MoESys: A Distributed and Efficient Mixture-of-Experts Training and Inference System for Internet Services

Dianhai Yu*, Liang Shen*, Hongxiang Hao, Weibao Gong, Huachao Wu, Jiang Bian, *Member, IEEE*, Lirong Dai, *Member, IEEE*, Haoyi Xiong, *Senior Member, IEEE*

Abstract—While modern internet services, such as chatbots, search engines, and online advertising, demand the use of large-scale deep neural networks (DNNs), distributed training and inference over heterogeneous computing systems are desired to facilitate these DNN models. Mixture-of-Experts (MoE) is one the most common strategies to lower the cost of training subject to the overall size of models/data through gating and parallelism in a divide-and-conquer fashion. While DeepSpeed [1] has made efforts in carrying out large-scale MoE training over heterogeneous infrastructures, the efficiency of training and inference could be further improved from several system aspects, including load balancing, communication/computation efficiency, and memory footprint limits. In this work, we present a novel **MoESys** that boosts efficiency in both large-scale training and inference. Specifically, in the training procedure, the proposed **MoESys** adopts an *Elastic MoE training* strategy with *2D prefetch* and *Fusion communication* over *Hierarchical storage*, so as to enjoy efficient parallelisms. For scalable inference in a single node, especially when the model size is larger than GPU memory, **MoESys** builds the CPU-GPU memory jointly into a ring of sections to load the model, and executes the computation tasks across the memory sections in a round-robin manner for efficient inference. We carried out extensive experiments to evaluate **MoESys**, where **MoESys** successfully trains a Unified Feature Optimization [2] (UFO) model with a Sparsely-Gated Mixture-of-Experts model of 12B parameters in 8 days on 48 A100 GPU cards. The comparison against the state-of-the-art shows that **MoESys** outperformed DeepSpeed with 33% higher throughput (tokens per second) in training and 13% higher throughput in inference in general. Particularly, under unbalanced MoE Tasks, e.g., UFO, **MoESys** achieved 64% higher throughput with 18% lower memory footprints.

Index Terms—Large Models for Internet Services, MoE, Distributed Training, Distributed Inference

1 Introduction

In recent years, there has been significant evolution in internet services, and the integration of artificial intelligence has made deep learning models indispensable in the internet ecosystem [3]–[6]. Particularly, large deep neural network (DNN) models such as BERT and GPT have gained increasing popularity due to their remarkable performance in text and language processing applications [7], leading to the reliance of various internet services, including chat bots, online advertising platforms, recommender systems, search engines, and translation tools, on these models to provide users with the desired accuracy and customization [8]–[12]. While the utilization of large models has significantly enhanced the performance of internet services, it has come at the cost of expanding the parameter scale to tens of billions, such as the GPT-3 model with 175B parameters [13], [14], Ernie3.0 Titan with 260B parameters [15], and Megatron-Turing NLG with 530B parameters [16]. However, these densely activated models necessitate abundant computing resources and extensive training time. For instance, the training of Megatron-Turing NLG with 530B parameters, one of the largest densely activated models, required three months using over 2000 NVIDIA A100 GPUs [16], making it financially expensive and hindering the development of models with even larger parameter scales. Moreover, the inference performance of these super large-scale models seldom meets the current industrial demands [5], [17].

Various ad-hoc strategies have been employed to improve the efficiency of training large-scale models. One such approach is the AIBox concept, applied specifically in the training of Click-Through Rate (CTR) prediction models to reduce costs. This method involves sparsifying feature embeddings and leveraging a distributed multi-GPU setup to over-parameterize the model [18]. AIBox primarily focuses on certain layers for processing high-dimensional data, with an aim to scale the model. Conversely, in the realm of pre-trained language models, multi-task learning has been adopted, especially for multilingual neural machine translation [19]. Models like MT5 [20], MASSively [21], and MultiNLI [22] differ from densely activated models but require significant computational resources to surpass existing benchmarks.

To address these challenges, Mixture-of-Experts (MoE) based sparsely activated neural networks have been introduced for training larger models with minimal or no additional computational resources, while still achieving improved training outcomes [23]–[26]. MoE architectures activate only a subset of parameters based on the input data, unlike densely activated models. This selective activation results in a sub-linear increase in computational costs relative to model size. For instance, GLaM's largest variant [27] possesses 1.2T parameters with 64 experts per MoE layer, yet only activates a 95B-parameter subnet (8% of 1.2T) for each input token. Training this model saves two-thirds of the power required for GPT-3 (175B) [13], while halving the

This work was supported in part by (1) project CEIEC-2022-ZM02-0247 and (2) Beijing Municipal Science and Technology Project (No. Z231100010323002)
**The first two author contributed equally to this work. D. Yu, L. Shen, H. Hao, W. Gong, H. Wu, J. Bian, H. Xiong are with Baidu, Inc., Beijing, China. L. Dai is with Department of Electronic Engineering and Information Science, University of Science and Technology of China, Heifei, China. Corresponding author is Jiang Bian (email: jiangbian03@gmail.com).*

computational resources needed during inference. Despite all the benefits, MoE models still face numerous challenges and limitations, especially in computation, communication, and storage:

- *Computation* – The computation cost per GPU remains constant in MoE models, but increases with the total number of experts. Training performance suffers due to expert imbalance, where some are overtrained and others underutilized [25]. Solutions include auxiliary losses [24], random expert selection [28], and noise in routing [25]. However, these focus more on scheduling than computation and require substantial CPU resources. Inefficient computational task allocation and redundant operations, like H2D and D2H transfers, reduce efficiency and increase latency [29].
- *Communication* – In MoE models, imbalances in routing strategies persist despite advanced learning methods [24], [25], [30], [31]. Unbalanced data leads to inconsistent progress and redundant waiting in multi-task training. For example, the Switch Transformer model requires four AlltoAll communications per MoE layer, leading to performance degradation due to routing conflicts and blocking in unknown network topologies [25].
- *Storage* – The memory and storage capacity limits MoE model sizes. While dense models are constrained by training time, MoE models scale better due to their sublinear computing cost increase. A dense model with 1 trillion parameters requires 3 months to train on 3072 NVIDIA A100 GPUs, but an MoE model can be trained in weeks [14]. However, the model's scalability depends on device memory capacity. The differences in I/O latency between HBM in GPUs, CPU memory, and SSDs cause delays, necessitating efficient storage management for sparsely activated training [32].

Our Contributions. To overcome the aforementioned challenges and limitations of MoE, we introduce a novel unified framework **MoESys**, based on an open-source platform for MoE training and inference. The non-trivial contributions in **MoESys** are as follows,

- A novel distributed framework named **MoESys** is designed, which is capable of scaling MoE models to trillions of parameters, fully utilizes the clusters including HBM, CPU memory and even SSDs to break the memory wall and achieves efficient training scheduling. Notably, **MoESys** incorporates advanced techniques such as 2D prefetch scheduling and fusion communication, further enhancing the efficiency of heterogeneous storage systems.
- A new inference method based on the ring memory is employed by dynamic graph scheduling, which can integrate the computation and communication as much as possible and accelerate the inference procedure without using additional machines for larger-scale MoE models.
- Several effective training strategies have been initially devised in **MoESys** for NLP and CV tasks, aimed at scaling up multi-task learning without requiring additional memory. These strategies include load balancing, embedding partition, and resource-aware communication.
- We conduct comprehensive industrial-level experiments to showcase the significant performance gain using **MoESys**, where the practice in this work could benefit the future development of large-scale MoE training and inference.

We organize the rest of this manuscript as follows. In Section 2, we review the previous efforts on the design of MoE. Section 3 introduces the novel design of **MoESys** respectively. Additionally, we reveal details of the practical implementation strategies adopted in **MoESys** in Section 4. To demonstrate the effectiveness and efficiency of **MoESys**, we conduct comprehensive experiments and analyze the results in Section 5. Finally, we conclude this work and look forward to the future direction in Section 6.

2 Related Work

In this section, we review the relevant works in the field from the perspectives of large models for internet services and their training and inference systems.

2.1 Internet Services and Large Models

Large Language Models (LLMs) are revolutionizing internet services such as search engines, chatbots, online advertising, and cloud applications [8]–[12], [33]. Organizations are increasingly using custom LLMs tailored to specific needs. These domain-specific models enhance internet service quality and customer experience, being more efficient and faster than general-purpose LLMs, particularly for applications involving proprietary data. An example is BloombergGPT [34], a custom LLM by Bloomberg, which significantly impacts online finance services by rapidly evaluating financial data for risk assessments, financial sentiment analysis, and potentially automating accounting and auditing. Despite its large size of 50 billion parameters, BloombergGPT avoids traditional single-model training, favoring a Mixture-of-Experts (MoE) system for better efficiency and effectiveness. MoE models have shown great promise in natural language processing, with strategies focusing on routing enhancements [28], [35] to improve model quality and performance. Notice that, the GLaM [27] framework demonstrates that the largest MoE with 1.2 trillion parameters is more energy-efficient, using only one-third of the energy required for training GPT-3.

In light of the scaling law, there's a growing trend to increase model sizes. MoE-based models with billions or even trillions of parameters, like CPM-2 [36], M6-T [31], M6-10T [37], and GLaM [27], are showing superior generalization in language processing and multi-modal tasks. Baidu's UFO [2] model, another MoE-based framework, emphasizes deployment efficiency and big data utilization. It features a super network comprising multiple subtasks, with a routing strategy selecting the appropriate subtask for training.

2.2 MoE Training and Inference Systems

The rising popularity of the Mixture of Experts (MoE) training approach has led to the release of several open-source MoE training frameworks and systems by various scientific research bodies and corporations. DeepSpeed-MoE integrates multiple distributed parallel techniques like data parallelism and tensor slicing to effectively utilize MoE parallelism, allowing for the training of larger models. It also introduces PR-MoE, a new sparsely activated model for MoE inference, and employs model compression to reduce model sizes, alongside an efficient communication strategy to

improve latency [38], [39]. FastMoE, another distributed MoE training system, offers a user-friendly hierarchical interface and straightforward guidelines for integrating Megatron-LM and Transformer-XL with data and tensor slicing parallelism [29], [40], [41]. Unlike DeepSpeed, FastMoE focuses on reducing network traffic through an advanced optimization method. The INFMoE inference system suggests an optimal computation sequence and parameter offloading using a greedy algorithm to address workload imbalances and minimize the impact of data movement, especially when offloading to CPUs, while maintaining computational efficiency [36]. Fairseq-MoE is a framework tailored for training custom models in areas like summarization, translation, and language modeling. Tutel enhances Fairseq's communication and computation capabilities, leading to a performance boost of around 40%. Notably, these improvements in Tutel have been incorporated into DeepSpeed for MoE model training [42]–[44].

Furthermore, model scale and data size are two crucial factors that significantly impact the performance and effectiveness of model training. However, exploring further in this field poses a substantial challenge for scientific institutions and enterprises due to the enormous computational and storage resource requirements involved. To address this challenge, the design of sparsely activated model has emerged in recent years and gained traction in the industry. Unlike densely activated models that involve computing all parameters, the sparsely activated model dynamically selects a subset of parameters for training based on the input data. This approach enables linear parameter scaling without increasing the computational workload, thus making larger models built on the Mixture-of-Experts (MoE) architecture more feasible and efficient.

3 MoESys Design

MoESys is an innovative system for distributed training and inference, utilizing a Mixture-of-Experts architecture to enhance scalability and efficiency. Its main objective is to adhere to predefined memory latency goals while operating within existing storage limits. A notable advancement in this area is DeepSpeed's Zero-infinity approach [45], which has successfully trained a model with over 30 trillion parameters using 512 V100 GPUs across NVIDIA DGX-2 nodes. This pioneering technique circumvents memory bottlenecks by fully exploiting a range of storage mediums, such as High Bandwidth Memory (HBM) in GPUs, CPU memory, and SSDs. This enables the training of exceptionally large models on singular devices. To refine storage use and boost training efficacy, both the Zero strategy [45] and a parameter prefetching method are implemented. However, there is a need to consider the reduced longevity and diminished performance of SSDs when near maximum capacity [46]. Moreover, DeepSpeed's current prefetching approach does not accommodate the heterogeneity of parameters specific to the Mixture-of-Experts design. **MoESys** addresses these issues by introducing an innovative prefetching scheduling technique. This method enhances both training and inference by tailoring to the distinct attributes of various parameters, effectively leveraging multi-tiered storage solutions to optimize system performance.

3.1 Overall Design of Architecture

MoESys employs a two-phase approach, namely the training phase and the inference phase, as illustrated in Figure 1. During the training phase, large-scale models are trained offline utilizing a variety of strategies. Once the model convergence is achieved, the parameters are saved for future use. On the other hand, the inference phase involves deploying the trained model to the cloud through graph optimization and pruning operations. This deployment facilitates convenient query services for users.

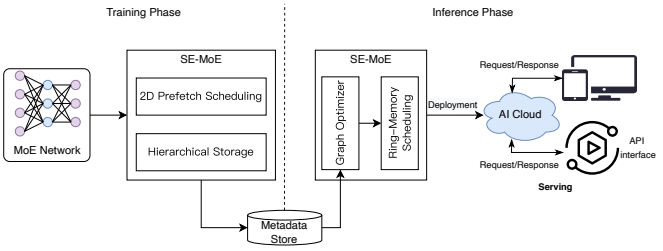


Fig. 1: **MoESys**'s architecture diagram

3.2 Training Phase

To enhance the efficiency of MoE training and address issues pertaining to Solid-State Drives (SSDs) and scheduling in the context of training large-scale models, a novel approach has been introduced [47]. In this method, MoE model parameters are divided into two categories according to their activation characteristics. Parameters in the first category are sparsely activated during training, such as those in the switching feed-forward network (FFN) layer, while the second category includes densely activated parameters, like those in the multi-head attention layer. Given that sparse parameters, which form a substantial part of the MoE model, may surpass GPU storage capacities, **MoESys** has restructured the MoE training system architecture, as shown in Figure 2. This restructure utilizes a variety of storage mediums to meet the memory demands of both sparse and dense parameters. To counteract the performance issues arising from data transfer across different storage types, a new technique termed 2D prefetch scheduling has been implemented. The following sections will delve into a comprehensive discussion of our training framework, concentrating specifically on two principal components: *Hierarchical Storage* and *2D Prefetch Scheduling*.

3.2.1 Hierarchical Storage

In the context of large-scale Mixture-of-Experts (MoE) models, the increasing scale of parameters has led to storage becoming a significant bottleneck in model training. Typically, the stored parameter states consist of three components: trainable parameters, parameter gradients, and corresponding optimizer states. Considering the different storage media available, the storage devices can be classified into three categories: GPU-Node, CPU-Node, and SSD-Node. Since dense parameters are extensively utilized for computation and do not occupy the majority of storage space, their parameter states are stored exclusively on the GPU-Node to minimize data movement. In contrast, sparse parameters, which are selectively activated during training and consume

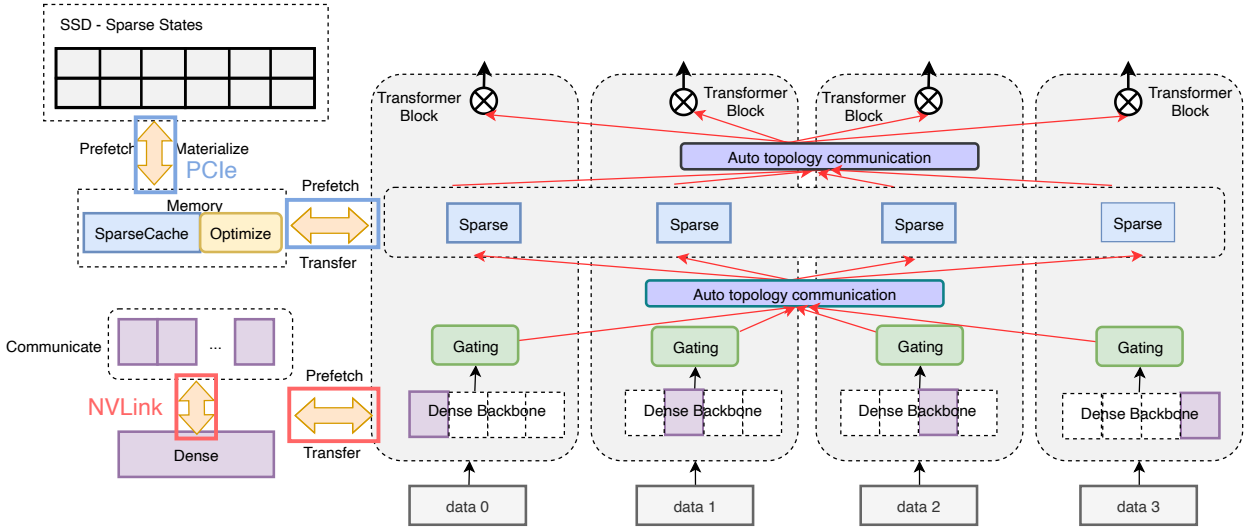


Fig. 2: Overall MoE training: This is an example of the MoE training with four devices. In accordance with the parameter state property of the MoE model, the parameter states are stored in both GPUs and SSDs. With this heterogeneous storage setup, we can effectively utilize the NVLink and PCIe bandwidth concurrently, leveraging their capabilities in two dimensions.

a significant amount of storage space compared to dense parameters, have their parameter states stored on the SSD-Node and are transferred to the GPU-Node when required for calculations. By strategically allocating the corresponding parameter states to hierarchical storage based on the computational and storage characteristics of parameters, the storage capacity of devices can be maximally utilized.

In light of the constraints posed by storage nodes, this work introduces a set of theoretical formulas to articulate the correlation between various storage devices and the storage requirements of parameter states when utilizing the ADAM optimizer [48]. Typically, each storage device is configured with eight GPUs. We denote the aggregate count of dense and sparse parameters as D and S respectively, and L as the total number of MoE layers. The capacities of SSD memory, CPU memory, and GPU memory in a single device are represented by M_{SSD}, M_{CPU}, and M_{GPU}, in that order. Moreover, N signifies the quantity of devices. We also introduce a variable, α, to quantify the likelihood of activation of sparse parameters during training, with α ranging between 0 and 1.

For the GPU-Node, it stores the dense parameter states used in forward propagation (FWD), backward propagation (BWD), and parameter updating. This includes parameters such as param fp16, grad fp16, master param fp32, momentum fp32, variance fp32, with a total size of $2D + 2D + 4D + 4D + 4D = 16D$ bytes. Furthermore, it accommodates sparse parameters and their corresponding gradients, with a size of $4\alpha S/L$ bytes, accounting for the selective activation of sparse parameters. The CPU-Node serves as a cache to hold high-frequency sparse parameter states, occupying $16\alpha S$ bytes. Lastly, the SSD-Node stores all sparse parameter states on the device, including master param fp32, momentum fp32, and variance fp32, with a size of $12S$ bytes.

$$\begin{aligned} \textbf{GPU-Node}: &\quad 16D + 4\alpha S/L \leq M_{GPU} \cdot N \\ \textbf{CPU-Node}: &\quad 16\alpha S \leq M_{CPU} \cdot N \\ \textbf{SSD-Node}: &\quad 12S \leq M_{SSD} \cdot N \end{aligned} \tag{1}$$

The scale of the entire MoE model:

$$P = S + D \tag{2}$$

The storage mechanism for sparse parameters typically involves saving them on SSDs. Nonetheless, SSDs encounter limitations due to their flash media, limited PCIe bandwidth, and constraints of the NVMe protocol. These factors contribute to increased latency and a restricted number of erasures, posing challenges in MoE training scenarios that require frequent write operations. To address these challenges, we turn our focus to Intel Optane Persistent Memory (Optane PMem) [49], an innovative storage medium that merges the benefits of byte-level addressing, similar to DRAM, with the long-term storage ability of SSDs. Optane PMem is connected to the CPU's integrated memory controller (IMC) via the DIMM (Dual Inline Memory Module) interface and communicates using DDR-T, a protocol developed for DDR4's electrical/mechanical interface. This configuration allows for byte-level addressing through CPU commands, enhancing bandwidth and decreasing latency. Significantly, Optane PMem functions in two modes: memory mode and AppDirect mode. For our specific requirement of storing parameter files on Optane PMem, we choose the AppDirect mode and set the namespace to FSDAX. By exploiting the features of Ext4, direct load and store operations are possible, circumventing both the CPU's page cache and the kernel, which facilitates seamless data transfer free from interruptions or context switches.

3.2.2 2D Prefetch Scheduling

The implementation of hierarchical storage for the preservation of both sparse and dense parameter states in MoE training introduces considerable time overhead due to the necessity of transferring these states across various devices.

To mitigate this, a 2D prefetch scheduling strategy is proposed, allowing for the simultaneous processing of dense and sparse schedules during MoE training. This strategy facilitates the concurrent computation of parameters with the scheduling procedure.

In greater detail, this strategy, particularly when applied to the dense parameter subset as defined by the ZeRO-3 strategy, enables prefetching of the entire dense parameter set post inter-rank communication along the horizontal axis, utilizing the rapid transfer speeds of NVLink. This approach is instrumental in achieving data parallelism, as demonstrated in Algorithm 1. In this methodology, prefetching occurs alongside the computation and communication processes of the current layer. To be more specific, while the i^{th} layer undergoes computation and communication, prefetch scheduling for the $(i+1)^{th}$ layer's parameters is conducted in parallel. This simultaneous prefetching approach guarantees the readiness of parameters for the subsequent layer when required, significantly reducing idle times and boosting overall computational efficiency.

Algorithm 1: Scheduling on Dense Parameters

$d_i^{'}$: Dense parameter state slices in i^{th} layer
d_i: total dense parameters in i^{th} layer

Function `DenseSchedule(`i`)`**:**
  Get dense parameters in i^{th} layer d_{slice}
  $d = AllGather(d_i^{'})$
End Function

In a similar vein, the prefetching of sparse parameters takes place through the PCIe bandwidth in the vertical dimension of the device. Given that sparse parameters are stored in SSDs, we mitigate access to SSDs for sparse parameter states by implementing a cache mechanism in the CPU memory, akin to the LFU (Least Frequently Used) mechanism [50]. CPU caches are responsible for storing selectively activated sparse parameter states used in FWD/BWD calculations and parameter updates. When a prefetch request is received, it is prioritized to retrieve the requested sparse parameters from the CPU caches. If these parameters are not found in the CPU caches, they are subsequently retrieved from the SSDs. Moreover, when the CPU caches become full or when the sparse parameter update cycle period is reached, the sparse parameter states from the CPU caches are used to update the corresponding parameter states on the SSDs.

As the CPU memory on each machine only caches frequently activated sparse parameters, we only need to prefetch the parameters of one or more expert layers, which are cached in the CPU memory, to the corresponding GPU memory in advance. By prefetching parameters in advance, the waiting time for computation can be significantly reduced. From a global perspective, by utilizing the bandwidth of NVLink and PCIe in two dimensions, we can simultaneously prefetch dense and sparse parameters, effectively reducing the scheduling gap caused by heterogeneous storage and greatly enhancing training efficiency. In the following sections, we present a detailed explanation of the CPU cache mechanism, as depicted in Algorithm 2. Additionally, we maintain historical hit information for each sparse parameter,

Algorithm 2: Scheduling on Sparse Parameters

Parameters:
p_s: sparse parameter states in i^{th} layer
$caches_{cpu}$: CPU caches
CPU_{size}: the maximum capacity of the CPU caches to store sparse parameter states
$hits$: the frequency of hits for a specific sparse parameter in the hash table
$threshold$: hit threshold
β: attenuation coefficient
K: the step size of moving average
$steps = 0$: cycle steps
$acc_{caches} = 0$: cumulative caches

Function `SparseSchedule(`i`)`**:**
  if p_s *in* $caches_{cpu}$ **then**
    Get p_s from $caches_{cpu}$
    $hits[p_s]$ += 1
  else if acc_{caches} + 1 < CPU_{size} **then**
    $hits[p_s] = 1$
    acc_{caches} += 1
    Fetch p_s from SSDs to $caches_{cpu}$
  else
    foreach p_a *in* $hits$ **do**
      $hit_a = hits[p_a]$
      if $hit_a \geq threshold$ *and* $\min(hits.values()) == hit_a$ **then**
        Update the states of p_a on SSDs
        Delete the states of p_a in $caches_{cpu}$
        Delete $hits[p_a]$
        Fetch p_s from SSDs to $caches_{cpu}$
  $steps$ += 1
  if $steps == K$ **then**
    $hits \cdot \beta$ ▷ moving average
    $steps = 0$
  $p_s \longrightarrow GPU$ ▷ transfer p_s to the corresponding GPU
End Function

which is recorded in a hash table referred to as $hits$. Specifically, if a parameter p_s is requested and has been used in the previous FWD, we increment its count in the $hits$ table. When the CPU caches have reached their maximum capacity, we update the sparse parameter states with the lowest hit frequency that surpasses the hit threshold.

In the MoE model training, each node determines whether to activate its experts in the next iteration based on the recorded expert selection results and the maintained experts' information. If activation is needed, further decisions are made based on the historical hit information recorded in a hash table to determine whether to send prefetch requests. Firstly, to avoid introducing additional CPU operations before sending prefetch requests, it is essential to place the hash table that records historical hit information on the GPU Node. Since each node only stores a portion of the sparse parameters in the SSD (not the full set), it is only necessary to maintain historical hit information for the corresponding sparse parameters. This approach distributes the GPU space cost across all computing nodes, making it negligible. Secondly, the process of selecting experts by the

Gate network inherently requires All-to-All communication to synchronize the selection results across each node in the Expert Parallelism Group. The prefetch scheduling simply reuses the results of this All-to-All communication, so no additional communication operations are introduced. Additionally, the time complexity of a hash table is O(1), meaning each prefetch operation involving searches, insertions, or deletions can be completed in constant time, thus not introducing additional computational costs.

The distinct and non-interfering characteristics of dense and sparse parameters in the model facilitate the simultaneous implementation of prefetch strategies. This approach optimally leverages the bandwidth capacities of both NVLink and PCIe. While the GPU is engaged in prefetching the parameter state for the upcoming layer, it can also simultaneously execute computations for the current layer. This dual-operation mode efficiently combines the tasks of computation and parameter readiness.

3.3 Inference Phase

Numerous studies [27], [43] have demonstrated that Mixture-of-Experts (MoE) models exhibit significantly higher training efficiency compared to dense models. However, during inference, the presence of numerous parameters, many of which are ineffective, poses a challenge of increased storage requirements compared to dense models. Knowledge distillation [25], [51]–[53] has emerged as a popular approach for reducing model size while preserving accuracy. In this context, DeepSpeed [39] has proposed the Mixture-of-Students (MoS) architecture to enhance the accuracy of the student models. Specifically, to achieve low latency and high throughput at a large scale for MoE models, various parallelism techniques have been devised [39], including expert-slicing, expert parallelism, tensor-slicing, and others. However, the inference of MoE models at an unprecedented scale often neglects the consideration of multiple storage devices when the number of machines is limited.

In the following subsections, we present the approach adopted by **MoESys** to achieve high efficiency throughout the training and inference deployment. We optimize the graph training process and propose innovations in the MoE inference architecture based on ring memory. This architecture addresses the memory wall challenge and ensures optimal performance to the greatest extent possible.

3.3.1 Graph Optimization

The training phase of **MoESys** incorporates dynamic graph training, which offers significant advantages in terms of debugging and flexibility. In contrast, for enhanced stability and efficiency, the inference and deployment stages utilize a static graph. Figure 3 illustrates the overall process of inference, which comprises six key steps:

- Graph Fusion – The original graph is merged with the corresponding distributed strategy to accommodate ultra-large-scale distributed training. This step involves eliminating parameter redundancy.
- Distillation and Compression – The numerous experts in the teacher network are compressed through distillation and compression techniques, resulting in a student network with fewer experts.

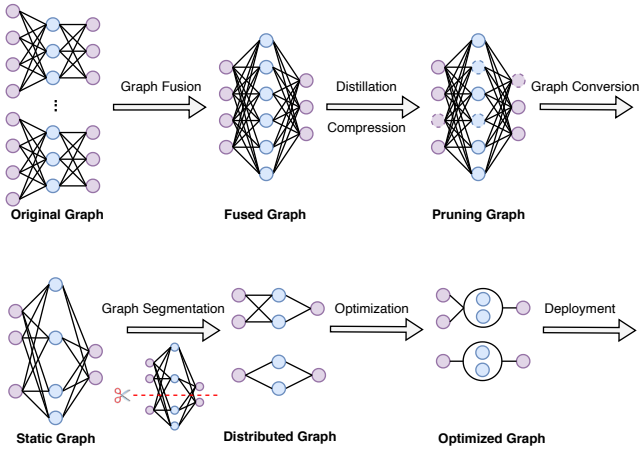


Fig. 3: Inference Pipeline in MoE.

- Graph Conversion – The dynamic graph is converted into a static graph to enable subsequent optimization and deployment processes. Due to space limit, we introduce the detailed strategy of conversion in the external link[1].
- Graph Segmentation – Based on available inference resources and specific requirements, a rational distributed strategy is chosen either manually or automatically to partition the static graph into multiple distributed sub-graphs. Additional communication is added as needed.
- Optimization – Pertinent Intermediate Representation (IR) Pass optimizations, such as kernel fusion, are applied to the distributed sub-graphs to further improve inference performance.
- Deployment – The optimized sub-graphs are deployed on servers to provide efficient and reliable services.

It is important to note that **MoESys** combines highly optimized transformers and MoE-related kernels. We leverage optimized methods, such as Fused Multi-head Attention, which have been successfully employed in NVIDIA's BERT implementation for MLPerf 1.1 [54]. These optimizations effectively reduce kernel launch time. For the MoE model, we have developed unique kernels to improve H2D/D2H (Host-to-Device/Device-to-Host) transfer time by utilizing CUDA Pinned Memory and customizing AlltoAll communication. Our aim is to minimize the number of layer transitions as much as possible. The details of these optimizations and their impact on the performance of **MoESys** are presented and discussed in Section 5.4.

3.3.2 Ring Memory Offloading

In order to facilitate the inference of large-scale MoE models with limited resources, it is essential to employ an offloading strategy to address the storage challenge. However, the speed of data movement often becomes a limiting factor for inference performance. Consequently, numerous methods aim to conceal the impact of data movement by maximizing the overlap between data movement and inference calculations, thereby reducing the waiting time for calculations. In this work, we propose a dynamic scheduling strategy for offloading sparse parameters, specifically expert

1. https://www.paddlepaddle.org.cn/documentation/docs/en/guides/jit/basic_usage_en.html

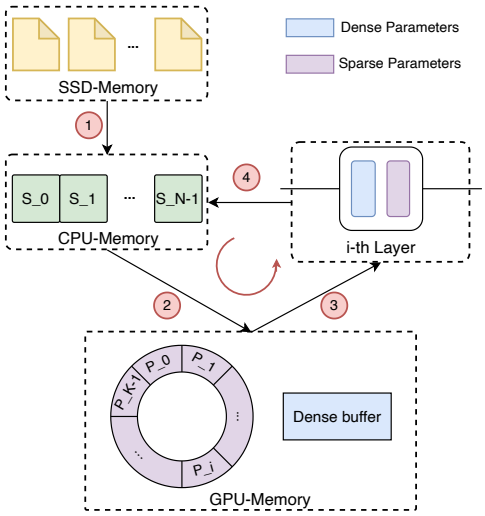


Fig. 5: The scheduling and timeline of the ring memory offloading process can be summarized into the following essential steps: ① Load N copies of parameters from files in SSD memory, ② Load K copies of parameters from CPU memory, ③ Execute the computation for the i-th layer, ④ Release the i-th parameter and trigger asynchronous copy process to replace P_i with S_{K+i}.

parameters in the MoE model. The objective is to maintain efficient performance by concurrently moving parameters from CPU memory while performing inference computations in GPU memory. By overlapping these operations, we aim to minimize the overall latency and enhance the efficiency of the inference process.

The structure of the MoE model during its inference stage, illustrated in Figure 4, demonstrates a layer-specific independence of parameters, reminiscent of the switch transformer architecture [25]. This design feature enables the staggering of computation and offloading tasks, thereby facilitating their concurrent execution. Considering an MoE inference model comprising N decoder layers, each layer's expert parameters are replicated N times and stored on the CPU device. Concurrently, other parameters, such as embeddings, are maintained within the dense buffer of the GPU device. In addition, K replicas of the expert parameters are also cached within the GPU device.

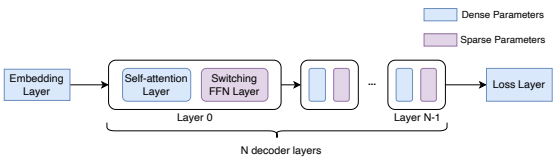


Fig. 4: Switching Layers in MoE Inference Model.

As depicted in Figure 5, upon completion of the computation pertaining to the i-th layer, the corresponding parameter P_i in the GPU memory can be released. Concurrently, the S_{K+i} expert parameter of the $(K+i)$-th layer can be asynchronously loaded from the CPU memory to occupy the previously utilized space by P_i. This procedure, referred to as calculation-released-load, facilitates the maintenance of a fixed number of K expert parameter duplicates on the GPU device. These duplicates are stored in the ring memory, thereby mitigating memory fragmentation. By leveraging distinct CUDA streams, the expert loading from the CPU and the computation process can be partially overlapped. Moreover, by ensuring a substantial ring memory size and incorporating a greater number of decoder layers in the MoE inference model, the level of overlap can be significantly optimized. For an evaluation of the inference performance using the ring memory approach, please consult Section 5.4.

4 Implementation Strategies

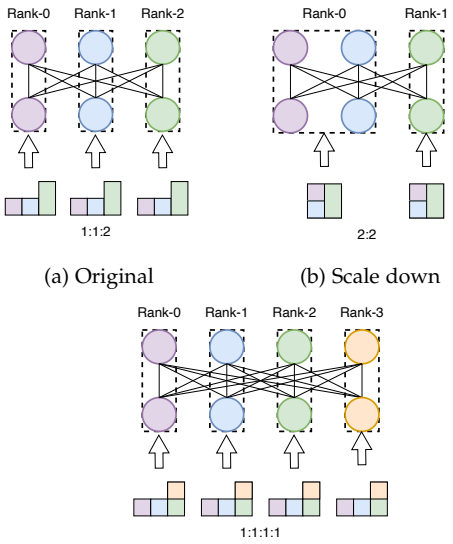
The distinctive architecture of the MoE model gives rise to inherent challenges in both the training and inference processes. In order to tackle the issue of load imbalance caused by uneven input data, we have devised the Elastic MoE Training approach. Furthermore, recognizing the significant involvement of cross-machine communication in MoE, we have delved into Resource-aware Communication techniques to enhance efficiency across diverse clusters. Lastly, to overcome storage limitations stemming from the use of oversized vocabularies in various tasks, we have developed and implemented a novel embedding partition method within the framework of data parallelism, distinct from the approach employed in tensor-slicing parallelism.

4.1 Elastic MoE Training

Load imbalance significantly influences training efficiency, especially in multi-task training scenarios employing the MoE framework. A prominent instance of this is seen in the UFO task, where the differential input data volumes across various tasks lead to unequal computation durations, thus exacerbating load imbalances. This discrepancy manifests in two primary forms: one, the breaching of memory capacity limits due to the processing of disproportionately large batch sizes by individual task nodes, a consequence of aggregating data from other nodes; and two, the retardation of synchronous communication caused by the lag of the slowest node. This phenomenon, known as the "Cask Effect" [55], results in reduced computational efficiency.

To tackle these challenges, we implement the elastic MoE training method, which dynamically adjusts the number of training nodes to ensure load balance based on the estimated workload of each task. In practical terms, for lighter tasks, combining multiple nodes proves more resource-efficient, as long as storage capacity is not a limiting factor (see Figure 6b). Conversely, for heavier tasks, we introduce additional nodes to distribute the workload across more computing resources. Simultaneously, we partition the input data of heavy-duty tasks to achieve load balance and employ data parallelism to ensure parameter synchronization (see Figure 6c). These elastic training approaches effectively mitigate performance degradation resulting from load imbalance. You can find detailed performance comparisons in Section 5.3.

In elastic training, load imbalance often leads to situations where some devices are idle, waiting for others to complete their computations, creating what is known as "bubbles." These bubbles not only reduce computational efficiency but also increase training costs. To address this, we have introduced a method of scaling up or down the computational devices dynamically. This method aims to reduce waiting



(a) Original (b) Scale down

(c) Scale up

Fig. 6: Different methods supported by elastic MoE training: (a) the original training with load imbalance, in which the ratio of each node data quantity is 1:1:2; (b) combining multiple nodes with light-duty tasks, in which the ratio of each node data quantity is 2:2; (c) adding extra nodes to handle heavy-duty tasks, in which the ratio of each node data quantity is 1:1:1:1.

times and enhance the FLOPS (floating-point operations per second) utilization rate per device, thereby boosting the throughput (tokens/s/card) of each compute device. The decision to scale up or down should be based on specific training requirements and cost considerations:

- Upscaling: When there is a need to increase the overall end-to-end throughput, we typically scale up by adding more compute devices. This reduces the total training time for the model.
- Downscaling: In situations where resources are constrained, and cost control is crucial, we opt for downscaling by reducing the number of compute devices, thereby lowering the cost of training the model.

Irrespective of the scaling direction, both methods effectively enhance the FLOPS utilization rate, reduce idle waiting times for compute devices, and decrease the overall "bubble" time during training.

4.2 Resource-Aware Communication

In the process of training and inference of the MoE model, a significant volume of AlltoAll communication is necessitated between devices in the context of expert parallelism. This communication procedure has the potential to become a performance bottleneck, as multiple instances of AlltoAll communication may contend for the finite network resources concurrently. Upon analysis of network topologies in typical clusters, it is observed that data interaction across clusters exhibits relatively slower transmission speeds compared to interactions within a single cluster. This disparity arises from factors such as congested message pathways and higher traffic costs associated with inter-cluster communication.

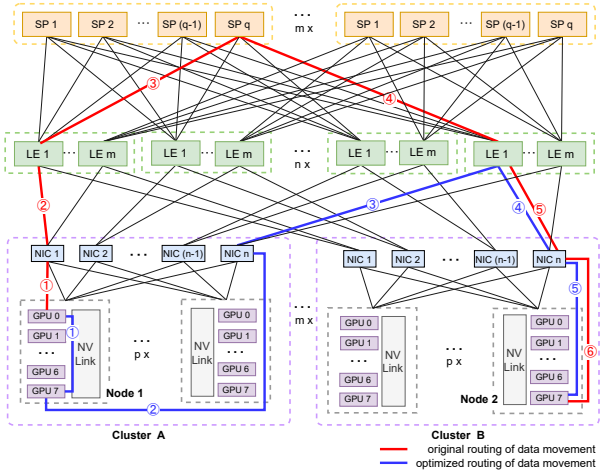


Fig. 7: Network topology and message pathways for data movement

By utilizing NVLink, intra-node communication incurs minimal time and resource overhead as it avoids traversing any Network Interface Cards (NICs) or switches. However, inter-node communication within a cluster or across clusters necessitates traversing a congested message path that involves NICs and switches [56], resulting in increased time consumption for traffic scheduling. Consider a network consisting of m clusters and p nodes that share a common set of NICs within each cluster. The leaf switches (LE) and spin switches (SP) are organized into n and m groups, respectively. As depicted in Figure 7, the leaf switches of the i-th group establish direct connections only to NICs with a rank of i from different clusters. The spin switches facilitate communication between leaf switches. It is important to note that the bandwidth of the spin switch is lower than that of the leaf switch. Hence, it is preferable to maximize the utilization of the leaf switch for data exchange, aiming for improved performance. For instance, let us consider a scenario where all GPU0s are connected to NIC1 and all GPU7s are connected to NICn. We observe that data movement between GPU0 of Node1 in Cluster A and GPU7 of Node2 in Cluster B traverses the switch routing path $[LE1, SPq, LE1]$ as indicated by the red lines. This incurs higher communication costs and the potential for resource contention with other interactions. An alternative approach involves a two-step process: first, transferring data from GPU0 to GPU7 within Node1 using NVLink, and then performing cross-cluster communication between the corresponding pair of NICs with rank 7, without crossing any switches except $LE1$. This is depicted by the blue lines. Such an approach enables the optimal utilization of NVSwitch bandwidth and enhances network traffic optimization.

Hence, the speed at which data is exchanged between GPUs of the same rank within a node surpasses that of GPUs with different ranks within the same node. To leverage the network topology's characteristics, we propose an optimized approach for Hierarchical AlltoAll communication that takes into account the available resources during both training and inference. As depicted in Figure 8, to avoid cross-node communication involving GPUs of different ranks, we initially utilize intra-node AlltoAll communication through

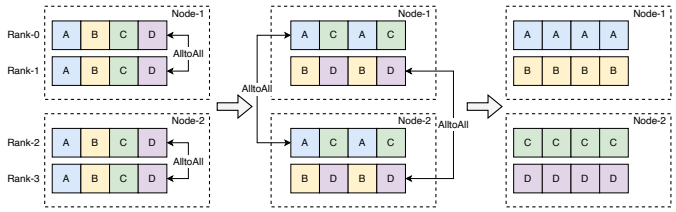


Fig. 8: Hierarchical AlltoAll

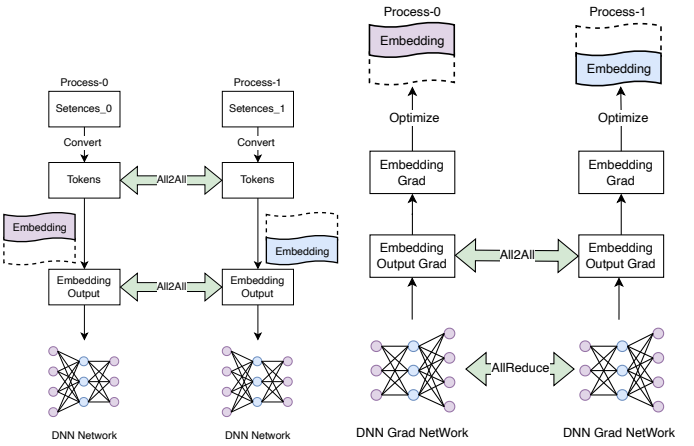
NVSwitch connections to collect the data. Afterward, we group GPUs with identical ranks for inter-node AlltoAll communication, allowing communication across machines without incurring unnecessary costs related to crossing different channels.

Furthermore, this approach enhances peer-to-peer communication between nodes by a factor of p, where p denotes the number of GPUs within a single node. This increase in inter-node communication capacity enables the optimal utilization of inter-node bandwidth. In contrast to DeepSpeed's AlltoAll design [39], which is primarily aimed at addressing the issue of small per-port communication volumes in all-to-all communication through a layered approach for tensor fusion to facilitate larger packet communication, our approach is specifically tailored to the network topology of our experimental cluster. We concentrate on maximizing the utilization of NVLink connections and mitigating network congestion. Therefore, our Hierarchical AlltoAll structure is a direct response to our specific network topology and would adapt if the cluster's topology were to change. This distinction underscores the customization of our method to our particular hardware and network architecture, which differs from the more generalized approach adopted by DeepSpeed.

4.3 Embedding Partition in Data Parallelism

In the context of ultra-large-scale model training, the embedding table often constitutes the largest parameter set within the entire model, thereby necessitating restrictions on its storage due to the model's scale. Numerous studies have focused on researching embedding partitioning techniques. For instance, Megatron [40] has successfully employed column-wise partitioning of the embedding table in tensor-slicing parallelism to reduce training memory requirements. Additionally, EmbRace [57] has proposed a column-wise partitioning approach within the embedding table to achieve more balanced communication. However, an efficient processing method for handling embedding partitioning in scenarios where the input data for each process is inconsistent remains elusive. In such cases, ensuring efficient processing becomes challenging due to the varying nature of the inputs across different devices.

In this study, our primary focus is on addressing the challenge of embedding partitioning within the context of data parallelism, as illustrated in Figure 9. To elaborate, when we consider an embedding table with dimensions $[V, H]$ being distributed among N training processes, we employ a column-wise partitioning scheme. This scheme assigns a $[V, \frac{H}{N}]$ shard to each worker. Consequently, each training process possesses an embedding representation that



(a) Forward stage (b) Backward and optimization stage

Fig. 9: Example data flow of Embedding Partition in data parallelism. The embedding table is column-wise partitioned among processes. In the forward stage, AlltoAll communication is called twice: one is for exchanging input data and another is for exchanging embedding lookup results. In the backward stage, AlltoAll is called once to swap the gradient of the embedding table and using the gradients updates the embedding table.

pertains only to a subset of the vocabulary. As a result, before querying the embedding table, the input data of each process must be exchanged with other processes through AlltoAll communication. This exchange allows the acquisition of embedding results corresponding to the local partial vocabulary. Subsequently, to obtain accurate results for the input data processed by each worker, the embedding results are exchanged once more through AlltoAll communication, effectively serving as the inverse procedure of the previous communication step. It is essential to note that during the backward stage, the gradient information also needs to be exchanged to recover the embedding table gradient.

In contrast to the traditional embedding sharding approach [58], which is similar to tensor model parallelism and partitions along the vocabulary dimension, our method is specifically designed for data parallelism. Given that each GPU card processes different data, sharding along the vocabulary dimension is not feasible. Instead, we employ sharding along the hidden_size dimension of the vocabulary, ensuring that each computing device can access the complete vocabulary. Additionally, we utilize AlltoAll communication to complete the hidden layer, accommodating the varying data inputs across devices.

Significantly, this approach effectively reduces the storage requirements of the embedding table within the data parallelism framework. It achieves this by introducing only three instances of AlltoAll communication and eliminating the need for AllReduce synchronization for embedding table gradients in data parallelism.

5 Experiment

In this section, we present a comprehensive experimental evaluation of MoE models using the proposed **MoESys**

system. Our evaluation focuses on both training and inference aspects of the MoE models. In the training phase, we assess the efficiency of the MoE-based GPT model across different configurations. For the inference phase, we analyze the performance of the ring memory-based offloading strategy, considering various model sizes. Moreover, we investigate several efficient methods implemented in the **MoESys** system, including the widely-used UFO model. It is important to note that our results reporting disregards the performance of the individual models themselves. This is because the converged models, whether based on baseline MoEs or utilizing **MoESys**, achieve comparable performance levels. Hence, our evaluation primarily focuses on the efficacy and efficiency of the proposed **MoESys** system.

5.1 Platform

MoESys is implemented based on the PaddleFleetX [59] architecture of PaddlePaddle. After some fundamental performance optimizations, PaddleFleetX demonstrates certain performance advantages over other standard models. For a more direct comparison, we would like to highlight the performance comparison between Paddle and Megatron as of March 2023. This comparison provides valuable insights into how PaddleFleetX, and by extension MoESys, stands in relation to other prominent frameworks in terms of efficiency and effectiveness. Please refer to Table 1 for detailed comparative data. The experimental data demonstrates PaddleFleetX's enhanced performance over Megatron-LM across several model configurations, with a notable throughput superiority that ranges from a substantial 14.2% for smaller models (0.35 billion parameters) to a marginal 0.4% for the extensive 175 billion parameter models. This improved throughput is consistent despite marginal differences in memory usage where Megatron-LM occasionally leads, particularly with smaller models. PaddleFleetX also showcases a more efficient utilization of GPU resources, as evidenced by higher TFLOPS/s per GPU and a closer approach to the theoretical peak FLOP/s utilization across varying model sizes.

5.2 Large-Scale MoE Training

We conducted training experiments using GPT models based on the MoE architecture, leveraging A100 GPUs (80 GB), and employing a combination of data parallelism and expert parallelism. In the MoE system, there are two main parts: the Dense parameters (Backbone) and the Sparse parameters (Expert). The Dense part employs data parallelism, meaning that different input data is processed in parallel. After the backward computation is complete, the Dense parameters are synchronized through Allreduce communication. On the other hand, the Sparse parameter part involves expert parallelism. Here, routing communication between experts is used to send the required data to the designated compute devices. This is accomplished through AlltoAll communication. The backward pass also utilizes AlltoAll communication for gradient synchronization. This approach allows us to efficiently leverage both data and expert parallelism, optimizing the performance of the MoE system. The evaluation of these models was performed using Gshard [60] and top1-gating metrics. Specifically, we utilized pure fp16 precision and the AdamW [61] optimizer during the training process.

Compared to the high precision FP32 method, there are two lower precision training approaches: "AMP" (Automatic Mixed Precision) and "pure fp16". Unlike AMP, pure fp16 is a commonly used, faster training method. This method is well-established and has been applied in model training, for example in Megatron and DeepSpeed. Our approach is consistent with the methods used in Megatron/DeepSpeed. The term "pure" in "pure fp16" is in contrast to AMP, meaning that all model parameters in pure fp16 training are of fp16 type. However, it's important to note that not all operations are computed in fp16. Certain operations, like softmax, use fp32 for computation. We also employ techniques like MasterWeight to mitigate the impact of lower precision training on model accuracy. Our use of pure fp16 in the experiments is a standard practice, and we maintain this strategy consistently when comparing with other frameworks.

Table 2 presents the throughput results for different configurations. The table lists the parameter sizes (*Parameters (B)* in billions) in ascending order from top to bottom, while keeping the number of attention heads (*Attention heads*), hidden layer size (*Hidden size*), vocabulary size (*Vocab size*), and number of layers (*Layers*) constant. Consequently, the number of experts (*Experts*), GPUs (*GPUs*), and batch size (*Batch size*) increase twofold. Both DeepSpeed and our proposed **MoESys** exhibit training speeds that double accordingly. It is worth noting that the first line of the table represents the performance on a single node equipped with eight GPUs, while the subsequent lines depict results for multi-node scenarios.

Our observations indicate that, compared to the state-of-the-art MoE system, DeepSpeed [2], **MoESys** achieves approximately a 28% speedup in single-node training and at least a 33% speedup in multi-node training for MoE models with over 100 billion parameters. Furthermore, **MoESys** reduces the GPU memory usage of each rank by nearly 12 GB. Therefore, in large-scale MoE training, our proposed **MoESys** system demonstrates comparable training speeds while consuming relatively less memory compared to the benchmark model, DeepSpeed.

5.3 Ablation Study in MoE Training

The experimental evaluation in this section highlights the benefits of efficient ***implementation strategies*** on a large-scale model, as discussed in Section 4. Each of these strategies is evaluated independently and compared against traditional/baseline methods.

5.3.1 Elastic MoE Training

In order to assess the efficiency of elastic MoE training, we conducted experiments using the UFO [2] model, which is based on the MoE architecture and trained on A100 GPUs with 80 GB of memory. We designed four tasks with batch sizes of 512, 256, 128, and 128, respectively, to simulate an imbalanced training process.

Following the elastic sparse training methodology outlined in Section 4.1, we adjusted the overall training workload by adding additional computing nodes. Specifically,

2. https://github.com/microsoft/Megatron-DeepSpeed

TABLE 1: The standard model's performance on the established baselines.

Parameters (Billions)	Hidden size	Layers	Attention heads	GPUs	Data parallel	Group sharded parallel	Tensor parallel	Pipeline parallel	Batch size	PaddleFleetX (vs Megatron-LM)				Difference of throughput
										Throughput (tokens/s)	Memory usage (MB)	TFLOPS/s per GPU	Theoratical peak FLOP/s (%)	
0.35	1024	24	16	8	8	1	1	1	64	291,585 (255,401)	31,919 (33,217)	102 (89)	32.7% (28.5%)	+14.2%
1.3	2048	24	16	8	8	1	1	1	64	115,682 (109,537)	39,775 (39,537)	150 (142)	48.1% (45.5%)	+5.6%
6.7	4096	32	32	16	1	16	1	1	128	44,605 (39,936)	35,442 (35,403)	149 (116)	47.8% (43.0%)	+11.7%
175	12288	96	96	128	1	1	8	16	1536	14,634 (14,571)	34,912 (34,708)	160 (159)	51.3% (50.9%)	+0.4%

TABLE 2: Results for large-scale MoE training on GPT models with different configurations.

Parameters(B)	Attention heads	Hidden size	Vocab size	Layers	Experts	GPUs	Batch size	Speed(tokens/s)		Memory(GB)	
								DeepSpeed	MoESys	DeepSpeed	MoESys
13.9	64	4096	50304	12	8	8	8	24165	**31085**	68.9	**56.8**
26.8					16	16	16	43691	**59136**	66.2	**53.9**
52.6					32	32	32	82957	**113456**	66.8	**54.5**
104.1					64	64	64	157728	**209970**	66.3	**54.4**
207.2					128	128	128	283706	**376968**	66.4	**54.3**

TABLE 3: Results for elastic MoE training with multiple tasks.

	Task number	Parameters(M)	Total batch size	Batch size per task	GPUs	GPUs per task	Total Speed (samples/s)	Speed per GPU (samples/s)
Load imbalanced	4	83	1024	512/256/128/128	4	1/1/1/1	250.4	62.6
Load balanced					8	4/2/1/1	591.9	74.0

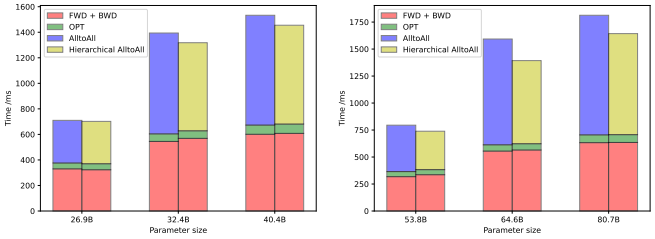
we allocated 4 GPUs for Task-1 and 2 GPUs for Task-2. To ensure fairness, we calculated the average speed of each GPU to mitigate the impact of increasing the number of nodes. The results, as presented in Table 3, indicate that compared to the *Load imbalanced* configuration, the *Load balanced* configuration achieved an approximate 18.2% improvement in throughput per GPU. It is important to note that the *Load balanced* approach was derived from our designed elastic MoE training, and the results highlight its effectiveness and efficiency, respectively.

Furthermore, we applied a task-based MoE load balancing mechanism to the training of the billion-scale visual model VIMER-UFO 2.0. This approach supports dynamic expansion of task numbers and parallel training of multiple tasks and multiple experts. Under the same experimental environment (32x A100 80GB GPUs), we achieved a training performance of 697 images per second. This represents a significant improvement in throughput by 64% compared to 425 images per second using the Pytorch v1.10 framework. Additionally, the memory footprint was reduced to 45 GB per GPU, a decrease of 18%.

5.3.2 Resource-Aware Communication

In this subsection, we conducted training of MoE models on different numbers of nodes and with varying model sizes to demonstrate the potential benefits of the **MoESys** designs. The results are presented using a stacked bar chart, which illustrates the time consumption for key components of the training process: 1) the forward stage (*FWD*), 2) the backward stage (*BWD*), 3) the optimization stage (*OPT*), and 4) the communication stage. Specifically, we compare the proposed *Hierarchical AlltoAll* approach to the baseline *AlltoAll* method in the communication stage, while keeping other components constant across the different settings.

As depicted in Figure 10, the first three components (green and pink bars) show similar performance between the AlltoAll baseline and Hierarchical AlltoAll across all parameter sizes. The performance gaps are primarily caused



(a) 2 nodes with 16 GPUs (b) 4 nodes with 32 GPUs

Fig. 10: MoE Training Time Breakdown

by the two top bars (purple for AlltoAll baseline and yellow for Hierarchical AlltoAll) representing the communication stage. It is evident that with the adoption of Hierarchical AlltoAll in **MoESys**, the computation time does not increase significantly, while the communication time decreases dramatically. Furthermore, as the model parameter size increases, the efficiency gap in communication becomes more significant between AlltoAll and hierarchical AlltoAll. This indicates that the proposed **MoESys** can amplify the performance improvement in communication when dealing with large-scale models. This effect is evident by observing the gap between the purple and yellow bars from left to right along the horizontal axis.

For the most substantial improvement observed in the experiment, which involved a MoE model with 80.7 billion parameters across four nodes with 32 GPUs, the overall end-to-end training performance improved by 10.3%. Additionally, the communication stage achieved a 15.5% speedup using the Hierarchical AlltoAll strategy.

5.3.3 Embedding Partition in Data Parallelism

To assess the performance of embedding partition in data parallelism, we conducted training of a MoE model on a dataset with an extremely large vocabulary. The ablation study included a baseline approach using the non-segment

TABLE 4: Performance of the Embedding Partition in Data Parallelism.

Batch size	GPUs	Experts	Vocab size	Hidden size	Parameter(M)	Memory (GB)		Speed (tokens/s)	
						Baseline	Embedding Partition	Baseline	Embedding Partition
8	8	**4**	50304	2048	72	4.68	**2.43**	289452	**300451**
				4096	180	7.51	**4.67**	152144	**167352**
				8192	400	15.81	**8.63**	80421	**91687**
8	8	**8**	50304	2048	100	7.46	**5.78**	144159	**150161**
				4096	300	12.80	**9.70**	86237	**95890**
				8192	700	27.80	**20.49**	40605	**46938**
8	8	**16**	50304	2048	227	18.55	**15.73**	53428	**55725**
				4096	781	31.41	**27.98**	26321	**29047**
				8192	1320	63.25	**55.75**	12065	**13902**

TABLE 5: Performance comparison among the proposed strategies.

Batch size	Vocab size	Layers	Hidden size	Experts	Peak Memory (GB)				Speed per GPU (tokens/s)			
					Baseline	Elastic Training	Resource-Aware Communication	Embedding Partition	Baseline	Elastic Training	Resource-Aware Communication	Embedding Partition
8	50304	12	2048	8	28	23	28	25	30604	35194	33664	31033

embedding strategy, while the remaining settings remained consistent with **MoESys**. The results in Table 4 demonstrate that the embedding partition strategy within a single machine effectively reduces GPU memory consumption when processing vocabularies of large sizes. The study maintains a constant batch size and number of GPUs but varies the number of experts, hidden size, and parameter magnitude. Embedding partitioning consistently exhibits significant reductions in memory usage (e.g., a drop from 15.81 GB to 8.63 GB with 4 experts and hidden size 8192) and enhancements in processing speed (as seen with the increase from 80421 to 91687 tokens/s for the same configuration). These improvements are maintained across various model complexities, indicated by varying numbers of parameters and experts, suggesting that embedding partitioning provides a scalable solution for optimizing large-scale neural networks in data-parallel scenarios.

5.3.4 Cross-wise Comparison

In order to determine the dominant strategy and its contribution to the overall performance gain in the **MoESys** system, we conducted a cross-wise comparison among the proposed strategies. While we have individually showcased the performance gains of each strategy compared to the baselines, their relative performance among each other requires further investigation. We are particularly interested in identifying the strategy that is most dominant and contributes the most to the **MoESys** system. To conduct this comparison, we performed experiments measuring the peak memory usage and average computation speed on the GPU in parallel. The results are summarized in Table 5. The baseline refers to the MoE without any of the proposed strategies, and as expected, it exhibits the highest peak memory consumption and the lowest computation speed. Among the proposed strategies, Elastic Training shows the least peak memory occupancy, while the Hierarchical AlltoAll strategy (utilized for resource-aware communication) consumes relatively more peak memory. Regarding GPU computation speed, Elastic Training again outperforms the other three strategies.

As shown in Fig. 11 directly, compared to a baseline scenario with no optimizations, our elastic training strategy shows the most significant improvements in terms of peak memory usage and GPU computation speed, contributing greatly to overall performance enhancement. Additionally, the topology-aware hierarchical AllToAll strategy and the DP-Embedding sharding strategy also contribute to a noticeable proportion of performance improvement in MoESys.

5.4 MoE Inference

The inference experiments consist of two parts: the first part evaluates the performance of the MoE inference system using different models with varying numbers of parameters (in the billions), while the second part assesses the effectiveness of the offloading strategy proposed in Section 3.3.2.

5.4.1 Effectiveness

TABLE 6: Performance of MoE inference.

Parameters(B)	GPUs	Batch size	Speed(tokens/s)	
			DeepSpeed	**MoESys**
10.0	1	1	4303	**4551**
106.5	8	8	27215	**29681**
209.6	16	16	35310	**40059**

Inference tasks typically require less memory compared to training tasks in many scenarios. Consequently, it is feasible to perform downstream tasks using a single GPU with a 10-billion-parameter MoE model. To evaluate the inference performance of large-scale MoE models, we conducted experiments on a text generation task. The results presented in Table 6 indicate that **MoESys** achieves approximately 13% faster inference speed compared to DeepSpeed for MoE models with over 200 billion parameters. This considerable performance improvement underscores the effectiveness of **MoESys** in the inference process.

5.4.2 Ring Memory Offloading

We conducted experiments to evaluate the inference performance of the expert offloading strategy using ring memory on a system equipped with 16 A100(40G) GPUs. The experiment focused on a MoE model with 32 experts and 58.2 billion parameters. We measured the time consumed for computation in GPU memory as well as the movement of experts between CPU and GPU memory. Figure 12 illustrates the results, indicating that the performance of the overlapped MoE inference system remains largely unaffected by CPU

offloading. The findings demonstrate that this strategy achieves a favorable balance between computation and data movement. Additionally, it enables the MoE inference systems to reduce GPU memory consumption by at least 30% compared to inference without ring memory offloading.

Fig. 11: Cross-wise comparison among proposed strategies.

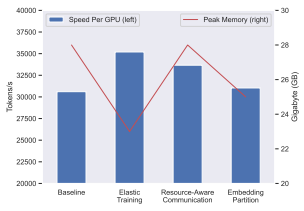


Fig. 12: Evaluation of MoE inference with and without overlapping offloading.

5.5 Summary

The aforementioned experiments comprehensively examine the efficiency and efficacy of the proposed designs within the **MoESys** systems. Particularly, when applied to large-scale deep learning models such as the GPT series, **MoESys** exhibits superior performance in terms of training and inference speed, as well as memory consumption, in comparison to the established benchmark DeepSpeed. As an industrially-relevant MoE system, **MoESys** is demonstrated to further enhance the development of distributed MoE designs in practical real-world applications.

6 Conclusion and Future Works

To address the needs of modern internet services that demand the use of large-scale DNNs, we have presented **MoESys**, a novel training and inference system based on MoE that boosts efficiency in both large-scale training and inference. Our proposed system adopts an elastic training strategy with 2D prefetch and fusion communication over hierarchical storage for efficient parallelisms. For scalable inference in a single node, **MoESys** builds CPU-GPU memory jointly into a ring of sections and executes computation tasks across memory sections in a round-robin manner. Our experiments demonstrate that **MoESys** achieves superior performance compared to the state-of-the-art DeepSpeed, outperforming DeepSpeed by 33% in training throughput and 13% in inference throughput, with 64% higher throughput and 18% lower memory footprints under MoE tasks based on large language models and foundation vision models.

Our future work focuses on developing a unified sparse training and inference system that considers parameter-server and scheduling across multiple dimensions, exploring efficient methods for sparse training within the **MoESys** framework, and enhancing our system collaboration with resource platforms for sustainable research. We also aim to implement a comprehensive evaluation framework that accurately assesses the comparative performance of diverse parallel computing strategies in a way that is fair, informative, and contributes constructively to the field of scalable machine learning architectures.

References

[1] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *SIGKDD*, 2020, pp. 3505–3506.

[2] L. Zhang, Y. Luo, Y. Bai, B. Du, and L.-Y. Duan, "Federated learning for non-iid data via unified feature learning and optimization objective alignment," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 4420–4428.

[3] F. Al-Doghman, N. Moustafa, I. Khalil, Z. Tari, and A. Zomaya, "Ai-enabled secure microservices in edge computing: opportunities and challenges," *IEEE Transactions on Services Computing*, 2022.

[4] J. Bian, J. Huang, S. Ji, Y. Liao, X. Li, Q. Wang, J. Zhoy, Y. Wang, and D. Dou, "Feynman: Federated advertising for ecosystems-oriented mobile apps recommendation," *IEEE Transactions on Service Computing*, 2023.

[5] H. Liu, Q. Gao, J. Li, X. Liao, H. Xiong, G. Chen, W. Wang, G. Yang, Z. Zha, D. Dong *et al.*, "Jizhi: A fast and cost-effective model-as-a-service system for web-scale online inference at baidu," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 3289–3298.

[6] J. Bian, H. Xiong, Z. Wang, J. Zhou, S. Ji, H. Chen, D. Zhang, and D. Dou, "Afcs: Aggregation-free spatial-temporal mobile community sensing," *IEEE Transactions on Mobile Computing*, 2022.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[8] J. Bian, J. Huang, S. Ji, Y. Liao, X. Li, Q. Wang, J. Zhou, D. Dou, Y. Wang, and H. Xiong, "Feynman: Federated learning-based advertising for ecosystems-oriented mobile apps recommendation," *IEEE Transactions on Services Computing*, 2023.

[9] Z. Cui, X. Xu, X. Fei, X. Cai, Y. Cao, W. Zhang, and J. Chen, "Personalized recommendation system based on collaborative filtering for iot scenarios," *IEEE Transactions on Services Computing*, vol. 13, no. 4, pp. 685–695, 2020.

[10] C. Zhang, J. Zhou, X. Zang, Q. Xu, L. Yin, X. He, L. Liu, H. Xiong, and D. Dou, "Chase: Commonsense-enriched advertising on search engine with explicit knowledge," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 4352–4361.

[11] L. Zou, S. Zhang, H. Cai, D. Ma, S. Cheng, S. Wang, D. Shi, Z. Cheng, and D. Yin, "Pre-trained language model based ranking in baidu search," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 4014–4022.

[12] Q. Liu, Y. Tian, J. Wu, T. Peng, and G. Wang, "Enabling verifiable and dynamic ranked search over outsourced data," *IEEE Transactions on Services Computing*, vol. 15, no. 1, pp. 69–82, 2019.

[13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[14] 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.

[15] S. Wang, Y. Sun, Y. Xiang, Z. Wu, S. Ding, W. Gong, S. Feng, J. Shang, Y. Zhao, C. Pang *et al.*, "Ernie 3.0 titan: Exploring larger-scale knowledge enhanced pre-training for language understanding and generation," *arXiv preprint arXiv:2112.12731*, 2021.

[16] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[17] S. Wang, Y. Zheng, and X. Jia, "Secgnn: Privacy-preserving graph neural network training and inference as a cloud service," *IEEE Transactions on Services Computing*, 2023.

[18] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 412–428, 2020.

[19] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[20] L. Xue, N. Constant, A. Roberts, M. Kale, R. Al-Rfou, A. Siddhant, A. Barua, and C. Raffel, "mT5: A massively multilingual pre-trained text-to-text transformer," in *Proceedings of the 2021 Conference of the*

North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Jun. 2021, pp. 483–498.

[21] R. Aharoni, M. Johnson, and O. Firat, "Massively multilingual neural machine translation," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 3874–3884.

[22] Y. Wang, C. Zhai, and H. Hassan, "Multi-task learning for multilingual neural machine translation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1022–1034.

[23] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*.

[24] 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.

[25] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *arXiv preprint arXiv:2101.03961*.

[26] A. Shah, R. Ganesan, S. Jajodia, H. Cam, and S. Hutchinson, "A novel team formation framework based on performance in a cybersecurity operations center," *IEEE Transactions on Services Computing*, 2023.

[27] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat *et al.*, "Glam: Efficient scaling of language models with mixture-of-experts," *arXiv preprint arXiv:2112.06905*.

[28] S. Zuo, X. Liu, J. Jiao, Y. J. Kim, H. Hassan, R. Zhang, T. Zhao, and J. Gao, "Taming sparsely activated transformer with stochastic experts," *arXiv preprint arXiv:2110.04260*, 2021.

[29] 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*.

[30] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, Q. Zhang, and K.-K. R. Choo, "An energy-efficient sdn controller architecture for iot networks with blockchain-based security," *IEEE Transactions on Services Computing*, vol. 13, no. 4, pp. 625–638, 2020.

[31] A. Yang, J. Lin, R. Men, C. Zhou, L. Jiang, X. Jia, A. Wang, J. Zhang, J. Wang, Y. Li *et al.*, "M6-t: Exploring sparse expert models and beyond," *arXiv preprint arXiv:2105.15082*, 2021.

[32] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE micro*, vol. 31, no. 5, pp. 7–17, 2011.

[33] X. Liu, S. X. Sun, and G. Huang, "Decentralized services computing paradigm for blockchain-based data governance: Programmability, interoperability, and intelligence," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 343–355, 2019.

[34] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "Bloomberggpt: A large language model for finance," *arXiv preprint arXiv:2303.17564*, 2023.

[35] M. Lewis, S. Bhosale, T. Dettmers, N. Goyal, and L. Zettlemoyer, "Base layers: Simplifying training of large, sparse models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6265–6274.

[36] Z. Zhang, Y. Gu, X. Han, S. Chen, C. Xiao, Z. Sun, Y. Yao, F. Qi, J. Guan, P. Ke *et al.*, "Cpm-2: Large-scale cost-effective pre-trained language models," *AI Open*, vol. 2, pp. 216–224, 2021.

[37] J. Lin, A. Yang, J. Bai, C. Zhou, L. Jiang, X. Jia, A. Wang, J. Zhang, Y. Li, W. Lin *et al.*, "M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining," *arXiv preprint arXiv:2110.03888*, 2021.

[38] Y. J. Kim, A. A. Awan, A. Muzio, A. F. C. Salinas, L. Lu, A. Hendy, S. Rajbhandari, Y. He, and H. H. Awadalla, "Scalable and efficient moe training for multitask multilingual models," *arXiv preprint arXiv:2109.10465*, 2021.

[39] 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," *arXiv preprint arXiv:2201.05596*, 2022.

[40] 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.

[41] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019.

[42] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[43] M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. Lin, J. Du, S. Iyer, R. Pasunuru *et al.*, "Efficient large scale language modeling with mixtures of experts," *arXiv preprint*, 2021.

[44] Microsoft, "Tutel: An efficient mixture-of-experts implementation for large dnn model training," https://www.microsoft.com/en-us/research/blog/tutel-an-efficient-mixture-of-experts-implementation-for-large-dnn-model-training/, 2021.

[45] 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.

[46] wikipedia, "Solid-state drive," https://en.wikipedia.org/wiki/Solid-state_drive, 2022.

[47] S. Tang, B. He, C. Yu, Y. Li, and K. Li, "A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 71–91, 2020.

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[49] Intel, "Memory optimized for data-centric workloads," https://www.intel.cn/content/www/cn/zh/architecture-and-technology/optane-dc-persistent-memory.html, 2018.

[50] L. B. Sokolinsky, "Lfu-k: An effective buffer management replacement algorithm," in *International Conference on Database Systems for Advanced Applications*. Springer, 2004, pp. 670–681.

[51] Q. Huang, Z. Yuan, Z. Xing, Z. Zuo, C. Wang, and X. Xia, "1+1>2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application," *IEEE Transactions on Services Computing*, 2022.

[52] S. Shleifer and A. M. Rush, "Pre-trained summarization distillation," *arXiv preprint arXiv:2010.13002*, 2020.

[53] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[54] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf *et al.*, "Mlperf training benchmark," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 336–349, 2020.

[55] X. Chen, M. Zhao, X. Yang, Z. Li, Y. Liu, Z. Li, and Y. Liu, "The cask effect of multi-source content delivery: Measurement and mitigation," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 261–270.

[56] Z. Ling, J. Luo, W. Yu, M. Yang, and X. Fu, "Extensive analysis and large-scale empirical evaluation of tor bridge discovery," in *2012 Proceedings IEEE INFOCOM*. IEEE, 2012, pp. 2381–2389.

[57] S. Li, Z. Lai, D. Li, X. Ye, and Y. Duan, "Embrace: Accelerating sparse communication for distributed training of nlp neural networks," *arXiv preprint arXiv:2110.09132*, 2021.

[58] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park *et al.*, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 993–1011.

[59] P. Contributors, "Paddlefleetx: An easy-to-use and high-performance one-stop tool for deep learning," https://github.com/PaddlePaddle/PaddleFleetX, 2022.

[60] S. Gross, M. Ranzato, and A. Szlam, "Hard mixtures of experts for large scale weakly supervised vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6865–6873.

[61] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2018.