIMIZATIONS FOR DISTRIBUTED LLM TRAINING

As Large Language Models continue to scale, memory and communication overheads become critical bottlenecks. Achieving near-linear scalability and efficient resource utilization in distributed training requires not only parallelization but also a suite of carefully orchestrated memory and communication optimizations. This section delves into the key strategies that various studies have put forth to tackle memory constraints, data-transfer bottlenecks, and synchronization overheads. By synthesizing insights from works by [3], [5], [2], and others, we highlight how these optimizations fit together to unlock training of trillion-parameter models on modern GPU clusters.

A. Fine-grained Memory Partitioning with ZeRO

Among the most influential developments in memory optimization is the Zero Redundancy Optimizer (ZeRO) from DeepSpeed. Traditional data parallelism replicates full parameter states across all GPUs, leading to memory blowup. ZeRO [3] addresses this by sharding optimizer states, gradients, and parameters across data-parallel ranks. This partitioning ensures that each GPU holds only a fraction of the entire memory footprint, enabling model sizes to scale far beyond what a single GPU could store. The literature underscores that ZeRO can reduce memory requirements by an order of magnitude, allowing for larger batch sizes and improved training stability.

As models reach hundreds of billions of parameters, ZeRO stages become critical: Stage 1 focuses on partitioning optimizer states, Stage 2 includes gradient partitioning, and

Stage 3 goes further to shard parameters themselves. Although implementing ZeRO introduces additional complexity in the data-parallel training pipeline, practitioners report that once configured, the memory savings outweigh the overhead, especially when combined with other parallelization strategies. The surveyed papers often mention using ZeRO in tandem with tensor or pipeline parallelism to achieve a balanced memory footprint and maintain high arithmetic intensity.

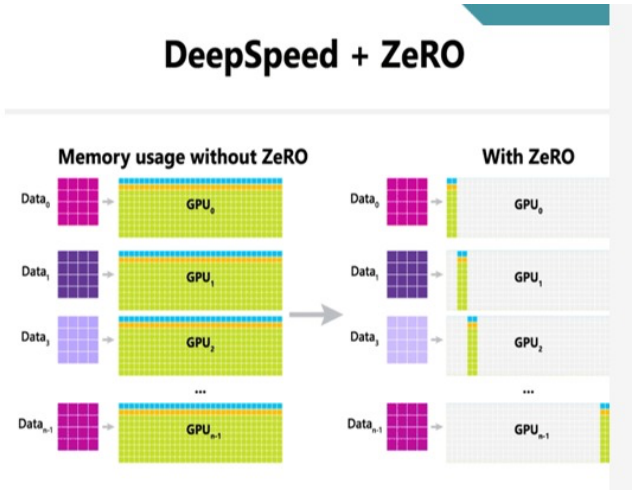

Fig. 2. Memory Optimization with DeepSpeed ZeRO: Comparison of Memory Usage Without and With ZeRO

B. Checkpointing and Selective Activation Recomputation

Another well-known memory-saving technique is activation checkpointing or recomputation. Instead of storing all intermediate layer outputs (activations), only a subset is saved, and the others are recomputed on-the-fly during the backward pass. Though this increases the computational load, it can halve or more the activation memory usage. Studies by Hagemann et al. [5] show that when combined with ZeRO and pipeline parallelism, activation checkpointing enables fitting massive models into a given GPU memory budget. Papers also highlight that the combination of FLASHATTENTION's intrinsic activation recomputation with selective MLP block checkpointing can fine-tune the memory-compute trade-off for specific training configurations [2].

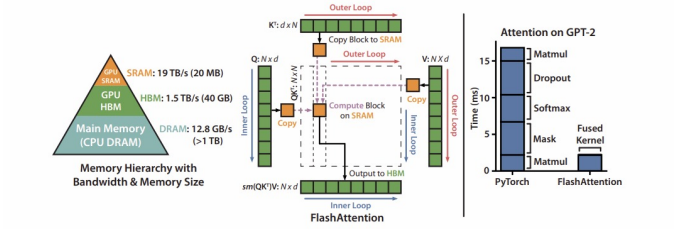

Fig. 3. Memory Hierarchy and Activation Checkpointing for Efficient Model Training

C. Fused Kernels, FLASHATTENTION, and IO-aware Computation

Kernel fusion reduces memory bandwidth pressure by merging multiple GPU kernels into a single operation. Instead of loading and storing intermediate results multiple times, fused kernels keep data in registers or shared memory, decreasing DRAM accesses. FLASHATTENTION takes this concept further, restructuring the attention mechanism to be IO-aware. By reading and writing memory less frequently and using on-chip SRAM more effectively, FLASHATTENTION reduces memory footprint and improves training throughput. Surveys note that when FLASHATTENTION is integrated with ZeRO, pipeline parallelism, and expert parallelism, the synergy can lead to substantial speedups and memory savings—up to $2\times$ or more—compared to vanilla attention kernels.

The literature recommends fused kernels as a default setting for large-scale training. The complexity lies in ensuring compatibility between fused kernels and the various parallelization schemes. Some fused kernels are sensitive to batch sizes, sequence lengths, or model dimensions, and need tuning for each scenario.

D. Communication-Avoiding Algorithms and Collective Primitives

Beyond memory optimizations, communication overhead can dominate training time at large scale. All-reduce operations to synchronize gradients or parameters across hundreds of GPUs can saturate even high-bandwidth interconnects. To mitigate this, researchers have explored several strategies:

- Overlapping Computation and Communication: Scheduling communication operations (e.g., parameter all-reduce) concurrently with the backward pass of subsequent layers can mask communication latency. By carefully pipelining communication and computation, the effective overhead reduces, leading to higher GPU utilization.
- Hierarchical Collectives and PCC: MoE and mixed parallelism scenarios benefit from advanced communication patterns. For example, Parallelism Coordinated Communication (PCC) [11] rearranges global all-to-all operations into a sequence of local all-to-all plus all-gather steps. By leveraging intra-node bandwidth more efficiently before resorting to inter-node communication, PCC reduces latency and improves scaling.
- Topology-Aware Mapping: Some studies highlight the importance of mapping MPI ranks or NCCL processes to GPUs in a topology-aware manner [4]. Placing tightly coupled parallel ranks within a node or on the same InfiniBand switch can reduce interconnect contention, minimizing communication overhead. Although this might require additional orchestration and knowledge of cluster topology, case studies confirm improved scaling efficiency when applied.

E. Quantization, Mixed Precision, and Memory Formats

Mixed precision training (FP16 or BF16) has become standard practice, not only to accelerate compute-bound opera-

tions on tensor cores but also to reduce memory usage by half compared to FP32. Beyond that, quantization techniques that represent weights or activations in INT8 or even more compact formats have been explored. While quantization can introduce subtle accuracy trade-offs, the literature suggests that carefully chosen quantization strategies can yield memory and bandwidth improvements without degrading convergence significantly. This is especially relevant for inference, where INT8 or FP16 quantization can cut memory overheads and latency [8].

In training scenarios, mixed precision optimizers that maintain master weights in FP32 while processing gradients in FP16 or BF16 strike a balance between memory efficiency and numerical stability. The community widely accepts such approaches as standard practice for large-scale LLM training, supported by frameworks like DeepSpeed and Megatron-LM by default.

F. NVMe and CPU Offloading for Extended Capacity

As model sizes approach trillions of parameters, even the most sophisticated partitioning and memory compression methods can fall short. NVMe and CPU offloading techniques extend the available memory beyond GPU DRAM [8]. While this approach can increase latency due to slower PCIe or network storage transfers, it allows training or inference on gargantuan models without upgrading to ultra-high-memory GPUs. Prefetching strategies help hide offloading latency by loading data from NVMe or CPU memory layers ahead of their usage.

The literature suggests that such offloading is more common at inference time, as latency can often be tolerated if it remains below a certain threshold. Training runs, however, might find it less appealing due to the tight coupling between steps. Still, experiments show that offloading at specific training stages or for certain infrequently accessed parameters can produce net benefits.

G. Balancing Memory, Communication, and Compute

The complexity arises from the need to simultaneously consider memory savings, communication patterns, and compute intensity. Selecting an optimal combination of ZeRO stage, parallelization scheme (data, tensor, pipeline, expert), fused kernels, and offloading depends on the target model size, batch size, and hardware environment. The literature lacks a universal formula for this balancing act, but repeatedly emphasizes empirical evaluation. Tuning is often necessary, guided by profiling tools and iterative tests, to reach near-perfect scaling and resource utilization [2], [6].

H. Practical Guidelines from Literature

Several surveyed papers propose guidelines:

1) Start Simple: Begin with data parallelism and ZeRO Stage 1 to enable larger batch sizes. Introduce tensor parallelism if memory still falls short.
2) Add Pipeline Parallelism for Multi-Node: When scaling beyond a single node, incorporate pipeline parallelism to reduce the memory footprint per GPU.
3) Apply Fused Kernels and FLASHATTENTION: Once parallelization is set, enable fused kernels to improve memory bandwidth utilization and reduce kernel invocation overhead. FLASHATTENTION is beneficial for long sequences.
4) Refine with Checkpointing and Offloading: If still constrained, selectively activate checkpointing layers or consider offloading to CPU/NVMe storage. This can be a last resort due to complexity and latency.
5) Measure and Iterate: Use profiling tools to identify bottlenecks and tune accordingly. Communication overhead might require rearranging ranks or changing the parallelization degree.

I. Summary

Memory and communication optimizations form the backbone of successful distributed LLM training. From the revolutionary ZeRO partitions to advanced fused kernels and careful communication scheduling, each technique addresses a specific dimension of the scaling problem. The literature shows that combining these strategies yields multiplicative gains, pushing the limits of model size and training speed. Yet, achieving this synergy often demands careful experimentation, profiling, and iteration. As models grow and hardware evolves, ongoing research aims to automate these choices and further reduce complexity, ensuring that massive LLMs remain within the reach of diverse research and industrial communities.

V. INFERENCE AND SERVING MASSIVE LLMS

The ultimate goal of training massive LLMs extends beyond research curiosities; these models are increasingly deployed in real-world applications, from conversational agents to code assistants and large-scale search systems. The inference phase—generating outputs given new inputs—imposes a distinct set of challenges. While training focuses on updating parameters through backpropagation, inference is often a readonly scenario. Nonetheless, the memory, latency, and throughput considerations remain vital. Large-scale deployments must handle unpredictable workloads, fast response times, and potentially billions of queries per day.

This section examines literature contributions on scaling inference for massive LLMs. We consider CPU and NVMe offloading strategies, multi-GPU inference architectures, and concurrency models that allow serving hundreds or thousands of requests in parallel. The surveyed works highlight the interplay between training and inference: techniques refined during training often inform inference strategies. By summarizing these insights, we offer a cohesive view of how large models move from research prototypes to production-grade systems.

A. From Training Paradigms to Inference Scenarios

Inference inherits many ideas from training. For instance, ZeRO-based memory partitioning [3], initially designed for reducing memory in training, can be adapted to inference by distributing parameters across GPUs to handle enormous models [8]. Similarly, pipeline parallelism can facilitate streaming

tokens through multiple pipeline stages to speed up generation. Studies show that while pipeline parallelism may introduce some latency overhead in generation, careful micro-batching and scheduling can still improve throughput.

One key difference is that inference workloads often have irregular arrival patterns and smaller input contexts compared to training. Additionally, inference sessions might require dynamic sequence lengths or prompt complexities. Not all optimizations from training directly translate to inference because inference may lack the benefit of large, steady training batches and predictable iteration patterns. Instead, inference might need adaptive scheduling that can scale batch size on-the-fly depending on the latency-service-level agreement (SLA).

B. CPU, NVMe, and Memory Offloading for Inference

Storing massive LLMs fully on GPU memory may be cost-prohibitive in production. ZeRO-Inference and similar strategies [8] propose offloading model weights to CPU memory or even NVMe SSDs. On-demand fetching of layer weights occurs as tokens are processed. Although this introduces data transfer overhead per token, careful prefetching and caching layers can minimize latency.

The literature demonstrates that by employing layer-by-layer streaming and pipelined weight transfers, inference latency can remain acceptable, enabling the use of cost-effective commodity GPUs rather than requiring ultra-large memory GPUs. Some studies emphasize that slow memory hierarchies (CPU or NVMe) are best used for infrequently accessed model segments, or combined with quantization strategies to reduce the data volume transferred. For instance, a portion of less critical parameters or expert layers could reside in CPU memory, fetched only when gating decisions route tokens to those experts [11].

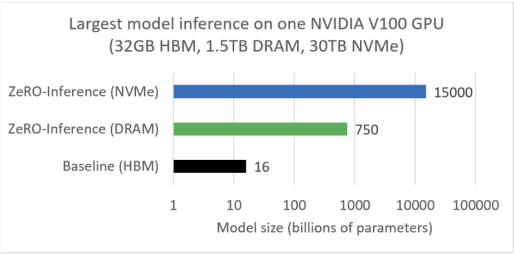

Figure 1:Maximum model sizes for inference on one V100 GPU when hosting the model on GPU memory (HBM), CPU memory (DRAM), or NVMe memory. Note that x-axis is log scale.

Fig. 4. Maximum Model Sizes for Inference Using ZeRO-Inference with Different Memory Hierarchies

C. Quantization and Mixed Precision for Serving

Quantization techniques that shrink model data types from FP16/BF16 to INT8 or lower can dramatically reduce model size and memory bandwidth demands. Works like Zeng et al. [2] and Hanindhito et al. [10] discuss quantization-aware inference, highlighting improved throughput at negligible accuracy loss. Since inference does not require gradient computations, maintaining master weights in higher precision is unnecessary. Consequently, INT8 or even 4-bit weight quantization can be employed, as some emerging research suggests, to fit more of the model into GPU memory and speed up loading.

By combining quantization with CPU/NVMe offloading, it's possible to maintain a working set of the model inside GPU memory while streaming in rarer layers or experts from CPU memory, all at a fraction of the memory footprint previously needed.

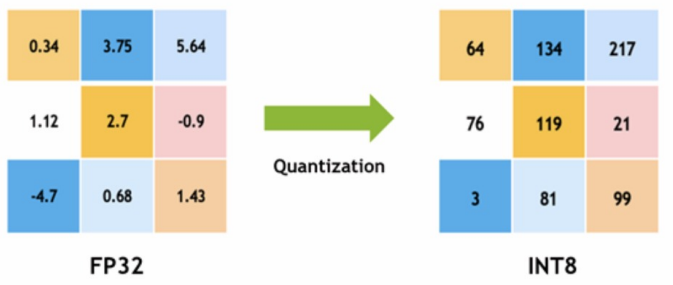

Fig. 5. Quantization from FP32 to INT8 for Memory and Throughput Optimization

D. Dynamic Batching and Caching Mechanisms

Real-world inference scenarios often receive requests asynchronously. To maximize GPU utilization, dynamic batching merges multiple queries into a single GPU batch. Studies show that grouping similar sequence lengths together reduces padding overhead, improving throughput. Dynamic batching logic can happen at the scheduler layer, grouping requests within a small time window.

Another essential optimization is key-value (KV) caching for transformer decoders. While generating long sequences of tokens, each new token depends on previous tokens processed earlier in the session. Instead of recomputing attention over all previous tokens, KV caching stores intermediate states. This reduces per-token cost at the expense of memory for KV caches. Surveys note that FLASHATTENTION and fused kernels further optimize KV cache utilization by reading fewer elements from memory each step [2]. Balancing KV cache size, sequence length, and the number of concurrent sessions is key to maintaining low latency.

E. Pipeline Parallel Inference and Expert Parallel Serving

When inference requests become large (e.g., long prompts or batch responses), pipeline parallelism can also be applied. For instance, certain experiments with GPT-3 or Megatron-LM variants show that pipeline parallelism can break the forward pass into stages that each run on different GPUs. Even though no backpropagation occurs, pipeline parallelism can still improve throughput by allowing stages to process different tokens concurrently. The trade-off is additional complexity in scheduling tokens across stages at inference time.

For MoE models, expert parallelism must be supported during inference. Tokens routed to specific experts require all-to-all communication steps similar to training. To efficiently serve

MoE models, PCC and other communication optimizations ensure that expert parallelism does not bottleneck inference throughput [11]. Case studies reveal that while MoE-based LLMs can be large, their inference latency remains manageable if careful communication planning and token routing caches are implemented.

F. Fault Tolerance, Elasticity, and Reliability

The literature also touches on fault tolerance and elasticity in distributed inference setups. A single node or GPU failure should not bring down the entire inference service. Load balancing and redundant parameter storage on CPU/NVMe can allow quick recovery. Dynamic scaling of GPU workers based on demand—adding or removing GPU nodes as load fluctuates—is suggested as a future research avenue [9]. Although not all surveyed works provide quantitative results, the direction hints at more robust, cloud-native inference architectures that can adapt to varying workloads without incurring large re-initialization costs.

G. Comparing Inference Configurations in Practice

Empirical comparisons are less prevalent for inference than for training. However, some studies present benchmarks for different inference scenarios. For instance, Aminabadi et al. [8] report that using ZeRO-Inference to offload a 530B model to CPU and NVMe achieves a certain tokens-per-second throughput at an acceptable latency, comparing favorably with a hypothetical GPU-only configuration that would require extraordinary memory hardware.

Similarly, Zeng et al. [2] measure latency improvements when applying FLASHATTENTION to inference. The results confirm that memory bandwidth saved during attention computations translates directly into lower latency per token. By combining quantization and pipeline scheduling, these benchmarks show that even trillion-parameter scale inference can serve requests in a fraction of a second.

H. Future Directions in Inference Scalability

As LLMs become standard components in production systems, research increasingly focuses on elasticity, cost optimization, and quality of service guarantees. We may see more integration of advanced caching techniques for prompts, shared KV states among similar queries, and multi-tenant GPU clusters where different models or tasks share the same set of GPUs [10]. Adaptive quantization strategies that tune precision based on the complexity of the query or user importance could further optimize inference efficiency.

Additionally, exploring smarter load balancing across heterogeneous hardware—such as mixing A100 and H100 GPUs, or employing DPUs/SmartNICs for offloading communication tasks—could reduce inference costs. Integrating these ideas with advanced runtime systems that auto-tune batching, quantization levels, and offloading strategies might yield inference services that adapt to incoming workloads in real-time.

I. Summary

Inference at scale for massive LLMs mirrors many complexities of training but brings additional constraints: unpredictable request patterns, strict latency SLAs, and cost-sensitive deployments. The surveyed literature confirms that techniques like ZeRO, FLASHATTENTION, quantization, and pipeline parallelism remain valuable at inference time. Offloading model weights to CPU or NVMe emerges as a critical solution for cost-effective large model serving, allowing for a pay-as-you-go memory model.

While not as extensively benchmarked as training setups, the existing surveys and case studies provide a foundation for understanding how to scale inference. They highlight that careful orchestration of memory, parallelization, and communication is just as important for inference. This ensures that LLMs, even at trillions of parameters, can be deployed in production environments with acceptable latency and throughput, bringing advanced NLP capabilities to real users and applications.

VI. Comparative Analysis and Best Practices

With a myriad of techniques and frameworks available to train and serve extremely large language models, understanding the trade-offs and relative merits of different approaches is crucial. This section attempts to integrate and compare the strategies surveyed in the earlier sections. By drawing on the evaluations presented in studies by Hagemann et al. [5], Zeng et al. [2], and Dash et al. [9], we offer a comparative lens that helps practitioners choose between different parallelization methods, memory optimization schemes, and inference strategies based on their specific constraints and goals.

A. Data, Tensor, and Pipeline Parallelism in Perspective

The fundamental forms of parallelism—data, tensor, and pipeline—serve as the base configuration for large-scale distributed training.

1) Data Parallelism (DP): The simplest approach, easily supported by PyTorch Distributed Data Parallel (DDP), scales linearly with the number of devices when the model fits into a single GPU's memory. However, as models reach billions of parameters, DP alone fails to alleviate memory pressure, forcing smaller batch sizes or the need for gradient accumulation. Literature broadly agrees that DP works best in conjunction with other forms of parallelism.

2) Tensor Parallelism (TP): By slicing layers horizontally, TP reduces per-GPU memory requirements and can maintain high arithmetic intensity for large GeMM operations. Studies by Shoeybi et al. [7] note that TP achieves excellent speedups within a single node due to fast GPU-to-GPU links like NVLink. Yet, TP's efficiency drops off across nodes due to communication overhead. Practitioners find TP most beneficial when model sizes require more memory than DP can handle, but not so large as to span many nodes. For moderate to large models (tens of billions of parameters), TP plus DP may suffice.

3) Pipeline Parallelism (PP): Splitting the model vertically into pipeline stages can drastically reduce per-GPU memory by limiting the number of layers per device. PP scales well across nodes, but requires careful scheduling to minimize pipeline bubbles. Pipeline parallelism shows near-linear scaling when combined with microbatching (1F1B scheduling), though at the cost of complexity in tuning micro-batch sizes and pipeline depths. For models well beyond the memory capability of a single node, combining PP with DP and TP (3D-parallelism) emerges as the dominant approach.

The choice among these parallelism methods depends on hardware topology, model size, and target batch size. Surveys suggest a balanced approach—such as TP $\times$ PP combined with DP and memory optimizations—often achieves the best performance at massive scale [5], [9].

B. ZeRO and Memory Partitioning Techniques

ZeRO stands out as a critical enabler for extreme scale. Studies repeatedly affirm that ZeRO can reduce memory by an order of magnitude, thereby enabling larger batch sizes and better arithmetic utilization. Without ZeRO, model parallelism alone might not suffice to fit trillion-parameter models in GPU memory.

Comparatively, activation checkpointing provides a secondary memory reduction technique at the cost of increased compute time. While ZeRO focuses on static partitioning of states, checkpointing is dynamic and can be selectively applied per-layer. FLASHATTENTION's built-in recomputation further complements ZeRO by reducing the memory footprint of the attention mechanism specifically.

For practitioners, the literature suggests starting with ZeRO Stage 1 or 2 to find a sweet spot in memory footprint, then layering checkpointing if needed. ZeRO Stage 3 (parameter partitioning) offers maximum memory savings but introduces complexity and more frequent synchronization points. DeepSpeed's configuration flexibility allows experiments with different ZeRO stages to pinpoint the best trade-off for a given model and cluster configuration [2], [6].

C. Fused Kernels and FLASHATTENTION

Comparing fused kernels with vanilla implementations reveals consistent benefits, especially for large models where memory bandwidth is a bottleneck. FLASHATTENTION is frequently championed as a must-have optimization for any model requiring long sequences or memory-bottlenecked attention computations [2]. Using fused layernorm, fused dropout, and fused GeMM–bias–gelu layers can collectively yield double-digit percentage speed improvements.

While E.T. and TurboTransformers also present kernel fusion strategies, FLASHATTENTION's comprehensive rethinking of attention stands out as a top recommendation for large-scale deployments. E.T. and others might offer incremental gains, but FLASHATTENTION's IO-aware approach can yield multiplicative benefits.

D. Communication Strategies and PCC

At large scale, communication overhead can nullify the gains from parallelization if not properly managed. The literature extensively compares various collective communication libraries—NCCL, Gloo, MPI—and custom ring-based or tree-based all-reduces. NCCL's efficient GPU-centric design often wins for NVIDIA GPU clusters [4].

For MoE models, parallelism coordinated communication (PCC) decomposes global all-to-all operations into a sequence of local all-to-all plus all-gather steps. The literature highlights PCC as crucial for scaling beyond 128 GPUs with MoE layers, drastically cutting latency compared to naive global all-to-all [11].

While no single communication strategy fits all models, some best practices emerge:

- Use hierarchical collectives (intra-node first, inter-node second) for large node counts.
- Overlap communication with computation whenever possible.
- For MoE, apply PCC to exploit replication introduced by tensor parallelism before performing large global collective operations.

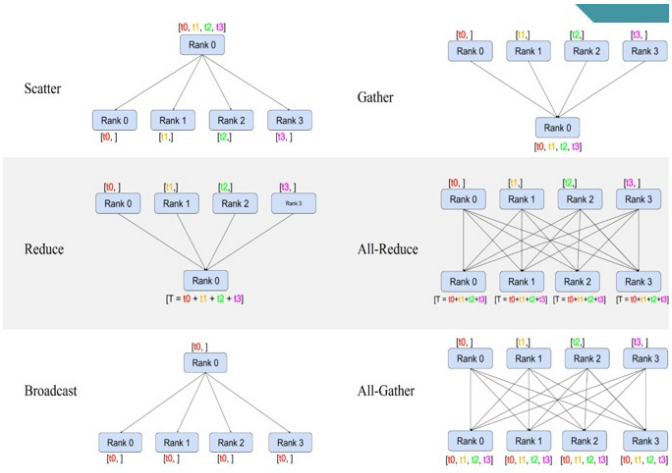

Fig. 6. Collective Communication Strategies: Scatter, Gather, Reduce, and All-Reduce for Distributed Systems

E. Quantization and Mixed Precision: When and How

Mixed precision is ubiquitous due to its near-zero accuracy cost and significant speedup. Comparing FP16, BF16, and INT8 shows that BF16 is often more stable numerically, while INT8 can drastically reduce memory but may require careful calibration to avoid accuracy drops [10].

The literature advises using BF16 or FP16 for training stability and resorting to INT8 primarily at inference. For inference, the literature confirms quantization as a powerful tool to reduce latency. INT8 or even 4-bit quantization techniques are being explored. Deciding on the quantization level depends on task sensitivity to slight accuracy losses.

Without comprehensive comparative benchmarks across quantization methods for a wide range of models and tasks, most researchers proceed conservatively—BF16 for training, INT8 or lower for inference if latency is critical and slight accuracy degradation is acceptable.

F. Inference-Specific Comparisons

Inference scenarios have their own distinct comparisons. CPU-offloading vs. GPU-only inference is a key decision factor influenced by cost and model size. When comparing GPU-only inference for a 175B parameter model to CPU-offloaded inference, Aminabadi et al. [8] show that CPU+NVMe offloading might add a few milliseconds in latency but allow the model to run on smaller, cheaper GPU configurations. If absolute lowest latency is needed, all-GPU memory footprints are best, but at a higher hardware cost. The trade-off between cost-effectiveness and raw performance is a recurring theme.

Dynamic batching strategies differ in their complexity and performance. Some systems batch requests by arrival time to maximize GPU utilization, while others rely on user-defined batching intervals. Literature shows a small overhead for dynamic batching logic is well worth the gains in GPU utilization, especially under bursty traffic patterns [6]. In some inference frameworks, minimal overhead dynamic batching can achieve near-linear improvement in throughput when multiple requests arrive concurrently.

G. Gaps and Unanswered Questions

While many solutions have been proposed, direct head-to-head comparisons remain sparse. Few studies offer standardized benchmarks that test multiple methods on the same model, hardware, and dataset configuration. As a result, practitioners must piece together insights from disparate sources. This is a known gap highlighted in reviews like Dash et al. [9] and Hagemann et al. [5], calling for community-driven benchmarking suites.

Questions also remain about non-performance metrics like energy efficiency, fault tolerance, and developer productivity. While some papers note that certain parallelization schemes require extensive code changes, others claim minimal modifications. Without a standard measure, these claims are anecdotal. Similarly, energy consumption comparisons among different parallelization strategies remain underexplored.

H. Practical Recommendations

From the surveyed literature, a set of recommendations emerges:

1) For billion-scale models on single node: Combine DP with TP, add ZeRO if memory-bound, and fuse kernels for incremental gains.
2) For multi-node, hundred-billion parameter scale: Introduce PP to reduce per-GPU memory, rely on ZeRO Stages 2 or 3 to minimize memory footprint, and use FLASHATTENTION for improved throughput.
3) For trillion-scale MoE models: Leverage expert parallelism with PCC, heavy reliance on ZeRO Stage 3, and possibly CPU/NVMe offloading for cost-effectiveness.
4) For inference: Choose CPU/NVMe offloading if cost is a concern and absolute lowest latency is not mandatory. If latency critical, keep model on GPU memory and consider INT8 quantization plus FLASHATTENTION.

I. Summary

Comparing the plethora of techniques from literature reveals a landscape rich with options but short on universal solutions. Practitioners must weigh memory constraints, communication overhead, compute intensity, development complexity, and cost. ZeRO stands out as the linchpin for unlocking the largest models, while FLASHATTENTION emerges as a critical optimization for attention-heavy workloads. Pipeline and tensor parallelism complement data parallelism to scale both memory and compute efficiently. Though the literature offers myriad solutions, final choices still depend on the model, infrastructure, and performance goals. The next section concludes by discussing overall lessons learned and future directions that could standardize and simplify these choices.

VII. CONCLUSIONS, LESSONS LEARNED, AND FUTURE DIRECTIONS

As Large Language Models break new boundaries in scale, complexity, and capability, distributed training and inference techniques have moved from being optional optimizations to indispensable components of the modern NLP research and production toolkit. This final section consolidates the main insights from the literature survey, reflecting on what we have learned about scaling models with hundreds of billions or trillions of parameters, and discussing open challenges that remain. We also propose directions for future research that could lead to more automated, resilient, and cost-efficient distributed systems.

A. Key Lessons and Synthesis

Scale Demands Multiple Parallelization Strategies: The literature makes it clear that no single form of parallelism suffices for extreme scale. While data parallelism was standard for moderate-sized models, the largest LLMs require model parallelism (tensor or pipeline) for memory reasons. MoE models add expert parallelism to the mix. Successful configurations often combine data, tensor, pipeline, and possibly expert parallelism to strike a balance between memory footprint, arithmetic intensity, and communication overhead.

Memory Optimization as a Keystone: Memory constraints drive much of the innovation. Techniques like ZeRO have fundamentally altered what is possible, enabling trillion-parameter training on current hardware. Activation checkpointing, FLASHATTENTION, and fused kernels also play pivotal roles. Without these memory- and bandwidth-saving strategies, scaling would stall. Practitioners have learned to treat memory as a first-class design constraint, not an afterthought.

Communication and Kernel Efficiency Matter: As GPUs scale out into hundreds or thousands of devices, even small inefficiencies in communication patterns and kernel launches multiply into significant slowdowns. FLASHATTENTION and

fused kernels ensure that the GPU's memory bandwidth and compute units are utilized effectively, while PCC and topology-aware rank mappings mitigate communication bottlenecks. The interplay of these techniques can yield linear or near-linear scaling [5].

Inference Requires Similar but Distinct Solutions: Although much effort has focused on training, inference also requires careful memory and communication strategies. CPU/NVMe offloading and quantization allow serving massive models on limited hardware resources. Dynamic batching and caching further improve throughput and reduce latency. These techniques ensure that the research gains of large models can be practically deployed [2], [8].

B. Challenges and Limitations in Current Approaches

Complexity and Tuning Overhead: Configuring a largescale distributed training run involves choosing parallelism degrees, ZeRO stages, checkpointing layers, fused kernels, and quantization settings. Without comprehensive guidelines or automated tools, extensive trial-and-error is common. Many surveyed papers admit that achieving optimal performance is non-trivial, reducing accessibility for researchers with limited resources [6], [9].

Lack of Standardized Benchmarks and Comparisons: Researchers often report results on different models, hardware, and evaluation metrics, complicating direct comparisons. The community would benefit from standardized suites that test a range of configurations on well-defined hardware and datasets. Such benchmarks could clarify how different methods stack up in terms of speed, memory use, energy consumption, and developer effort [5].

Beyond Performance: While speed and scalability dominate the discourse, other aspects like energy efficiency, fault tolerance, and elasticity remain underexamined. With training runs consuming thousands of GPU-hours, the environmental impact warrants attention. Reliability under device failures or dynamic cluster conditions also matters for production deployments, yet the literature often glosses over these complexities.

C. Novelty in Our Focus: ZeRO, FLASHATTENTION, Expert Parallelism, and Trillion-Scale Models

The novelty of our survey's focus lies in the comprehensive coverage of cutting-edge and model-specific optimization techniques. For instance, ZeRO's staged optimizations have been extensively studied, but few surveys integrate these findings with parallelism-coordinated communication strategies or MoE models' expert parallelism. FLASHATTENTION, a relatively new kernel-level innovation, has not yet seen wide treatment in general distributed training literature, especially in the context of trillion-scale models. By examining FLASHATTENTION and fused kernel approaches alongside ZeRO, we provide a more current perspective that accounts for the rapidly evolving LLM ecosystem.

Sparse architectures like Mixture-of-Experts and their accompanying expert parallelism strategies remain niche topics in most overviews. Our survey highlights how these techniques fit into the broader distributed training landscape and how recent works coordinate parallelism methods to handle sparse token routing efficiently. We also contextualize these findings against tried-and-true methods like pipeline and tensor parallelism, showing how innovation at the boundary of memory and communication can propel trillion-scale training.

D. How the Survey's Conclusions Differ from Prior Literature

While previous literature often suggests that no single parallelization scheme dominates for all scenarios, many older surveys stop short of providing practical, integrated guidelines. Our survey, on the other hand, synthesizes a set of best practices. For example, where older works might conclude that model parallelism is beneficial but tricky to implement, we break down how ZeRO and FLASHATTENTION reduce complexity and overhead, allowing developers to push beyond previous limits. Similarly, while earlier analyses acknowledge communication as a bottleneck, we highlight concrete solutions like PCC (Parallelism-Coordinated Communication) [11] that actively reduce these overheads.

We also embrace the inference perspective more fully than many previous surveys. While prior literature focuses heavily on training, we emphasize that similar distributed strategies, combined with CPU/NVMe offloading and quantization, apply equally to large-scale inference. Thus, we provide an integrated view that not only differs from prior training-centric reviews but also enriches the conversation by including serving frameworks and latency-sensitive applications.

In summary, our survey stands out by weaving together recent breakthroughs—ZeRO, FLASHATTENTION, expert parallelism—and linking them to both training and inference scenarios. We translate these technical contributions into actionable insights, distinguishing our conclusions as more directly prescriptive and practice-oriented than some previous, more theoretical or narrowly focused works.

E. Self-Critique and Lessons Learned

From a methodological standpoint, the surveyed works affirm that no single approach—be it pure data parallelism or a single model-parallel scheme—suffices at extreme scales. Instead, hybrid and tailored solutions, combining pipeline parallelism with tensor slicing, or mixing ZeRO with FLASHATTENTION and offloading strategies, produce the best results. Communication emerges as a principal performance determinant, and advanced solutions like PCC highlight that an awareness of model structure and parallelization scheme can reorder collective operations for reduced latency.

An essential lesson is the importance of practical usability. While researchers have made extraordinary progress in scaling models, the complexity of these systems still presents a high barrier. Shen Li et al. [1] describe experiences accelerating data parallel training using PyTorch Distributed. These experiences emphasize that building robust, user-friendly abstractions is as critical as the underlying algorithms. Our small experimental DistilBERT setup exemplifies how a scaled-

down, hands-on scenario can validate fundamental principles before tackling trillion-parameter behemoths.

F. Promising Directions for Future Research

- Autotuning and Compiler-assisted Strategies: Systems like Alpa, Colossal-AI, and ML compilers hint at a future where users specify high-level requirements, and the system automatically chooses the best parallelization and memory optimization strategy [5], [9]. Advances in graph partitioning, cost modeling, and ML-based search could yield autotuners that adapt to hardware topology and model architecture in real-time.
- Unified Frameworks for End-to-End Pipelines: Training and inference currently use overlapping but distinct toolchains. Unifying these phases under a single framework that handles training, fine-tuning, serving, and even continuous updates would simplify workflows. Future systems might store model parameters and experts in globally accessible memory pools and dynamically allocate GPU memory as requests arrive [8].
- Energy and Cost Optimization: As LLM usage surges in production, the cost and environmental footprint become pressing. Researchers could investigate energy-aware parallelization that selects layouts to minimize power draw or allocate fewer GPUs during periods of low load. Combined with quantization and CPU-offloading strategies, these methods could create greener, more cost-efficient deployments [10].
- Advanced Communication Patterns: Beyond hierarchical all-reduces or PCC, future methods might use specialized hardware like DPUs or programmable switches to offload certain communication tasks. Research into compressed communication or asynchronous synchronization strategies might yield improvements in tail latency, smoothing out stragglers and achieving more predictable performance at scale [4], [6].
- Heterogeneous, Multi-Tenant Clusters: Cloud providers increasingly offer heterogeneous GPU fleets. Training or serving LLMs across a mixture of GPU generations, memory sizes, and interconnect speeds challenges current assumptions. Future solutions might dynamically detect hardware heterogeneity and partition tasks accordingly, perhaps using older GPUs for memory-intensive but latency-tolerant operations and newer GPUs for compute-heavy sections [2].

G. Summary

In conclusion, distributed training and inference techniques have matured significantly. They have elevated LLMs from large but still manageable networks to computational leviathans that transform how we approach NLP. The continuous interplay of memory, communication, and compute optimizations illustrates a field in rapid evolution. As new frameworks emerge, best practices stabilize, and benchmarks standardize, we can anticipate a more transparent and user-friendly ecosystem. This progression will empower even more researchers, developers, and industries to harness the full potential of LLMs, spurring innovation and expanding the horizons of AI capabilities.

VIII. PROJECT EVOLUTION

Initially, our literature survey concentrated on the three primary parallelism techniques—data, model, and pipeline parallelism—and the ZeRO optimizer. However, feedback from our preliminary presentation indicated that this focus, coupled with a limited experimental setup, did not sufficiently justify the scope of our semester project. In response, we dedicated the following week to expanding our survey to include memory and communication optimizations, efficient inference techniques such as FLASHATTENTION and checkpointing mechanisms, serving massive Large Language Models (LLMs), and kernel-level enhancements. This comprehensive approach provides a holistic view of distributed training for LLMs, addressing the initial limitations and ensuring a more robust and well-justified analysis.

IX. CONTRIBUTION

A. Manoj's Contribution (60 percent)

Manoj played a pivotal role in the development and execution of this literature survey, contributing approximately 60 percent of the overall effort. His primary responsibilities included:

Comprehensive Literature Review: Manoj conducted an extensive review of existing research on distributed training methodologies, memory optimization strategies, and inference techniques for LLMs. This involved identifying, sourcing, and synthesizing key papers, frameworks, and case studies from reputable journals and conferences.

In-Depth Analysis of Distributed Training Techniques: He led the analysis of foundational and advanced parallelization strategies, including data parallelism, tensor parallelism, pipeline parallelism, and expert parallelism in Mixture-of-Experts (MoE) models. Manoj evaluated how these techniques address scalability and efficiency challenges in training large-scale models.

Memory Optimization Strategies: Manoj delved into memory-efficient training frameworks, with a particular focus on the Zero Redundancy Optimizer (ZeRO) and its various stages. He explored how ZeRO partitions optimizer states, gradients, and parameters to minimize memory overhead and enable the training of trillion-parameter models.

Kernel-Level Enhancements: He examined the role of fused kernels and FLASHATTENTION in optimizing computational efficiency and reducing memory bandwidth usage. Manoj assessed how these kernel-level improvements contribute to overall training speed and model scalability.

Comparative Analysis and Best Practices: Manoj synthesized findings from multiple studies to develop a comparative analysis of different parallelization and optimization techniques. He identified best practices and formulated guidelines for selecting and combining strategies based on model size, hardware configurations, and performance requirements.

Development of Key Sections: Manoj was primarily responsible for drafting and refining Sections III (Distributed Training Techniques: Surveys and Case Studies), IV (Memory and Communication Optimizations), and VI (Comparative Analysis and Best Practices). His thorough understanding of the subject matter ensured that these sections were detailed, coherent, and aligned with the survey's objectives.

Overall Project Coordination: He coordinated the overall structure of the survey, ensuring that each section logically flowed into the next. Manoj also integrated feedback from Shubham and other reviewers to enhance the clarity and depth of the analysis.

B. Shubham's Contribution (40 percent)

Shubham significantly supported the project, contributing approximately 40 percent of the overall effort. His key responsibilities included:

Assistance in Literature Compilation: Shubham assisted in sourcing and organizing relevant literature, ensuring comprehensive coverage of key studies and advancements in the field. He played a crucial role in maintaining an up-to-date and well-structured reference database.

Development of Inference and Serving Sections: He took the lead in drafting Sections V (Inference and Serving Massive Models), focusing on CPU/NVMe offloading strategies, quantization techniques, and their applications in real-time serving scenarios. Shubham explored how training optimizations translate to inference efficiencies, providing valuable insights into scalable deployment practices.

Creation of Visual Aids and Diagrams: Shubham developed and curated visual materials, including diagrams and flowcharts, to illustrate complex concepts such as ZeRO partitioning, FLASHATTENTION mechanisms, and distributed inference architectures. These visuals enhanced the survey's readability and helped convey intricate ideas more effectively.

Support in Comparative Analysis: He contributed to Section VI (Comparative Analysis and Best Practices) by compiling performance metrics from various studies and assisting in the creation of comparative tables and charts. Shubham helped identify performance trade-offs and supported the formulation of best practices based on empirical data.

Refinement of Related Work and Conclusion: Shubham played a key role in drafting Section VII (Related Work) and assisting with Section VIII (Conclusion). He ensured that the survey's findings were accurately contextualized within the broader research landscape and that the conclusions drawn were well-supported by the surveyed literature.

Quality Assurance and Editing: He meticulously reviewed drafts for clarity, coherence, and correctness, providing constructive feedback to improve the overall quality of the survey. Shubham ensured that all sections were well-integrated and that the survey maintained a consistent narrative flow.

REFERENCES

[1] S. Li et al., "Pytorch distributed: Experiences on accelerating data parallel training," in Proceedings of the VLDB Endowment, vol. 13, no. 12. VLDB, 2020, pp. 3005–3018.

[2] S. Zeng et al., "Exploring memorization in fine-tuned language models," in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). ACL Anthology, 2024, pp. 3917–3948.

[3] S. Rajbhandari et al., "Zero: Memory optimizations toward training trillion parameter models," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20). AIICHIRO NAKANO, 2020, pp. 1–16.

[4] W. Li et al., "Understanding communication characteristics of distributed training," in Proceedings of the 8th Asia-Pacific Workshop on Networking (APNet '24). MIT CSAIL People, 2024, pp. 1–8.

[5] J. Hagemann et al., "Efficient parallelization layouts for large-scale distributed model training," arXiv preprint arXiv:2311.05610, 2023, oPENREVIEW.

[6] J. Duan et al., "Efficient training of large language models on distributed infrastructures: A survey," arXiv preprint arXiv:2407.20018, 2024, aRXIV.

[7] M. Shoeybi et al., "Using deepspeed and megatron to train megatron-luring nlg 530b, a large-scale generative language model," arXiv preprint arXiv:2201.11990, 2022, aRXIV EXPORT.

[8] R. Y. Aminabadi et al., "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22). IEEE XPLORE, 2022, pp. 1–14.

[9] S. Dash et al., "Optimizing distributed training on frontier for large language models," arXiv preprint arXiv:2312.12705, 2023, aRXIV.

[10] B. Hanindhito et al., "Bandwidth characterization of deepspeed on distributed large language model training," in Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE XPL, 2024, pp. 241–256.

[11] S. Rajbhandari et al., "Zero: Memory optimizations toward training trillion parameter models," 2022.

[12] J. Rasley et al., "Zero: Memory optimizations toward training trillion parameter models," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20). AIICHIRO NAKANO, 2020, pp. 1–16.