Efficient Training of Large Language Models on Distributed Infrastructures: A Survey

Jiangfei Duan*, Shuo Zhang*, Zerui Wang*, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, Xipeng Qiu, Dahua Lin, Yonggang Wen, Xin Jin, Tianwei Zhang and Peng Sun✉

Abstract— Large Language Models (LLMs) like GPT and LLaMA are revolutionizing the AI industry with their sophisticated capabilities. Training these models requires vast GPU clusters and significant computing time, posing major challenges in terms of scalability, efficiency, and reliability. This survey explores recent advancements in training systems for LLMs, including innovations in training infrastructure with AI accelerators, networking, storage, and scheduling. Additionally, the survey covers parallelism strategies, as well as optimizations for computation, communication, and memory in distributed LLM training. It also includes approaches of maintaining system reliability over extended training periods. By examining current innovations and future directions, this survey aims to provide valuable insights towards improving LLM training systems and tackling ongoing challenges. Furthermore, traditional digital circuit-based computing systems face significant constraints in meeting the computational demands of LLMs, highlighting the need for innovative solutions such as optical computing and optical networks.

Index Terms—Large Language Models; Distributed Training; Machine Learning Systems.

✦

1 INTRODUCTION

Large Language Models (LLMs) are transforming the AI industry, demonstrating remarkable capabilities across a wide range of tasks and applications, including personal assistants [1], code copilot [2], chip design [3] and scientific discovery [4]. The success of this revolution is built on the unprecedented scale of transformer-based LLMs like GPT [5], LLaMA [6], Gemini [7], etc. Moreover, it is evidenced that the scaling of LLMs has not plateaued [8]. This trend has significantly shifted the design of underlying training systems and infrastructures, since LLM typically follows a relatively fixed architecture and its training exclusively occupies huge GPU clusters over extended periods. For example, the pre-training of LLaMA-3 takes approximately 54 days with 16K H100-80GB GPU on Meta's production cluster [9].

LLM training highlights significant challenges for today's training system and infrastructure in terms of "SER", i.e., *Scalability, Efficiency*, and *Reliability*. Scalability demands

- Jiangfei Duan and Dahua Lin are with Shanghai AI Laboratory and the Chinese University of Hong Kong.
  E-mail: {dj021, dhlin}@ie.cuhk.edu.hk
- Shuo Zhang and Xipeng Qiu are with Shanghai AI Laboratory and Fudan University.
  E-mail: {zhangshuo, qiuxipeng}@pjlab.org.cn
- Zerui Wang is with Shanghai AI Laboratory and Shanghai Jiao Tong University.
  E-mail: wangzerui@pjlab.org.cn
- Qinghao Hu, Yonggang Wen and Tianwei Zhang are with Nanyang Technological University.
  E-mail: {qinghao.hu, ygwen, tianwei.zhang}@ntu.edu.sg
- Xin Jin is with School of Computer Science, Peking University.
  E-mail: xinjinpku@pku.edu.cn
- Lijuan Jiang, Wenwen Qu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang and Peng Sun are with Shanghai AI Laboratory.
  E-mail: {jianglijuan, quwenwen, wangguoteng, wengqizhen, yanhang, zhangxingcheng, sunpeng}@pjlab.org.cn
- * Equal Contribution.

that both infrastructure and systems adapt seamlessly to massive clusters of tens of thousands of GPUs or AI accelerators, while preserving training correctness and model accuracy. This requires innovative solutions in hardware configuration, networking, and training framework. Efficiency focuses on maximizing resource utilization across the entire cluster, often measured by Model FLOPs Utilization (MFU). Achieving high MFU involves optimizing computation, minimizing communication overhead, and efficiently managing memory at an unprecedented scale. Reliability are critical given the prolonged duration of LLM training, usually lasting weeks to months. The system must maintain consistent performance and be resilient to various types of failures, including hardware malfunctions, network issues, and software errors. It should be capable of swiftly detecting and recovering from these failures without significant loss of progress or training quality. These interrelated challenges necessitate a holistic approach to system and infrastructure design, pushing the boundaries of large-scale distributed computing and opening new avenues for research and innovation in high-performance machine learning systems.

This survey paper aims to provide a comprehensive overview of the advancements in LLM training systems and infrastructure, addressing the aforementioned challenges. This survey spans from the distributed training infrastructure to training systems. We examine innovative approaches to infrastructure design, covering advancements in GPU clusters, high-performance networking, and distributed storage systems tailored for LLM workloads. We also explore the key aspects of distributed training systems, including parallelism strategies, computation, communication and memory optimizations that enhance the scalability and efficiency. We also delve into fault tolerance mechanisms for improving training reliability. By synthesizing recent advancements and identifying future research direc-
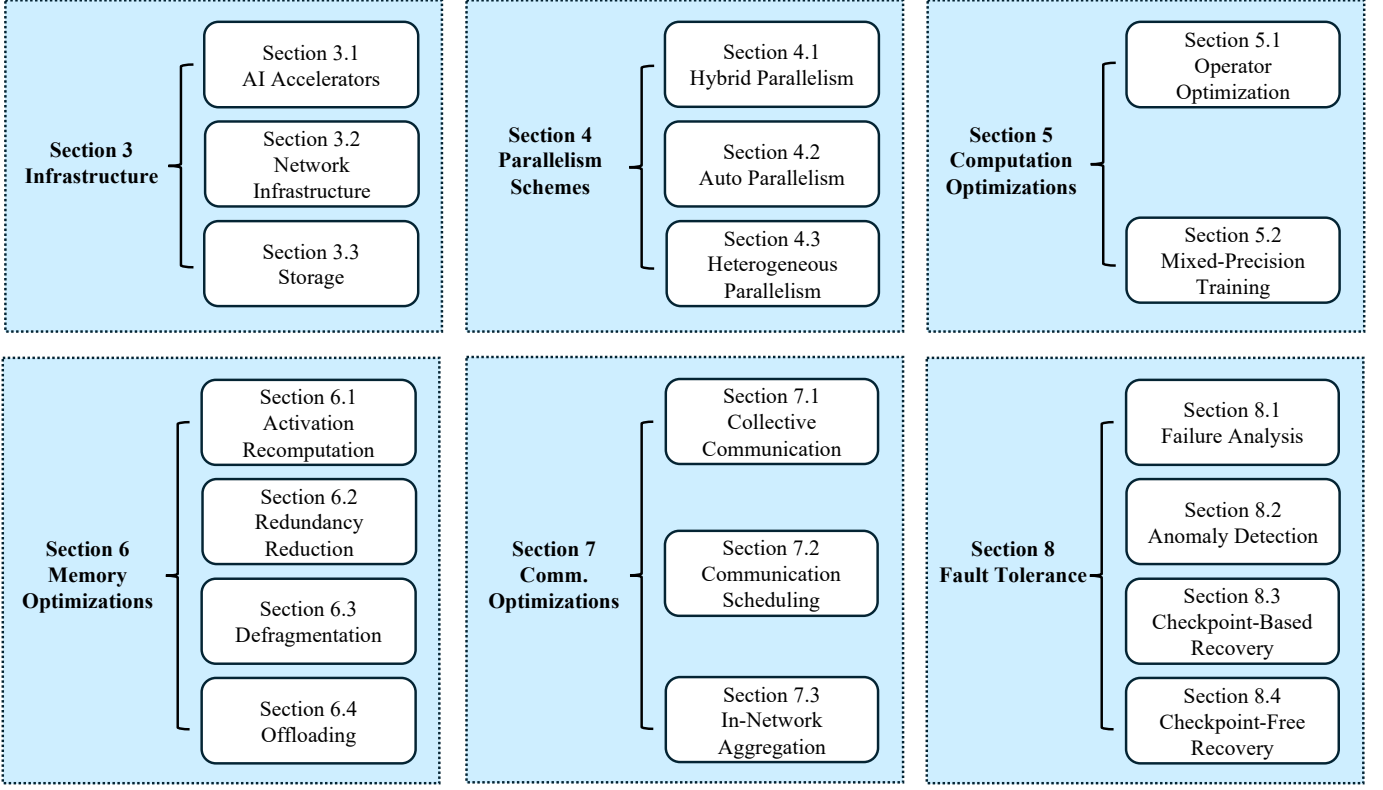
Fig. 1: Overall structure of this survey.

tions, this survey aims to provide researchers and practitioners with insights into the most promising avenues for improving LLM training systems. Our goal is to offer a valuable resource that not only addresses current challenges but also paves the way for future innovations in large-scale machine learning infrastructure.

Organization. Fig. 1 shows the organization of this survey. Section 2 discusses the background information about LLM architecture, characteristics and challenges of LLM training. In Section 3, we summarize the key aspects of the training infrastructure, including AI accelerators, network infrastructure and storage systems. In Section 4, we investigate the parallelism schemes for distributed LLM training. In Section 5, we discuss computation optimizations to harness the unprecedented computational capabilities. In Section 6, we discuss techniques to optimize the memory footprint in LLM training. In Section 7, we introduce communication optimizations to minimize the communication overhead. In Section 8, we first present a failure analysis, the cover approaches to enable fast failure detection and recovery. Finally, we conclude the survey in Section 9.

2 BACKGROUND

2.1 Transformer-based LLMs

The current state-of-the-art Large Language Models (LLMs) are predominantly transformer-based. Their core architecture is built around the attention mechanism [10], which allows the model to dynamically weigh the importance of different words in a sentence. Fig. 2 depicts the typical architecture of a transformer layer [10], which can be stacked multiple times to construct an LLM. The input text is first tokenized into individual tokens, which are then converted into a token vector X via an embedding layer. To preserve the sequential nature of the text, the token vector is embedded with positional information. The resulting token vector is then fed into the transformer layer, which consists of an Attention block and a Feed-Forward Neural Network (FFN) block.

Suppose the input token vector is $X = [x_1, x_2, \cdots, x_n]$. These tokens are first transformed into query Q, key K and value V tensors via linear transformation. The attention mechanism computes the attention output as follows:

$$\text{Attention}(Q, K, V) = \texttt{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V \qquad (1)$$

where d is the dimension of the key tensor. This formula ensures that the LLM can focus on relevant parts of the input sequence by calculating a weighted sum of the values, where the weights are derived from the similarity between the queries and keys. After the attention layer, the output is passed through a FFN for further processing.

Nowadays, LLMs generally follow the original decoder-only transformer architecture but incorporate modifications to the attention mechanism and FFN to enhance efficiency and performance. The original attention mechanism, known as Multi-Head Attention (MHA) [10], suffers from quadratic computational complexity and high memory consumption due to the key-value cache. To address these issues, several variants such as Multi-Query Attention (MQA) [11], Group-Query Attention (GQA) [12] and Multi-Latent Attention (MLA) [13] have been proposed. A notable advancement of the FFN component is the Mixture-of-Experts (MoE) [14],
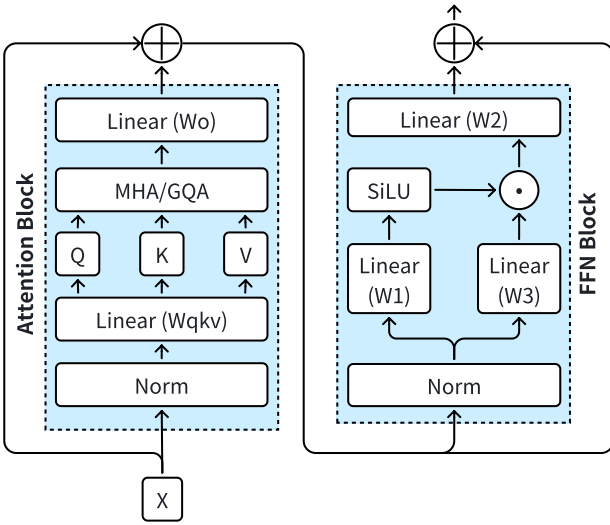
Fig. 2: A typical Transformer layer contains an Attention block and a Feed-Forward Network (FFN) block.

[15] architecture, which employs a sparsely activated FFN. In MoE, only a subset of the FFN layers (or experts) are activated for each input, significantly reducing the computational load while maintaining high model capacity.

2.2 LLM Training Workloads Characteristics

The characteristics of LLM training workloads diverge significantly from traditional deep learning workloads, primarily due to their complexity and scale. These unique characteristics affect training system design, performance, scalability, and resource utilization. Here, we highlight key differences and requirements for LLMs.

(1) Homogeneous Model Architecture. Unlike prior DL workloads that employed various model architectures (e.g., LSTM [16], CNN [17]) for different tasks, LLMs predominantly use the Transformer architecture [10]. Models like GPT [5], LLaMA [6], InternLM [18] and MOSS [19] all share this common foundation. This architectural uniformity presents significant potential for optimizing system performance for a particular model architecture.

(2) Unprecedented Scale and Training Duration. LLM training operates at an unparalleled scale, typically updating models with hundreds of billions of parameters with terabyte-scale training datasets. Such magnitude necessitates distributed training across huge GPU clusters and presents challenges in maintaining high efficiency. Besides, the training of LLMs can span weeks or months, demanding robust fault tolerance mechanisms and efficient checkpointing strategies to safeguard against data loss and facilitate the resumption of interrupted training sessions.

(3) Specialized Software Optimization. To accommodate the enormous model size of LLMs, specialized systems implement advanced techniques to optimize execution. For example, Megatron [20] and Alpa [21] accelerate training through hybrid parallelism. DeepSpeed [22] reduces memory consumption by integrating state-sharding optimizers.

(4) Shift in Training Paradigm. Traditional DL workloads follow a task-specific paradigm, training models on domain-specific data for particular tasks, such as translation. In contrast, LLMs adopt a self-supervised training approach on extensive datasets to create foundation models, which are then adapted for various downstream tasks. This paradigm shift represents a substantial change in the model development pipeline, including pretraining and alignment phases, and results in distinct workload characteristics compared to previous DL workloads. From the datacenter perspective, LLM development involves numerous small-scale workloads that are associated with pretraining, including alignment (i.e., fine-tuning) and periodical evaluation workloads [23].

2.3 LLM Training Challenges

The unique characteristics of LLM training workloads give rise to significant challenges in developing efficient training systems and infrastructure. These challenges primarily manifest in three critical areas: *scalability, efficiency,* and *reliability*. Each of these challenges stems directly from the massive scale of LLMs and the complexity of their training processes, requiring innovative solutions that push the boundaries of distributed computing and machine learning systems. Below, we detail these challenges and their implications for LLM training:

(1) Scalablity. The success of LLMs is largely attributed to their scale, with performance often improving as LLMs grow larger [8]. However, the scaling of model size introduces substantial scalability challenges, since training LLMs requires increasingly large clusters of GPUs or specialized AI accelerators. First, building scalable infrastructure that can provide massive computations and memory capacity is essential. This involves designing and deploying large clusters of GPUs or specialized AI accelerators, high-performance networking to connect these devices, and distributed storage systems capable of handling enormous datasets and model checkpoints. The challenge lies in ensuring that these components work together efficiently at scale, managing heat dissipation, power consumption, and hardware failures in large-scale deployments. Second, designing scalable training systems that can effectively utilize massive accelerators in parallel is crucial. This includes designing parallelization strategies and communication algorithms that can achieve near-linear scalability with thousands of accelerators while maintaining consistent LLM accuracy.

(2) Efficiency. The enormous computational requirements of LLM training translate to high training costs, making it imperative to maximize the efficiency of hardware and software systems. The efficiency can be measured with MFU (Model FLOPs Utilization), which quantifies how effectively the system uses available computing resources. However, achieving high efficiency at scale remains a significant challenge. For instance, LLaMA3 achieves only 38% to 41% MFU on 16K GPUs [9], highlighting the difficulty in maintaining high utilization as systems scale. Maximizing efficiency demands the optimization in parallelism, computation, communication and memory. First, the parallelism of distributed LLM training demands careful design to minimize communication demands. Second, optimized computation operators and lower precision arithmetic are crucial

to achieve high GPU FLOPS utilization. Third, communication overhead needs to minimized to reduce GPU idle time. Finally, efficient memory optimizations are required to hold LLMs in existing hardwares and reduce FLOPs waste of recomputation.

(3) Reliability. Ensuring the reliability of LLM training over extended periods is paramount. As training jobs can span weeks or months on large clusters of tens of thousands of GPUs, the probability of training failures increases, necessitating fast failure detection and recovery mechanism for resilient LLM training. First, LLM training jobs may crash due to various errors, making it hard to identify the exact fault reason across tens of thousands GPUs quickly. Second, the hang of LLM training jobs results in all GPUs becoming idle due to the synchronous nature of training, resulting in significant waste. Moreover, some intricate anomalies, like redundant link failures or stragglers, may not cause immediate crashes but can lead to training slowdowns. This instability can result in reduced training efficiency. To tackle these challenges, robust anomaly detection systems capable of detecting both catastrophic failures and performance degradations are essential. Additionally, implementing fault-tolerant training frameworks that can seamlessly handle node failures and network issues is crucial.

2.4 Related Survey

This work focuses on the efficient training system and infrastructure of transformer-based LLMs, including the design of underlying distributed infrastructure, paradigm of parallelism, optimizations of computation and communication, efficient memory management and resilience of training system. We also investigate the efficient training systems of emerging workloads such as MoE, a promising efficient LLM variant, and fine-tuning, a necessary stage to align the capabilities of LLMs. However, this work does not cover the evolution of promising LLM architectures [24], [25] and the algorithms for training [26], instruction tuning [27] and alignment [28] towards powerful and safe LLMs. While previous works [29]–[31] have discussed some aspects of LLM training systems, their primary focus was not on the design of efficient training systems and infrastructure. Wan et al. [29] aims to provide a holistic view of efficient LLM advances in model- and data-centric methods. Liu et al. [30] covers the training and inference deployment techniques of LLMs. Xu et al. [31] targets on discussing the resource-efficient strategies for LLM development in both algorithmic and systemic aspects. This work also discuss the approaches for quantized LLM training and efficient LLM fine-tuning, but we focus on the systemic approaches. The algorithmic approaches to compress and fine-tune LLMs are discussed by Zhu et al. [32] and Han et al. [33]. The discussion scope of this work does not include advanced optimization algorithms [34] and distributed DNN training systems [35]. While Liang et al. [36] have extensively reviewed auto parallelism approaches, their focus is on general DNNs rather than LLMs specifically.

3 INFRASTRUCTURE FOR LLM TRAINING

In this section, we explore the infrastructure design for training LLMs, encompassing accelerators, networks, storage,
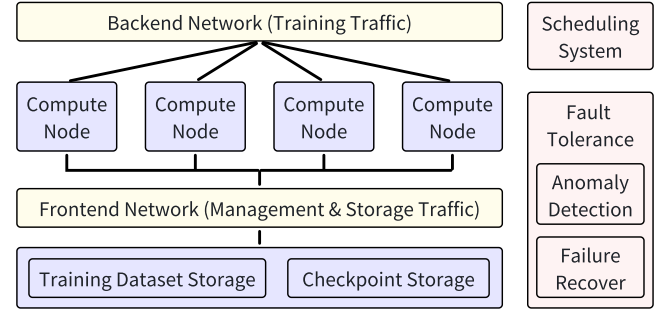

Fig. 3: Infrastructure overview for distributed LLM training.

and scheduling systems (Fig. 3).

3.1 AI Accelerators

The rapid progress of LLMs is significantly benefiting from the evolution of GPUs and AI accelerators, which are crucial for improving the model training performance.

3.1.1 NVIDIA Graphics Processing Unit (GPU)

NVIDIA GPUs have become an essential component for distributed LLM training due to their superior ability to handle parallel computations. These processors are built with a multitude of compact, high-efficiency cores that can execute numerous tasks simultaneously. The design of GPUs is ideally matched for the matrix and vector operations in LLM training. They offer support for various numerical precision formats such as FP32, TF32, FP16, BF16, FP8, INT8, and even FP4. This allows researchers to well balance the training speed and accuracy, making LLM training more efficient [110]. NVIDIA's GPU programming language (i.e. CUDA) makes it easier to manage how tasks are split and processed in parallel on GPUs. This helps researchers harness the full power of GPUs for training advanced LLMs.

A typical GPU comprises an array of Streaming Multiprocessors (SMs), with each SM housing several cores that share an instruction unit but are capable of executing distinct threads in parallel. The shared memory within each SM allows for effective data exchange and synchronization among threads, which is essential for optimizing the memory access patterns required for LLM computations. Furthermore, GPUs are furnished with high-bandwidth memory (HBM), which accelerates data transfer and mitigates memory access bottlenecks in computationally intensive tasks. The latest GPU architectures, such as NVIDIA's Ampere [37], Hopper [38] and Blackwell [39], are continuously pushing the boundaries of LLM computation. They offer enhanced memory bandwidth and capacity, increased floating-point operations per second (FLOPS), and specialized mixed-precision computing units like Tensor Cores. Notably, NVIDIA's Hopper architecture introduces a significant advancement with the Transformer Engine [111], a feature that leverages mixed FP8 and FP16 precisions to expedite the training of Transformer-based LLMs.

3.1.2 Other AI Accelerators

Distributed LLM training on AMD GPUs has become a reality, particularly on Frontier [112], the world's first exas-

AI Accelerators	NVIDIA GPUs	Ampere [37], Hopper [38], Blackwell [39]
	Other AI Accelerators	AMD GPU [40], GAUDI [41], TPU [42], Graphcore IPU [43], Cerebras CS-2 [44]

	Chip-to-Chip	**Cube-Mesh Topology:** NVLink-1.0 [45] **Fully-Connected Topology:** NVSwitch [46], Infinity Fabric [47] **2D/3D-Torus Topology:** TPUv2 [48], TPUv3 [49], TPUv4 [42]
	Node-to-Node	GPUDirect-RDMA [50], InfiniBand EDR/HDR/NDR [51], RoCE-v1 [52], RoCE-v2 [53], iWARP [54]
Network Infrastructure	Network Topology	**HPC Topology:** Clos [55], BCube [56], DCell [57], Jellyfish [58], Torus [59], Dragonfly [60], Dragonfly+ [61] **Training-Optimized Topology:** Rail-Optimized [62], HPN [63], Rail-Only [64], BiGraph [63], HammingMesh [65] Reconfigurable Topology: SiP-ML [66], TopoOpt [67], TPUv4 [42]
	Load Balancing & CC	**Load Balancing:** ECMP [68], Multi-Flow with Enhanced-ECMP [9], Packet spraying [69], Ethereal [70], HPN [63], MegaScale [71] **Congestion Control (CC):** PFC [72], TIMELY [73], Swift [74], DCQCN [75], [76], HPCC [77], EQDS [78], RoCC [79], MLTCP [80], CASSINI [81], MLT [82]

Storage Systems	Checkpoint Storage	Tectonic [83], HDFS [84], Ceph Object Storage [85]
	Training Data Storage	Lustre [86], GPFS [87], BeeGFS [88], Alluxio [89], JuiceFS [90], Quiver [91], Fluid [92]

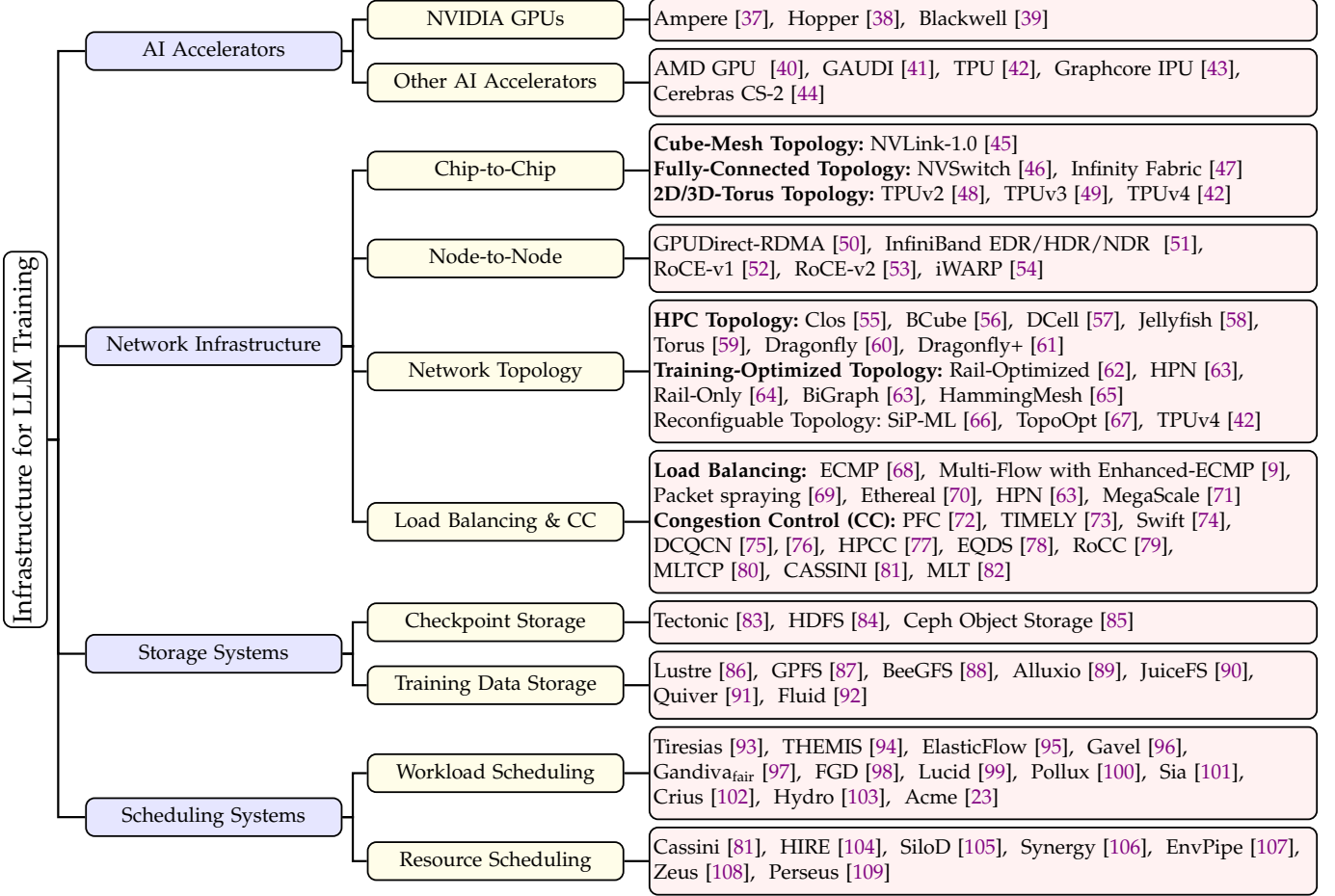| Scheduling Systems	Workload Scheduling	Tiresias [93], THEMIS [94], ElasticFlow [95], Gavel [96], Gandiva$_{fair}$ [97], FGD [98], Lucid [99], Pollux [100], Sia [101], Crius [102], Hydro [103], Acme [23]
	Resource Scheduling	Cassini [81], HIRE [104], SiloD [105], Synergy [106], EnvPipe [107], Zeus [108], Perseus [109]

(Infrastructure for LLM Training)

Fig. 4: Studies on infrastructure optimizations for distributed LLM training.

cale supercomputer. Each Frontier node is equipped with 8 MI250X [40] AMD GPUs, each with 64 GB of HBM and a theoretical FP16 peak performance of 191.5 TFLOPS. This configuration presents an unparalleled opportunity for training trillion-parameter models efficiently. The key to unlocking this potential lies in adapting existing CUDA-based tools and frameworks to the ROCm platform [113], [114]. Notably, ROCm-enabled versions of FlashAttention [115] and FlashAttention2 [116] have been developed, allowing for the efficient execution of attention.

Various AI accelerators with powerful computing and software optimizations have been developed to train LLMs. GAUDI [41] offers a heterogeneous compute architecture comprising two Matrix Multiplication Engines and a cluster of fully programmable tensor processor cores, enabling efficient handling of LLM training operations. This processor can support the training of a GPT-3 model with 175 billion parameters using 384 GAUDI2 cards [117]. Google TPUv4 [42] supercomputer has 4096 chips, supporting LLM training at an average of approximately 60% of peak FLOPS. Graphcore Bow Pod64 [43], a one-rack setup with 64 Bow-class IPUs, achieves 22 petaFLOPS. It supports GPT-3 model training with 256 IPUs. Cerebras CS-2 [44] is a wafer-scale deep learning accelerator comprising 850,000 processing cores, each providing 48KB of dedicated SRAM memory. It is used to train Cerebras-GPT, a family of open compute-optimal language models [118].

3.2 Network Infrastructure

Communication overhead is a major obstacle to scaling LLM training [119], [120]. For example, reducing model gradients during training can lead to over 90% of the training time being spent on communication [121]. To address this, the research community has focused on improving communication infrastructure for LLM training.

3.2.1 Chip-to-Chip Communications

Chip-to-chip communication is pivotal for data transfer between AI accelerators within a node, significantly impacting the efficiency of LLM training. Traditionally, this communication has relied on PCI Express (PCIe) [122], which employs a tree topology—a hierarchical structure where multiple devices connect to a single root complex. Over the years, PCIe has improved its bandwidth: PCIe 3.0 offers approximately 1 GB/s per lane, totaling about 16 GB/s for a configuration with 16 lanes; PCIe 4.0 doubles the bandwidth to 2 GB/s per lane, while PCIe 5.0 further increases it to 4 GB/s per lane. Despite these enhancements, PCIe's inherent limitations in bandwidth, latency, and scalability render it suboptimal for LLM training [123]. To address these limitations, specialized chip-to-chip interconnects like NVLink [45] are increasingly preferred for LLM training. Compared to PCIe, these advanced interconnects provide significantly higher bandwidth and lower latency by utilizing various
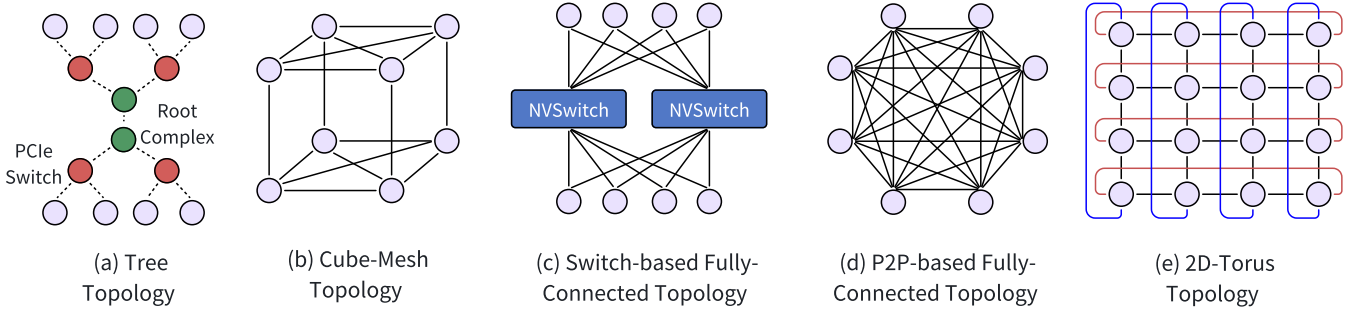
Fig. 5: Five chip-to-chip topologies: tree topology, cube-mesh topology, switch-based fully-connected topology, P2P-based fully-connected topology, and 2D-torus topology.

topologies: *cube-mesh*, *fully-connected*, and *3D-torus*. Additionally, shared memory models, specialized communication protocols, and synchronization mechanisms also play crucial roles.

Cube-Mesh Topology. NVLink-1.0 [45] offers a direct and high-speed connection between GPUs, with each link providing 160 GB/s of bidirectional bandwidth. This architecture enables the formation of planar mesh structures for four GPUs and a cube-mesh topology for eight GPUs, which can be configured into a DGX-1 server. This cube-mesh configuration, although not an all-to-all connection, significantly enhances data communication efficiency and training performance on GPUs.

Fully-Connected Topology. Many interconnects utilize either switch-based or P2P-based fully connected topologies to improve chip-to-chip communication performance. NVIDIA uses NVSwitch [46] to achieve switch-based all-to-all interconnections among GPUs. In the DGX-2 [124] system, six NVSwitches fully connect each of the sixteen GPUs to all others, providing a bidirectional bandwidth of 300 GB/s between any two GPUs. This bandwidth increased to 600 GB/s with NVSwitch 2.0 and further to 900 GB/s with NVSwitch 3.0. Intel, AMD, and Huawei Ascend utilize P2P-based fully-connected topology for their accelerators, where each chip directly connects to every other chip within the same node using Ethernet or Infinity Fabric [47]. Compared to the switch-based topology, the bandwidth between two GPUs in a P2P-based topology is limited by the bandwidth of the directly connected link.

2D/3D-Torus Topology. Google's TPU systems utilizes Torus network topology [59] for chip-to-chip communication. It establishes connectivity by linking each TPU chip to its four adjacent neighbors in a grid, with wraparound edges, thus forming a toroidal structure. This architectural design ensures low latency and high bandwidth due to the availability of multiple direct paths between chips. Specifically, the TPUv2 [48] supercomputer employs a 16x16 2D torus configuration, encompassing 256 chips, interconnected via high-speed Inter-Chip Interconnect (ICI) links. The TPUv3 [49] supercomputer utilizes a 32x32 2D torus, comprising 1024 chips. Advancing from the 2D torus design, the TPUv4 [42] supercomputer organizes compute resources into multi-machine cubes with a 3D torus topology. Each TPU machine contains four chips arranged in a 2x2x1 mesh,

interconnected through the ICI links. Sixteen of these TPU machines are combined to form a data center rack, where ICI links within the rack interconnect to create a 4x4x4 mesh, resulting in a 3D torus structure. This advanced configuration significantly enhances communication efficiency and scalability, particularly beneficial for LLM training.

3.2.2 Node-to-Node Communications

Remote Direct Memory Access (RDMA) [54] enables high-speed and low-latency data transfer between nodes. RDMA allows direct memory access from the memory of one computer to another without involving either node's operating system. GPUDirect-RDMA [50] enhances this process by enabling direct communication between GPUs across different nodes, bypassing the CPU entirely. This technology is particularly beneficial for LLM training, as it accelerates the synchronization of model parameters and gradients. The two most prevalent RDMA technologies are InfiniBand [51] and RDMA over Converged Ethernet (RoCE) [52].

InfiniBand is a high-speed, low-latency networking technology widely utilized in HPC (High Performance Computing) environments, such as the Eagle supercomputer [125]. This technology necessitates a dedicated network infrastructure, reflecting its design focus on delivering superior performance. Over the years, InfiniBand has significantly evolved in terms of bandwidth capabilities, advancing from EDR (Enhanced Data Rate) at 100 Gbps to HDR (High Dynamic Range) at 200 Gbps, and more recently to NDR (Next Data Rate) at 400 Gbps per link [126]. RoCE leverages the existing Ethernet infrastructure to deliver RDMA capabilities. This approach offers a more cost-effective and easier-to-deploy solution, particularly in data centers that already utilize Ethernet. RoCE is available in two versions: RoCE-v1 [52], which operates as an Ethernet link layer protocol, and RoCE-v2 [53], which operates over UDP. Industry leaders such as ByteDance and Meta have employed these technologies to scale LLM training. Another RDMA protocol, the Internet Wide Area RDMA Protocol (iWARP) [54], enables RDMA over TCP/IP networks. However, due to its comparatively limited performance, iWARP is not commonly used for distributed LLM training [127].

3.2.3 Network Topology

In LLM training clusters, the network architecture is structured into frontend and backend components (Fig. 3). The
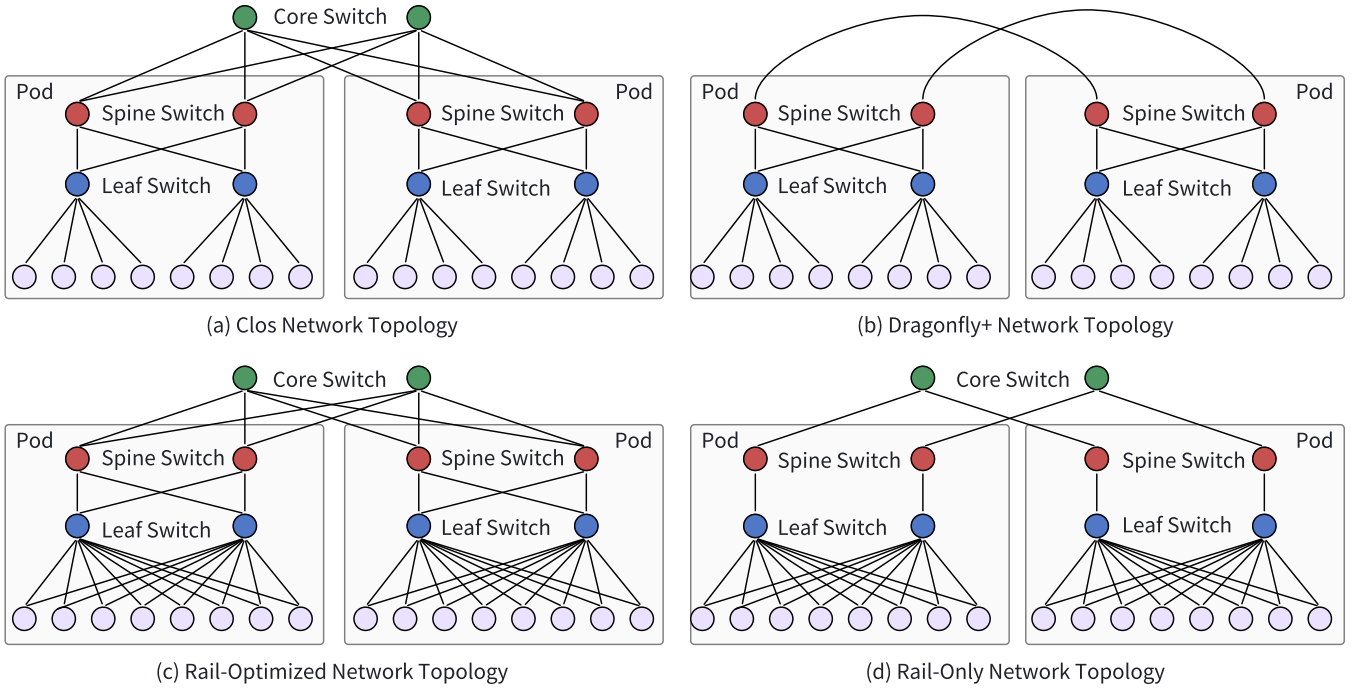
Fig. 6: Four typical network topologies in large-scale GPU clusters: Clos topology, Dragonfly+ topology, rail-optimization topology, and rail-only topology.

frontend network handles a variety of traffic, such as job management, model inference, and storage activities, while the backend network is dedicated to the high-volume traffic generated during the training process. Our primary focus in optimizing LLM training lies in improving the performance and efficiency of this backend network, so as to scale AI accelerators to tens of thousands.

HPC Network Topology. Conventional topologies for HPC environments can also be used in AI clusters for distributed training, such as Clos [55], BCube [56], DCell [57], Jellyfish [58], Torus [59], Dragonfly [60] and Dragonfly+ [61]. The Clos network architecture, commonly known as a Fat-Tree topology, is widely used in LLM training clusters. In a Clos-based cluster, each server, equipped with one or more NICs, is organized into racks connected to leaf switches. These leaf switches link to spine switches, providing inter-rack connectivity and forming a pod. The pods are further interconnected with core switches, facilitating any-to-any communication across servers within the cluster. For example, Meta's last-generation GPU cluster architecture, supporting up to 24,000 GPUs, consists of eight pods with full-fat bandwidth between them and use a 7:1 oversubscription ratio on the core layer [9]. Meta uses 24,000 GPUs to this cluster for training Llama 3 405B.

Training-Optimized Topology. Many network topologies are co-designed with distributed training algorithms. The rail-optimized topology [62] enhances the connection from GPUs to leaf switches. Within each rail, GPUs that share the same index across different servers are interconnected via the same leaf switch. This configuration improves collective communication performance by reducing network interference between data flows. The SuperPod architecture utilizes

a rail-optimized network and is capable of connecting over 16,000 GPUs [128]. ByteDance employs a three-layer rail-optimized network to connect more than 10,000 GPUs in its MegaScale system design [71]. However, rail-optimized network designs could be less efficient because they necessitate connecting GPUs to distant switches, which requires costly and power-hungry optical transceivers. These optical components increase power consumption and heat, leading to higher rates of network failures, which is significant for distributed LLM training. Alibaba further optimizes the rail-optimized topology with a 2-tier, dual-plane architecture named HPN [63]. This architecture employs the latest 51.2Tbps single-chip switch and supports 1,000 GPUs in a tier1 network and up to 15,000 GPUs within one pod.

The network traffic analysis of GPT/OPT-175B model training reveals that 99% of GPU pairs do not carry any traffic, and less than 0.25% of GPU pairs handle pipeline/tensor parallel and data parallel traffic [64]. Based on these findings, the rail-only topology [64] eliminates the connection between different rails on rail-optimized network. Each rail is connected by a dedicated but separate Clos network. Communication across GPUs on different rails is managed by forwarding the data through internal chip-to-chip interconnects. This method could effectively reduce costs while maintaining performance. HammingMesh [65] organizes GPUs into groups with a 2D-torus topology and connects these 2D-torus groups via sparsely connected switches. This design aims to save costs without compromising training performance. Given GPUs that are only connected by PCIe, BiGraph [129] proposes a new network architecture that exports intra-node GPU communication to outside the node, bypassing PCIe bandwidth bottlenecks. It features a two-layer network interconnected via a Clos architecture,

with unique shortest paths for communication that support application-controlled traffic routing.

Reconfiguable Topology. Reconfigurable network can be dynamically adjusted to optimize communication patterns for improving the training performance. They typically utilize optical switching and customized configurations to improve bandwidth utilization, flexibility, and scalability of the network infrastructure. Driven by Silicon Photonic (SiP) interfaces, SiP-ML [66] advances two principal architectures: SiP-OCS and SiP-Ring. SiP-OCS incorporates a fully connected configuration that maximizes the bandwidth by employing commercially accessible optical circuit switches, linking GPUs to all switches via Tbps SiP interfaces. Conversely, the SiP-Ring utilizes a switchless ring configuration, reducing the reconfiguration latency by integrating micro-ring resonators within the SiP interfaces. Wang et al. proposed TopoOpt [67] for co-optimizing the network topology and parallelization strategies in distributed training. This approach not only optimizes the computational and communication demands but also addresses the physical layer of network topology. TPUv4 [42] features Optical Circuit Switches (OCS), allowing for dynamic reconfiguration of the 3D-Torus based interconnect topology, optimizing the data flow for the varied and intensive communication patterns that characterize LLM training. For instance, with 512 chips, TPUv4 offers flexibility in 3D-Torus topologies such as 4x4x32 or 8x8x8.

3.2.4 Load Balancing & Congestion Control

Load Balancing. The network traffic of LLM training is characterized by a small number of elephant flows. Specifically, LLM training demonstrates periodic bursts of network traffic due to gradient synchronization [63]. Each burst demands significant network bandwidth. Moreover, each compute node involved in LLM training generates very few connections [63]. The conventional load-balancing technique, ECMP (Equal-Cost Multi-Path routing) [68], uses hash algorithms to evenly distribute traffic across equivalent paths, such as those from leaf switches to spine switches in a Clos topology. However, this hash-based scheme is inefficient for handling LLM training traffic, which consists of a small number of elephant flows. When multiple elephant flows are routed to the same link, it can result in congestion and high latency.

Various strategies have been developed to address the load balancing challenges in large-scale GPU clusters. During the Llama 3 405B training, the collective library establishes 16 network flows between two GPUs, rather than a single flow, thereby reducing traffic per flow and enhancing load balancing opportunities [9]. Additionally, the Enhanced-ECMP (E-ECMP) protocol effectively distributes these 16 flows across different network paths by hashing on additional fields in the RoCE header of packets. Packet spraying [69] distributes packets from a flow across all available parallel links, which can lead to out-of-order packets. NICs need to process out-of-order RDMA packets. Based on the traffic pattern of LLM training, Ethereal [70] demonstrates that greedily assigning paths to each flow can uniformly distribute the load across all network paths and resolve the ECMP hash conflict problem. In a large-scale

GPUs cluster, HPN [63] achieves efficient load balancing by identifying precise disjoint equal paths and balancing the load within the collective communication library. MegaScale [71] shows that a rail-optimized topology can also mitigate ECMP hashing conflicts.

Congestion Control. Lossless transmission is crucial in RDMA clusters. Priority-based Flow Control (PFC) [72] is a flow control mechanism that prevents packet loss. When congestion occurs in a PFC-enabled queue on a downstream device, the device instructs the upstream device to halt traffic in the queue, thereby ensuring zero packet loss. Since PFC is a coarse-grained mechanism, it can lead to head-of-line blocking [130], which significantly diminishes network throughput. To address these challenges, various general-purpose congestion control schemes have been developed. These techniques include TIMELY [73], Data Center Quantized Congestion Notification (DCQCN) [75], [76], Swift [74], High Precision Congestion Control (HPCC) [77], Edge-Queued Datagram Service (EQDS) [78], and Robust Congestion Control (RoCC) [79]. These schemes monitor network congestion, adjust data rate to alleviate congestion, and recover the rate to minimize throughput reduction.

When there are concurrent training jobs, many congestion control schemes leverage the bursty and periodic traffic patterns to interleave the network traffic effectively. MLTCP [80] interleaves the communication phases of jobs that compete for bandwidth based on a key insight: training flows should adjust their congestion window size based on the number of bytes sent in each training iteration. CASSINI [81] optimizes job placement on network links by considering the communication patterns of different jobs. MLT [82] leverages the characteristics of LLM training, where the gradients of earlier layers are less important than those of later layers, and larger gradients are more significant than smaller ones. Consequently, in the event of communication congestion, MLT prioritizes queuing or discarding packets based on the importance of the gradients within them at the switch level to mitigate communication congestion issues.

3.3 Storage

The storage system plays a critical role in distributed LLM training and need to meet several key requirements. Firstly, it should align with the computing power of GPUs to maximize their utilization and avoid resource wastage caused by storage bottlenecks. Secondly, it should support the storage of large-scale structured and unstructured training datasets and be scalable in distributed processing environments. Additionally, the storage and retrieval of model checkpoints present challenges in LLM training, requiring the system to meet the writing and reading bandwidth dictated by the model size and training duration. Lastly, the storage system should satisfy traditional enterprise-level requirements, such as data protection, high availability, and security.

3.3.1 Storage Systems for Checkpoint

The model checkpoint size is enormous in LLM training. As the number of parameters increases, so does the volume of data that needs to be written, demanding greater write bandwidth from the storage system. For example, the checkpoint size is 980GB for an LLM with 70B parameters. Numerous storage systems have been deployed in large-scale

GPU data centers to manage model checkpoints. Meta's distributed filesystem, Tectonic [83], enables thousands of GPUs to save and load model checkpoints simultaneously, providing efficient and scalable storage solutions for extensive training operations [131]. At ByteDance, HDFS [84] is employed for centralized model checkpoint maintenance, ensuring consistency and reliability at scale [71]. To mitigate bandwidth bottlenecks during checkpoint recovery, a common approach designates a single worker to read the checkpoint partition from HDFS and then broadcast it to other workers sharing the same data. Distributed object stores, such as Ceph Object Storage [85], offer easier scalability. This advantage stems from their lack of a hierarchical directory tree or namespace, simplifying consistency maintenance. Due to these benefits, object stores have become widely adopted for model checkpoint storage.

3.3.2 Storage Systems for Training Data

The raw dataset for LLM training is substantial. LLaMA 3 was trained on over 15 trillion tokens, which is more than seven times larger than LLaMA 2's dataset [6]. Each token requires about 2 bytes, equaling roughly 30 TB of data. Preparing datasets for training involves extensive preprocessing steps, including data crawling and cleaning, requiring significant experimentation. Typically, the data processed during these steps exceeds 100 times the final size of the training dataset [132]. For instance, the WanJuan-CC dataset [132] selectively extracts approximately 68 billion documents, generating around 1 trillion high-quality tokens, equivalent to a data size of 2 TB, after discarding 99% of the raw data. Therefore, the total data volume for LLM training is expected to exceed tens of PB.

Parallel file systems such as Lustre [86], GPFS [87], and BeeGFS [88] are frequently deployed on leading high-performance computing systems to ensure efficient I/O, persistent storage, and scalable performance. These systems are also widely used in training clusters for data loading, providing the necessary infrastructure to handle large-scale training data efficiently. Additionally, it is crucial for file systems to enable engineers to perform interactive debugging on jobs utilizing thousands of GPUs, as code changes need to be immediately accessible to all nodes [131].

During the training of most LLMs, each token is typically encountered only once. However, employing data caching remains crucial to mitigate I/O bottlenecks during data loading. This strategy involves prefetching training data from slower backend storage to faster cache storage. Alluxio [89] and JuiceFS [90] enhance LLM training by efficiently caching training data from underlying storage systems such as HDFS or object storage. Quiver [91] supports the transparent reuse of cached data across multiple jobs and users operating on the same dataset. Fluid [92] leverages Alluxio for data caching, incorporating a mechanism that enables on-the-fly autoscaling of the cache based on I/O conditions.

3.4 Scheduling

LLM training workloads generally run on large-scale multi-tenant infrastructures (e.g., GPU clusters, public clouds) where users share cluster resources. Effective scheduling mechanisms are crucial for managing these workloads, ensuring efficient resource utilization and task execution [133].

Unlike *task-level* scheduling (e.g., pipeline scheduling [134]–[136]), which focuses on fine-grained optimization of single-job execution (§4.1.3), *cluster-level* scheduling aims to optimize resource allocation and task scheduling across the entire cluster. We categorize existing *cluster-level* scheduling systems into two types, workload scheduling and resource scheduling, according to their primary optimization aspects.

3.4.1 Workload Scheduling

Schedulers tailored for DL training workloads have been actively explored in recent years [93]–[95], [137]–[141]. To enhance resource utilization, three advanced features are commonly implemented: (1) *heterogeneous-aware* schedulers (e.g., Gavel [96], Gandiva$_{fair}$ [97]) focus on optimizing job allocation across different GPU generations; (2) *job-packing* schedulers (e.g., FGD [98], Lucid [99]) enable fine-grained GPU sharing to fully facilitate hardware capability; (3) *adaptive-scaling* schedulers (e.g., Pollux [100], Sia [101]) dynamically adjust the number of GPUs as well as the training hyper-parameters to accelerate training progress. However, these schedulers are designed for general DL workloads and may not be directly applicable to LLMs due to the unique characteristics of LLM workloads [23].

To better manage LLM workloads, several recent studies have proposed systems tailored to LLMs. Crius [102] jointly considers hybrid parallelism (§4.1) and hardware-affinity within heterogeneous clusters. It investigates the workflow efficiency of integrating adaptive parallelism configuration at the cluster scheduling level, offering significant opportunities to improve the efficiency of training multiple LLMs concurrently. To achieve highly efficient hyperparameter tuning for LLMs, Hydro [103] scales down the model to a smaller surrogate model for hyperparameter search, and then fuses multiple models into a single entity to enhance the hardware utilization. Additionally, Hydro extends the resources for tuning workloads by interleaving them with pipeline-enabled LLM pretraining tasks, effectively utilizing the pipeline bubbles. Acme [23] further characterizes the workload mixture of the LLM development workflow and proposes a system to efficiently schedule associated jobs related to LLM training, including decoupled evaluation scheduling for timely model quality feedback as well as LLM-involved failure diagnosis and automatic recovery.

3.4.2 Resource Scheduling

In addition to workload scheduling, associated resource scheduling (e.g., CPU, memory, and networking) is another critical aspect of cluster-level management. For networking, Cassini [81] enables the interleaving of bandwidth demands during the up and down phases of different jobs by using an affinity graph to determine time-shift values for adjusting communication phases. HIRE [104] introduces an innovative in-network computing scheduling system for data-center switches, significantly reducing the network detours and tail placement latency. For storage, SiloD [105] treats data cache and remote I/O as first-class resources for joint allocation, showing significant throughput improvements. For CPU and memory, Synergy [106] enhances training efficiency by optimizing CPU core allocations instead of relying on GPU-proportional allocation. Additionally, some works focus on energy conservation. EnvPipe [107] leverages the
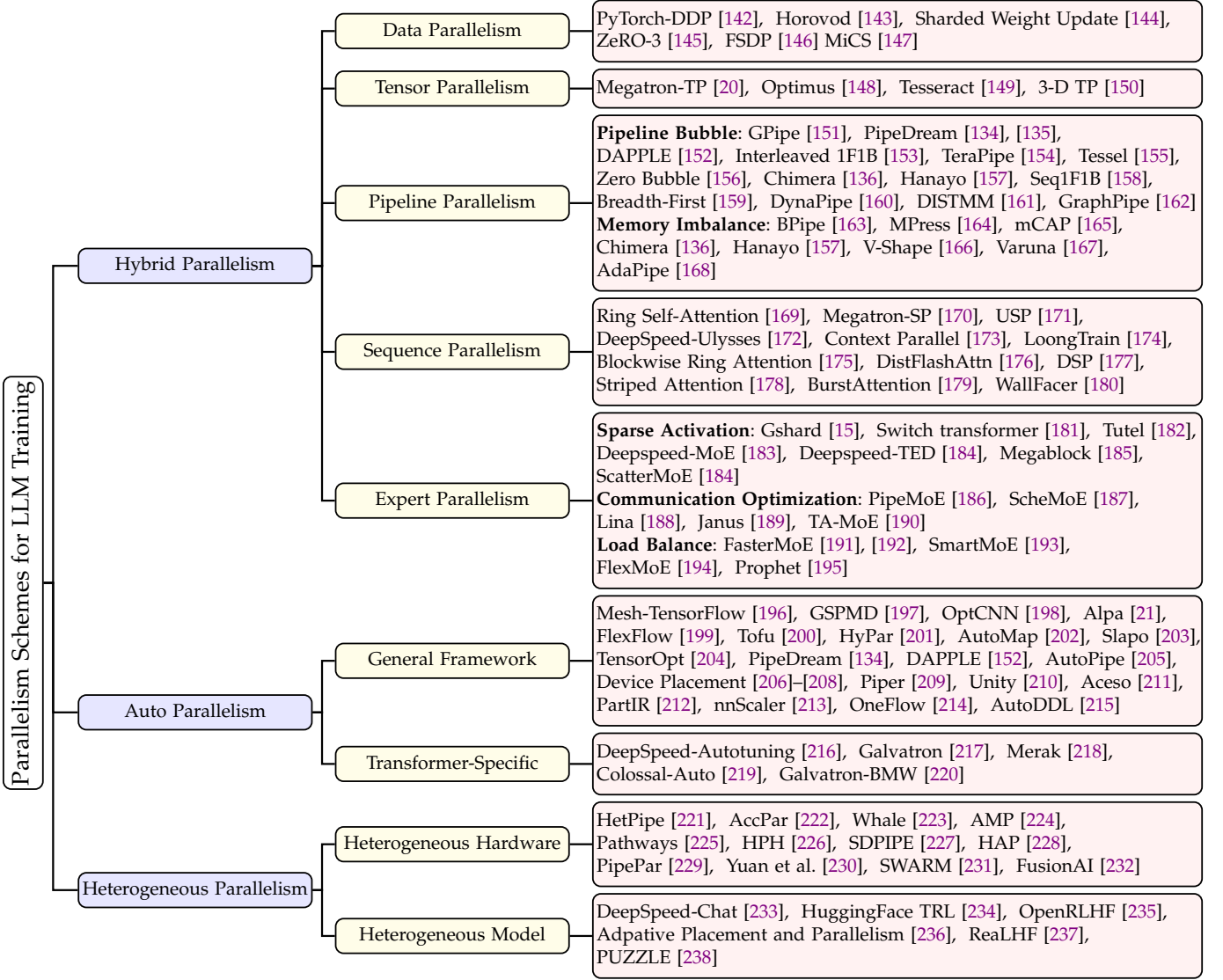
Data Parallelism	PyTorch-DDP [142], Horovod [143], Sharded Weight Update [144], ZeRO-3 [145], FSDP [146] MiCS [147]
Tensor Parallelism	Megatron-TP [20], Optimus [148], Tesseract [149], 3-D TP [150]
Pipeline Parallelism	**Pipeline Bubble**: GPipe [151], PipeDream [134], [135], DAPPLE [152], Interleaved 1F1B [153], TeraPipe [154], Tessel [155], Zero Bubble [156], Chimera [136], Hanayo [157], Seq1F1B [158], Breadth-First [159], DynaPipe [160], DISTMM [161], GraphPipe [162] **Memory Imbalance**: BPipe [163], MPress [164], mCAP [165], Chimera [136], Hanayo [157], V-Shape [166], Varuna [167], AdaPipe [168]
Sequence Parallelism	Ring Self-Attention [169], Megatron-SP [170], USP [171], DeepSpeed-Ulysses [172], Context Parallel [173], LoongTrain [174], Blockwise Ring Attention [175], DistFlashAttn [176], DSP [177], Striped Attention [178], BurstAttention [179], WallFacer [180]
Expert Parallelism	**Sparse Activation**: Gshard [15], Switch transformer [181], Tutel [182], Deepspeed-MoE [183], Deepspeed-TED [184], Megablock [185], ScatterMoE [184] **Communication Optimization**: PipeMoE [186], ScheMoE [187], Lina [188], Janus [189], TA-MoE [190] **Load Balance**: FasterMoE [191], [192], SmartMoE [193], FlexMoE [194], Prophet [195]
General Framework	Mesh-TensorFlow [196], GSPMD [197], OptCNN [198], Alpa [21], FlexFlow [199], Tofu [200], HyPar [201], AutoMap [202], Slapo [203], TensorOpt [204], PipeDream [134], DAPPLE [152], AutoPipe [205], Device Placement [206]–[208], Piper [209], Unity [210], Aceso [211], PartIR [212], nnScaler [213], OneFlow [214], AutoDDL [215]
Transformer-Specific	DeepSpeed-Autotuning [216], Galvatron [217], Merak [218], Colossal-Auto [219], Galvatron-BMW [220]
Heterogeneous Hardware	HetPipe [221], AccPar [222], Whale [223], AMP [224], Pathways [225], HPH [226], SDPIPE [227], HAP [228], PipePar [229], Yuan et al. [230], SWARM [231], FusionAI [232]
Heterogeneous Model	DeepSpeed-Chat [233], HuggingFace TRL [234], OpenRLHF [235], Adpative Placement and Parallelism [236], ReaLHF [237], PUZZLE [238]

Fig. 7: Studies on parallelism schemes for distributed LLM training.

bubble time in the pipeline parallelism, stretching the execution time of pipeline units by lowering the SM frequency to save energy. Zeus [108] automatically configures the batch size and GPU power limit for improving energy efficiency during training. Perseus [109] introduces an efficient graph cut-based iterative algorithm to obtain the iteration time-energy Pareto front for large model training job.

4 PARALLELISM SCHEMES FOR LLM TRAINING

The continual growing scales of LLMs demand substantial computational resources and memory capacity. Distributed training, leveraging large-scale HPC clusters, has emerged as a crucial approach to efficiently train these models. In this section, we investigate various parallelism schemes proposed to enhance the utilization of HPC clusters for LLM training. We categorize these approaches into three main groups: Hybrid Parallelism, Auto Parallelism, and Heterogeneous Parallelism. Hybrid Parallelism combines multiple handcrafted parallelization strategies, such as data parallelism, tensor parallelism, pipeline parallelism, sequence parallelism, and expert parallelism. Auto Parallelism automatically determines the optimal parallelization strategy based on the model and hardware characteristics. Heterogeneous Parallelism exploits the heterogeneity in hardware or model for efficient training. This includes techniques that leverage different types of accelerators or leverage the heterogeneity within a single model (e.g., RLHF training) to improve the overall training efficiency on HPC clusters.

Most of today's state-of-the-art parallelization strategies adopt a Single Program Multiple Data (SPMD) programming model, akin to the MPI paradigm [239], where the same program runs across multiple processors, each working on different pieces of data [225]. For example, data, model and sequence parallelism utilizes the SPMD programming model. This approach ensures uniformity and consistency in operations, making it well-suited for large-scale, distributed training environments. Some strategies explore to break the restriction of SPMD and further improve the resource utilization with the Multiple Program Multiple Data (MPMD) model, where different programs (or different parts of a program) run on different processors, handling

different parts of the data or model [225]. For example, pipeline parallelism runs different parts of an LLM on different devices. In addition, auto parallelism and heterogeneous parallelism can leverage both SPMD and MPMD models to increase the resource utilization. Therefore, we discuss these approaches according to the dimensions along which parallelism occurs and whether the computing devices employed are homogeneous or heterogeneous, rather than focusing on the underlying programming models.

4.1 Hybrid Parallelism

Hybrid parallelism typically combines multiple handcrafted parallelization strategies to partition different parallelizable dimensions of an LLM. These strategies include data parallelism, tensor parallelism, pipeline parallelism and sequence parallelism, as illustrated in Fig. 8. The combination of data parallelism, tensor parallelism, and pipeline parallelism is also referred to as 3D parallelism.

4.1.1 Data Parallelism

Data parallelism is the most commonly used parallelization strategy for distributed training due to its high scalability and ease of implementation. It follows the Single Program Multiple Data (SPMD) programming model. Data parallelism partitions the input training data along the batch dimension, where each GPU processes its assigned segment of data, as shown in Fig. 8(a). Throughout the training process, the data first undergoes forward computation with the full model weights layer by layer, and then performs backward computation in the reverse order. Each layer produces gradients that will be aggregated across all GPUs using collective communication operations for optimizer updates.

Data parallelism incorporates various sharding strategies, which significantly influence the memory footprint and communication overhead. Supposing the global world size is W (i.e. the number of devices), a sharding factor F is introduced to control the sharding strategy used [146], defined as the number of devices across which parameters are partitioned ($1 \leq F \leq W$). We have the following scenarios. **Full replication** ($F = 1$): this sharding strategy is simplified as vanilla data parallelism. Pytorch-DDP [240] and Horovod [143] fully replicate the model across all devices and use All-Reduce for gradient aggregation. They also divide the gradients into small buckets to overlap the gradient communication with backward computation.

Full sharding ($F = W$). This sharding strategy comes with the lowest memory consumption but the most communication overhead ($1.5\times$ over vanilla data parallelism). Full sharding strategy fully shards the model, where each device holds only $\frac{1}{W}$ of the model parameters. The full weights and gradients are communicated and recovered on-demand before the computation, and immediately discarded afterwards. ZeRO-3 [145] employs per-parameter sharding to shard the full model and utilizes All-Gather and Reduce-Scatter for unsharding and sharding communication, respectively. Sharded Weight Update [144] also employs per-parameter sharding but focuses more on sharding the redundant parameter update computation across all the devices. FSDP (Fully Sharded Data Parallelism) [146] achieves the same functionality by sharding model parameters in the grain of module units and provides more user-friendly API.

Hybrid sharding ($1 < F < W$). In this strategy [146], all the devices are divided into a $N \times M$ device mesh. The model parameters are sharded along the N dimension of the mesh, and replicated along the M dimension. MiCS [147] invokes All-Gather collective to gather the sharded parameters and All-Reduce to aggregate the gradients. FSDP [146] replaces the All-Reduce with Reduce-Scatter to reduce the memory and communication overhead. Compared to full replication and full sharding, hybrid sharding is more flexible to provide a trade-off between memory consumption and communication overhead via adjsuting F based on the model architecture and hardware constraints.

4.1.2 Tensor parallelism

Tensor parallelism (Fig. 8(b)), also known as intra-layer model parallelism, is a technique proposed to enable the training of LLMs across multiple GPUs. It partitions the parameter tensors of each layer along multiple dimensions, effectively distributing the model parameters across the available GPUs. Tensor parallelism communicates intermediate activation tensors, the size of which is much smaller than that of parameters and gradients communicated in data parallelism, except for long context LLM training scenarios. However, in tensor parallelism, it is challenging to overlap the communication with computation, necessitating the use of high-bandwidth connections. Consequently, tensor parallelism is more commonly employed in a single GPU node.

Tensor parallelism can be divided into 1-D [20], 2-D [148], 2.5-D [149] and 3-D [150] parallelism according to the dimensionality of the partition. There are two parameter matrices in both the MLP and self-attention module of transformer-based LLMs. Megatron-LM [20] first adopts 1-D tensor parallelism to partition the first parameter matrix along its column, and the second parameter matrix along its row. It replicates the input and output tensors of each partitioned module and introduces two All-Reduce collective communication across all GPUs to fit an LLM into multiple GPUs. Inspired by Scalable Universal Matrix Multiplication Algorithm (SUMMA) [241] and Cannon's algorithm [242] for 2-D parallel matrix multiplication, Optimus [148] further partitions the input and parameter tensors in 2 dimensions to improve the communication and memory efficiency of 1-D tensor parallelism. Tesseract [149] extends the 2.5-D matrix multiplication method [243], which is proposed to improve the efficiency of Cannon's algorithm, to LLM training and proposes the 2.5-D tensor parallelism to overcome the abundance of unnecessary communication resulting from the increasing model size. 3-D tensor parallelism [150] adopts and improves the 3-D parallel matrix multiplication algorithm [244] for linear operations and achieves a perfect load balance across multiple devices for LLM training.

4.1.3 Pipeline Parallelism

Pipeline parallelism (Fig. 8(c)) [151], also known as inter-layer model parallelism, is proposed to accommodate large models across multiple GPUs, particularly across different nodes. Pipeline parallelism partitions the layers of a model into multiple stages, where each stage consists of a consecutive set of layers in the model and is mapped to a set of GPUs. Unlike tensor parallelism, which typically demands
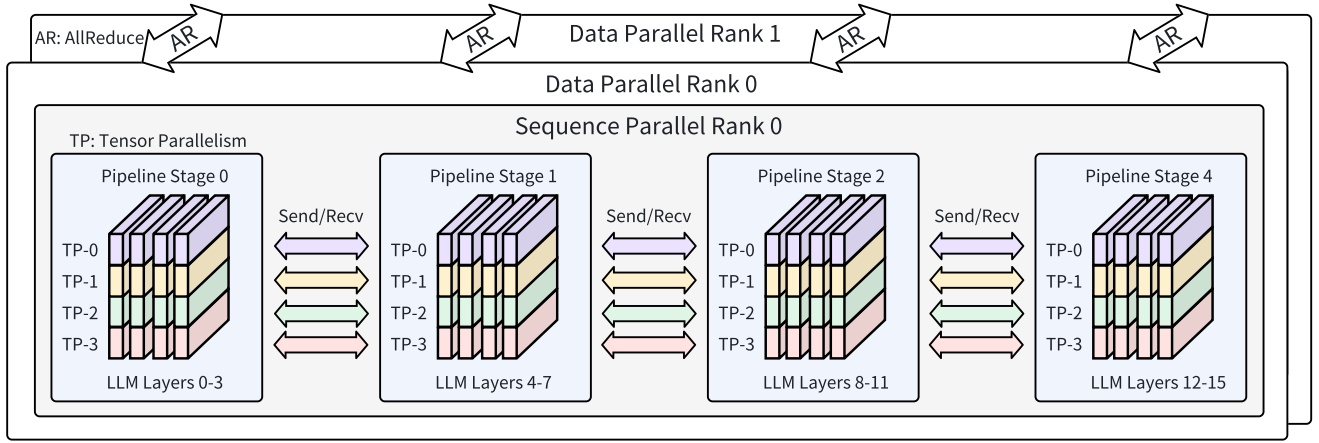
Fig. 8: An example of 3D-parallelism with data parallelism, tensor parallelism, and pipeline parallelism.

high-bandwidth connections like NVLink for communication, pipeline parallelism only necessitates the exchange of intermediate tensors at designated cutting points, resulting in less frequent communication requirements. Therefore, pipeline parallelism is suitable for scaling up the LLM training across multiple GPU nodes that are connected with small bandwidth. For example, Strati et al. [245] adopt pipeline parallelism to fully utilize geo-distributed resources to overcome the shortage of GPUs. Due to the data dependency of different stages, pipeline parallelism typically splits the input data into multiple micro-batches for pipelining to enable the efficient training of giant models. However, it comes with two significant problems. First, the pipeline bubble problem reduces the utilization of GPUs due to the time spent on waiting for the output of the previous stage. Second, there exists memory consumption imbalance across different stages since the former stages need to hold more active micro-batches than the latter for better pipelining and higher utilization. We detail each problem below.

Pipeline Bubble. Efficient micro-batch scheduling algorithms are proposed to reduce pipeline bubbles. GPipe [151] introduces a fill-drain schedule that injects all micro-batches at once for forward pass execution, followed by backward passes. Gpipe incurs significant pipeline bubbles due to the warm up and cool down of both forward and backward passes. PipeDream [134], [135] introduces a 1F1B (1 Forward 1 Backward) schedule, which executes the backward pass of a micro-batch as soon as the corresponding forward pass has completed, to reduce the pipeline bubbles in the asynchronous scenario. DAPPLE [152] employs an early backward schedule to first inject a fixed number of micro-batches at the beginning of each stage and then interleave forward and backward passes with round robin. Interleaved 1F1B [153] adapts the 1F1B schedule but assigns multiple stages to each GPU (i.e. looping pipeline placement). The pipeline bubble is reduced at the cost of higher communication and peak memory consumption. Chimera [136] introduces a bidirectional pipeline to reduce bubbles with weight duplication. Hanayo [157] further proposes a wave-like pipeline that assigns multiple symmetrical stages to one GPU to improve the pipeline utilization. Zero bubble [156] splits the backward computation into two parts: activation

and parameter gradient computation. It schedules the forward and activation gradient computation with 1F1B and then fills the bubbles with parameter gradient computation, which reduce bubbles with higher peak memory consumption. Breadth-First [159] runs all micro-batches at once in looping pipeline placement to reduce the communication overhead when combined with sharded data parallelism. TeraPipe [154] splits the micro-batch along the sequence dimension and exploits more fine-grained token parallelism to reduce pipeline bubbles. However, The memory overhead of TeraPipe is large since it is based on the GPipe schedule. Seq1F1B [158] splits the sequence into chunks and utilizes the 1F1B schedule to reduce the peak memory consumption while achieving low pipeline bubble rates. DynaPipe [160] uses a dynamic micro-batching approach to the multi-task training of LLMs with variable-length input. It introduces a memory-aware adaptive scheduling algorithm and ahead-of-time communication planning to further reduce the pipeline bubble rates. Tessel [155] is a two-phase approach, including repetitive pattern construction and schedule completion, to automatically search for the efficient pipeline schedule for a specified partition strategy. DISTMM [161] launches doubled micro-batches to bypass the dependency barrier caused by the large batch requirement of multi-modal training, thus reducing idle cycles. GraphPipe [162] preserves the DNN graph topology and partitions it into stages that can be concurrently executed to improve pipeline utilization and reduce memory consumption.

Memory Imbalance. Pipeline parallelism typically injects more micro-batches into the beginning stages to improve the pipeline utilization, resulting in higher activation memory consumption in these stages. To resolve this problem, BPipe [163] and MPress [164] employ D2D (device to device) transfer to swap intermediate activation tensors from high-load GPU to light-load GPU during runtime. MPress also combines the activation recomputation to reduce the memory footprint. Chimera [136] introduces a bidirectional pipeline that combines two pipelines in different directions together to achieve more balanced memory consumption. Each GPU holds two symmetric stages, leading to weight duplication. Hanayo [157] turns the bidirectional pipeline

into two data parallel pipelines to remove the weight duplication and achieves a balanced memory consumption by assigning multiple stages to one GPU symmetrically. V-Shape [166] partitions the model into stages twice the number of devices, where the two halves of stages are placed in reverse order. By varying the offsets between stages, V-Shape makes a trade-off between peak memory consumption and bubble utilization. mCAP [165] utilizes an incremental-profiling approach to partition models evenly across GPUs with respect to peak memory usage.

Peak memory consumption limits the number of active micro-batches in pipeline parallelism, thus its efficiency. Activation recomputation can be employed to reduce the peak memory consumption effectively. Varuna [167] combines pipeline parallelism and activation recomputation to achieve this goal. It designs a static rule-based schedule enumerated for a given pipeline with an opportunistic policy to hide jitter and reduce bubbles. The static schedule is generated based on constraints including activation recomputation timing, activation memory management, and backward computation prioritization. To resolve the memory imbalance with low recomputation overhead, AdaPipe [168] adopts adaptive recomputation to support different recomputation strategies for different stages, and adaptive partitioning based on the 1F1B schedule to balance the computation of each stage.

4.1.4   Sequence parallelism
The context window of today's LLMs grows rapidly, and the most powerful LLM can support millions of tokens [7]. Such ultra long sequence leads to significant memory and computation requirements for LLM training: linearly increasing memory footprint of activations and quadratic complexity of attention mechanism. Recomputing activations in the backward can reduce the peak memory consumption but also introduce significant overheads (30% with full recompute). Large tensor parallelism degree incurs significant communication overhead. Sequence parallelism (Fig. 8(d)) [169], [170] is proposed to accommodate the long sequence training and distribute the computation in multiple GPUs efficiently within the memory capacity. It divides the input data into multiple chunks along the sequence dimension and each chunk is fed to one GPU for computation. Since sequence parallelism replicates the model parameters, it is typically combined with tensor and pipeline parallelism to scale up the LLM training. When used together with tensor parallelism, sequence parallelism distributes the memory and computation of attention on multiple GPUs, but incurs redundant memory consumption and computation in the non-tensor parallel regions of a transformer layer. Megatron-SP [170] splits these computations along the sequence dimension to reduce redundant the activation computation and memory consumption without increasing the communication.

Although sequence parallelism partitions the memory, computation and communication across multiple GPUs, the quadratic causal attention still presents remarkable challenges in training efficiency, including the key-value tensor communication overhead, IO-awareness attention computation overhead and load imbalance among GPUs due to the causal attention mask. Most sequence parallelism approaches for attention are ring-based [169], [173], [175], [176], [178], [179]. Ring Self-Attention [169] leverages sequence parallelism and calculates the self-attention with ring-style communication to scale up the context window of LLM training. It first transmits the key tensors among GPUs to calculate the attention scores in a circular fashion, and then calculates the self-attention output based on the attention scores and value tensors transmitted in a similar fashion. DistFlashAttn [176] transmits the key-value tensor chunks concurrently to utilize IO-awareness FlashAttention [115] kernel and balance the computation of different GPUs by filling the idle cycles of earlier tokens with later tokens. Megatron Context Parallel [173] also leverages the FlashAttention kernel and removes the unnecessary computation resulted from low-triangle causal masking. It further balances the computation among GPUs by exchanging half of the chunk with the symmetric GPU. DistFlashAttn and Context Parallel also overlap the key-value tensor communication and attention computation with separate CUDA streams. Striped Attention [178] resolves the imbalance by assigning each GPU a subset of tokens distributed uniformly throughout the sequence, instead of contiguous chunks. BurstAttention [179] computes the attention with FlashAttention on each GPU and utilizes double buffers to overlap the communication and computation. Blockwise Ring Attention [175] extends Ring Self-Attention [169] to blockwise attention, which computes the attention in small blocks to reduce the memory footprint. Inspired from N-body simulation, WallFacer [180] first divides GPUs into subgroups and replicates query and key-value tensors in each subgroup via asynchronous AllGather. The attention computation leverages multiple ring-style P2P communication to enhance efficiency. A final asynchronous ReduceScatter is needed to distribute the attention output.

DeepSpeed-Ulysses [172] differs from previous ring-based approaches by splitting the head dimension instead of sequence dimension and leverages All-to-All to shift the partition dimension from sequence to head. DeepSpeed-Ulysses can be seamlessly combined with existing attention implementation, e.g., FlashAttention, and the workload among GPUs is naturally balanced. However, the parallelism degree of DeepSpeed-Ulysses is restricted by the number of heads, especially for LLMs using MQA [11] and GQA [12]. LoongTrain [174] and USP [171] are concurrent work that integrate the advantages of DeepSpeed-Ulysses and Ring Attention. They organize the GPUs into 2-dimension mesh, forming hybrid ulysses- and ring-style process groups. During training, they first perform All-to-All among ulysses groups to switch partition from sequence to head dimension, and then perform attention computation with Ring-Attention among ring groups. LoongTrain further proposes Double-Ring-Attention to fully utilize available bandwidth for inter-node communication and overlap communication with computation. DSP [177] dynamically switches the parallelism dimension according to the computation stage in multi-dimensional transformers, like DiT [246].

4.1.5   Expert Parallelism
The Mixture-of-Experts (MoE) is currently the most popular sparse model in LLMs. While significantly increasing
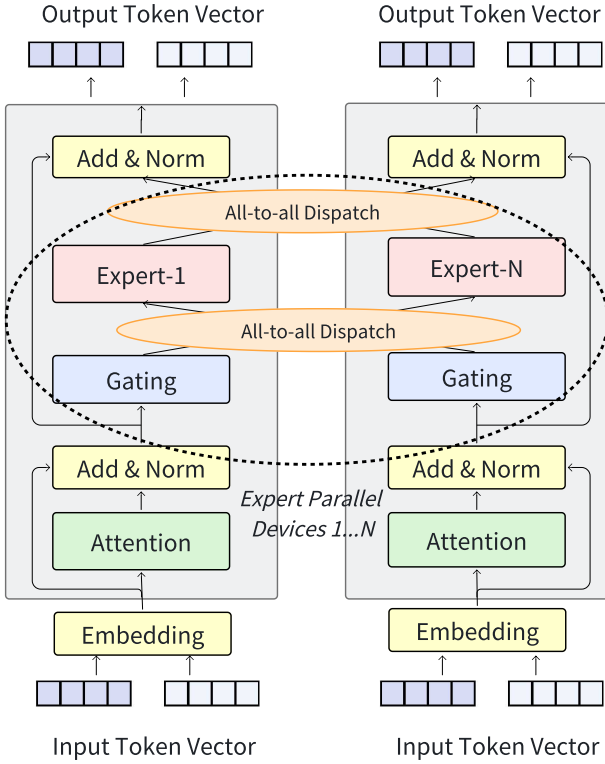
Fig. 9: Expert parallelism. The dotted line highlights the MoE components within the transformer model, where each device maintains one expert for expert parallelism and collaborate based on All-to-All communication.

the number of parameters in LLMs, MoE does not greatly increase the computational cost with conditional computations [247]. The basic framework of MoE, as shown in Fig. 9, consists of multiple expert networks that handle different subsets of training data and a gate network that applies routing algorithm to assign the input tokens to different experts networks. MoE enables the training of large models with parameters beyond the trillion scale and is claimed to be used in popular LLM models such as Mixtral 8x7B [248] and DeepSeek2 [13].

Sparse Activation. With the increasing model size, all experts cannot be accommodated and trained on a single device. Therefore, GShard [15] extends the idea of MoE to Transformers in distributed settings, where experts are distributed across different workers and collaborates with All-to-All communication, as shown in Fig. 9. Subsequent research for expert parallelism generally follows the same paradigm. For example, the Switch Transformer [181] incorporates the design of distributed MoE training on the T5 model. But unlike the top-2 routing algorithm used in GShard, the Switch Transformer routes each token to only the top-1 expert to maximize computational efficiency. Additionally, DeepSpeed-MoE [183] proposes a new distributed MoE architecture that applies shared experts in each worker and places more experts in deeper layers to balance communication costs with training accuracy.

Expert parallelism can be effectively integrated with conventional 3D parallelism. For example, GShard, Switch Transformer, and DeepSpeed-MoE all treat expert parallelism as an orthogonal dimension of hybrid parallelism. For efficient hybrid training, DeepSpeed-TED [249] presents a hybrid parallel algorithm that combines data, tensor, and expert parallelism to enable the training of MoE models. The authors partition the MoE parameters into "tiles" of a predefined size to avoid high optimizer memory spikes and propose communication optimizations like Duplicate Token Dropping (DTD) and activation checkpointing to eliminate duplicate data in All-to-All communication. However, it is challenging to choose the optimal hybrid parallelism plan due to the dynamic nature of MoE, and switching between different parallelism strategies during runtime also incurs substantial overhead. Therefore, some research like Tutel [182] designs an adaptive parallelism switching algorithm that applies the same distribution model layout for all possibly optimal strategies, and can dynamically switch the parallelism strategy at every iteration without any extra overhead.

Since General Matrix Multiplications (GeMMs) require the size of all experts' inputs to be consistent, existing MoE training frameworks often perform token dropping and padding to match the same expert capacity, which wastes computation. Megablocks [185] optimizes grouped GeMMs by implementing Block Sparse Matrix Multiplication and supports different batch sizes for expert computations in a single kernel to avoid unnecessary token dropping in MoE training. Another framework that supports grouped GeMMs is ScatterMoE [184], which implements the ParallelLinear kernel that fuses grouped GeMMs and scattered read and write operations to reduce the memory footprint for top-k ($k \geq 2$) gating.

Communication Optimization. All-to-all communication in expert parallelism can seriously affect the training efficiency of MoE, especially in poor network environments. Existing distributed training systems try to optimize the performance of MoE by overlapping communication tasks with computing tasks so that some communication costs can be hidden. For example, Tutel [182] divides the input tensors into groups along the expert capacity dimension and overlaps computation and communication among different groups to hide All-to-All overhead. FasterMoE [191], [192] uses a similar strategy to Tutel but splits the tensor along the expert dimension. Additionally, Tutel [182] also optimizes the All-to-All kernel implementation by aggregating small messages into a single large chunk inside the nodes before exchanging data among different nodes. This optimization is also used in FasterMoE and ScheMoe [187]. Based on the overlap strategy in Tutel, PipeMoE [186] models the execution time of both communication and computation tasks based on the workloads and designs an adaptive algorithm to find the optimal number of partitions to minimize training time. ScheMoE [187] considers data compression approaches before All-to-All communication and modularizes time-consuming operations, including data compression, collective communication, and expert computation. ScheMoE then proposes an adaptive optimal scheduling algorithm to pipeline communication and computing operations to improve training efficiency.

Expert parallelism usually interacts with other parallel

strategies in MoE training. It is possible to reduce communication overhead by fine-grained task scheduling. For example, Lina [188] analyzes the All-to-All overhead of MoE during distributed training and inference systematically and finds that All-to-All latency is prolonged when it overlaps with AllReduce operations. Lina proposes prioritizing All-to-All over AllReduce to improve its bandwidth and reduce its blocking period in distributed training. Additionally, Lina incorporates tensor partitioning and pipelining to perform micro-op scheduling similar to Tutel. Lina also dynamically schedules resources based on expert popularity during inference to minimize overhead. Janus [189] designs a data-centric paradigm that keeps data in place and moves experts between GPUs based on a Parameter Server. The data-centric paradigm uses fine-grained asynchronous communication and allows experts to move between GPUs using non-blocking communication primitives such as pull. Janus implements a topology-aware strategy to effectively pull experts between nodes and supports expert prefetching to pull all external experts to local CPU memory.

There are some research optimizes MoE training from model-system co-design perspective. For example, TA-MoE [190] proposes a topology-aware routing strategy for large-scale MoE training. TA-MoE abstracts the dispatch problem into an optimization objective to obtain the target dispatch pattern under different topologies and designs a topology-aware auxiliary loss based on the dispatch pattern. This approach adaptively routes the data to fit the underlying topology without sacrificing model accuracy.

Load Balance. Due to the sparse and conditional computing nature of MoE, a popular expert may receive more tokens than others in expert parallelism (usually caused by a poor routing algorithm), leading to serious load imbalance and affecting the training efficiency of MoE. FasterMoE [192] proposes the shadowing experts approach, which dynamically broadcasts the parameters of popular experts to all other GPUs based on the workload of previous iterations. By spreading the workload of popular experts across different devices, the shadowing experts approach reduces the impact of skewed expert popularity. SmartMoE [193] adopts a two-stage approach to search for the optimal parallel plan for load balance. First, SmartMoE designs a data-sensitive performance model that divides parallel plans into pools, where the cost of switching parallel modes within a pool is relatively low. Then, SmartMoE can switch to the appropriate parallelism (referred to as expert placement in SmartMoE) to keep load balance during the online stage. FlexMoE [194] found that the distribution of expert-to-device mapping does not shift significantly over a short period, so it introduces fine-grained replicated expert parallelism that duplicates heavy experts across multiple devices. FlexMoE monitors data workload and uses three placement adjustment primitives (i.e., expand, shrink, migrate) to generate optimal placement solutions if the balance ratio is exceeded. Prophet [195] presents a systematic, fine-grained, and efficient load balancing training method for large-scale MoE models. Taking the MoE model, device pool, and token distribution as inputs, Prophet's planner iteratively searches and evaluates expert placements and finally outputs a well-balanced expert placement. Additionally, Prophet hides the

overhead of these resource allocation operations using a layer-wise scheduling strategy.

4.2 Auto Parallelism

Given an arbitrary DNN model and a GPU cluster, there exists a vast array of options for parallelism, encompassing the partitioning of individual layers and their partitioning degrees. It is a time-consuming and knowledge-intensive process to design handcrafted hybrid parallelism approaches that can maximize the training efficiency, requiring expert understanding of the model architecture, hardware characteristics, and intricate trade-offs involved in parallelization strategies. Moreover, the efficient implementation of optimal parallelization strategies often necessitates substantial human efforts. To address these challenges, auto parallelism emerges as a promising solution, which seeks to automatically determine the most effective parallelization strategy for a given DNN model on a specific GPU cluster. By leveraging sophisticated algorithms and heuristics, auto parallelism systems can analyze the model architecture, hardware specifications, and performance characteristics to identify the optimal combination of parallelism techniques, such as data, tensor, and pipeline parallelism. This approach streamlines the process of optimizing the distributed training across various models and infrastructures, enhancing the overall efficiency and reducing the manual effort. Furthermore, auto parallelism can adapt to changing hardware configurations and model architectures, automatically adjusting the parallelization strategy to maintain the optimal performance. In the following, we categorize existing auto parallelism systems into general and transformer-specific frameworks, according to the targeted model architecture.

4.2.1 General Framework

General auto parallelism frameworks focus on automatically parallelizing general DNNs on a specific computation cluster. These frameworks typically follow a three-step process: (1) defining the search space of parallelization strategies; (2) developing performance models to measure the training efficiency of different strategies; (3) designing algorithms to efficiently identify the optimal parallelization strategy. Below we investigate different approaches according to the search space they cover.

Some works have explored the search space of hybrid data and pipeline parallelism strategies for DNN training optimization. These approaches focus on partitioning DNNs automatically and designing pipeline schedules to improve the pipeline utilization. PipeDream [134] measures the efficiency of pipeline partitions with the execution time of the slowest stage and develops a dynamic programming algorithm to partition the DNN evenly by minimizing the slowest stage. DAPPLE [152] builds an analytical model to estimate the execution time of one partition strategy and uses dynamic programming to determine the optimal pipeline partition. AutoPipe [205] constructs a simulator to simulate the pipeline execution and proposes a heuristic algorithm to obtain the balanced partition. AutoPipe also automatically splits the micro-batch to reduce the latency of the warm-up stage. Some device placement approaches [206]–[208] use reinforcement learning to predict the optimal operator placement for pipeline parallelism.

Researchers also explore the automated data and model parallelism by partitioning operators along different dimensions. OptCNN [198] partitions operators along all divisible dimensions in their output tensor and utilizes an analytical performance model to pick the optimal parallelization strategy, including the parallelizable dimensions and degree of parallelism, which defines how to parallelize an individual layer across different devices. FlexFlow [199] further extends the search space to Sample-Operator-Attribute-Parameter (SOAP), which includes almost all the divisible dimensions in input and output tensors, and introduces a novel execution simulator for accurate performance modeling. FlexFlow efficiently finds an optimal parallelization strategy with MCMC sampling. Tofu [200] and HyPar [201] develop dynamic programming algorithms that minimize the total communication cost rather than the end-to-end performance, to identify the optimal partition for each operator in the hybrid data and model parallelism space. TensorOpt [204] optimizes the parallelization strategy under a given memory budget with a frontier tracking algorithm. AutoMap [202] employs Monte Carlo Tree Search (MCTS) to select a sequence of partitioning rules defined by PartIR [212] for a set of selected important operators via a learned scorer. The whole parallelization strategy is propagated from the strategy via the selected operators.

Recent works also design approaches for automated data, model and pipeline parallelism. Piper [209] designs a two-level dynamic programming approach to find the optimal hybrid data, tensor and pipeline parallelism combined with activation recomputation. It first divides the model into small partitions for the pipeline and then splits operators within each partition. Alpa [21] formulates a comprehensive space by viewing parallelisms as two hierarchical levels: inter-operator and intra-operator parallelism. Then it automatically derives an efficient parallel execution plan at each parallelism level. Unity [210] jointly optimizes the parallelization and algebraic transformations by representing them as substitutions on a unified parallel computation graph. Aceso [211] proposes an iterative bottleneck alleviation approach to significantly reduce the search time. It identifies a performance bottleneck at every step and adjusts the strategy to mitigate the bottleneck until convergence. nnScaler [213] introduces three primitives to enable the composition of the search space with arbitrary partitioning and spatial-temporal scheduling of the partitioned model. Domain experts can apply constraints to the primitives to build effective and small search spaces, which can be automatically explored with low overheads. AutoDDL [215] customizes a coordinate descent algorithm by iteratively updating the SBP [214] distributions for each layer and quickly discover an optimal strategy with near-optimal communication cost.

General auto parallelism frameworks demand efficient system support for various parallelization strategies, in addition to fast optimization algorithms for optimal parallelization strategy discovery. This is because parallelism often involves complex computation and communication operators, especially for model parallelism that partitions operators. Prior works have developed efficient systems that enable a wide range of parallelization strategies, either by building upon modern DL frameworks [21], [213] or

implementation from scratch [199]. Mesh-TensorFlow [196] observes the intrinsic complexity of implementing a parallelization strategy, and first proposes to abstract the device cluster into a multi-dimensional mesh, and abstract parallelism into partitioning the iteration space (i.e. tensor dimensions). By mapping the tensor and mesh dimensions, a hybrid data and model parallelism strategy can be easily implemented with high performance. For example, data and model parallelisms split the batch and hidden dimensions, respectively. GSPMD [197] further provides a unified way to achieve various general parallelism schemes with simple tensor sharding annotations based on JAX [250] and XLA [251]. OneFlow [214] proposes SBP (Split, Broadcast, Partial-value) abstraction for partition and allows users to specify the placement and SBP signature for tensors to implement different parallelization strategies. PartIR [212] decouples the model from its partitioning and designs a compiler stack for users to compose SPMD sharding strategies incrementally via a schedule. Similar to TVM [252], Slapo [203] defines a comprehensive set of schedule primitives for parallelization and subgraph optimization like operator fusion and activation checkpointing. These schedules are decoupled from execution and preserves the original model structure for progressive optimization.

4.2.2 Transformer-Specific Framework

As LLMs are based on the transformer architecture, recent works tailor automated systems for transformers. DeepSpeed-Autotuning [216] automatically tunes the system knobs to figure out good performance-relevant configurations in the user-defined tuning space, including the degree of parallelism. Galvatron [217] designs a dynamic programming algorithm to generate the most efficient hybrid data, tensor and pipeline parallelism strategy. Merak [218] introduces an automatic model partitioner for non-intrusive automatic parallelism and a high-performance 3D parallel runtime engine to enhance the utilization of available resources. Colossal-AI [219], [253] provides a unified interface for modular usage of hybrid data, tensor, sequence and pipeline parallelism. Galvatron-BMW [220] extends the space of Galvatron to include sharded data parallelism and activation recomputation, and searches for the optimal strategy considering both the memory consumption and computation while maximizing the hardware utilization.

4.3 Heterogeneous Parallelism

The escalating computational demands of LLM training have spurred advancements in heterogeneous hardware, which harnesses diverse computing resources and globally distributed devices. This heterogeneity is also reflected in model architectures, particularly with Reinforcement Learning from Human Feedback (RLHF). Utilizing heterogeneous hardware and diverse model architectures has become essential for the efficient training of LLMs.

4.3.1 Heterogeneous Hardware

The massive computational requirements of LLM training have driven the evolution of accelerators, leading to clusters with mixed device types and uneven interconnect bandwidths. Additionally, modern data and computing clusters

are often distributed globally due to factors such as power shortages. These phenomena have motivated the adoption of heterogeneous parallelisms, which leverage diverse computing resources and geographically distributed devices to accelerate LLM training.

Some works leverage heterogeneous computing resources, such as CPUs, GPUs, and specialized accelerators, to enhance the performance of LLMs. The distinct computation, memory capacity and interconnect bandwidth of these devices introduce challenges for efficient LLM pre-training. HetPipe [221] partitions the heterogeneous cluster into multiple virtual works. Each virtual work processes mini-batches with the pipeline parallelism, and different virtual works employ asynchronous data parallelisms to improve the throughput. AccPar [222] proposes flexible tensor partitioning to balance the computation of different accelerators and uses dynamic programming to automatically decide tensor partitioning among heterogeneous devices for DNNs. Whale [223] proposes a unified abstraction to ease the efforts for parallel training of giant models on heterogeneous clusters. It seamlessly adapts to heterogeneous GPUs through automatic graph optimizations and balances the workloads with hardware information. AMP [224] utilizes a heterogeneous-aware performance model to find the optimal hybrid data, tensor and pipeline parallelism strategy. HPH [226] arranges different GPUs into stages according to the compute-communication ratio in descending order and formulates the model partitioning as an integer programming problem to minimize the iteration time. Pathways [225] employs a sharded dataflow model and asynchronous gang-scheduling to efficiently execute ML models on heterogeneous cluster. SDPIPE [227] introduces a semi-decentralized scheme that decentralizes the communication model synchronization and centralizes the process of group scheduling for pipeline parallelism to utilize heterogeneous devices. HAP [228] uses an A*-based search algorithm to generate the optimal tensor sharding strategy, sharding ratio across heterogeneous devices and the communication methods for distributed training. PipePar [229] proposes a dynamic programming algorithm to partition the model into stages for pipeline considering both the heterogeneity of GPUs and network bandwidths.

Some other works explore geo-distributed devices, featuring the low network bandwidth, to enhance the training efficiency. Yuan et al. [230] partition the LLMs into computational tasklets and propose a novel scheduling algorithm to efficiently utilize a group of heterogeneous devices connected by a slow heterogeneous network for hybrid data and pipeline parallelism. SWARM parallelism [231] partitions the model into equal-sized stages and prioritizes routing inputs to stable peers with lower latency for workload balance. It also adaptively moves devices across stages to maximize the training throughput. FusionAI [232] splits the training computation graph (DAG) into subgraphs (sub-DAG) and generates a load balanced task schedule to utilize heterogeneous consumer GPUs connected with low bandwidth for pipeline training. Communication compression approaches, like CocktailSGD [254], can also be leveraged to train LLMs efficiently in low-bandwidth clusters.
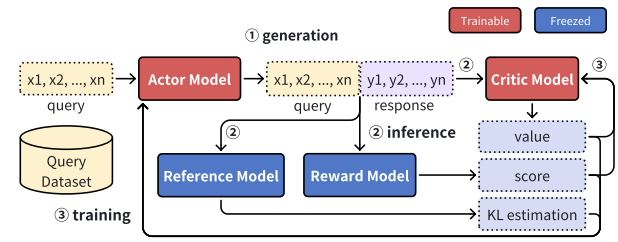

Fig. 10: An example of RLHF. **Inference process**: ① The actor model generates a response from a given query. ② The critic model, reward model, and reference model use the query and response pairs to generate the value, score, and KL divergence required for training through inference. **Training process**: ③ The actor model and critic model use the data collected in the inference process to update their weights through gradient descent.

4.3.2 Heterogeneous Model

During the LLM training process, heterogeneity is not only reflected in the hardware, but also in the model. Training may involve the interaction of several different models. A specific example is Reinforcement Learning from Human Feedback (RLHF). RLHF is a training method that aims to align AI systems more closely with human preferences [255], leveraging human's advantages in judging appropriate behavior rather than demonstrating. This method has received widespread attention, especially for fine-tuning large language models. However, due to the particularity of the Proximal Policy Optimization (PPO) [256] algorithm, the model heterogeneity is introduced into the RLHF training, making the training process of RLHF very different from pre-training and supervised fine-tuning.

In principle, RLHF consists of three different stages: the stage 1 is supervised fine-tuning, the stage 2 is the training of the reward model, and the stage 3 is PPO training. Model heterogeneity is presented in stage 3, as shown in Fig. 10. The PPO training stage consists of two different processes, namely the inference process that generates data, and the training process that updates the weights of the actor model and critic model. PPO training is performed via the collaboration of these two processes. Moreover, the training stage introduces higher memory cost, as we need to serve several copies of auto-regressive generation models and reward models at the same time, and more time costs, because we must wait for the experience generation to be completed before updating the weights.

Many frameworks have been proposed for RLHF training. For instance, DeepSpeed-Chat [233] uses Hybrid Engine to seamlessly switch model partitioning between training and inference, such as using tensor parallelism to improve throughput during inference and using ZeRO [145] or LoRA [257] to improve memory utilization during training, providing outstanding system efficiency for RLHF training. HuggingFace TRL [234] can make full use of various parameter-efficient fine-tuning (PEFT) methods, such as LoRA or QLoRA [258], to save memory cost, and use a dedicated kernel designed by unsloth [259] to increase the training speed of RLHF. ColossalAI-Chat [253] is another

end-to-end RLHF training framework that also supports LoRA and supports the use of ZeRO [145] to reduce memory redundancy.

However, the above work adopts a flattening strategy for model placement, that is, placing the four models in RLHF on the same device, and then using methods such as ZeRO or LoRA to minimize memory cost. But using only ZeRO will lead to memory bottlenecks when training larger models, while using efficient parameter fine-tuning strategies such as LoRA will damage model performance. To solve this problem, OpenRLHF [235] uses Ray [260] and vLLM [261] to distribute the reward models to different devices, avoiding placing all four models in PPO on the same device. Similarly, Adpative Placement and Parallelism (APP) framework [236] proposed two other model placement strategies, namely Interleaving Strategy and Separation Strategy. It captures the fact that the generation part and the training part can run independently during PPO training, and some serialization can be eliminated by placing them on different devices, which introduces additional communication but can overlap well with computing.

Meanwhile, there are some works that apply the parallel strategies in the first two stages to the stage 3 of RLHF in a fine-grained scheduling manner. For example, ReaLHF [237] switches the most suitable parallel mode for different sub-stages in stage 3 by redistributing parameters, which greatly increases the optimization space. PUZZLE [238] reschedules the order of task execution according to the affinity of different stages, so that stages with better affinity can effectively cover execution and improve training efficiency.

5 COMPUTATION OPTIMIZATIONS

Today's AI accelerators offer unprecedented computational capabilities in terms of FLOPs. However, effectively utilizing these FLOPs to their full potential requires sophisticated optimization techniques. This section introduces systems and techniques of computation optimizations to effectively utilize GPU FLOPs. We first elaborate operator optimizations including the core attention operator optimizations and automatic optimizations via compilers. Remarkable performance for operator and computing graphs is gained based on exploiting massive parallelism and efficient multi-level memory access concerning the underlying hardware features. Second, mixed-precision training is detailed where computations are accelerated benefiting from reduced precision. 16-Bit floating point mixed training has been the de facto method in most training systems. Low-bit fixed points as low as 1-bit have been studied and employed for high training efficiency.

5.1 Operator Optimizations

Operator optimizations can be categorized into manual and automatic optimizations. Manual optimizations mainly focus on the attention operator, while automatic optimizations are applied more broadly.

5.1.1 Manually Optimized Attention Operator

Attention, as the core of transformer architectures, plays a crucial role in the training efficiency of LLMs. Given a query q, and lists of keys $k_1, k_2, ..., k_n$ and values $v_1, v_2, ..., v_n$, where $q, k_i, v_i \in \mathbb{R}^d$, the attention is computed as follows,

$$s_i = dot(q, k_i), \quad s_i^{'} = softmax(s_i) = \frac{e^{s_i}}{\sum_j e^{s_j}}, \quad o_i = \sum_i v_i s_i^{'}.$$

The self-attention exhibits quadratic time and memory complexity relative to sequence length. The substantial memory consumption and frequent access to high-bandwidth memory (HBM) imposed by self-attention constrain both the performance and the context length of transformer models. Extensive work is presented to optimize self-attention. We focus on exact attention optimizations while lossy optimizations, like linear attention, are out of our scope.

Memory-efficient attention is primarily proposed to mitigate the large memory cost. Rabe et al. [287] prove that self-attention needs $O(logn)$ memory complexity instead of $O(n^2)$. By employing lazy softmax, the division by $\sum_j e^{s_j}$ in softmax can be delayed to the very end of the attention operation. Thus the summation could be processed incrementally which requires just a scalar (i.e. $O(1)$) to maintain the intermediate result but not change the output. The self-attention requires extra $O(logn)$ memory complexity to keep the additional index into the list of queries to compute the results to all queries sequentially.

The FlashAttention series further demonstrate fast and memory-efficient exact attention with IO-awareness, high parallelism, and balanced workloads on GPU. In FlashAttention [115], an IO-aware tiling algorithm is proposed to reduce the number of memory reads/writes between slow HBM and fast on-chip SRAM based on the online softmax. More specifically, the softmax could be calculated one block at a time by tracking the normalization statistics including the maximum score and the sums of exponentiated scores. The tiling algorithm thus fuses all the computation operation chain in self-attention including matrix multiply, softmax, matrix multiply, etc, in one cuda kernel for reduced HBM access. FlashAttention-2 [116] further improves the low-occupancy and unnecessary shared memory reads/writes in FlashAttention with additional parallelism in sequence length dimension and improved warp-level scheduling for data sharing inside a thread block. Besides, the popular training systems [174] generally employ FlashAttention-2 for high performance. FlashAttention-3 [262] speeds up attention on H100 GPU by excavating the newly presented hardware capabilities as the former FlashAttention implementations are based on A100 GPU. An interleaved block-wise GEMM and softmax algorithm is redesigned based on FlashAttention-2 to hide the non-GEMM operations in softmax with the asynchronous WGMMA instructions for GEMM. Besides, by leveraging the asynchrony of the Tensor Cores and Tensor Memory Accelerator (TMA), overall computation is overlapped with data movement via a warp-specialized software pipelining scheme. Blockwise Parallel Transformer (BPT) [263] further reduces the substantial memory requirements by extending the tiling algorithm in FlashAttention to fuse the feedforward network.

The attention operation is also optimized on various architectures by leveraging hardware-specific features. For instance, SWattention [264] designs a two-level blocking attention algorithm to exploit the underlying hardware of the
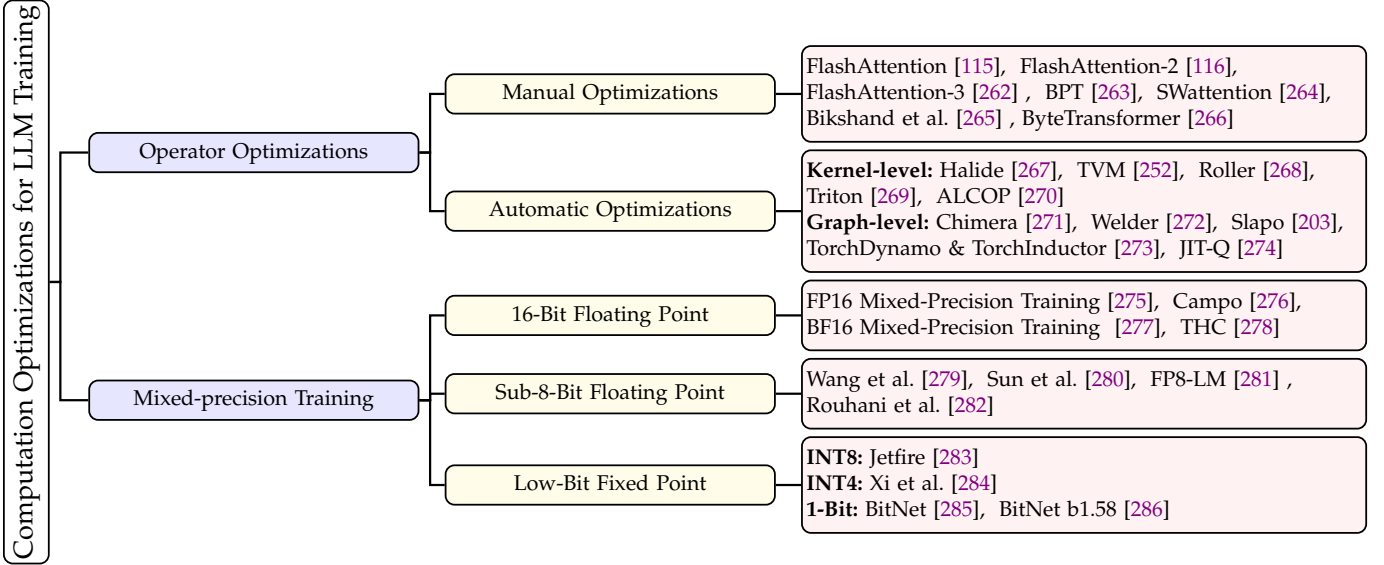
Computation Optimizations for LLM Training

Operator Optimizations	Manual Optimizations	FlashAttention [115], FlashAttention-2 [116], FlashAttention-3 [262] , BPT [263], SWattention [264], Bikshand et al. [265] , ByteTransformer [266]
	Automatic Optimizations	**Kernel-level:** Halide [267], TVM [252], Roller [268], Triton [269], ALCOP [270] **Graph-level:** Chimera [271], Welder [272], Slapo [203], TorchDynamo & TorchInductor [273], JIT-Q [274]
Mixed-precision Training	16-Bit Floating Point	FP16 Mixed-Precision Training [275], Campo [276], BF16 Mixed-Precision Training [277], THC [278]
	Sub-8-Bit Floating Point	Wang et al. [279], Sun et al. [280], FP8-LM [281] , Rouhani et al. [282]
	Low-Bit Fixed Point	**INT8:** Jetfire [283] **INT4:** Xi et al. [284] **1-Bit:** BitNet [285], BitNet b1.58 [286]

Fig. 11: Studies on computation optimizations for distributed LLM training.

new Sunway architecture, building upon FlashAttention. Similarly, Bikshand et al. [265] implement FlashAttention-2 on the H100 GPU using the Cutlass library. They utilize the TMA and WarpGroup Matrix-Multiply-Accumulate (WG-MMA) instructions to optimize data copying and GEMM operations, respectively. Additionally, tensor layout transformations and software pipelining of data copying and computations between the two GEMMs are carefully designed based on the Cutlass library.

Attention mechanisms are also optimized for variable-length sequences, which are common in distributed LLM training. These variable-length sequences can incur significant memory and computation costs if padded to the maximum length. FlashAttention-2 efficiently handles variable-length inputs by parallelizing the sequence length dimension inseparably. ByteTransformer [266] focuses on padding-free transformers for variable-length inputs, maintaining a position array during computation. This array records the mapping relationship of valid tokens between the original tensor and the intermediate packed tensor. The fused Multihead Attention algorithm for long sequences employs optimized grouped GEMM for unpadded tensors. This optimization reduces the memory and computation overhead associated with padding, thereby enhancing performance.

5.1.2 Automatic Optimizations via Compilers

DNN compilers play an important role in optimizing key computations in LLM training. Highly efficient kernels of operators are generated automatically which mitigates the burden of library-based kernel optimizations on diverse hardware vendors to a great extent. Operator fusion is performed by analyzing the computation graphs automatically in the training process.

Efficient Operator Kernel Generation. Halide [267] and TVM [252] generate high-performance operator implementations automatically, relying on multiple effective schedule primitives that exploit parallelism and data locality on various backends. Furthermore, Roller [268] optimizes the

cost of searching for optimal alternatives in the large search space of kernel implementations. It primarily generates a tile kernel consisting of Load, Store, and Compute interfaces, following which the complete operator kernel is constructed by a scale-up-then-scale-out approach. Triton [269] provides a C-based language and compiler that facilitates expressing and optimizing tile tensor programs for competitive performance. In particular, effective optimizations such as hierarchical tiling and shared memory allocation are supported via machine-dependent compiling passes. ALCOP [270] performs automatic load-compute pipelining to overlap the high-latency memory access with computations for operators on GPUs. Multi-stage pipelining is utilized by pipeline buffer detection as well as sophisticated index analysis and substitution in complicated loop structures.

Graph-level Optimizations for Operator Fusion. With the disparity of speed of computing cores and memory bandwidth enlarging, modern DNNs are restricted by memory access. Data reuse among inter-operators is excavated via operator fusion using compilers. Plenty of compiler works [288]–[291] performs operator fusion by setting expert rules. Particularly, Chimera [271] works on optimizing compute-intensive operator chains. The operator chain is firstly decomposed into a series of computation blocks and the optimal block execution order is then selected to maximize data reuse according to an analytical model. In addition, replaceable microkernels are designed to leverage hardware-specific intra-block optimizations. Welder [272] lowers the computing graph into a tile-level data-flow graph whose nodes are operator tiles and edges are marked with the memory level of the tensor data reused by the connected nodes. Operator fusion combinations that maximize data reuse across different levels of memory hierarchies are searched at the tile level.

Pytorch2 [273] presents two extensions, i.e. a Python-level JIT compiler TorchDynamo and the corresponding compiler backend TorchInductor, to enable more robust graph compilation on various back-ends for remarkable

performance improvement without sacrificing the flexibility of Python. Slapo [203] proposes a schedule language to decouple model execution from definition. Declaring a set of schedule primitives, users could convert the model for high-performance kernels. JIT-Q [274] proposes just-in-time quantization for weights which enables storing only a high-precision copy of weights during training and creates low-precision weight copies based on the in-memory ALU augmentations of the commercial PIM (processing-in-memory) solutions.

5.2 Mixed-precision Training

Low-precision training is an effective methodology to reduce the computation, storage, and communication costs in training large-scale models. Nowadays LLM training generally leverages FP16 and BF16 data types. In particular, BF16 can represent the same range of values as that of FP32. BF16 training is utilized in models such as BLOOM [292] since the loss slowly diverges when the loss scalar becomes too low in FP16 [293]. However, fast bfloat16 support is only available on TPUs, or GPUs developed with or after the NVIDIA Ampere series. Furthermore, mixed-precision training and techniques such as loss scaling are exploited to ensure numerical stability due to the limited dynamic range represented by reduced precision. 8-Bit or even lower-bit training is also becoming the focus of quantitative research.

5.2.1 16-Bit Floating Point

Popular training systems often employ FP16/BF16 mixed-precision strategies to reduce precision during training, as highlighted by works like Megatron-LM [20] and Colossal-AI [253]. The FP16 mixed-precision training scheme [275] utilizes the IEEE half-precision format to store weights, activations, and gradients for forward and backward arithmetic operations. To maintain model accuracy at reduced precision, a single-precision copy of weights is kept for accumulation at each optimizer step. Loss scaling is also applied to preserve the values of small-magnitude gradients. Campo [276] optimizes the casting cost incurred by conversions between FP32 and FP16 through automatic graph rewriting. This is crucial since the casting cost can sometimes negate the performance benefits of low precision. Campo also employs offline-trained linear regression models to predict casting costs and execution times for FP32 and FP16 operations. BF16 [277] is also widely used in mixed-precision training across various fields [294], [295]. It has the same representational range as FP32 and does not require hyperparameter tuning for convergence. In addition, THC [278] addresses computational overhead in parameter server architectures by eliminating the need for decompression and compression. THC enables direct aggregation of compressed gradient values through the Uniform Homomorphic Compression property, thus enhancing efficiency.

5.2.2 Sub-8-Bit Floating Point

With the newly released chips characterized by lower precision data types such as FP8, mixed-precision training is designed to train with lower precision. Newly designed data formats combined with the techniques to ensure numerical stability are primarily leveraged to enable FP8 training for deep learning neural networks. Wang et al. [279] use a new FP8 floating point format for numerical representation of data as well as computations. Chunk-based computations and stochastic rounding are utilized in the floating point accumulation and weight update process, respectively, to preserve model accuracy. Sun et al. [280] propose hybrid 8-bit floating point training across the whole spectrum of deep learning models without accuracy degradation. The novel hybrid FP8 formats utilize different exponent bits and mantissa bits for forward and backward propagation, respectively, since forward and backward passes have different optimal balances between range and precision. Besides, the techniques such as loss scaling are used to avoid accuracy degradation. With the maturation of more accelerators with FP8 data types, an FP8 automatic mixed-precision framework (FP8-LM) [281] for training LLMs based on NVIDIA H100 GPU [296] is proposed, where 8-bit gradients, optimizer states, and distributed parallel training are gradually incorporated and FP8 low-bit parallelism including tensor, pipeline, and sequence parallelism is specified. Besides, precision decoupling and automatic scaling are designed to solve the data underflow or overflow issues due to the narrower dynamic range and reduced precision. FlashAttention-3 also employs block GEMM quantization and incoherent processing that exploits hardware support for FP8 low-precision on H100 GPU. Furthermore, Rouhani et al. [282] train LLMs at sub-8-bit weights, activations, and gradients with minimal accuracy loss by utilizing micro scaled data formats that associate scaling factors with fine-grained sub-blocks of a tensor.

5.2.3 Low-Bit Fixed Point

Low-bit fixed point training is also studied for LLM training. Jetfire [283] maintains an INT8 data flow where inputs and outputs are loaded and stored in INT8 data formats to accelerate both compute-bound linear operators and memory-bound non-linear operators. In addition, tiling algorithms are utilized to excavate shared memory data access with a per-block quantization method, where higher precision computations are performed, i.e. INT32 for WMMA tensor core operations for linear operators and FP32 for non-linear operations, to maintain the accuracy of pretrained transformers. Xi et al. [284] propose a novel INT4 training algorithm for transformer models. In the forward propagation, the activation matrix is firstly transformed into a block diagonal Hadamard matrix to alleviate the accuracy degradation caused by outliers in the activation, and the transformed matrix is then quantized. In the backward propagation, bit splitting and leverage score sampling are exploited to choose informative gradients for quantization based on the structural sparsity of activation gradients.

Recently, low-precision training for LLMs has advanced to using 1-bit precision. BitNet [285] employs a novel low-bit precision matrix multiplication within transformer blocks, utilizing weights that are 1-bit and activations that are 8-bit. The model weights are centralized around zero to maximize capacity within the limited numerical range, then binarized to +1 or -1 using the signnum function. To ensure training stability and accuracy, the gradients, optimizer states, and a high-precision latent weight copy are maintained for parameter updates. Building on BitNet, BitNet b1.58 [286] further
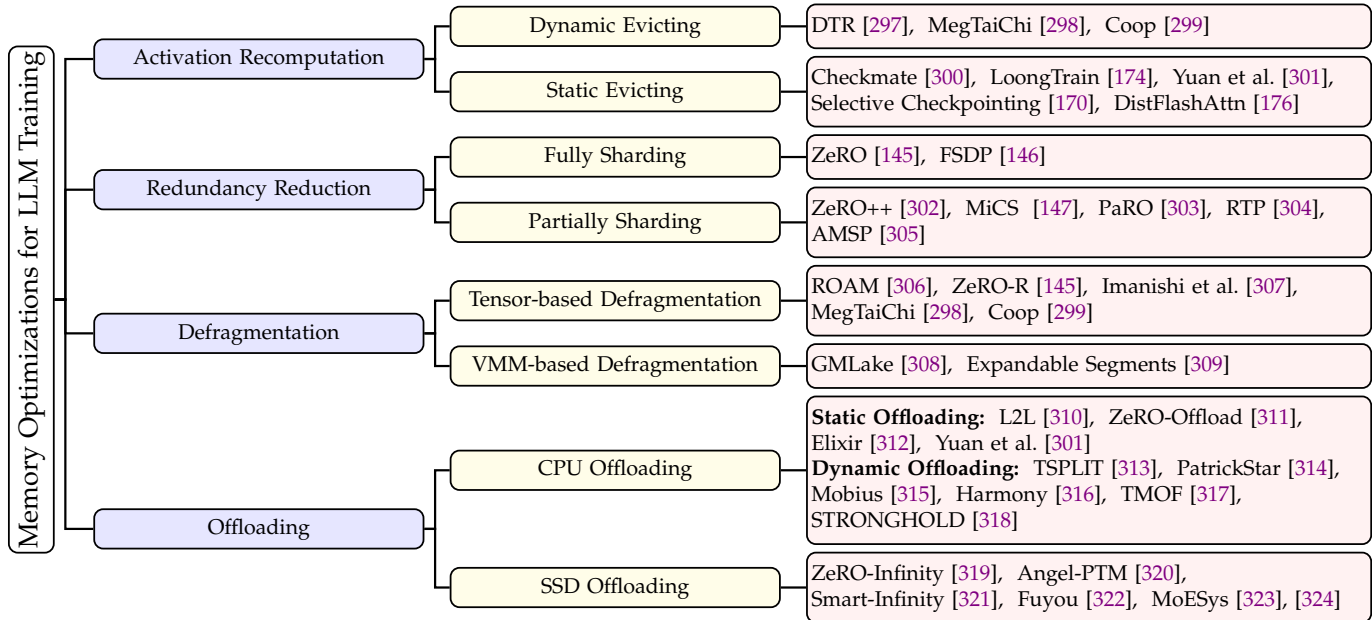
Fig. 12: Studies on memory optimizations for distributed LLM training.

enhances modeling capability by lowering model weights to ternary values {-1, 0, 1}. The weight matrix is scaled by its average absolute value, and each value is rounded to the nearest integer among -1, 0, and +1.

6 MEMORY OPTIMIZATIONS

Memory consumption during the training of LLMs can be categorized into four key components: model states, activations, temporary buffers, and memory fragmentation.

- **Model States:** Model states encompass the memory consumed by the optimizer states, gradients, and model parameters. In mixed-precision training [275], model parameters and activations are stored in 16-bit precision. When training a model with Φ parameters, 4Φ bytes are needed to store parameters and gradients. The 32-bit copies of the parameters, momentum, and variance each require 4Φ bytes, totaling 12Φ bytes. Therefore, the overall memory requirement for storing model states is 16Φ bytes.
- **Activations:** Activations refer to the tensors generated during the forward pass. These tensors are essential for gradient computation during the backward phase.
- **Temporary Buffers:** Temporary buffers are used to store intermediate results. For example, operations such as gradient AllReduce often fuse gradients in a bucket into a single flattened buffer before applying the operation to enhance throughput.
- **Memory Fragmentation:** Memory fragmentation can lead to scenarios where memory requests fail despite having a large amount of available memory. This occurs because usable memory can become fragmented, and there is insufficient contiguous memory to satisfy the memory request [145].

To address memory constraints of LLM training, various memory-efficient techniques have been proposed. These in-clude activation recomputation strategies, which trade increased computation for reduced memory usage; redundancy reduction methods that minimize data duplication across training processes; defragmentation techniques that optimize memory allocation and deallocation to reduce fragmentation and improve memory utilization; and swap and offload approaches that leverage CPU memory and NVMe SSDs to supplement GPU memory. Figure 12 outlines the taxonomy of these optimizations for memory-efficient LLM training.

6.1 Activation Recomputation

During the backward phase of model training, activations are essential for computing gradients. As model sizes increase, the memory required to store these activations during training can exceed GPU memory capacity, thereby limiting the scale of the models that can be trained. Activation recomputation [325] provides a solution by strategically discarding certain activations during the forward pass and recomputing them as needed during the backward pass. This approach has become a de facto method for reducing memory consumption in LLM training. The key to effective activation recomputation is balancing the memory savings against the additional computational overhead.

We categorize these methods into two primary approaches: static evicting and dynamic evicting. Static evicting methods typically involve the formulation of evicting strategies tailored to specific model architectures or modules. In contrast, dynamic evicting methods make decisions in real-time without prior knowledge of the model. Although static approaches necessitate modifications for new models, the structure of the majority of LLMs share similar architectures, enabling the general application of these strategies during LLM training. Despite their inherent flexibility, dynamic evicting methods have not been widely adopted in the training of LLMs. Nevertheless, we still ex-

plore some related works in this section for further reference.

6.1.1 Static Evicting

Static evicting involves establishing a fixed plan for discarding activations during the forward pass and later recomputing them during the backward pass. Checkmate [300] formulates this activation recomputation problem as a mixed integer linear program to determine the optimal rematerialization plan for static deep learning models. However, Checkmate struggles to scale to large models like LLMs due to the vast search space.

Recently, several works have proposed customized activation recomputation policies tailored for LLM training. Selective-checkpointing [170] selectively discards the activations of memory-intensive attention modules. FlashAttention [115] fuses the attention module into a single kernel, and also employs selective-checkpointing to reduce memory consumption. DistFlashAttn [176] addresses the high computation overhead in long sequences caused by the recomputation of attention modules, employing a rematerialization-aware gradient checkpointing strategy. Specifically, DistFlashAttn places checkpoints at the output of the FlashAttention kernel instead of at the Transformer layer boundary, thereby removing recomputation in the attention module during the backward pass and only requiring storage of its output. LoongTrain [174] introduces selective-checkpoint++, which further optimizes the checkpointing process, particularly for training with long sequences, by adding attention modules to a *whitelist*. This method saves the attention output and softmax statistics (`softmax_lse`). During the forward pass, it saves the outputs of the modules in the whitelist, and during the backward pass, it retrieves these stored outputs instead of recomputing them, continuing the computation graph and thus reducing the need for recomputing attention.

Unlike recent works that predominantly focus on handcrafted checkpointing policies on attention modules for LLM training, Yuan et al. [301] carefully measure the minimum computation cost required to reconstruct each activation tensor during model training. They derive a Pareto frontier of memory and computation costs by enumerating all possible checkpointing methods. From this Pareto frontier, they select a solution that optimally balances computation and memory costs.

6.1.2 Dynamic Evicting

Dynamic evicting makes real-time decisions on which activations to discard and recompute based on the current state of the training process. DTR [297] proposes a greedy online algorithm to heuristically evict and rematerialize tensors at runtime for both static and dynamic models. MegTaiChi [298] introduces a dynamic tensor evicting that leverages the access patterns of tensors tracked at runtime. Coop [299] proposes to mitigate the memory fragmentation issue caused by activation recomputing methods due to evicting tensors without considering their contiguity. Coop employs an efficient sliding window algorithm to ensure that only contiguous memory blocks are evicted, thereby minimizing memory fragmentations.

6.2 Redundancy Reduction

Traditional data parallel approaches replicate the entire model state across all GPUs, which leads to substantial redundant memory usage. Redundancy reduction techniques are proposed to optimize memory usage by eliminating or reducing memory redundancies on each device. These techniques often seek to balance memory efficiency with the induced communication overhead, thereby facilitating training of larger scale or batch size with acceptable costs.

6.2.1 Fully Sharding

The Zero Redundancy Optimizer (ZeRO) [145] optimizes memory redundancies by fully sharding model states across all GPUs through three stages: ZeRO-1, ZeRO-2, and ZeRO-3. ZeRO-1 globally distributes optimizer states across all GPUs. During the training, each GPU conducts independent forward and backward propagation to compute gradients, which are subsequently synchronized across all GPUs within the data parallel group using an ReduceScatter operation. Each GPU is responsible for updating specific shard of the model parameters. Following this, the updated model parameter shards are collected from other GPUs using an AllGather operation, ensuring that all GPUs have the latest model parameters. ZeRO-1 reduces the memory consumption of optimizer states from 12Φ to $12\Phi/N$, where N is the size of data parallelism. Building upon ZeRO-1, ZeRO-2 further shards the gradients across all GPUs and each GPU only updates its parameter shard, reducing the memory required for holding gradients from 2Φ to $2\Phi/N$. ZeRO-3 partitions parameters in addition to optimizer states and gradients. Each GPU only holds a part of the parameters. When the parameters from remote GPUs are needed for the upcoming computation, they are collected by an AllGather operation and discarded afterward. In ZeRO-3, each GPU holds only the weights, gradients, and optimizer states corresponding to its specific parameter partition, reducing the overall memory consumption from 16Φ to $16\Phi/N$. ZeRO is widely adopted by numerous frameworks, such as DeepSpeed [183], PyTorch-FSDP [146], and ColossalAI [253].

6.2.2 Partially Sharding

ZeRO faces communication challenges since the latency of collective communication operations increases with the communication scale. There exists a trade-off between memory utilization and communication cost in distributed LLM training. Optimizing communication overhead can be achieved by sharding the model states across smaller groups of GPUs, which are smaller sets of GPUs within a large GPU cluster. This approach reduces inter-node communications and communication scales, though it may lead to higher memory usage due to increased redundancy of model states. The key is to balance the communication scale with memory utilization [305].

Several approaches building upon the ZeRO framework have been proposed to address the communication inefficiencies while improving memory utilization. ZeRO++ [302] partitions all model states globally across all devices following ZeRO-3 and further introduces a secondary shard of parameters within subgroups of GPUs. In the forward phase, it collects parameters leveraging the primary shard across

all GPUs and maintains a secondary shard of parameters within subgroups, typically within the same node. During the backward phase, it collects parameters from this secondary shard, reducing communication scale and inter-node communications. Additionally, ZeRO++ uses quantization to compress parameters and gradients, effectively diminishing communication volume with a trade-off in accuracy. MiCS [147] and FSDP [146] shards all model state components within subgroups and replicates them across subgroups, thereby reducing communication scale and consequently communication latency, leading to enhanced training performance. AMSP [305] and PaRO [303] incorporate three flexible sharding strategies, including Full-Replica, Full-Sharding, and Partial-Sharding, allowing each component within the model states to independently choose a sharding strategy. AMSP formulates an optimization problem to find the optimal sharding strategy that minimizes communication costs under memory constraints. In addition, AMSP proposes a customized communication and computation overlap strategy, incorporating these flexible sharding strategies to achieve optimized training efficiency. RTP (Rotated Tensor Parallelism) [304] seeks to minimize memory duplication by strategically sharding activations and rotating weights/gradients.

6.3 Defragmentation

GPU memory fragmentation refers to the scattered, unusable chunks of GPU memory that arise between adjacent tensors. This problem is particularly pronounced during the training of LLMs due to the varying lifetimes of different tensors and the inefficient memory allocation and deallocation schemes of general deep learning frameworks, such as PyTorch [240] and TensorFlow [326]. Furthermore, memory optimization techniques like recomputation and offloading exacerbate the issue by introducing more frequent and irregular memory allocation and deallocation requests [299], [306], [308]. The fragmentation problem could cause high peak memory and out-of-memory (OOM) errors limiting the batch size and overall training efficiency. Defragmentation efforts are proposed to mitigate these issues through memory management techniques.

6.3.1 Tensor-based Defragmentation

Deep learning frameworks typically use a caching allocator with a memory pool to enable fast memory allocation and deallocation without requiring device synchronization. Several approaches have been proposed to reduce memory fragmentation based on the tensor allocation and deallocation scheme in the caching allocator. ROAM [306] co-optimizes the execution order of operators and tensor allocation by considering tensors' lifetimes and sizes. It introduces an efficient tree-based algorithm to search for an execution plan that maximizes tensor reuse and reduces data fragmentation. ROAM has been evaluated in single-GPU scenarios, specifically with the largest model being the 1.5B GPT-2 XL [5], but it has not yet been tested in distributed training scenarios with larger models, where computation graphs can become significantly larger. Imanishi et al. [307] present an offline optimization approach by modeling tensor allocation as a 2D bin-packing problem. In this model,

each tensor allocation is represented as a vertically movable rectangle, reflecting periodic allocation patterns during model training. They propose a heuristic algorithm using simulated annealing to optimize the topological ordering of allocations, aiming to minimize fragmentation. While effective, this method may struggle with scalability issues when applied to LLMs due to the high number of allocations and complex patterns involved. MegTaiChi [298] and Coop [299] consider memory fragmentation issues when evicting activation tensors for reducing memory consumption.

6.3.2 VMM-based Defragmentation

GMLake [308] and PyTorch expandable segments [309] propose to mitigate fragmentation by utilizing the virtual memory management (VMM) functions of the low-level CUDA driver application programming interface. This low-level API provides developers with direct control over the GPU's virtual memory operations, such as reserving, mapping, and managing virtual memory addresses. Building on this, GMLake [308] introduces a virtual memory stitching mechanism that consolidates non-contiguous memory blocks into larger ones through virtual memory address mapping, minimizing data movement and copying. Similarly, PyTorch's expandable segments [309] enable allocated memory segments to be expanded to larger sizes for reuse. Both approaches are transparent to different models and memory-efficient training techniques and can be seamlessly integrated into existing deep learning frameworks. Furthermore, GMLake demonstrates excellent scalability on multi-GPUs with minimal overhead and does not require modification to user code. PyTorch-v2.1 has also integrated expandable segments.

6.4 Offloading

To enable efficient training of LLMs on fewer GPUs, various works leveraging swap and offload methods have been proposed. These techniques transfer parts of the computation and data from GPU memory to external resources, which are inexpensive and slower but enjoy vast capacity.

6.4.1 CPU Offloading

Numerous studies have proposed methods to efficiently utilize CPU memory to enhance distributed LLM training. These techniques can be broadly categorized into two main approaches: Static Offloading and Dynamic Offloading.

Static Offloading. Static offloading methods involve a predetermined allocation of model components between GPU and CPU memory. L2L [310] manages and moves tensors layer by layer. L2L synchronously fetches tensors required for the upcoming computational layers into GPU memory while keeping the tensors for the remaining layers stored in CPU memory. L2L allows scaling the models to arbitrary depth but fails to scale across multi-GPUs. In contrast, ZeRO-Offload [311] concentrates on multi-GPU training. It holds model parameters on GPU, and stores optimizer states and gradients on CPU memory. In addition, it offloads optimizer update computation to the CPU. This method enables the training of up to 70B models with 16 V100s. However, ZeRO-Offload can leave some GPU memory unused and suffers from slow CPU optimizer updates [312].

To address this issue, Elixir [312] employs a search engine to find the optimal combination of memory partitioning and offloading by leveraging pre-runtime model profiling. Unlike ZeRO-Offload, Elixir effectively utilizes all available GPU memory by partitioning both the model states and optimizer chunks between GPU and CPU. Mobius [315] tackles multi-GPU training on commodity servers with limited inter-GPU bandwidth and high communication contention by introducing a pipeline parallelism scheme. This scheme assigns each GPU multiple stages and dynamically swaps them between GPU and CPU memory. Additionally, Mobius optimizes communication through prefetching and cross-mapping to reduce overhead and contention. Yuan et al. [301] propose to mitigate the activation bottleneck by offloading and reloading activations at the granularity of pipeline stages while maximizing the overlap between activation transmission with computation, thereby avoiding slowing the training process. Compared to other offloading efforts, this work focuses more on improving the balance between computation and memory utilization rather than training with extremely tight memory budgets.

6.4.2 Dynamic Offloading

Dynamic offloading methods adaptively allocate partitions of model or tensors between GPU and CPU memory based on real-time optimization of memory utilization and data transmission. STRONGHOLD [318] proposes to dynamically offload model states between GPU and CPU memory and maintain a suitable working window size to minimize GPU stalls during offloading. Harmony [316] employs a heuristic-based scheduler to map computation and model states to physical devices. Harmony reduces the overhead for offloading with reduced swaps and fast peer-to-peer swaps. TMOF [317] introduces disjoint swapping and bidirectional overlapping coordination mechanisms to prevent PCIe channel contention in swapping and offloading. For MoE models, MPipeMoE [327] designs an adaptive and memory-efficient pipeline parallelism algorithm. Specifically, MPipeMoE employs efficient memory reusing strategies by eliminating memory redundancies and an adaptive selection component to decide whether to offload or recompute the required tensors to reduce memory requirements.

To facilitate better memory management, some studies have proposed systems that break tensors into finer-grained units. TSPLIT [313] and PatrickStar [314] are two dynamic memory management systems that optimize peak GPU memory usage. TSPLIT splits tensors into micro-tensors and performs operations at the micro-tensor level, enabling precise and dynamic memory operations. PatrickStar organizes model data into memory chunks that are dynamically distributed between CPU and GPU memory and optimizes CPU-GPU data transmission as well as bandwidth utilization. Additionally, TSPLIT uses a model-guided planning algorithm to find optimal memory configurations for each tensor, while PatrickStar employs runtime memory tracing, chunk eviction strategies, and device-aware operator placement to further minimize data movement between CPU and GPU.

6.4.3 SSD Offloading

To facilitate the training of trillion-scale LLMs, where methods solely relying on CPU offloading are insufficient, several works have been proposed for offloading data to both CPU memory and NVMe SSDs during training. ZeRO-Infinity [319] offloads all the partitioned model states to CPU or NVMe memory and offloads activation only to CPU memory. This method supports training models with up to 32T parameters on 32 nodes (a total of 512 V100s). However, the CPU offloading for activations still requires extensive CPU memory. For instance, approximately 0.76 TB of CPU memory is needed to store activation checkpoints for training a 10T model, and around 4 TB for 100T models. Fuyou [322] focuses on training LLMs on commodity servers with limited CPU memory capacity and a single GPU. Compared to ZeRO-Infinity, Fuyou further offloads the activations to SSDs and incorporates SSD-CPU communication as an additional optimization dimension. It also proposes a synchronous out-of-core CPU optimizer that overlaps with the backward propagation stage and introduces an automatic activation swapping mechanism, thereby maximizing GPU utilization. Smart-Infinity [321] proposes to reduce the secondary storage bandwidth requirements by using near-storage processing devices for parameter update. MoESys [323], [324] combines various storage devices (GPU, CPU memory, and SSDs) to save the sparse parameter states and dense parameter states and propose a 2D prefetch scheduling strategy to MoE training so that the computation of parameters can be overlapped with the scheduling.

7 COMMUNICATION OPTIMIZATIONS

Different parallelism mechanisms introduce varying patterns of network communication traffic. For instance, tensor parallelism requires AllReduce operations across the tensor parallelism ranks. Data parallelism, on the other hand, necessitates AllReduce operations for gradient synchronization across data parallelism ranks at the end of each iteration. Pipeline parallelism involves passing activation values to the next stage at the end of each stage. Typically, training frameworks place tensor or sequence parallel communication groups, which demand high bandwidth, within high-bandwidth domains (e.g., the same node), while placing data parallel or pipeline parallel communication groups, which have lower bandwidth requirements, between high-bandwidth domains. Fig. 13 shows the communication heatmap of LLM training in practice and well reflects the data traffic brought by different parallel strategies. From this heatmap, it can be observed that the LLM training communication traffic exhibits a clear pattern and hierarchy, with most communication occurring within smaller scopes, and only a little fraction of the traffic crossing the entire cluster. This insight has inspired approaches like rail-optimized topology [62], which reduces unnecessary core switches to cut costs.

This section introduces systems and techniques for optimizing the collective communication performance of distributed LLM training. As shown in Fig. 14, we first discuss collective communication libraries, which utilize both predefined and synthesized algorithms. Next, we explore com-
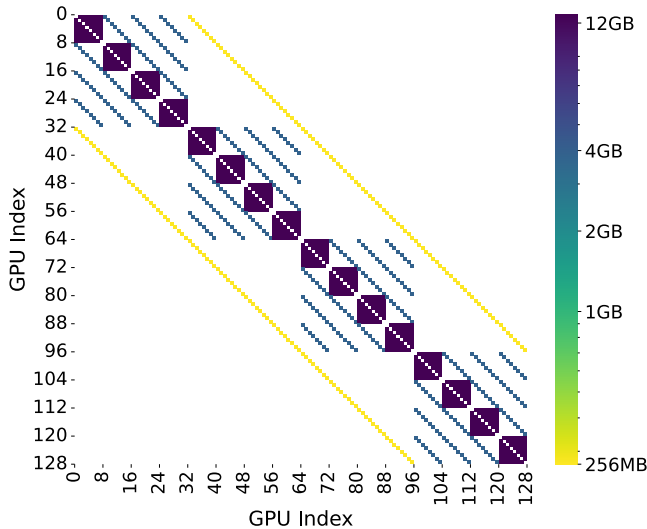
Fig. 13: Communication traffic heatmap for InternLM-2 102B pre-training using 128 GPUs during a single iteration, with tensor parallelism (TP) size 8, pipeline parallelism (PP) size 4, data parallelism (DP) size 4 and ZeRO stage 1 (ZeRO-1) size 4. The prioritization of topology arrangement is TP >DP/ZeRO-1 >PP. There are four different data traffic loads: ① the AllReduce of TP; ②③ ReduceScatter/AllGather of DP/ZeRO-1; ④ Send/Recv of PP. The communication for TP utilizes the fully-connected topology of NVSwitch, resulting in sixteen dense square traffic patterns along the diagonals in the diagram, with each pattern representing a node. The cross-node communication traffic for DP and ZeRO-1 are shown in the diagram as six symmetric diagonal lines within the four 32x32 rectangular topologies. It is important to note that DP/ZeRO-1 also involves intra-node communication traffic, which accumulates into the same heatmap grid as TP. Due to its relatively small communication volume, PP forms two yellow lines on the heatmap at coordinates ((32, 0), (128, 96)) and ((0, 32), (96, 128)). (In this diagram, all communications use the ring-based collective algorithm)

munication scheduling techniques designed to reorganize communication operations to overlap with computation, thereby reducing delays and accelerating the training process. Finally, we delve into in-network aggregation (INA), which leverages the computational capabilities of network devices to perform aggregation operations, such as summing gradients of deep learning models.

Compressing model parameters and gradients effectively reduces communication overhead during distributed LLM training. Various studies explore sparse communication and quantization approaches. For example, ZeRO++ [302] adopt quantization on weights to shrink down each model parameter from FP16 to INT8 data type before communicating. However, these works typically involve lossy sparsification or quantization techniques. We do not survey lossy data compression techniques in this section, as they are beyond the scope of this work.

7.1 Collective Communication

The Message Passing Interface (MPI) is a widely adopted programming model for large-scale scientific applications on parallel computing architectures. MPI has several implementations, including OpenMPI [328], MPICH2 [329], and MVAPICH [330]. These libraries provide a variety of CUDA-aware primitives such as AllReduce, AllGather, and ReduceScatter, which are essential for distributed LLM training. In practice, current training frameworks prefer collective communications tailored to specific AI accelerators with pre-defined or synthesized algorithms.

7.1.1 Pre-Defined Collective Communication Algorithm

NVIDIA's NCCL [331] and AMD's RCCL [332] are highly optimized libraries that typically outperform MPI-based collective communication libraries on their respective AI accelerators. These libraries usually select pre-defined algorithms to perform collectives based on conditions such as network topology and input tensor size.

Ring Algorithm. The Ring algorithm is used for collective communications like AllReduce to move data across all GPUs. With this algorithm, the input tensor is split into multiple chunks and transferred one by one during the operation. This pipeline reduces the idle time that each device spends waiting for data. Baidu used the bandwidth-optimal ring AllReduce algorithm [333] for distributed deep learning model training. Horovod [143] replaced the Baidu ring-AllReduce implementation with NCCL and designed a user-friendly interface for distributed training.

Tree Algorithm. The latency of the Ring algorithm increases with the number of GPU devices [346]. The Double Binary Tree algorithm [334] was proposed to solve this problem. Double binary trees rely on the fact that half or fewer ranks in a binary tree are nodes and half or more ranks are leaves. Therefore, a second tree can be built using leaves as nodes and vice-versa for each binary tree. This algorithm is implemented in MPI-based libraries, NCCL and RCCL.

Hybrid Algorithm. Several approaches propose using hybrid algorithms to handle collective communication tasks on training clusters with heterogeneous intra-node and inter-node communication bandwidth. Two-level AllReduce [335] divides a single AllReduce operation into three steps: intra-node Reduce utilizing PCIe/NVLINK, inter-node AllReduce utilizing network, and intra-node Broadcast. 2D-Torus AllReduce [336] and ACCL [337] decompose a single AllReduce operation into three phases: intra-node ring-based ReduceScatter, inter-node tree-based AllReduce, and intra-node ring-based AllGather. BlueConnect [338] breaks down a single AllReduce operation into numerous parallelizable ReduceScatter and AllGather operations. Each operation can be mapped to different network fabrics, leveraging the best-performing pre-defined implementation for each specific fabric. Plink [339] could probes network topology and efficiently generates two-level hybrid communication plans, exploiting locality in datacenter networks.

7.1.2 Synthesized Collective Communication Algorithm

Several approaches have emerged that synthesize collective communication algorithms and kernels specifically tailored
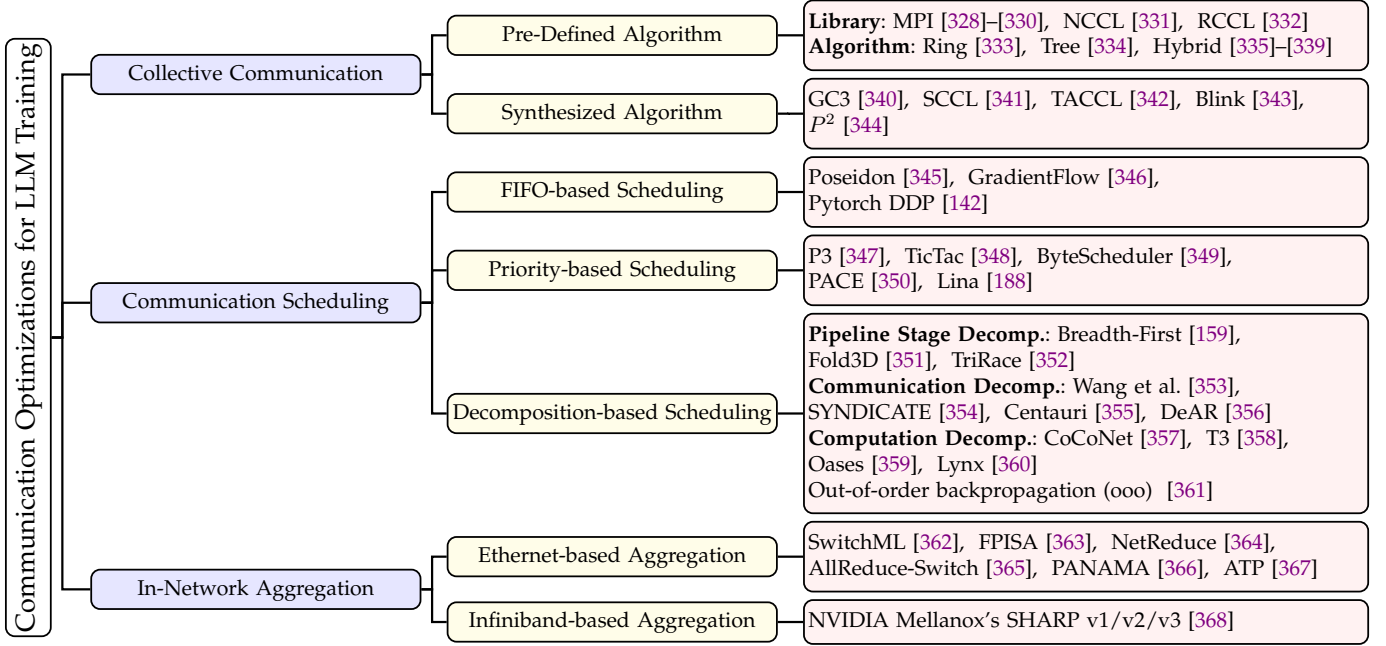
Fig. 14: Studies on communication optimizations for distributed LLM training.

to the hardware topology, aiming to outperform generic pre-defined algorithms in many cases. GC3 [340] introduces a data-oriented domain-specific language (DSL) for designing custom collective communication algorithms. It includes an optimizing compiler that translates these algorithms into executable forms optimized for specific hardware configurations. SCCL [341] encodes the collective communication synthesis problem as an SMT (Satisfiability Modulo Theories) formula. This approach aims to derive exact schedules for Pareto-optimal algorithms, optimizing both latency and bandwidth utilization. TACCL [342] formulates the problem of finding optimal communication algorithms as a mixed integer linear program (MILP). It leverages a communication sketch abstraction to efficiently gather essential information and reduce the search space, with the goal of minimizing overall execution time. Blink [343] dynamically constructs a topology with suitable link capacities by probing available link sets for each job at runtime. Using this topology, it optimizes communication rates through the creation of packet generation trees, and generating CUDA code. P^2 [344] utilizes parallel matrices to partition the parallel axis at the system level, thereby generating topology-aware parallel placement and reduction strategies. By simulating and predicting communication costs, this method reduces the number of actual evaluations required.

7.2 Communication Scheduling

Communication scheduling in distributed training reorganizes communication operations to overlap with computation, thereby reducing delays and accelerating the training process. The key concept of communication scheduling involves reordering communication operations based on the data dependencies of parallel training. Hybrid parallel LLM training necessitates multidimensional communication scheduling schemes to manage communications generated by data, pipeline, tensor, and sequence parallelism, as well as their combinations.

7.2.1 FIFO-based Scheduling

During the backward phase, rather than waiting for all gradient calculations to complete before initiating communication, communication can begin as soon as each gradient is ready. This wait-free backpropagation approach leverages a dependency-directed acyclic graph to manage tasks efficiently. Poseidon [345] employs a First-In-First-Out (FIFO) queue to schedule AllReduce operators, ensuring that each layer starts its communication once its gradients are generated. Motivated by the efficiency of collective communications on large tensors, GradientFlow [346] and Pytorch DDP [142] fuse multiple sequential AllReduce communication operations into a single operation. This method avoids transmitting a large number of small tensors over the network by waiting for a short period of time and then combining multiple gradients into one AllReduce operation during the backward phase.

7.2.2 Priority-based Scheduling

The FIFO scheme is often sub-optimal because the generated communication sequence of in the backward phase differs from the computation sequence in the forward phase. This mismatch can lead to communication blocking computation, even when overlap is enabled. Consequently, many approaches employ priority queues to schedule communication operators efficiently. P3 [347] schedules AllReduce operations at a finer granularity, overlapping gradient communication of the current layer with the forward computation of the next layer. Unlike FIFO queue-based scheduling, this method divides layers into fixed-sized slices and prioritizes synchronizing slices based on the order in which they are processed in forward propagation. Therefore, the

first layer gets the highest priority, with priority decrementing towards the end. When utilizing the parameter-server architecture for distributed model training, TicTac [348] prioritizes transfers that accelerate the critical path within the underlying computational graph.

ByteScheduler [349] and PACE [350] are proposed to generalize priority-based communication scheduling across training frameworks. Specifically, ByteScheduler [349] introduces a unified abstraction to facilitate communication scheduling without disrupting the original dependencies within framework engines. ByteScheduler achieves good performance by using Bayesian optimization to automatically tunes two critical parameters: partition size and credit size. PACE [350] implements preemptive communications by segmenting primitive AllReduce operations into smaller pieces. The preempted AllReduce operators can be resumed at a later time. This preemption strategy prevents head-of-line blocking of large communication tensors. Additionally, PACE uses a dynamic programming approach to fuse small communication tensors to reduce the overhead caused by handling a large number of small tensors, thereby achieving more efficient bandwidth utilization.

To improve bandwidth efficiency in MoE systems, Lina [188] prioritizes All-to-All operations over AllReduce. Typically, expert-parallel (All-to-All) and data-parallel (AllReduce) processes use separate CUDA streams, causing potential overlap and bandwidth sharing without coordination. Lina breaks tensors into smaller chunks, ensures All-to-All operations get full bandwidth while allowing AllReduce micro-ops to run during idle time. In addition, micro-ops enable overlap All-to-All operations with expert computations.

7.2.3 Decomposition-based Scheduling

Several advancements have focused on decomposing communication and computation operations into fine-grained tasks, reordering these operations with greater flexibility to maximize overlap and optimize execution efficiency.

Pipeline Stage Decomposition. When using conventional pipeline parallelism, each GPU stores a contiguous segment of layers. Breadth-First [159] further splits these contiguous stages into finer-grained stages distributed across different GPUs, forming a loop by connecting the first and last GPUs, so each GPU is assigned multiple stages. This allows the given micro-batch to reach the end of the pipeline earlier, reducing pipeline bubbles. Breadth-First uses a breadth-first scheduling strategy to achieve greater computation-communication overlap. Fold3D [351] employs an all-in-all-out scheduling strategy to overlap the pipeline's gradient synchronization process with computation. This involves further folding model fragments within the pipeline, where each device contains two model fragments, allowing one fragment's gradient synchronization to overlap with another fragment's forward or backward computation.

Asynchronous pipeline parallelism relaxes data dependencies between gradients and parameter updates. Leveraging this characteristic, TriRace [352] postpones parameter updates to maximize computation overlap with gradient communication. Additionally, TriRace decomposes bidirectional P2P communication between pipeline stages into two separate unidirectional operations and prioritizes them based on critical path analysis.

Communication Decomposition. Communication primitives could be decomposed into fine-grained operations with high scheduling flexibility. Wang et al. [353] decomposed communication operations (e.g., AllGather and ReduceScatter), into a series of fine-grained peer-to-peer collections. In addition, computational operations (e.g., Einstein Summation) were divided into fine-grained tasks, each performing a part of the computation. This decomposition creates more opportunities for overlapping communication with computation. SYNDICATE [354] segments communication operations into smaller sub-operations, termed Motifs, and employs a Central Optimizer using Markov Chain Monte Carlo search to achieve optimal overlap execution plans. Centauri [355] adopts a different approach by using Primitive Partition, Group Partition, and Workload Partition to decompose communication operations into fine-grained atomic operations. These operations are then scheduled using Workload-aware Scheduling, Backward Scheduling, and Elastic Scheduling to maximize overlap efficiency. DeAR [356] also decomposes communication primitives, specifically breaking down AllReduce into AllGather and ReduceScatter. This decomposition allows subsequent operations to overlap with the forward propagation process of the model, thus eliminating the need to wait for the completion of both communication steps.

Computation Decomposition. When using tensor parallelism, an AllReduce communication is required to synchronize the matrix multiplication outputs in the forward phase. CoCoNet [357] facilitates the overlap of matrix multiplication and AllReduce by partitioning the output into smaller blocks and immediately initiating the AllReduce kernel after computing each result block within the matrix multiplication kernel. To minimize waiting time for the AllReduce kernel, the data blocks are fed into the matrix multiplication kernel in a carefully scheduled order. T3 [358] applies a hardware-software co-design approach, which transparently overlaps matrix multiplication with communication while minimizing resource contention. At the hardware level, T3 introduces a track-and-trigger mechanism to orchestrate the producer's compute and communication activities. Additionally, it employs compute-enhanced memories to handle the attendant compute operations required by the communication processes.

The backward pass generates two types of gradients: the output gradient, which is used to calculate the gradients of the preceding layer, and the weight gradient, which is used to update the layer's weight parameters. These weight gradients need to be synchronized with other ranks using AllReduce. Conventional frameworks simultaneously perform gradient computation for both weights and outputs. Out-of-order backpropagation (ooo-backprop) [361] decouples the gradient computations for weights and outputs, scheduling the weight gradient computations flexibly out of their original order. This allows more critical computations to be prioritized and scheduled accordingly. Consequently, ooo-backprop optimizes overall performance by scheduling communications based on this out-of-order computation strategy. This scheme is also used by Zero Bubble [156] to

reduce the bubble rate of pipeline parallelism.

With activation checkpointing enabled, training frameworks need to recompute activations during the backward phase. This recomputation also involves AllReduce communication when using tensor parallelism. Oases [359] reduces redundant communication in recomputation by always placing AllReduce communication as the last forward communication operation of a recomputation unit, and further splits the batch into smaller sub-batches, allowing the communication and computation of two batches to overlap. Lynx [360] also exploits the potential of recomputation and communication overlap, using two recomputation scheduling algorithms, OPT and HEU, to search for the optimal or near-optimal recomputation scheduling strategy, achieving the best overlap and training performance.

7.3 In-Network Aggregation

In-network aggregation (INA) uses the computational capabilities of network devices to perform aggregation operations like summing gradients of deep learning models. This technique has been previously proposed to accelerate big data processing. Notably, frameworks like NetAgg [369], SwitchAgg [370], and CamDoop [371] have demonstrated significant performance advantages by executing data aggregation at switch-attached high-performance middleboxes or servers within a direct-connect topology. Many approaches have been proposed to apply in-network aggregation to deep learning model training, aiming to reduce the data exchanged between nodes during AllReduce operations on gradients in the backward phase [372].

7.3.1 Ethernet-based Aggregation

Many Ethernet-based in-network aggregation systems depend on programmable switches, and can be leveraged for distributed LLM training. SwitchML [362] supports offloading collective communication operations to programmable network switches during the backward phase of distributed training. Since complete model updates can exceed the storage capacity of a switch, SwitchML streams the aggregation through the switch, processing the aggregation function on a limited number of vector elements at a time. There are two limitations of SwitchML. First, when dealing with floating-point operations, SwitchML cannot directly perform collective communications (such as AllReduce) for floating-point tensors. Instead, it converts floating-point values into 32-bit integers using a block floating-point-like approach. Second, SwitchML is primarily implemented on DPDK, and while there is an RDMA-capable implementation, it is difficult to integrate with training frameworks.

To better facilitate distributed model training, FPISA [363] implements floating-point computation as a P4 [373] program running directly on a programmable switch. Therefore, training frameworks could offload collective communication operations on FP16 tensors to switches without converting them to 32-bit integers. NetReduce [364] supports in-network aggregation compatible with RoCE, fully utilizing the congestion control and reliability design of RoCE without the need for costly network protocol processing stacks in switches. NetReduce is prototyped with an FPGA board attached to an Ethernet switch. AllReduce-Switch [365] is closely related to NetReduce and compatible with its network protocol. It introduces a novel switch architecture tailored for in-network aggregation tasks and has implemented a prototype using FPGA hardware. PANAMA [366] and ATP [367] have also contributed to the field with their in-network aggregation frameworks designed for shared environments. PANAMA focuses on optimizing network load by managing bandwidth allocation among multiple active training jobs concurrently. It addresses the challenge that traditional congestion controls may not adequately support simultaneous training operations. ATP, on the other hand, enables multiple concurrent tenants to run several jobs simultaneously, emphasizing the support for diverse workloads in shared environments.

Certain works are tailored for specific training workloads, making them unsuitable for distributed LLM training. For example, Libra [374] is designed for sparse model training using a parameter server architecture. It offloads the aggregation of frequently updated parameters to programmable switches, leaving infrequently updated parameters to be handled by servers. This approach effectively reduces server load. On the other hand, iSwitch [375] is designed for parameter aggregation in reinforcement learning training tasks. Although its FPGA-based implementation supports native floating point operations, it operates at a significantly lower bandwidth. Furthermore, iSwitch stores an entire gradient vector during aggregation, which is feasible for reinforcement learning workloads but does not scale well for large-scale models, especially LLMs.

7.3.2 Infiniband-based Aggregation

NVIDIA Mellanox's Scalable Hierarchical Aggregation Protocol (SHARP) [368] is a proprietary in-network aggregation scheme available in certain InfiniBand switches and NVIDIA GPUs. Built on InfiniBand, SHARP leverages link-layer flow control and lossless guarantees and employs dedicated on-chip FPUs for collective offloading. SHARPv1 was introduced on InfiniBand EDR switches, and SHARPv2 was enhanced on InfiniBand HDR switches with features such as support for collective communications (e.g., Barrier, Reduce, AllReduce and Broadcast), integer and floating-point operations (16/32/64 bits), and GPUDirect RDMA. SHARPv2 also uses streaming aggregation for large vector reductions at line rate, integrates with NCCL, and is easily usable by existing training frameworks. Enabled on the latest InfiniBand NDR switches, SHARP is production-ready for distributed LLM training and has been deployed in many training clusters. In addition to Infiniband, NVIDIA's NVSwitch-v3 [46] also integrates SHARP to speed up collective operations in GPU-based clusters.

8 FAULT TOLERANCE

LLM training typically involves extended training periods ranging from weeks to months, utilizing clusters of tens of thousands of GPUs. The vast array of components involved, spanning from the underlying infrastructure to training system optimizations, necessitates robust fault tolerance mechanisms to ensure the reliability of training processes. This is because a single point of failure in any part of the system can result in a suspension of the training process due to the
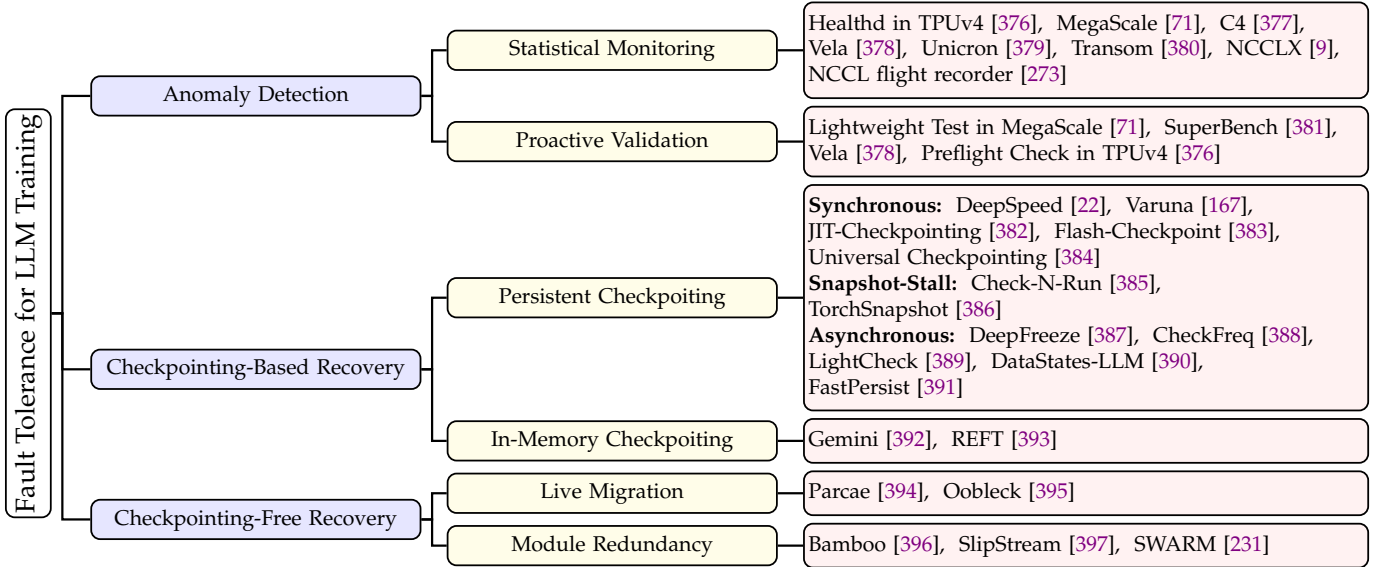
Fault Tolerance for LLM Training			
Anomaly Detection	Statistical Monitoring	Healthd in TPUv4 [376], MegaScale [71], C4 [377], Vela [378], Unicron [379], Transom [380], NCCLX [9], NCCL flight recorder [273]	
	Proactive Validation	Lightweight Test in MegaScale [71], SuperBench [381], Vela [378], Preflight Check in TPUv4 [376]	
Checkpointing-Based Recovery	Persistent Checkpoiting	**Synchronous:** DeepSpeed [22], Varuna [167], JIT-Checkpointing [382], Flash-Checkpoint [383], Universal Checkpointing [384] **Snapshot-Stall:** Check-N-Run [385], TorchSnapshot [386] **Asynchronous:** DeepFreeze [387], CheckFreq [388], LightCheck [389], DataStates-LLM [390], FastPersist [391]	
	In-Memory Checkpoiting	Gemini [392], REFT [393]	
Checkpointing-Free Recovery	Live Migration	Parcae [394], Oobleck [395]	
	Module Redundancy	Bamboo [396], SlipStream [397], SWARM [231]	

Fig. 15: Studies on fault tolerance techniques for distributed LLM training.

synchronous nature of the training. In this section, we first present a failure analysis in LLM training, then investigate the approaches for fast failure detection and recovery.

8.1 LLM Failure Analysis

Empirical evidence from various sources underscores the frequency of failures in LLM training. For example, the training of Bloom experiences 1-2 GPU failures per week on average on a cluster with 384 GPUs [292]. Meta's comprehensive training records [398] of 175B OPT model on 992 A100 GPUs document over 40 interruptions within a two-week period, attributed to hardware, infrastructure, and other external factors. More recent studies further highlight this issue. Acme [23] reported failure occurrences every 1-2 days on average in their training process using over 1,000 A100 GPUs. ByteDance's MegaScale project [71], utilizing 12,288 Ampere GPUs, experiences over 100 times failures in several weeks. Meta's LLaMA3 experiences 466 job interruptions during a 54-day period of pre-training on a cluster of 16,384 H100 GPUs [9]. The frequent failure is primarily attributed to the immense complexity and scale of these systems and extended training periods. The whole training system encompasses a vast array of components as we investigated in previous sections. Moreover, the synchronized training further exacerbates this issue, as errors in any single node can cause the entire job to fail, making the system particularly vulnerable to even isolated hardware or software faults. Even a seemingly low 1.5% daily failure rate for a single node, as observed in Alibaba's cluster [383], translates to a staggering 84.8% daily failure rate when scaled to a system with 1,000 GPUs. However, the trend of scaling up the training system continues to grow, emphasizing the concomitant challenges for fault tolerance mechanisms to maintain system reliability.

The reasons behind these failures are multifaceted and stem from various components of the LLM training system. According to Acme [23], the most severe impact comes from hardware failures, such as issues with GPU (e.g., CUDA-Error, ECC-Error), NVLink, and network system (e.g., NCCL-Timeout-Error, Connection-Error). Similar observations are also delivered in Alibaba C4 [377]. C4 further observes that the majority of errors (about 82.5%) are confined to specific nodes or even individual devices, although most errors observed by users are NCCL error. LLaMA3 pre-training [9] also reports that 78% of the failures are hardware issues. Moreover, the latest generation GPUs (A100 and H100) tend to exhibit high error rates, likely due to rapid development, rushed delivery, and increased power consumption [377], [399]. Beyond hardware, software-related issues in distributed training frameworks, data preprocessing pipelines, or library dependencies can lead to crashes or unexpected behavior [23], [378], [399]. The complex nature of the models themselves can introduce instabilities such as loss spikes, numerical overflow or underflow, gradient explosions, or optimization difficulties [398], [400]. External factors like power outages or cooling system failures in data centers further contribute to system instabilities. For example, the high temperature in the cluster server room also tends to result in GPU overheating, which can cause NVLink-Error or ECC-Error [23] or unstable training speed [9].

These high frequent and multifaceted LLM failures lead to significant waste of GPUs. This inefficiency manifests in two primary ways: failure recovery and performance degradation. First, LLM training jobs periodically save checkpoints during runtime to maintain progress. Upon failure, system maintainers must first locate and diagnose the issue before restarting the training by rolling back to previous checkpoints. Some hardware failures, however, can be challenging to detect proactively and often require considerable time to diagnose and recover from, resulting in prolonged stalls in LLM training. Second, stragglers in the cluster, caused by network link failures [377] or abnormal computational slowdowns [71], can significantly decrease the MFU, further compounding the overall training inefficiency. The training of Meta's 175B OPT model exemplifies these inefficiencies [398]. While the ideal training time was estimated at

about 25 days based on the MFU, the actual training lasted approximately 57 days. This means that a staggering 56% of the total time was wasted handling various failures, underscoring the severe impact of system instabilities on resource utilization and training efficiency in LLM training.

8.2 Anomaly Detection

Rapid detection and diagnosis of LLM failures are crucial for maintaining training stability and efficiency. This process, known as anomaly detection, primarily employs two approaches: statistical monitoring and proactive validation.

8.2.1 Statistical Monitoring

Statistical monitoring is a systematic approach to observing and analyzing various metrics and indicators throughout the LLM training process. This method involves collecting, processing, and interpreting data to identify anomalies or deviations from expected behavior. In a typical setup, each GPU is assigned a dedicated monitoring process responsible for collecting basic information and runtime statistics [71], [378], [379]. These statistics are then transmitted to a central monitor node as heartbeat messages for further analysis. Nodes that fail to send heartbeat messages are considered to have failed. The primary objective of this monitoring system is to detect anomalies promptly, allowing for quick recovery that minimize training interruptions and maintain overall efficiency.

Most runtime statistics monitored in LLM training are hardware-related, encompassing both GPU and network metrics. Recent works [71], [378], [379] collect GPU-related statistics with NVIDIA DCGM [401], including SM block utilization, SM occupancy, SM pipe utilization, PCIe traffic rate, NVLink traffic rate and etc. One frequently occurring issue is GPU memory row-remapping, which seamlessly replaces known degraded memory cells with sparse ones in hardware. Vela [378] detects this by leveraging the DCGM_FI_DEV_ROW_REMAP_PENDING statistics from DCGM. Megascale [71] and Transom [380] also detect errors by analyzing errors occurred in training logs.

In addition to GPU metrics, network statistics are crucial for monitoring distributed training performance. MegaScale [71] tracks RDMA traffic metrics to detect potential anomalies. It also develops visualization system to identify inefficiency GPUs manually. Unicron [379] detects errors like NCCL timeout, TCP timeout, and task hangs with delayed notification during training. C4 [377] gathers connection specifics such as RDMA IP and QP numbers, along with message statistics including counts, sizes, and durations of transfers at the transport layer to detect training slowdowns and hangs. Collective communication activities can also be monitored with PyTorch's built-in NCCL flight recorder [273], which captures collective metadata and stack traces into a ring buffer for later diagnosis. Meta further co-designs NCCLX [9] with PyTorch, allowing PyTorch to access its internal state for fast and accurate failure detection. NCCLX traces the kernel and network activities of each NCCLX communication, which can help diagnose communication issues. Vela [378] implements an enhanced Multi-NIC health checker that collects node network bandwidth data for all 2-node pairs on every port. This information

can be utilized to detect nodes with degraded RoCE/GDR performance. Leveraging the key characteristics of LLMs training as prior knowledge, Transom [380] develops machine learning algorithms to do anomaly detection.

Statistical monitoring also enables the resiliency of Google's TPUv4 supercomputer [376]. Each TPUv4 machine is equipped with a `healthd` daemon that performs real-time monitoring of ICI (Inter-Chip Interconnect) links, PCIe links and TPU ASIC. Detected severe symptoms will notify cluster scheduler for appropriate action, such as evicting affected jobs or rescheduling them.

8.2.2 Proactive Validation

Proactive validation offers an alternative to reactive troubleshooting based on online statistical monitoring, aiming to validate the training system before failures occur. However, a trade-off exists between validation test time and accuracy, as comprehensive validation can significantly impact effective training time. MegaScale [71] introduces a suite of lightweight tests, including intra-network host and NCCL tests, to diagnose a wide spectrum of potential failures. Vela [378] employs a two-tiered strategy with lightweight tests running periodically on every node and more intrusive tests executed only when nodes are idle. Google's TPUv4 supercomputer implements a preflight check [376] before user jobs, consisting of an end-to-end check and an intent-driven checker for hardware health. SuperBench [381] resents a comprehensive benchmark suite for evaluating individual hardware components, incorporating a selector to balance validation time against potential issue-related penalties.

8.3 Checkpoint-Based Recovery

Periodically saving the model states, i.e., *checkpointing*, and resuming computation from the latest checkpoint after failures happen is the common practice for fault tolerant LLM training. However, this presents a dilemma: frequent checkpointing incurs high I/O overhead, while infrequent checkpointing results in substantial progress loss when failures occur. To address this dilemma, fast persistent and in-memory checkpointing approaches have been designed.

8.3.1 Persistent Checkpointing

Persistent checkpointing involves saving model states to non-volatile storage, e.g. SSD and remote cloud storage, ensuring data persistence across system failures. The process typically consists of two phases: first, the *snapshot* phase copies model states from GPU to CPU memory, and second, the *persist* phase writes the snapshots to persistent storage devices. Despite considerable I/O overhead due to the low bandwidth of storage devices, persistent checkpointing remains a widely used approach for fault tolerance due to its ease-of-use and reliability. Advanced persistent checkpointing approaches have been proposed to reduce training stall, thereby enabling more frequent checkpointing without significant performance penalties.

Synchronous Checkpointing. To keep consistency of model parameters, DeepSpeed's default synchronous checkpointing [22] and Varuna [167] periodically stalls the training process to perform checkpointing to persistent storage

synchronously on data parallel rank 0. This approach results in GPU idle time during both the snapshot and persist phases, leading to resource underutilization. Recognizing that most failures are attributable to a single GPU or network device, JIT-Checkpointing [382] proposes an alternative strategy. It takes just-in-time checkpoints immediately after failures occur, allowing training to resume from these JIT checkpoints. This approach significantly reduces the cost of wasted GPU time, limiting it to at most one mini-batch iteration of work. DLRover Flash-Checkpoint [383] accelerates the migration efficiency utilizing a distributed caching service. Universal Checkpointing [384] introduces a universal checkpoint representation to decouple the distributed checkpoints storage from parallelism techniques. Universal Checkpointing can seamlessly transform checkpoints from one parallelization strategy to another upon demands.

Snapshot-Stall Checkpointing. To reduce LLM training stalls while maintaining checkpoint consistency, Check-N-Run [385] decouples the snapshot and persist phases. It achieves atomic checkpointing by stalling training only during the snapshot phase and asynchronously persisting snapshots using dedicated background CPU processes. TorchSnapshot [386] further optimizes this process through tensor chunking and multi-threaded disk writing. By creating chunked snapshots, TorchSnapshot allows the persist phase to begin earlier through parallel writing, thereby reducing overall training stall time. MegaScale [71] and InternEvo [18] also adopt a snapshot-stall approach for fast checkpointing and recovery. The snapshot phase stalls training for several seconds to capture the model states, while the persist phase asynchronously transfers checkpoints from CPU memory to a distributed file system. MegaScale optimizes the recovery process by designating a single worker within the data parallel group to read from the distributed file system, thus mitigating the low bandwidth bottleneck. This worker then broadcasts the checkpoint data to other GPUs, enabling faster and more efficient recovery across the entire system. To save storage space, InternEvo also asynchronously moves checkpoints from expensive hot storage to cheaper cold storage.

Asynchronous Checkpointing. Asynchronous checkpointing aims to minimize training stall by executing the snapshot and persist phases concurrently with training. DeepFreeze [387] applies both lightweight (snapshot) and heavy (persist) persistence strategies in the background, sharding checkpoints across data-parallel GPUs to distribute I/O workload. CheckFreq [388] carefully pipelines the snapshot and persist phases with subsequent iteration's forward and backward passes, ensuring snapshot completion before the next parameter update. It also dynamically tunes checkpointing frequency to balance recovery costs and runtime overhead. LightCheck [389] exploits inter-iteration data dependencies, introducing layer-wise checkpointing pipeline to reduce stall. DataStates-LLM [390] addresses slow host memory allocation by pre-allocating pinned host memory for snapshots and utilizes efficient computation, snapshot, and persist layer-wise pipelining. FastPersist [391] identifies risks in fully asynchronous persist phases and synchronizes them with the next iteration's parameter update. It improves SSD bandwidth utilization through double-buffering

pinned memory and reduces hardware contention by using a subset of data-parallel ranks for checkpoint writing.

8.3.2 In-Memory Checkpointing

The low bandwidth of remote persistent storage severely restricts the frequency of checkpointing, in-memory checkpointing addresses the limitations by storing checkpoints in the memory of other compute nodes or dedicated in-memory storage systems, significantly reducing I/O overhead and enabling higher checkpointing frequencies. Gemini [392] proposes checkpointing to CPU memory for faster failure recovery, along with a checkpoint placement strategy to minimize checkpoint loss and a traffic scheduling algorithm to reduce interference with training. REFT [393] asynchronously caches model states to host memory and in-memory storage like Redis, bypassing checkpoint I/O and enabling high checkpointing frequency. It also leverages erasure coding to implement RAIM5 (inspired by RAID5 with "Disk" replaced by "Memory") that protects data against node failures. While these approaches significantly advance fault tolerance for LLM training by enabling more frequent checkpointing without performance penalties, they may not provide the same long-term data persistence as traditional storage-based methods. Consequently, hybrid approaches combining both in-memory and persistent checkpointing is necessary for comprehensive fault tolerance strategies.

8.4 Checkpoint-Free Recovery

Checkpoint-free recovery methods aim to minimize training stalls by eliminating the need to restart and roll back to previous checkpoints when failures occur. These techniques depend on automatic failure detection mechanisms to identify issues promptly. When a failure is detected, checkpoint-free approaches automatically address the problem and allow the training process to continue without interruption. By avoiding the time-consuming process of loading from a checkpoint and repeating computations, these methods can significantly reduce downtime and improve overall training efficiency. Checkpoint-free recovery strategies can be broadly categorized into two main approaches: live migration and module redundancy.

8.4.1 Live Migration

Live migration leverages the inherent redundancy present in distributed LLM training setups, particularly the model replicas across different data parallel pipelines, to restore model states in case of failure. When a failure is detected, live migration approaches dynamically reconfigure the parallelization strategy using the remaining healthy instances or by incorporating new instances into the training cluster. The current model states are then transferred to these reconfigured nodes, allowing the training process to continue with minimal interruption. Parcae [394] proposes three distinct migration mechanisms, each with different communication overheads, to efficiently transfer model states between varying parallelization strategies. Oobleck [395] takes a pipeline template-based approach to live migration. It maintains a set of predefined pipeline templates and, upon detecting a failure, swiftly instantiates new heterogeneous pipelines based on these templates.

8.4.2 Module Redundancy

Module redundancy, like live migration, also leverages the redundancy of model states. However, instead of restoring the latest model states across different GPUs, this approach continues training by routing computation to redundant modules. Bamboo [396] places a redundant pipeline stage in the GPU holding an adjacent pipeline stage within the same pipeline. This redundant stage performs redundant computations during training, utilizing pipeline bubbles, and is activated as a normal stage upon failure. SlipStream [397] leverages the redundancy across model replica pipelines, routing the computation of failed nodes to nodes in different data parallel pipelines. SWARM [231] proposes a similar solution but focuses more on poorly connected, heterogeneous, and unreliable devices. In addition to redundant computation, SWARM also incorporates instance migration to rebalance the pipeline, combining aspects of both redundant computation and live migration approaches.

9 CONCLUSION AND OUTLOOKS

The rise of LLMs has transformed AI, enabling applications like personal assistants, code generation, and scientific research. Models such as GPT, LLaMA, and Gemini have set new standards, but training these massive models, exemplified by LLaMA-3's 54-day process on 16,384 GPUs, presents challenges in scalability, efficiency, and reliability. Managing vast GPU clusters requires innovative hardware and networking solutions. Efficient training demands optimizing computation, communication, and memory use. Reliability involves robust mechanisms to detect and recover from failures over long training periods. This survey reviews recent advancements in LLM training systems and infrastructure, highlighting approaches to enhance scalability, efficiency, and reliability.

Traditional digital circuit-based computing systems, guided by Moore's Law and Dennard Scaling, are facing significant physical and economic constraints in meeting the computational demands for training and deploying LLMs. Consequently, the AI industry necessitates innovative solutions. One promising approach is large-scale optoelectronic integration technology, which replaces traditional digital circuits with integrated silicon photonics to enhance computational and communication capabilities [402]. This optoelectronic hybrid data center technology combines optical computing with optical networks, increasing single-node computing power and the efficiency of large-scale distributed computing. Several works have proposed leveraging optical networks for LLM training. For instance, TopoOpt [67] optimizes both the optical network topology and parallelization strategies in distributed training, enhancing computational and communication efficiency. TPUv4 [42] uses Optical Circuit Switches to dynamically reconfigure its 3D-Torus interconnect topology, improving data flow for the intensive communication patterns in LLM training. Additionally, Taichi [403] explores a distributed diffractive-interference hybrid photonic computing architecture to effectively scale the optical neural network to the million-neuron level with 160 tera-operations per second per watt (TOPS/W) energy efficiency. The future may necessitate a paradigm shift in LLM training and inference towards silicon photonics. However, this transition will require extensive innovation across system design and implementation.

REFERENCES

[1] X. L. Dong, S. Moon, Y. E. Xu, K. Malik, and Z. Yu, "Towards next-generation intelligent assistants leveraging llm techniques," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5792–5793.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[4] A. Jo, "The promise and peril of generative ai," *Nature*, vol. 614, no. 1, pp. 214–216, 2023.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[6] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[7] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[8] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[9] LlamaTeam. (2024) The llama 3 herd of models. [Online]. Available: https://ai.meta.com/research/publications/the-llama-3-herd-of-models

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[11] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019.

[12] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints," *arXiv.org*, 2023.

[13] DeepSeek-AI, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

[14] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[15] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[16] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations*, ser. ICLR '15, 2015.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[18] Z. Cai, M. Cao, H. Chen, K. Chen, K. Chen, X. Chen, X. Chen, Z. Chen, Z. Chen, P. Chu *et al.*, "Internlm2 technical report," *arXiv preprint arXiv:2403.17297*, 2024.

[19] T. Sun, X. Zhang, Z. He, P. Li, Q. Cheng, X. Liu, H. Yan, Y. Shao, Q. Tang, S. Zhang *et al.*, "Moss: An open conversational large language model," *Machine Intelligence Research*, pp. 1–18, 2024.

[20] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[21] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing *et al.*, "Alpa: Automating inter- and {Intra-Operator} parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 559–578.

[22] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.

[23] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo *et al.*, "Characterization of large language model development in the datacenter," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 709–729.

[24] C. Zhou, Q. Li, C. Li, J. Yu, Y. Liu, G. Wang, K. Zhang, C. Ji, Q. Yan, L. He *et al.*, "A comprehensive survey on pretrained foundation models: A history from bert to chatgpt," *arXiv preprint arXiv:2302.09419*, 2023.

[25] Y. Huang, J. Xu, Z. Jiang, J. Lai, Z. Li, Y. Yao, T. Chen, L. Yang, Z. Xin, and X. Ma, "Advancing transformer architecture in long-context large language models: A comprehensive survey," *arXiv preprint arXiv:2311.12351*, 2023.

[26] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[27] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, and G. Wang, "Instruction Tuning for Large Language Models: A Survey," *arXiv preprint arXiv:2308.10792*, 2023.

[28] Y. Wang, W. Zhong, L. Li, F. Mi, X. Zeng, W. Huang, L. Shang, X. Jiang, and Q. Liu, "Aligning large language models with human: A survey," *arXiv preprint arXiv:2307.12966*, 2023.

[29] Z. Wan, X. Wang, C. Liu, S. Alam, Y. Zheng, J. Liu, Z. Qu, S. Yan, Y. Zhu, Q. Zhang, M. Chowdhury, and M. Zhang, "Efficient Large Language Models: A Survey," *arXiv.2312.03863*, 2023.

[30] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong *et al.*, "Understanding llms: A comprehensive overview from training to inference," *arXiv preprint arXiv:2401.02038*, 2024.

[31] M. Xu, W. Yin, D. Cai, R. Yi, D. Xu, Q. Wang, B. Wu, Y. Zhao, C. Yang, S. Wang, Q. Zhang, Z. Lu, L. Zhang, S. Wang, Y. Li, Y. Liu, X. Jin, and X. Liu, "A Survey of Resource-efficient LLM and Multimodal Foundation Models," *arXiv preprint arXiv:2401.08092*, 2024.

[32] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," *arXiv preprint arXiv:2308.07633*, 2023.

[33] Z. Han, C. Gao, J. Liu, S. Q. Zhang *et al.*, "Parameter-efficient fine-tuning for large models: A comprehensive survey," *arXiv preprint arXiv:2403.14608*, 2024.

[34] X. He, F. Xue, X. Ren, and Y. You, "Large-scale deep learning optimizations: A comprehensive survey," *arXiv preprint arXiv:2111.00856*, 2021.

[35] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–37, 2020.

[36] P. Liang, Y. Tang, X. Zhang, Y. Bai, T. Su, Z. Lai, L. Qiao, and D. Li, "A survey on auto-parallelism of large-scale deep learning training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2377–2390, 2023.

[37] NVIDIA. NVIDIA Ampere Architecture. [Online]. Available: https://www.nvidia.com/en-us/data-center/ampere-architecture

[38] ——. NVIDIA Hopper Architecture. [Online]. Available: https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture

[39] ——. NVIDIA Blackwell Architecture. [Online]. Available: https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture

[40] R. Swaminathan, M. J. Schulte, B. Wilkerson, G. H. Loh, A. Smith, and N. James, "Amd instinct tm mi250x accelerator enabled by elevated fanout bridge advanced packaging architecture," in *2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. IEEE, 2023, pp. 1–2.

[41] Habana. (2023) Gaudi training platform white paper. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/784830/gaudi-training-platform-white-paper.html

[42] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles *et al.*, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[43] M. Emani, S. Foreman, V. Sastry, Z. Xie, S. Raskar, W. Arnold, R. Thakur, V. Vishwanath, and M. E. Papka, "A comprehensive performance study of large language models on novel ai accelerators," *arXiv preprint arXiv:2310.04607*, 2023.

[44] J.-P. Fricker, "The cerebras cs-2: Designing an ai accelerator around the world's largest 2.6 trillion transistor chip," in *Proceedings of the 2022 International Symposium on Physical Design*, 2022, pp. 71–71.

[45] NVIDIA, "Nvidia dgx-1 system architecture white paper," NVIDIA, White Paper, 2017.

[46] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, "3.2 the a100 datacenter gpu and ampere architecture," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 48–50.

[47] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "2.2 amd chiplet architecture for high-performance server and desktop products," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2020, pp. 44–45.

[48] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[49] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. P. Jouppi, and D. A. Patterson, "Google's training chips revealed: Tpuv2 and tpuv3." in *Hot Chips Symposium*, 2020, pp. 1–70.

[50] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia gpudirect over infiniband—a new model for gpu to gpu communications," *Computer Science-Research and Development*, vol. 26, pp. 267–273, 2011.

[51] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 10, 2001.

[52] Infiniband Trade Association, "Supplement to infiniband architecture specification volume 1 release 1.2.2 annex a16," pp. 1–17, 2010.

[53] ——, "Supplement to infiniband architecture specification volume 1 release 1.2.2 annex a17," pp. 1–17, 2010.

[54] RDMA Consortium. Architectural Specifications for RDMA over TCP/IP . [Online]. Available: http://www.rdmaconsortium.org

[55] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[56] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009, pp. 63–74.

[57] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008, pp. 75–86.

[58] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 225–238.

[59] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection networks*. Morgan Kaufmann, 2003.

[60] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 77–88, 2008.

[61] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, and E. Zahavi, "Dragonfly+: Low cost topology for scaling datacenters," in *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*. IEEE, 2017, pp. 1–8.

[62] K. Mandakolathur and S. Jeaugey. (2018) Doubling all2all performance with nvidia collective communication library 2.12. [Online]. Available: https://developer.nvidia.com/blog/nvswitch-leveraging-nvlink-to-maximum-effect/

[63] K. Qian, Y. Xi, J. Cao, J. Gao, Y. Xu, Y. Guan, B. Fu, X. Shi, F. Zhu, R. Miao *et al.*, "Alibaba hpn: A data center network for large language model training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024.

[64] W. Wang, M. Ghobadi, K. Shakeri, Y. Zhang, and N. Hasani, "Optimized network architectures for large language model training with billions of parameters," *arXiv preprint arXiv:2307.12169*, 2023.

[65] T. Hoefler, T. Bonato, D. De Sensi, S. Di Girolamo, S. Li, M. Heddes, J. Belk, D. Goel, M. Castro, and S. Scott, "Hammingmesh: a network topology for large-scale deep learning," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–18.

[66] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi, "Sip-ml: high-bandwidth optical network interconnects for machine learning training," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 657–675.

[67] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch, "{TopoOpt}: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 739–767.

[68] C. Hopps. (2000) Analysis of an equal-cost multi-path algorithm. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2992.html

[69] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *2013 proceedings ieee infocom*. IEEE, 2013, pp. 2130–2138.

[70] V. Addanki, P. Goyal, and I. Marinos, "Challenging the need for packet spraying in large-scale distributed training," *arXiv preprint arXiv:2407.00550*, 2024.

[71] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong *et al.*, "{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 745–760.

[72] IEEE. 802.1qbb – priority-based flow control. [Online]. Available: https://1.ieee802.org/dcb/802-1qbb

[73] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 537–550, 2015.

[74] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan *et al.*, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 514–528.

[75] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[76] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye, "Ecn or delay: Lessons learnt from analysis of dcqcn and timely," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 313–327.

[77] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: High precision congestion control," in *Proceedings of the ACM special interest group on data communication*, 2019, pp. 44–58.

[78] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley, and C. Raiciu, "An edge-queued datagram service for all datacenter traffic," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 761–777.

[79] P. Taheri, D. Menikkumbura, E. Vanini, S. Fahmy, P. Eugster, and T. Edsall, "Rocc: robust congestion control for rdma," in *Proceedings of the 16th International conference on emerging networking experiments and technologies*, 2020, pp. 17–30.

[80] S. Rajasekaran, S. Narang, A. A. Zabreyko, and M. Ghobadi, "Mltcp: Congestion control for dnn training," *arXiv preprint arXiv:2402.09589*, 2024.

[81] S. Rajasekaran, M. Ghobadi, and A. Akella, "{CASSINI}:{Network-Aware} job scheduling in machine learning clusters," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1403–1420.

[82] H. Wang, H. Tian, J. Chen, X. Wan, J. Xia, G. Zeng, W. Bai, J. Jiang, Y. Wang, and K. Chen, "Towards {Domain-Specific} network transport for distributed {DNN} training," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1421–1443.

[83] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, M. Shuey, R. Wareing, M. Gangapuram, G. Cao *et al.*, "Facebook's tectonic filesystem: Efficiency from exascale," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 217–231.

[84] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[85] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, 2006, pp. 307–320.

[86] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[87] F. Schmuck and R. Haskin, "{GPFS}: A {Shared-Disk} file system for large computing clusters," in *Conference on file and storage technologies (FAST 02)*, 2002.

[88] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/o characterization and performance evaluation of beegfs for deep learning," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[89] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–15.

[90] JuiceFS. Juicefs: A High-Performance, Cloud-Native, Distributed File System . [Online]. Available: https://github.com/juicedata/juicefs

[91] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 283–296.

[92] R. Gu, K. Zhang, Z. Xu, Y. Che, B. Fan, H. Hou, H. Dai, L. Yi, Y. Ding, G. Chen *et al.*, "Fluid: Dataset abstraction and elastic acceleration for cloud-native deep learning training jobs," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 2182–2195.

[93] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A {GPU} cluster manager for distributed deep learning," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 485–500.

[94] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient {GPU} cluster scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 289–304.

[95] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu, "Elasticflow: An elastic serverless training platform for distributed deep learning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 266–280.

[96] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '20. USENIX Association, 2020, pp. 481–498.

[97] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. Association for Computing Machinery, 2020.

[98] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang, "Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent," in *2023 USENIX Annual Technical Conference*, ser. USENIX ATC '23. USENIX Association, 2023, pp. 995–1008.

[99] Q. Hu, M. Zhang, P. Sun, Y. Wen, and T. Zhang, "Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs," in *Proceedings of the 28th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2023, pp. 457–472.

[100] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '21.   USENIX Association, 2021, pp. 1–18.

[101] S. Jayaram Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, and G. R. Ganger, "Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 642–657.

[102] C. Xue, W. Cui, H. Zhao, Q. Chen, S. Zhang, P. Yang, J. Yang, S. Li, and M. Guo, "A codesign of scheduling and parallelization for large model training in heterogeneous clusters," *arXiv preprint arXiv:2403.16125*, 2024.

[103] Q. Hu, Z. Ye, M. Zhang, Q. Chen, P. Sun, Y. Wen, and T. Zhang, "Hydro: Surrogate-Based hyperparameter tuning service in datacenters," in *17th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '23.   USENIX Association, 2023.

[104] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt, "Switches for hire: Resource scheduling for data center in-network computing," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 268–285.

[105] H. Zhao, Z. Han, Z. Yang, Q. Zhang, M. Li, F. Yang, Q. Zhang, B. Li, Y. Yang, L. Qiu *et al.*, "Silod: A co-design of caching and scheduling for deep learning clusters," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 883–898.

[106] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 579–596.

[107] S. Choi, I. Koo, J. Ahn, M. Jeon, and Y. Kwon, "{EnvPipe}: Performance-preserving {DNN} training framework for saving energy," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 851–864.

[108] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training," in *Proc. USENIX NSDI*, 2023.

[109] J.-W. Chung, Y. Gu, I. Jang, L. Meng, N. Bansal, and M. Chowdhury, "Perseus: Removing energy bloat from large model training," *arXiv preprint arXiv:2312.06902*, 2023.

[110] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia, "Towards efficient generative large language model serving: A survey from algorithms to systems," *arXiv preprint arXiv:2312.15234*, 2023.

[111] J. Choquette, "Nvidia hopper gpu: Scaling performance," in *2022 IEEE Hot Chips 34 Symposium (HCS)*.   IEEE Computer Society, 2022, pp. 1–46.

[112] D. Schneider, "The exascale era is upon us: The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second," *IEEE spectrum*, vol. 59, no. 1, pp. 34–35, 2022.

[113] S. Dash, I. R. Lyngaas, J. Yin, X. Wang, R. Egele, J. A. Ellis, M. Maiterth, G. Cong, F. Wang, and P. Balaprakash, "Optimizing distributed training on frontier for large language models," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*.   Prometeus GmbH, 2024, pp. 1–11.

[114] J. Yin, S. Dash, J. Gounley, F. Wang, and G. Tourassi, "Evaluation of pre-training large language models on leadership-class supercomputers," *The Journal of Supercomputing*, vol. 79, no. 18, pp. 20 747–20 768, 2023.

[115] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[116] T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," *arxiv:2307.08691*, 2023.

[117] C. Zhang, B. Sun, X. Yu, Z. Xie, W. Zheng, K. A. Iskra, P. Beckman, and D. Tao, "Benchmarking and in-depth performance study of large language models on habana gaudi processors," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1759–1766.

[118] N. Dey, G. Gosal, H. Khachane, W. Marshall, R. Pathria, M. Tom, J. Hestness *et al.*, "Cerebras-gpt: Open compute-optimal language

[119] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?" in *Proceedings of the Workshop on Network Meets AI & ML*, 2020, pp. 8–13.

[120] L. Dai, H. Qi, W. Chen, and X. Lu, "High-speed data communication with advanced networks in large language model training," *IEEE Micro*, 2024.

[121] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin, "Multi-resource interleaving for deep learning training," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 428–440.

[122] K. Kong, "Using pci express® as the primary system interconnect in multiroot compute, storage, communications and embedded systems," *White Paper, Integrated Device Technology*, p. 12, 2008.

[123] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[124] NVIDIA. (2018) Nvidia dgx-2: The world's most powerful system for the most complex ai challenges. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-2/

[125] A. C. Elster and T. A. Haugdahl, "Nvidia hopper gpu and grace cpu highlights," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, 2022.

[126] "Infiniband product guide," 2020, accessed: 2024-07-01. [Online]. Available: https://network.nvidia.com/files/doc-2020/br-infiniband-product-guide.pdf

[127] "Roce vs. iwarp competitive analysis," 2017. [Online]. Available: https://network.nvidia.com/sites/default/files/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf

[128] Nvidia. (2024) Dgx superpod architecture. [Online]. Available: https://docs.nvidia.com/dgx-superpod/reference-architecture-scalable-infrastructure-h100/latest/dgx-superpod-architecture.html

[129] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng *et al.*, "Eflops: Algorithm and system co-design for a high performance distributed training platform," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.   IEEE, 2020, pp. 610–622.

[130] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.

[131] Kevin Lee, Adi Gangidi and Mathew Oldham. Building Meta's GenAI Infrastructure. [Online]. Available: https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure

[132] J. Qiu, H. Lv, Z. Jin, R. Wang, W. Ning, J. Yu, C. Zhang, P. Chu, Y. Qu, R. Peng *et al.*, "Wanjuan-cc: A safe and high-quality opensourced english webtext dataset," *arXiv preprint arXiv:2402.19282*, 2024.

[133] Z. Ye, W. Gao, Q. Hu, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen, "Deep learning workload scheduling in gpu datacenters: A survey," *ACM Computing Surveys*, vol. 56, p. 1–38, 2024.

[134] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[135] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[136] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[137] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "{AntMan}: Dynamic scaling on {GPU} clusters for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 533–548.

[138] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[139] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters

for DNN training workloads," in *2019 USENIX Annual Technical Conference*, ser. USENIX ATC '19. USENIX Association, 2019, pp. 947–960.

[140] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, 2021.

[141] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 945–960.

[142] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[143] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[144] Y. Xu, H. Lee, D. Chen, H. Choi, B. Hechtman, and S. Wang, "Automatic cross-replica sharding of weight update in data-parallel training," *arXiv preprint arXiv:2004.13336*, 2020.

[145] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[146] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3848–3860, 2023.

[147] Z. Zhang, S. Zheng, Y. Wang, J. Chiu, G. Karypis, T. Chilimbi, M. Li, and X. Jin, "Mics: Near-linear scaling for training gigantic model on public," *Proceedings of the VLDB Endowment*, vol. 16, no. 1, pp. 37–50, 2022.

[148] Q. Xu and Y. You, "An efficient 2d method for training super-large deep learning models," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 222–232.

[149] B. Wang, Q. Xu, Z. Bian, and Y. You, "Tesseract: Parallelize the tensor parallelism efficiently," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.

[150] Z. Bian, Q. Xu, B. Wang, and Y. You, "Maximizing parallelism in distributed training for huge neural networks," *arXiv preprint arXiv:2105.14450*, 2021.

[151] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[152] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[153] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[154] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6543–6552.

[155] Z. Lin, Y. Miao, G. Xu, C. Li, O. Saarikivi, S. Maleki, and F. Yang, "Tessel: Boosting distributed execution of large dnn models via flexible schedule search," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 803–816.

[156] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble pipeline parallelism," in *The Twelfth International Conference on Learning Representations*, 2023.

[157] Z. Liu, S. Cheng, H. Zhou, and Y. You, "Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[158] S. Ao, W. Zhao, X. Han, C. Yang, Z. Liu, C. Shi, and M. Sun, "Seq1f1b: Efficient sequence-level pipeline parallelism for large language model training," *arXiv preprint arXiv:2406.03488*, 2024.

[159] J. Lamy-Poirier, "Breadth-first pipeline parallelism," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[160] C. Jiang, Z. Jia, S. Zheng, Y. Wang, and C. Wu, "Dynapipe: Optimizing multi-task training through dynamic pipelines," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 542–559.

[161] J. Huang, Z. Zhang, S. Zheng, F. Qin, and Y. Wang, "Distmm: Accelerating distributed multimodal model training," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1157–1171.

[162] B. Jeon, M. Wu, S. Cao, S. Kim, S. Park, N. Aggarwal, C. Unger, D. Arfeen, P. Liao, X. Miao *et al.*, "Graphpipe: Improving performance and scalability of dnn training with graph pipeline parallelism," *arXiv preprint arXiv:2406.17145*, 2024.

[163] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun, "Bpipe: Memory-balanced pipeline parallelism for training large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 16 639–16 653.

[164] Q. Zhou, H. Wang, X. Yu, C. Li, Y. Bai, F. Yan, and Y. Xu, "Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 556–569.

[165] H. Dreuning, H. E. Bal, and R. V. v. Nieuwpoort, "mcap: Memory-centric partitioning for large-scale pipeline-parallel dnn training," in *European Conference on Parallel Processing*. Springer, 2022, pp. 155–170.

[166] P. Qi, X. Wan, N. Amar, and M. Lin, "Pipeline parallelism with controllable memory," *arXiv preprint arXiv:2405.15362*, 2024.

[167] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: scalable, low-cost training of massive deep learning models," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 472–487.

[168] Z. Sun, H. Cao, Y. Wang, G. Feng, S. Chen, H. Wang, and W. Chen, "Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 86–100.

[169] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 2391–2404.

[170] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[171] J. Fang and S. Zhao, "Usp: A unified sequence parallelism approach for long context generative ai," *arXiv preprint arXiv:2405.07719*, 2024.

[172] S. A. Jacobs, M. Tanaka, C. Zhang, M. Zhang, R. Y. Aminadabi, S. L. Song, S. Rajbhandari, and Y. He, "System optimizations for enabling training of extreme long sequence transformer models," in *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, 2024, pp. 121–130.

[173] NVIDIA. (2023) Megatron context parallelism. [Online]. Available: https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html

[174] D. Gu, P. Sun, Q. Hu, T. Huang, X. Chen, Y. Xiong, G. Wang, Q. Chen, S. Zhao, J. Fang *et al.*, "Loongtrain: Efficient training of long-sequence llms with head-context parallelism," *arXiv preprint arXiv:2406.18485*, 2024.

[175] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," *International Conference on Learning Representations(ICLR)*, 2024.

[176] D. Li, R. Shao, A. Xie, E. P. Xing, J. E. Gonzalez, I. Stoica, X. Ma, and H. Zhang, "DISTFLASHATTN: Distributed Memory-efficient Attention for Long-context LLMs Training," *arxiv:2310.03294*, 2023.

[177] X. Zhao, S. Cheng, Z. Zheng, Z. Yang, Z. Liu, and Y. You, "Dsp: Dynamic sequence parallelism for multi-dimensional transformers," *arXiv preprint arXiv:2403.10266*, 2024.

[178] W. Brandon, A. Nrusimha, K. Qian, Z. Ankner, T. Jin, Z. Song, and J. Ragan-Kelley, "Striped attention: Faster ring attention for causal transformers," *arXiv preprint arXiv:2311.09431*, 2023.

[179] S. Ao, W. Zhao, X. Han, C. Yang, Z. Liu, C. Shi, M. Sun, S. Wang, and T. Su, "Burstattention: An efficient distributed attention framework for extremely long sequences," *arXiv preprint arXiv:2403.09347*, 2024.

[180] Z. Liu, S. Wang, S. Cheng, Z. Zhao, Y. Bai, X. Zhao, J. Demmel, and Y. You, "Wallfacer: Guiding transformer model training out of the long-context dark forest with n-body problem," *arXiv preprint arXiv:2407.00611*, 2024.

[181] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[182] C. Hwang, W. Cui, Y. Xiong, Z. Yang, Z. Liu, H. Hu, Z. Wang, R. Salas, J. Jose, P. Ram *et al.*, "Tutel: Adaptive mixture-of-experts at scale," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[183] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale," in *International conference on machine learning*. PMLR, 2022, pp. 18 332–18 346.

[184] S. Tan, Y. Shen, R. Panda, and A. Courville, "Scattered mixture-of-experts implementation," *arXiv preprint arXiv:2403.08245*, 2024.

[185] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "Megablocks: Efficient sparse training with mixture-of-experts," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[186] S. Shi, X. Pan, X. Chu, and B. Li, "Pipemoe: Accelerating mixture-of-experts through adaptive pipelining," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.

[187] S. Shi, X. Pan, Q. Wang, C. Liu, X. Ren, Z. Hu, Y. Yang, B. Li, and X. Chu, "Schemoe: An extensible mixture-of-experts distributed training system with tasks scheduling," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 236–249.

[188] J. Li, Y. Jiang, Y. Zhu, C. Wang, and H. Xu, "Accelerating distributed {MoE} training and inference with lina," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 945–959.

[189] J. Liu, J. H. Wang, and Y. Jiang, "Janus: A unified distributed training framework for sparse mixture-of-experts models," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 486–498.

[190] C. Chen, M. Li, Z. Wu, D. Yu, and C. Yang, "Ta-moe: Topology-aware large scale mixture-of-expert training," *Advances in Neural Information Processing Systems*, vol. 35, pp. 22 173–22 186, 2022.

[191] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang, "Fastmoe: A fast mixture-of-expert training system," *arXiv preprint arXiv:2103.13262*, 2021.

[192] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li, "Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 120–134.

[193] M. Zhai, J. He, Z. Ma, Z. Zong, R. Zhang, and J. Zhai, "Smartmoe: Efficiently training {Sparsely-Activated} models through combining offline and online parallelization," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 961–975.

[194] X. Nie, X. Miao, Z. Wang, Z. Yang, J. Xue, L. Ma, G. Cao, and B. Cui, "Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–19, 2023.

[195] W. Wang, Z. Lai, S. Li, W. Liu, K. Ge, Y. Liu, A. Shen, and D. Li, "Prophet: Fine-grained load balancing for parallel training of large-scale moe models," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2023, pp. 82–94.

[196] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," *Advances in neural information processing systems*, vol. 31, 2018.

[197] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni *et al.*, "Gspmd: general and scalable parallelization for ml computation graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[198] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks." in *ICML*, 2018, pp. 2279–2288.

[199] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[200] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[201] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 56–68.

[202] M. Schaarschmidt, D. Grewe, D. Vytiniotis, A. Paszke, G. S. Schmid, T. Norman, J. Molloy, J. Godwin, N. A. Rink, V. Nair *et al.*, "Automap: Towards ergonomic automated parallelism for ml models," *arXiv preprint arXiv:2112.02958*, 2021.

[203] H. Chen, C. H. Yu, S. Zheng, Z. Zhang, Z. Zhang, and Y. Wang, "Slapo: A schedule language for progressive optimization of large deep learning model training," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1095–1111.

[204] Z. Cai, X. Yan, K. Ma, Y. Wu, Y. Huang, J. Cheng, T. Su, and F. Yu, "Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1967–1981, 2021.

[205] W. Liu, Z. Lai, S. Li, Y. Duan, K. Ge, and D. Li, "Autopipe: A fast pipeline parallelism approach with balanced partitioning and micro-batch slicing," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 301–312.

[206] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International conference on machine learning*. PMLR, 2017, pp. 2430–2439.

[207] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1676–1684.

[208] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, "Efficient algorithms for device placement of dnn graph operators," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 451–15 463, 2020.

[209] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for dnn parallelization," *Advances in Neural Information Processing Systems*, vol. 34, pp. 24 829–24 840, 2021.

[210] C. Unger, Z. Jia, W. Wu, S. Lin, M. Baines, C. E. Q. Narvaez, V. Ramakrishnaiah, N. Prajapati, P. McCormick, J. Mohd-Yusof *et al.*, "Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 267–284.

[211] G. Liu, Y. Miao, Z. Lin, X. Shi, S. Maleki, F. Yang, Y. Bao, and S. Wang, "Aceso: Efficient parallel dnn training through iterative bottleneck alleviation," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 163–181.

[212] S. Alabed, B. Chrzaszcz, J. Franco, D. Grewe, D. Maclaurin, J. Molloy, T. Natan, T. Norman, X. Pan, A. Paszke *et al.*, "Partir: Composing spmd partitioning strategies for machine learning," *arXiv preprint arXiv:2401.11202*, 2024.

[213] Z. Lin, Y. Miao, Q. Zhang, F. Yang, Y. Zhu, C. Li, S. Maleki, X. Cao, N. Shang, Y. Yang *et al.*, "{nnScaler}:{Constraint-Guided} parallelization plan generation for deep learning training," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 347–363.

[214] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu *et al.*, "Oneflow: Redesign the distributed deep learning framework from scratch," *arXiv preprint arXiv:2110.15032*, 2021.

[215] J. Chen, S. Li, R. Guo, J. Yuan, and T. Hoefler, "Autoddl: Automatic distributed deep learning with near-optimal bandwidth cost," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[216] DeepSpeed. (2021) Deepspeed autotuning. [Online]. Available: https://www.deepspeed.ai/tutorials/autotuning

[217] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie, H. Zhang, and B. Cui, "Galvatron: Efficient transformer training over multiple gpus using automatic parallelism," *Proceedings of the VLDB Endowment*, vol. 16, no. 3, pp. 470–479, 2022.

[218] Z. Lai, S. Li, X. Tang, K. Ge, W. Liu, Y. Duan, L. Qiao, and D. Li, "Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models," *IEEE*

Transactions on Parallel and Distributed Systems, vol. 34, no. 5, pp. 1466–1478, 2023.

[219] Y. Liu, S. Li, J. Fang, Y. Shao, B. Yao, and Y. You, "Colossal-Auto: Unified Automation of Parallelization and Activation Checkpoint for Large-scale Models," *arXiv:2302.02599*, 2023.

[220] Y. Wang, Y. Jiang, X. Miao, F. Fu, S. Zhu, X. Nie, Y. Tu, and B. Cui, "Improving automatic parallel training via balanced memory workload optimization," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[221] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 307–321.

[222] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 342–355.

[223] X. Jia, L. Jiang, A. Wang, W. Xiao, Z. Shi, J. Zhang, X. Li, L. Chen, Y. Li, Z. Zheng *et al.*, "Whale: Efficient giant model training over heterogeneous {GPUs}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 673–688.

[224] D. Li, H. Wang, E. Xing, and H. Zhang, "Amp: Automatically finding model parallel strategies with heterogeneity awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 6630–6639, 2022.

[225] P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy *et al.*, "Pathways: Asynchronous distributed dataflow for ml," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 430–449, 2022.

[226] Y. Duan, Z. Lai, S. Li, W. Liu, K. Ge, P. Liang, and D. Li, "Hph: Hybrid parallelism on heterogeneous clusters for accelerating large-scale dnns training," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 313–323.

[227] X. Miao, Y. Shi, Z. Yang, B. Cui, and Z. Jia, "Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2354–2363, 2023.

[228] S. Zhang, L. Diao, C. Wu, Z. Cao, S. Wang, and W. Lin, "Hap: Spmd dnn training on heterogeneous gpu clusters with automated program synthesis," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 524–541.

[229] J. Zhang, G. Niu, Q. Dai, H. Li, Z. Wu, F. Dong, and Z. Wu, "Pipepar: Enabling fast dnn pipeline parallel training in heterogeneous gpu clusters," *Neurocomputing*, vol. 555, p. 126661, 2023.

[230] B. Yuan, Y. He, J. Davis, T. Zhang, T. Dao, B. Chen, P. S. Liang, C. Re, and C. Zhang, "Decentralized training of foundation models in heterogeneous environments," *Advances in Neural Information Processing Systems*, vol. 35, pp. 25 464–25 477, 2022.

[231] M. Ryabinin, T. Dettmers, M. Diskin, and A. Borzunov, "Swarm parallelism: Training large models can be surprisingly communication-efficient," in *International Conference on Machine Learning*. PMLR, 2023, pp. 29 416–29 440.

[232] Z. Tang, Y. Wang, X. He, L. Zhang, X. Pan, Q. Wang, R. Zeng, K. Zhao, S. Shi, B. He *et al.*, "Fusionai: Decentralized training and deploying llms with massive consumer-level gpus," *arXiv preprint arXiv:2309.01172*, 2023.

[233] Z. Yao, R. Y. Aminabadi, O. Ruwase, S. Rajbhandari, X. Wu, A. A. Awan, J. Rasley, M. Zhang, C. Li, C. Holmes *et al.*, "Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales," *arXiv preprint arXiv:2308.01320*, 2023.

[234] "Trl - transformer reinforcement learning." [Online]. Available: https://github.com/huggingface/trl

[235] J. Hu, X. Wu, W. Wang, D. Zhang, Y. Cao *et al.*, "Openrlhf: An easy-to-use, scalable and high-performance rlhf framework," *arXiv preprint arXiv:2405.11143*, 2024.

[236] Y. Xiao, W. Wu, Z. Zhou, F. Mao, S. Zhao, L. Ju, L. Liang, X. Zhang, and J. Zhou, "An Adaptive Placement and Parallelism Framework for Accelerating RLHF Training," *arxiv:2312.11819*, 2023.

[237] Z. Mei, W. Fu, K. Li, G. Wang, H. Zhang, and Y. Wu, "Realhf: Optimized rlhf training for large language models through parameter reallocation," *arXiv preprint arXiv:2406.14088*, 2024.

[238] K. Lei, Y. Jin, M. Zhai, K. Huang, H. Ye, and J. Zhai, "{PUZZLE}: Efficiently aligning large language models through {Light-

Weight} context switch," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 127–140.

[239] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*. Springer, 1994, pp. 213–218.

[240] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[241] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[242] L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969.

[243] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *European Conference on Parallel Processing*. Springer, 2011, pp. 90–109.

[244] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[245] F. Strati, P. Elvinger, T. Kerimoglu, and A. Klimovic, "Ml training with cloud gpu shortages: Is cross-region the answer?" in *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024, pp. 107–116.

[246] W. Peebles and S. Xie, "Scalable diffusion models with transformers," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 4195–4205.

[247] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[248] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[249] S. Singh, O. Ruwase, A. A. Awan, S. Rajbhandari, Y. He, and A. Bhatele, "A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 203–214.

[250] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne *et al.*, "Jax: composable transformations of python+ numpy programs," 2018.

[251] G. X. team. (2017) Xla: Optimizing compiler for machine learning. [Online]. Available: https://github.com/openxla/xla

[252] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[253] S. Li, H. Liu, Z. Bian, J. Fang, H. Huang, Y. Liu, B. Wang, and Y. You, "Colossal-ai: A unified deep learning system for large-scale parallel training," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 766–775.

[254] J. Wang, Y. Lu, B. Yuan, B. Chen, P. Liang, C. De Sa, C. Re, and C. Zhang, "Cocktailsgd: Fine-tuning foundation models over 500mbps networks," in *International Conference on Machine Learning*. PMLR, 2023, pp. 36 058–36 076.

[255] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," *arXiv:2203.02155*, 2022.

[256] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[257] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[258] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[259] "unsloth - finetune llama 3, mistral, phi-3 & gemma 2-5x faster with 80

[260] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed

framework for emerging {AI} applications," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.

[261] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[262] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *arXiv preprint arXiv:2407.08608*, 2024.

[263] H. Liu and P. Abbeel, "Blockwise parallel transformers for large context models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[264] R. Wu, X. Zhu, J. Chen, S. Liu, T. Zheng, X. Liu, and H. An, "Swattention: designing fast and memory-efficient attention for a new sunway supercomputer," *The Journal of Supercomputing*, pp. 1–24, 2024.

[265] G. Bikshandi and J. Shah, "A case study in cuda kernel fusion: Implementing flashattention-2 on nvidia hopper architecture using the cutlass library," *arXiv preprint arXiv:2312.11918*, 2023.

[266] Y. Zhai, C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, and Y. Zhu, "Bytetransformer: A high-performance transformer boosted for variable-length inputs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 344–355.

[267] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[268] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui *et al.*, "{ROLLER}: Fast and efficient tensor compilation for deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 233–248.

[269] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[270] G. Huang, Y. Bai, L. Liu, Y. Wang, B. Yu, Y. Ding, and Y. Xie, "Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[271] S. Zheng, S. Chen, P. Song, R. Chen, X. Li, S. Yan, D. Lin, J. Leng, and Y. Liang, "Chimera: An analytical optimizing framework for effective compute-intensive operators fusion," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1113–1126.

[272] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, "Welder: Scheduling deep learning memory access via tile-graph," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 701–718.

[273] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 929–947.

[274] M. Ibrahim, S. Aga, A. Li, S. Pati, and M. Islam, "Jit-q: Just-in-time quantization with processing-in-memory for efficient ml training," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 46–59, 2024.

[275] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[276] X. He, J. Sun, H. Chen, and D. Li, "Campo:{Cost-Aware} performance optimization for {Mixed-Precision} neural network training," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 505–518.

[277] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[278] M. Li, R. B. Basat, S. Vargaftik, C. Lao, K. Xu, M. Mitzenmacher, and M. Yu, "{THC}: Accelerating distributed deep learning using tensor homomorphic compression," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1191–1211.

[279] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," *Advances in neural information processing systems*, vol. 31, 2018.

[280] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[281] H. Peng, K. Wu, Y. Wei, G. Zhao, Y. Yang, Z. Liu, Y. Xiong, Z. Yang, B. Ni, J. Hu *et al.*, "Fp8-lm: Training fp8 large language models," *arXiv preprint arXiv:2310.18313*, 2023.

[282] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf *et al.*, "Microscaling data formats for deep learning," *arXiv preprint arXiv:2310.10537*, 2023.

[283] H. Xi, Y. Chen, K. Zhao, K. Zheng, J. Chen, and J. Zhu, "Jetfire: Efficient and accurate transformer pretraining with int8 data flow and per-block quantization," *arXiv preprint arXiv:2403.12422*, 2024.

[284] H. Xi, C. Li, J. Chen, and J. Zhu, "Training transformers with 4-bit integers," *Advances in Neural Information Processing Systems*, vol. 36, pp. 49 146–49 168, 2023.

[285] H. Wang, S. Ma, L. Dong, S. Huang, H. Wang, L. Ma, F. Yang, R. Wang, Y. Wu, and F. Wei, "Bitnet: Scaling 1-bit transformers for large language models," *arXiv preprint arXiv:2310.11453*, 2023.

[286] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," *arXiv preprint arXiv:2402.17764*, 2024.

[287] M. N. Rabe and C. Staats, "Self-attention does not need $o(n^2)$ memory," *arXiv preprint arXiv:2112.05682*, 2021.

[288] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[289] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[290] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 883–898.

[291] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai *et al.*, "Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 359–373.

[292] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[293] M. Cherti, R. Beaumont, R. Wightman, M. Wortsman, G. Ilharco, C. Gordon, C. Schuhmann, L. Schmidt, and J. Jitsev, "Reproducible scaling laws for contrastive language-image learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 2818–2829.

[294] K. Yang, Y.-F. Chen, G. Roumpos, C. Colby, and J. Anderson, "High performance monte carlo simulation of ising model on tpu clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–15.

[295] K. Fischer and E. Saba, "Automatic full compilation of julia programs and ml models to cloud tpus," *arXiv preprint arXiv:1810.09868*, 2018.

[296] J. Choquette, "Nvidia hopper h100 gpu: Scaling performance," *IEEE Micro*, 2023.

[297] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock, "Dynamic tensor rematerialization," *arXiv preprint arXiv:2006.09616*, 2020.

[298] Z. Hu, J. Xiao, Z. Deng, M. Li, K. Zhang, X. Zhang, K. Meng, N. Sun, and G. Tan, "Megtaichi: Dynamic tensor-based memory

management optimization for dnn training," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.

[299] J. Zhang, S. Ma, P. Liu, and J. Yuan, "Coop: Memory is not a commodity," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[300] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica, "Checkmate: Breaking the memory wall with optimal tensor rematerialization," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 497–511, 2020.

[301] T. Yuan, Y. Liu, X. Ye, S. Zhang, J. Tan, B. Chen, C. Song, and D. Zhang, "Accelerating the Training of Large Language Models using Efficient Activation Rematerialization and Optimal Hybrid Parallelism," *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024.

[302] G. Wang, H. Qin, S. A. Jacobs, C. Holmes, S. Rajbhandari, O. Ruwase, F. Yan, L. Yang, and Y. He, "ZeRO++: Extremely Efficient Collective Communication for Giant Model Training," *arXiv:2306.10209*, 2023.

[303] C. Wu, H. Zhang, L. Ju, J. Huang, Y. Xiao, Z. Huan, S. Li, F. Meng, L. Liang, X. Zhang, and J. Zhou, "Rethinking memory and communication cost for efficient large language model training," *arXiv preprint arXiv:2310.06003*, 2023.

[304] C. Luo, T. Zhong, and G. Fox, "Rtp: Rethinking tensor parallelism with memory deduplication," *arXiv preprint arXiv:2311.01635*, 2023.

[305] Q. Chen, Q. Hu, G. Wang, Y. Xiong, T. Huang, X. Chen, Y. Gao, H. Yan, Y. Wen, T. Zhang, and P. Sun, "Amsp: Reducing communication overhead of zero for efficient llm training," 2024.

[306] H. Shu, A. Wang, Z. Shi, H. Zhao, Y. Li, and L. Lu, "Roam: memory-efficient large dnn training via optimized operator ordering and memory layout," *arXiv preprint arXiv:2310.19295*, 2023.

[307] A. Imanishi and Z. Xu, "A Heuristic for Periodic Memory Allocation with Little Fragmentation to Train Neural Networks," in *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2024, pp. 82–94.

[308] C. Guo, R. Zhang, J. Xu, J. Leng, Z. Liu, Z. Huang, M. Guo, H. Wu, S. Zhao, J. Zhao *et al.*, "Gmlake: Efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 450–466.

[309] PyTorch. (2023) Pytorch expandable segments. [Online]. Available: https://github.com/pytorch/pytorch/pull/96995

[310] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, "Training large neural networks with constant memory using a new execution algorithm," *arXiv preprint arXiv:2002.05645*, 2020.

[311] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 551–564.

[312] H. Huang, J. Fang, H. Liu, S. Li, and Y. You, "Elixir: Train a large language model on a small gpu cluster," *arXiv preprint arXiv:2212.05339*, 2022.

[313] X. Nie, X. Miao, Z. Yang, and B. Cui, "Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 2615–2628.

[314] J. Fang, Z. Zhu, S. Li, H. Su, Y. Yu, J. Zhou, and Y. You, "Parallel training of pre-trained models via chunk-based dynamic memory management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 304–315, 2022.

[315] Y. Feng, M. Xie, Z. Tian, S. Wang, Y. Lu, and J. Shu, "Mobius: Fine tuning large-scale models on commodity gpu servers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 489–501.

[316] Y. Li, A. Phanishayee, D. Murray, J. Tarnawski, and N. S. Kim, "Harmony: overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2747–2760, 2022.

[317] S.-F. Lin, Y.-J. Chen, H.-Y. Cheng, and C.-L. Yang, "Tensor movement orchestration in multi-gpu training systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1140–1152.

[318] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang, "Stronghold: fast and affordable billion-scale deep learning model training," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–17.

[319] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–14.

[320] X. Nie, Y. Liu, F. Fu, J. Xue, D. Jiao, X. Miao, Y. Tao, and B. Cui, "Angel-ptm: A scalable and economical large-scale pre-training system in tencent," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3781–3794, 2023.

[321] H. Jang, J. Song, J. Jung, J. Park, Y. Kim, and J. Lee, "Smart-infinity: Fast large language model training using near-storage processing on a real system," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 345–360.

[322] C. Liao, M. Sun, Z. Yang, K. Chen, B. Yuan, F. Wu, and Z. Wang, "Adding nvme ssds to enable and accelerate 100b model fine-tuning on a single gpu," *arXiv preprint arXiv:2403.06504*, 2024.

[323] D. Yu, L. Shen, H. Hao, W. Gong, H. Wu, J. Bian, L. Dai, and H. Xiong, "Moesys: A distributed and efficient mixture-of-experts training and inference system for internet services," *IEEE Transactions on Services Computing*, 2024.

[324] L. Shen, Z. Wu, W. Gong, H. Hao, Y. Bai, H. Wu, X. Wu, J. Bian, H. Xiong, D. Yu *et al.*, "Se-moe: A scalable and efficient mixture-of-experts distributed training and inference system," *arXiv preprint arXiv:2205.10034*, 2022.

[325] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[326] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[327] Z. Zhang, Y. Xia, H. Wang, D. Yang, C. Hu, X. Zhou, and D. Cheng, "Mpmoe: Memory efficient moe for pre-trained models with adaptive pipeline parallelism," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[328] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[329] W. Gropp, "Mpich2: A new start for mpi implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting Linz, Austria, September 29–Oktober 2, 2002 Proceedings 9*. Springer, 2002, pp. 7–7.

[330] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The mvapich project: Evolution and sustainability of an open source production quality mpi library for hpc," in *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int'l Conference on Supercomputing (WSSPE)*, 2013.

[331] Nvidia. (2016) Nccl library. [Online]. Available: https://github.com/NVIDIA/nccl

[332] AMD. (2018) Rccl library. [Online]. Available: https://github.com/ROCm/rccl

[333] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[334] S. Jeaugey. (2019) Massively scale your deep learning training with nccl 2.4. [Online]. Available: https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/

[335] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[336] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Massively distributed sgd: Imagenet/resnet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[337] J. Dong, S. Wang, F. Feng, Z. Cao, H. Pan, L. Tang, P. Li, H. Li, Q. Ran, Y. Guo *et al.*, "Accl: Architecting highly scalable dis-

tributed training systems with highly efficient collective communication library," *IEEE micro*, vol. 41, no. 5, pp. 85–92, 2021.

[338] M. Cho, U. Finkler, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 241–251, 2019.

[339] L. Luo, P. West, A. Krishnamurthy, L. Ceze, and J. Nelson, "Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training," *Proc. of MLSys*, 2020.

[340] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong, "Gc3: An optimizing compiler for gpu collective communication," 2022.

[341] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing optimal collective algorithms," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 62–75.

[342] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh, "{TACCL}: Guiding collective algorithm synthesis using communication sketches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 593–612.

[343] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," pp. 172–186, 2020.

[344] N. Xie, T. Norman, D. Grewe, and D. Vytiniotis, "Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 548–566, 2022.

[345] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.

[346] P. Sun, Y. Wen, R. Han, W. Feng, and S. Yan, "Gradientflow: Optimizing network performance for large-scale distributed dnn training," *IEEE Transactions on Big Data*, vol. 8, no. 2, pp. 495–507, 2019.

[347] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 132–145, 2019.

[348] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 418–430, 2019.

[349] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[350] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 626–635.

[351] F. Li, S. Zhao, Y. Qing, X. Chen, X. Guan, S. Wang, G. Zhang, and H. Cui, "Fold3d: Rethinking and parallelizing computational and communicational tasks in the training of large dnn models," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1432–1449, 2023.

[352] S. Li, K. Lu, Z. Lai, W. Liu, K. Ge, and D. Li, "A multidimensional communication scheduling method for hybrid parallel dnn training," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[353] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang *et al.*, "Overlap communication with dependent computation via decomposition in large deep learning models," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2022, pp. 93–106.

[354] K. Mahajan, C.-H. Chu, S. Sridharan, and A. Akella, "Better together: Jointly optimizing {ML} collective scheduling and execution planning using {SYNDICATE}," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 809–824.

[355] C. Chen, X. Li, Q. Zhu, J. Duan, P. Sun, X. Zhang, and C. Yang, "Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 178–191.

[356] L. Zhang, S. Shi, X. Chu, W. Wang, B. Li, and C. Liu, "Dear: Accelerating distributed deep learning with fine-grained all-reduce pipelining," in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2023, pp. 142–153.

[357] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi, "Breaking the computation and communication abstraction barrier in distributed machine learning workloads," pp. 402–416, 2022.

[358] S. Pati, S. Aga, M. Islam, N. Jayasena, and M. D. Sinclair, "T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1146–1164.

[359] S. Li, Z. Lai, Y. Hao, W. Liu, K. Ge, X. Deng, D. Li, and K. Lu, "Automated tensor model parallelism with overlapped communication for efficient foundation model training," 2023.

[360] P. Chen, W. Zhang, S. He, Y. Gu, Z. Peng, K. Huang, X. Zhan, W. Chen, Y. Zheng, Z. Wang *et al.*, "Optimizing large model training through overlapped activation recomputation," 2024.

[361] H. Oh, J. Lee, H. Kim, and J. Seo, "Out-of-order backprop: An effective scheduling technique for deep learning," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 435–452.

[362] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling distributed machine learning with {In-Network} aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 785–808.

[363] Y. Yuan, O. Alama, J. Fei, J. Nelson, D. R. Ports, A. Sapio, M. Canini, and N. S. Kim, "Unlocking the power of inline {Floating-Point} operations on programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 683–700.

[364] S. Liu, Q. Wang, J. Zhang, Q. Lin, Y. Liu, M. Xu, R. C. Chueng, and J. He, "Netreduce: Rdma-compatible in-network reduction for distributed dnn training acceleration," *arXiv preprint arXiv:2009.09736*, 2020.

[365] Y. Liu, J. Zhang, S. Liu, Q. Wang, W. Dai, and R. C. C. Cheung, "Scalable fully pipelined hardware architecture for in-network aggregated allreduce communication," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 10, pp. 4194–4206, 2021.

[366] N. Gebara, M. Ghobadi, and P. Costa, "In-network aggregation for shared machine learning clusters," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 829–844, 2021.

[367] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "{ATP}: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 741–761.

[368] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir *et al.*, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 2016, pp. 1–10.

[369] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "Netagg: Using middleboxes for application-specific on-path aggregation in data centres," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 249–262.

[370] F. Yang, Z. Wang, X. Ma, G. Yuan, and X. An, "Switchagg: A further step towards in-network computation," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 185–185.

[371] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 29–42.

[372] H. Zhu, W. Jiang, Q. Hong, and Z. Guo, "When in-network computing meets distributed machine learning," *IEEE Network*, 2024.

[373] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[374] H. Pan, P. Cui, R. Jia, P. Zhang, L. Zhang, Y. Yang, J. Wu, J. Dong, Z. Cao, Q. Li *et al.*, "Enabling fast and flexible distributed deep learning with programmable switches," *arXiv preprint arXiv:2205.05243*, 2022.

[375] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 279–291.

[376] Y. Zu, A. Ghaffarkhah, H.-V. Dang, B. Towles, S. Hand, S. Huda, A. Bello, A. Kolbasov, A. Rezaei, D. Du, S. Lacy, H. Wang, A. Wisner, C. Lewis, and H. Bahini, "Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 761–774.

[377] J. Dong, B. Luo, J. Zhang, P. Zhang, F. Feng, Y. Zhu, A. Liu, Z. Chen, Y. Shi, H. Jiao *et al.*, "Boosting large-scale parallel training efficiency with c4: A communication-driven approach," *arXiv preprint arXiv:2406.04594*, 2024.

[378] T. Gershon, S. Seelam, B. Belgodere, M. Bonilla, L. Hoang, D. Barnett, I. Chung, A. Mohan, M.-H. Chen, L. Luo *et al.*, "The infrastructure powering ibm's gen ai model development," *arXiv preprint arXiv:2407.05467*, 2024.

[379] T. He, X. Li, Z. Wang, K. Qian, J. Xu, W. Yu, and J. Zhou, "Unicron: Economizing Self-Healing LLM Training at Scale," *arXiv preprint arXiv:2401.00134*, 2023.

[380] B. Wu, L. Xia, Q. Li, K. Li, X. Chen, Y. Guo, T. Xiang, Y. Chen, and S. Li, "TRANSOM: An Efficient Fault-Tolerant System for Training LLMs," *arXiv preprint arXiv:2310.10046*, 2023.

[381] Y. Xiong, Y. Jiang, Z. Yang, L. Qu, G. Zhao, S. Liu, D. Zhong, B. Pinzur, J. Zhang, Y. Wang *et al.*, "Superbench: Improving cloud ai infrastructure reliability with proactive validation," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 835–850.

[382] T. Gupta, S. Krishnan, R. Kumar, A. Vijeev, B. Gulavani, N. Kwatra, R. Ramjee, and M. Sivathanu, "Just-in-time checkpointing: Low cost error recovery from deep learning training failures," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 1110–1125.

[383] A. Group. (2024) Dlrover: An automatic distributed deep learning system. [Online]. Available: https://github.com/intelligent-machine-learning/dlrover

[384] X. Lian, S. A. Jacobs, L. Kurilenko, M. Tanaka, S. Bekman, O. Ruwase, and M. Zhang, "Universal checkpointing: Efficient and flexible checkpointing for large scale distributed training," *arXiv preprint arXiv:2406.18820*, 2024.

[385] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "{Check-N-Run}: A checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 929–943.

[386] "Torchsnapshor: A performant, memory-efficient checkpointing library for pytorch applications, designed with large, complex distributed workloads in mind." [Online]. Available: https://github.com/pytorch/torchsnapshot

[387] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CC-GRID)*. IEEE, 2020, pp. 172–181.

[388] J. Mohan, A. Phanishayee, and V. Chidambaram, "{CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 203–216.

[389] M. Chen, Y. Hua, R. Bai, and J. Huang, "A cost-efficient failure-tolerant scheme for distributed dnn training," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 150–157.

[390] A. Maurya, R. Underwood, M. M. Rafique, F. Cappello, and B. Nicolae, "Datastates-llm: Lazy asynchronous checkpointing for large language models," *arXiv preprint arXiv:2406.10707*, 2024.

[391] G. Wang, O. Ruwase, B. Xie, and Y. He, "Fastpersist: Accelerating model checkpointing in deep learning," *arXiv preprint arXiv:2406.13768*, 2024.

[392] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. E. Ng, and Y. Wang, "Gemini: Fast failure recovery in distributed training with in-memory checkpoints," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 364–381.

[393] Y. Wang, S. Shi, X. He, Z. Tang, X. Pan, Y. Zheng, X. Wu, A. C. Zhou, B. He, and X. Chu, "Reliable and Efficient In-Memory Fault Tolerance of Large Language Model Pretraining," *arXiv preprint arXiv:2310.12670*, 2023.

[394] J. Duan, Z. Song, X. Miao, X. Xi, D. Lin, H. Xu, M. Zhang, and Z. Jia, "Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances," *arXiv preprint arXiv:2403.14097*, 2024.

[395] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury, "Oobleck: Resilient distributed training of large models using pipeline templates," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 382–395.

[396] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu, "Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 497–513.

[397] S. Gandhi, M. Zhao, A. Skiadopoulos, and C. Kozyrakis, "Slipstream: Adapting pipelines for distributed training of large dnns amid failures," *arXiv preprint arXiv:2405.14009*, 2024.

[398] M. Research. (2022) Metaseq: Opt-175 logbook. [Online]. Available: https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf

[399] Y. Gao, X. Shi, H. Lin, H. Zhang, H. Wu, R. Li, and M. Yang, "An empirical study on quality issues of deep learning platform," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2023, pp. 455–466.

[400] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[401] NVIDIA, "Nvidia data center gpu manager (dcgm)," https://developer.nvidia.com/dcgm, 2017.

[402] M. Steinman. (2023) Taking stock of new data center computing paradigm. [Online]. Available: https://www.eetimes.com/taking-stock-of-new-data-center-computing-paradigm/

[403] Z. Xu, T. Zhou, M. Ma, C. Deng, Q. Dai, and L. Fang, "Large-scale photonic chiplet taichi empowers 160-tops/w artificial general intelligence," *Science*, vol. 384, no. 6692, pp. 202–209, 2024.