MLP-Offload: Multi-Level, Multi-Path Offloading for LLM Pre-training to Break the GPU Memory Wall

Avinash Maurya
Argonne National Laboratory
Lemont, IL, USA
amaurya@anl.gov

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@mcs.anl.gov

Bogdan Nicolae
Argonne National Laboratory
Lemont, IL, USA
bogdan.nicolae@acm.org

Abstract

Training LLMs larger than the aggregated memory of multiple GPUs is increasingly necessary due to the faster growth of LLM sizes compared to GPU memory. To this end, multi-tier host memory or disk offloading techniques are proposed by state of art. Despite advanced asynchronous multi-tier read/write strategies, such offloading strategies result in significant I/O overheads in the critical path of training, resulting in slower iterations. To this end, we propose *MLP-Offload*, a novel multi-level, multi-path offloading engine specifically designed for optimizing LLM training on resource-constrained setups by mitigating I/O bottlenecks. We make several key observations that drive the design of *MLP-Offload*, such as I/O overheads during the update dominate the iteration time; I/O bandwidth of the third-level remote storage tier remains unutilized; and, contention due to concurrent offloading amplifies I/O bottlenecks. Driven by these insights, we design and implement *MLP-Offload* to offload the optimizer states across multiple tiers in a cache-efficient and concurrency-controlled fashion to mitigate I/O bottlenecks during the backward and update phases. Evaluations on models up to 280B parameters shows that *MLP-Offload* achieves 2.5× faster iterations compared to the state-of-the-art LLM training runtimes.

CCS Concepts

• **Information systems → Data management systems**; **Storage management**; • **Computing methodologies → Distributed artificial intelligence**.

Keywords

Parallel and Distributed LLM and Transformer Training, Multi-Level Asynchronous Offloading, Mixed-Precision Training, Hybrid Memory and Storage Tier Management, Low-Resource Cost-Effective Training

ACM Reference Format:
Avinash Maurya, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2025. *MLP-Offload*: Multi-Level, Multi-Path Offloading for LLM Pre-training to Break the GPU Memory Wall. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3712285.3759864

1 Introduction

Large Language Models (LLMs) have revolutionized a broad range of domains thanks to text summarizing and knowledge distillation, enabling researchers to navigate complex scientific literature more efficiently. A natural evolution towards more general foundational models (FMs) capable of capturing complex correlations between different data modalities (e.g. using cross-attention) are beginning to unlock an even bigger impact on scientific progress.

In a quest for advancing emergent behavior (capabilities not explicitly trained for but emerging spontaneously due to the massive scale and exposure to vast amounts of data during training), FMs' scale and complexity continuously increase, requiring larger training infrastructures and incurring enormous costs. Pre-training and even fine-tuning were not even feasible without large-scale HPC systems: ChatGPT-3 was trained on over 10,000 GPUs [12] and cost over $4 million in each training session. Meta's LLaMA model (released Feb. 2023), follows the same pattern: it used 2,048 Nvidia A100 GPUs to train on 1.4 trillion tokens, taking about 21 days [41]. Even worse, LLMs are rapidly growing both in terms of parameters and training data sizes. For example, ChatGPT-4 is estimated to be 10× larger and was trained on 570× more data than ChatGPT-3. Under these circumstances, the scale and cost required to train or fine-tune FMs become prohibitively expensive.

Motivation: GPU Memory Wall. A large number of state-of-the-art techniques have been proposed to make efficient use of the computational capabilities and massive parallelism offered by a large number of GPUs at scale. For example, DeepSeek [23] has introduced several techniques in this direction. However, with GPU computations becoming more efficient, the bottleneck is shifting towards insufficient GPU memory capacities. For instance, the Gopher-280B model [28] requires 4.8 TB [30], demonstrating terabyte-scale GPU memory requirements to train models in the order of hundreds of billions of parameters. This bottleneck leads to a so-called *memory wall*: the ever-increasing size of model parameters and auxiliary data structures such as activations and optimizer state (450×/2 years) far exceed the GPU memory growth of newer

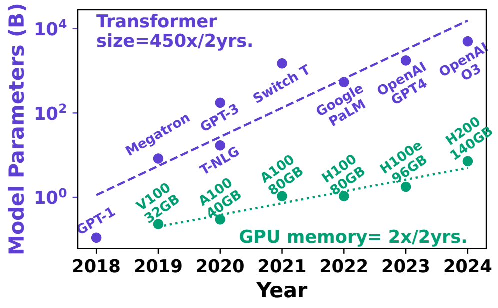

Figure 1: Model vs GPU memory growth.

generations of devices (2×/2 years) [10, 25, 38]. This trend, illustrated in Figure 1 is unsustainable. Furthermore, a majority of HPC datacenters need to serve multiple users running different workloads at the same time [6]. Thus, the ability to train and fine-tune LLMs and FMs on a limited number of GPUs is highly desirable compared with the alternative of waiting for a long time in the batch queue to gain access to a large number of GPUs [9, 15].

To address this challenge, state-of-the-art LLM training frameworks (e.g. DeepSpeed [31] and Megatron [33]) introduce several techniques such as ZeRO redundancy optimization (e.g. partitioning the large-optimizer state across data-parallel replicas) [29], quantization, mixed-precision training [33], etc. Still, this is not enough to alleviate the large GPU memory requirements. As a consequence, these techniques are often complemented with *offloading* of important data structures (model parameters, optimizer states) to the slower host memory (DRAM) [26]. When memory constraints are even tighter, the aggregated GPU and host memory capacity are not enough, prompting the need to further offload the data structures to third-level storage tiers (e.g., NVMe devices) [21, 30, 32]. While the slower memory and storage tiers have significantly higher capacities, they also have significantly lower I/O bandwidth. For example, when pre-training a 20B LLaMA model without offloading (model parameters and optimizer states reside on aggregated GPU memory of a single node), the average iteration duration is 0.4 seconds. Offloading the optimizer states, used during the update phase, to the host memory increases the duration of the iterations to 3.7 seconds on average. Further offloading to a node-local NVMe increases the duration of the iterations to 67 seconds (§ 3.1). Such a large I/O bottleneck results in an overall 170× slowdown compared with training without offloading. At larger scales, such as training the 70B-parameter LLaMA across multiple nodes, communication overheads begin to dominate iteration time, which in turn amortizes the relative cost of NVMe offloading. As a result, the slowdown of NVMe-offloaded training relative to GPU-only training is reduced by about 7×, highlighting the interaction between communication and I/O bottlenecks in distributed settings. Therefore, despite the scale, reducing the I/O bottlenecks when using third-level storage is an important challenge that needs to be solved in order to be able to benefit from a significantly higher overall memory capacity without a prohibitive performance penalty.

Limitations of State-of-the-art. Existing advanced LLM training runtimes such as Microsoft's DeepSpeed [31], PyTorch's Fully Sharded Data Parallel (FSDP) [53], Colossal-AI [19], etc. introduce novel asynchronous offloading techniques for both model and optimizer states. While FSDP allows only offloading to host memory, DeepSpeed and Colossal-AI runtimes enable NVMe offloading. Specifically, the model parameters and optimizer state are partitioned into model shards, which are distributed among worker processes (one per GPU) and collaborate using model parallelism. In turn, each model shard is decomposed into smaller chunks, called "subgroups", which are processed in combination with mixed-precision techniques one by one as follows. During the forward pass and backward pass, a copy of the FP16 parameters is used on the GPUs to compute the gradients of each subgroup. As the backward pass progresses, the gradients are flushed to host memory and from there to third-level storage in FP32 format. Then, the update phase is performed on the CPU to avoid excessive traffic between the host memory and GPU memory, which negates the computational speed-up of GPUs. During the update phase the FP32 model parameters, optimizer state and gradients of each subgroup are fetched from third-level storage to host memory, the CPU-based updates are performed, and then the updated model parameters and optimizer state are flushed back to the third-level storage, while at the same time an FP16 version is pushed to the GPUs. Efficient pipelining and asynchronous I/O techniques are used to overlap computations with transfers between the GPU memory, host memory and third-level storage using specialized engines such as DeepNVMe [32]. However, this is not enough to alleviate the limited I/O bandwidth of third-level storage. Furthermore, compute nodes are equipped with many GPUs that share the same host memory and third-level storage, resulting in competition for I/O bandwidth that is not mitigated. Another limitation of state-of-art approaches is the unnecessary back-and-forth movement between the host memory and third-level storage due to a suboptimal strategy to handle gradients and to reuse subgroups already available in the host memory.

Key Insights and Contributions. In this work, we present *MLP-Offload*, a novel approach that aims to alleviate the aforementioned limitations of state-of-the-art offloading techniques that make use of third-level storage to increase the overall memory capacity and I/O bandwidth. Specifically, we leverage a key observation that external storage of HPC systems, e.g., parallel file systems (PFS) and object stores, are typically underutilized during LLM training (except for occasional checkpointing [49]), therefore they can complement node-local NVMe storage to provide a significant boost of I/O bandwidth. Combined with concurrency control to mitigate contention for I/O bandwidth and a better reuse of cached subgroups from one iteration to another, these ideas enable *MLP-Offload* to achieve a 2.5× speedup over state-of-art approaches.

We summarize the key contributions as follows:

(1) We perform an in-depth characterization of resource utilization during different phases of training when the optimizer state is offloaded to SSDs. In particular, we highlight several fundamental observations for our proposal: the CPU memory, utilized as caching buffers for asynchronous data transfers, experiences thrashing at every iteration; the upscaling and offloading of gradients are redundant; modern PFS capable of delivering several GB/s worth of parallel bandwidth (e.g. using Infiniband) remains unused; CPU cores remain idle due to slow disk writes despite overlapping transfers with CPU-based optimizer updates (§ 3.1).

(2) Based on the above characterization, we propose a series of design principles: unified multi-level, multi-path asynchronous offloading using virtual tiers; optimized virtual tier concurrency control for multi-path I/O; cache-friendly ordering of model subgroup processing; and delayed in-place mixed-precision gradient conversion during updates (§ 3.2). These design principles are complemented by an I/O performance model detailed in § 3.3.
(3) We present *MLP-Offload*, an open-source implementation of the design principles that integrates with existing state-of-the-art LLM training runtimes such as DeepSpeed and Megatron. Our implementation insists on low-level optimizations such as process-exclusive multi-thread-shared locking mechanism in `libaio`– an optimized asynchronous POSIX I/O library; and efficient non-sequential ordering of asynchronous prefetch and flush operations to/from the disk (§ 3.5).
(4) We evaluate *MLP-Offload* in a series of experiments in which we pretrain up to 280B parameters model on 32×A100-40GB GPUs. Our approach accelerates both backward and update phases by 13.5× and 2.3×, respectively, and speeds up the end-to-end training by 2.5× as compared to the state-of-art DeepSpeed (§ 4).

2 Background and Related Work

Data, Pipeline, and Tensor Parallelism. Various parallelism techniques for data, pipelines, and tensors, have been widely adopted to accelerate the training of large models, such as Convolutional Neural Networks (CNNs) [52], Deep Learning Recommendation Models (DLRMs) [50], Large Language Models (LLMs) [51], and Large Vision Models (LVMs) [16]. Data parallelism accelerates training by running multiple model replicas, each running forward and backward passes in parallel with different input mini-batches and synchronizing at the end of the iteration in the update phase to combine the patterns learned from all mini-batches. Pipeline [13] and Tensor [5, 33] parallelism techniques split large models across GPUs with limited memory capacities. While pipeline parallelism splits the model vertically by placing a subset of model layers on a given GPU, tensor parallelism performs horizontal partitioning by splitting the model across all available GPUs. The combination of data, pipeline, and tensor-parallelism, often termed "3D parallelism", is used to effectively scale large-scale training across thousands of GPUs for CNNs, LVM, LLMs, DLRMs, etc. However, given the ever-growing large sizes of LLMs and their rapid adoption in various fields, optimizations beyond 3D parallelism are required to run in resource-constrained setups.

ZeRO Redundancy Elimination. State-of-the-art LLM training runtimes, e.g., PyTorch's Fully Sharded Data-Parallel (FSDP) [53], Microsoft DeepSpeed's Zero Redundancy (ZeRO) [29], Colossal-AI [19], use redundancy elimination techniques to remove redundant parts of the model and optimizer states across data parallel ranks to minimize GPU memory consumption. To this end, DeepSpeed, a widely used [34, 45] LLM training runtime, proposes three stages for eliminating redundancy across data parallel ranks: ZeRO-1 splits the optimizer states; ZeRO-2 splits both the optimizer states and gradients; and ZeRO-3 splits optimizer states, gradients, and model parameters [29]. As illustrated in Figure 2(a), training a P parameters model on N GPUs with full redundancy elimination (ZeRO-3) leads to $\sim \lceil O(\mathcal{P})/N \rceil$ order of memory savings at the expense of 1.5× higher communication overheads [29].

The ZeRO-3 technique partitions model states across GPUs, requiring frequent scatter-gather collectives to reconstruct layers on demand, significantly increasing communication costs. Consequently, ZeRO-3, despite its memory efficiency, cannot be seamlessly combined with pipeline parallelism, which relies on efficient inter-stage communication. Instead, ZeRO-3 employs a hybrid model and data parallelism strategy, where training on N GPUs typically results in N "virtual" data-parallel replicas. Unlike conventional data parallelism where the model is fully replicated, these replicas remain virtual because model states are dynamically fetched and synchronized across GPUs as required.

Mixed Precision Training. Mixed precision training, proposed by Baidu and Nvidia research [27], is another widely adopted approach to improve throughput and reduce the memory footprint in LLM training. This is illustrated in Figure 2(b) in the left dotted block. Specifically, mixed precision uses two different copies of the model parameters, one in `FP16` (or `BF16`), used to run the forward and backward passes, and another master copy in high-precision (`FP32`), used by the optimizer in the update phase to retain higher stability [18]. The activations and gradients produced by the forward and backward passes are in half-precision formats, leading to faster communications and faster computations. The low-precision `FP16` gradients are upscaled to `FP32` and used by the optimizer to perform the update phase. Several real-world LLMs, such as BLOOM-176B [45], OPT-175B [49], GPT-3 [4], and GLM-130B [47], are pre-trained using mixed-precision, thereby demonstrating the stability and efficiency of the mixed-precision approach.

Sharded Model and Optimizer States Into Subgroups. To reduce the intermediary memory required during computations, DeepSpeed's ZeRO-3 shards the model parameters and optimizer states of each rank/GPU into subgroups, as depicted in Figure 2(b). The subgroup sharding technique is unique to the DeepSpeed runtime and is unavailable on other runtimes such as FSDP or Colossal-AI. The size of these subgroups, M, is user-defined and specifies the total number of parameters per subgroup. The subgroups are evenly distributed among the GPUs. When using mixed precision for training, for each subgroup of M parameters, the forward and backward passes operate on the FP16 parameters and FP16 gradients, while the update step operates on the M corresponding FP32 parameters and FP32 optimizer state. This subgroup-style sharding allows for efficient piecewise computation and communication overlaps while minimizing memory footprint, as explained in ZeRO-Infinity [30].

Optimizer State Offloading. The optimizer state, held in FP32, is much larger than the model parameters and is only required during the update step. Thus, in memory-constrained scenarios, offloading it to the host memory (and further other multi-level tiers such as node-local NVMe) is a practical choice. The use of subgroups further facilitates swapping between the host memory and other multi-level tiers. This is shown in Figure 2(c). When optimizer offloading is enabled, updates are typically performed on the CPU because (a) GPU memory is typically fully utilized by FP16 model parameters, and (b) transferring FP32 optimizer

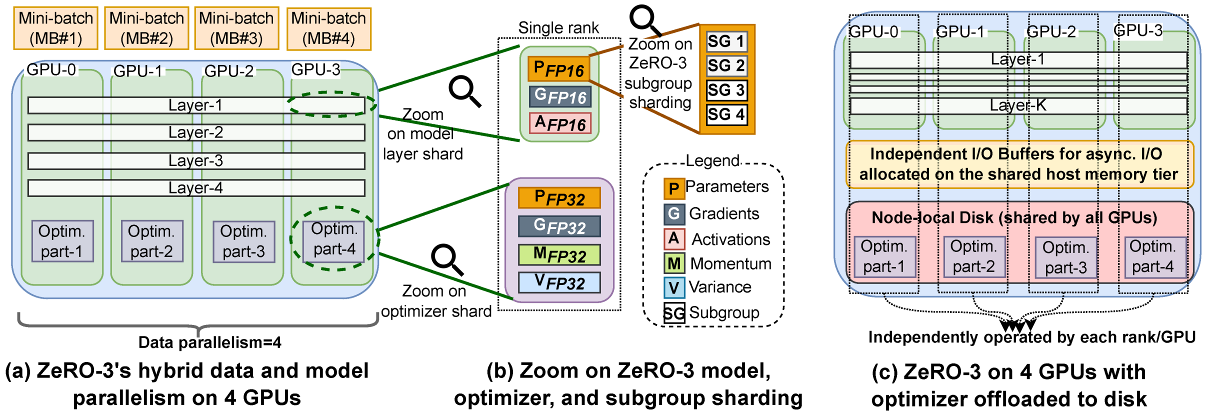
Mini-batch (MB#1)
Mini-batch (MB#2)
Mini-batch (MB#3)
Mini-batch (MB#4)
GPU-0
GPU-1
GPU-2
GPU-3
Layer-1
Layer-2
Layer-3
Layer-4
Optim. part-1
Optim. part-2
Optim. part-3
Optim. part-4
Data parallelism=4
(a) ZeRO-3's hybrid data and model parallelism on 4 GPUs
Zoom on model layer shard
Zoom on optimizer shard
Single rank
P FP16
G FP16
A FP16
P FP32
G FP32
M FP32
V FP32
Zoom on ZeRO-3 subgroup sharding
SG 1
SG 2
SG 3
SG 4
Legend
P Parameters
G Gradients
A Activations
M Momentum
V Variance
SG Subgroup
(b) Zoom on ZeRO-3 model, optimizer, and subgroup sharding
Layer-1
Layer-K
Independent I/O Buffers for async. I/O allocated on the shared host memory tier
Node-local Disk (shared by all GPUs)
Independently operated by each rank/GPU
(c) ZeRO-3 on 4 GPUs with optimizer offloaded to disk

Figure 2: DeepSpeed ZeRO-3 showing (a) hybrid model and data parallelism; wherein (b) the model parameters, activations, gradients, and optimizer states are sharded into subgroups; and (c) the optimizer states are offloaded to node-local NVMe.

states and gradients between GPU and CPU, even subgroup-by-subgroup, incurs high I/O overhead. To efficiently manage these subgroup updates, each GPU worker allocates an independent I/O buffer on shared host memory, with enough capacity to hold a configurable number of subgroups. These I/O buffers serve two primary purposes: (1) caching FP16 gradients during the backward pass, upscaling them to FP32, and asynchronously flushing the FP32 gradients to disk for consumption during the update step, and (2) offloading to disk using asynchronous swap-in/out of optimizer subgroups between the host memory and the disk.

Accelerating Model and Optimizer Offloading. Several recent efforts focus on mitigating the I/O bottlenecks encountered due to offloading for training, fine-tuning, and inference scenarios. For instance, Fuyou [21] and LoHan [22] perform pipelined overlapping activation, parameter, and optimizer swapping across GPU-CPU-SSD to accelerate fine-tuning for extremely constrained scenarios, wherein neither model nor optimizer fits on the GPU memory; and is therefore not applicable for the scenario targeted in our work. Smart-Infinity [14] uses computational storage devices (NVMe attached to FPGAs) to mitigate I/O between the NVMe and CPU, which accelerates the update phase but requires specialized NVMe devices. To mitigate the slowdown due to the expensive update phase, asynchronous update techniques such as one-step-delayed updates had been previously proposed in ZeRO-Offload [32]: it overlaps the next iteration on the GPUs (with stale parameters) with the still ongoing CPU-based updates. However, this optimization was later removed from the runtime due model inconsistencies [36]. DeepSpeed TwinFlow [43] and Deep Optimizer States [26] are complementary efforts that exclusively focus on optimizing CPU-only offloading and do not optimize disk-based offloading.

3 Our Contribution: *MLP-Offload*

3.1 Gap Analysis

The combination of redundancy elimination and multi-tier offloading enables the pre-training and fine-tuning of large LLM models that normally would not fit in the aggregated GPU memory. However, the differences in I/O bandwidth between GPU memory, host memory, and third-level storage are large, especially when considering node-local SSDs or other forms of disk storage [32]. As a consequence, I/O overheads may significantly delay especially the update phase, during which the offloaded subgroups need to be fetched, updated and flushed back.

Training Iterations are Dominated by the Update Phase. As a first step, we study the breakdown of a training iteration in terms of time spent in the forward pass, backward pass and the update phase, in order to check whether the update phase occupies a significant fraction of the overall iteration duration, which would validate the hypothesis that accelerating the update phase would accelerate the overall training iteration. To this end, we study DeepSpeed [31], configured to use redundancy elimination and multi-tier offloading that includes a local SSD. We run experiments on a single node with 4×H100-80GB GPUs (Table 1), and record a breakdown of the iterations. We indeed confirmed that the iteration is dominated by the update phase. For instance, in the case of a 40B model (setup and methodology described in § 4.1, respectively), the forward and backward passes finish in 0.6s (0.02%) and 28s (11%) of the total 242s iteration time, while the update phase finishes in 213s (89%). Thus, accelerating the update phase is an important step towards a faster end-to-end runtime.

Update Phases are Dominated by I/O when using SSD Offloading. We evaluate four model configurations, i.e., 20B, 40B, 70B, and 120B parameters model (Table 2). For the 20B model [44], the full FP32 parameters and optimizer state fit in the 512 GB host memory; therefore, we use it as a baseline. Larger model sizes offload their subgroups to an SSD. As shown in Figure 3, in the case of the 20B model, the update phase completes ~30× faster than in the case of larger models that offload the model subgroups to an SSD. Specifically, when subgroups are offloaded to the SSD, 99% of the update phase duration is spent in SSD I/O, despite DeepSpeed's optimized DeepNVMe engine that overlaps I/O with CPU computations. Thus, we can conclude that accelerating the fetches and flushes of offloaded subgroups to/from the host memory will significantly accelerate the update phase, which according to the previous experiments, will significantly accelerate the overall iteration.

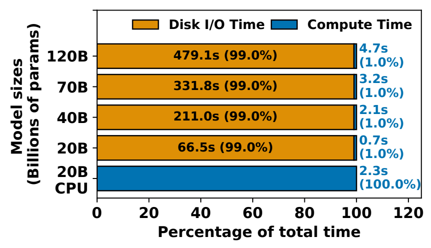

Figure 3: Fraction of time spent in disk I/O during the update phase

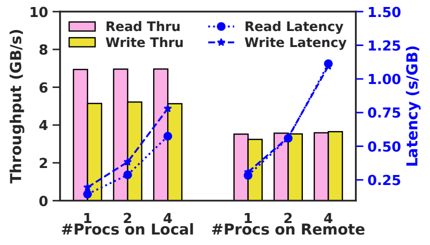

Figure 4: I/O bandwidth of SSD (local) vs. parallel file system (remote)

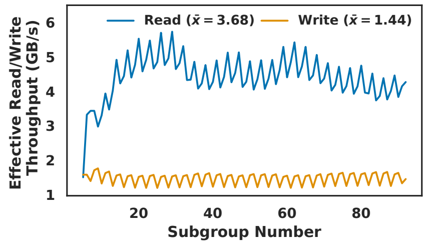

Figure 5: I/O bandwidth under concurrency per subgroup using a local SSD

I/O Bandwidth of Different Third-Level Storage. HPC systems are not only equipped with node-local SSDs but also external storage such as parallel file systems (PFS) and object stores (e.g. DAOS). Although external storage is shared by all compute nodes, and therefore they are subject to I/O competition, at a medium scale, it can provide a significant aggregated I/O bandwidth boost to complement the limited I/O bandwidth of node-local SSDs. To quantify this opportunity, we measure the raw read and write throughput of both node-local NVMe storage and remote PFS using microbenchmarks. Remote storage can offer higher throughput than node-local NVMe in some cases (e.g., Testbed-2 in Table 1). Additionally, given that each GPU process independently issues fetch and flush operations for its optimizer state subgroups, understanding I/O performance under contention is crucial. Figure 4 shows that while overall read/write throughputs remain constant as the number of concurrent processes increases, per-process latency (shown using lines and minor y-axis) worsens due to contention within the storage subsystem. This suggests that shared NVMe bandwidth saturation is a limiting factor, making I/O the primary bottleneck at scale.

I/O Bandwidth under Concurrency during Offloading. Finally, we analyze the effective read/write throughput perceived by the training runtime when offloading the optimizer states of a 40B model to a node-local NVMe. Figure 5 reveals oscillations in throughput due to runtime memory constraints, which restrict active subgroups in host memory to three at a time: one prefetched, one actively updated, and one flushed back to disk. The slow flush-back rate results in cases where the next subgroup is prefetched before the previous one is fully written, leading to intermittent spikes in read-throughput. However, the aggregated read and write throughput remains bottlenecked by NVMe's write bandwidth, reinforcing the I/O limitations observed in prior experiments.

3.2 Design Principles

Unified Multi-level, Multi-path Asynchronous Offloading using Virtual Tiers. Leveraging local SSDs for offloading is scalable because the SSDs can be leveraged independently using model parallelism to achieve a high aggregated I/O bandwidth. However, model subgroups offloaded to local storage, such as SSDs, introduce high I/O overheads that dramatically slow down the update phase, as discussed in § 3.1. On the other hand, external storage has significant potential to complement the limited I/O bandwidth of node-local SSDs and is typically under-utilized during pretraining (save for occasional checkpoints) [46]. It is this observation that we capitalize on to extend existing multi-level offloading techniques with support for multi-path I/O at each level. Specifically, we unify all alternative storage (local SSDs, parallel file systems (PFS), object stores) into a virtual, third-level tier that can be used to offload model subgroups from the host memory. Then, based on an I/O performance model (detailed in § 3.3), we assign the subgroups to the alternative storage tiers proportionally to their I/O bandwidth. Using this approach, we can parallelize I/O operations such that slow tiers finish roughly at the same time as fast tiers because they store fewer subgroups. This I/O load balancing allows our approach to avoid any of the tiers becoming a bottleneck, thereby maximizing the acceleration of I/O operations for the overlapping with the computations. Although we illustrate this principle for the third-level tier, it can be generalized to any level (e.g., second-level tier to combine GPU-CPU or GPU-GPU HBMs, DDR, CXL, etc.).

Optimized Virtual Tier Concurrency Control for Multi-Path I/O. To enable high scalability for model parallelism, we allow each worker process (typically attached to a single GPU), to apply I/O load balancing on alternative storage independently of other workers. Using this approach, we avoid the need for expensive global synchronization. However, each compute node features multiple GPUs, which means that multiple processes will compete for a shared I/O bandwidth to alternative storage tiers. Therefore, we introduce a lightweight concurrency control strategy at the node level that allows only one worker process on each compute node to access a given alternative storage at a time. With this restriction of exclusive access per alternative storage, the full I/O bandwidth is guaranteed to be available to a worker process, which achieves lower latency (Figure 4) and I/O load balancing. At the same time, the other worker processes are free to compute updates for prefetched groups using all the available CPU cores, and use other alternative storage(s) in parallel, resulting in a natural interleaving that achieves global I/O load balancing. Note that a worker is not limited to accessing an alternative storage using a single I/O thread; it can leverage the preferred I/O parallelism of the alternative storage (e.g., a PFS is faster when using multiple I/O threads [11]). Furthermore, if the preferred I/O parallelism cannot be saturated by a single worker, then the alternative storage can be subdivided further into multiple virtual tiers.

Cache-Friendly Ordering of Model Subgroup Processing. An important observation during the update phase of adaptive optimizers (e.g., ADAM [17]) is that the computations are embarrassingly parallel: each model subgroup has its own set of corresponding parameters, optimizer state and gradients that can be applied independently to obtain the new model parameter and optimizer state. Therefore, *the order in which the subgroups are independently processed is inconsequential and does not impact accuracy or convergence.* We exploit this observation to design a cache-friendly update strategy that leverages subgroups present on fast tiers (host memory) as much as possible. Specifically, in the first iteration, we start processing the subgroups in increasing order of IDs. After the update phase, the subgroups with a high ID will reside on fast (cache) tiers, while the subgroups with a low ID will reside on the slow tiers as they were evicted to slow tiers to make room for the subgroups with higher ID in the host memory. Then, in the second iteration, unlike state-of-the-art approaches, we reverse the order of processing the subgroups. This results in a significant acceleration of the update phase because a large number of subgroups with a high ID are already present on the fast tiers, thereby reducing I/O overheads due to memory thrashing. After the second iteration, the subgroups with low IDs end up on the fast tiers, which means we can take advantage of the increasing order again in the third iteration. Thus, we keep alternating between ascending and descending order of subgroup IDs which maximizes the benefits of caching.

Delayed In-place Mixed-Precision Gradient Conversion during the Update Phase. As mentioned in § 2, offloading is typically implemented in combination with mixed-precision training. In this case, state-of-the-art approaches typically push the FP16 gradients that are computed during the backward pass from the GPU to the host memory, where they are converted to FP32. From there, they are flushed to the third-level virtual tier. However, during the update phase, each subgroup offloaded to the third-level virtual tier needs to be brought back to the host memory. Since the subgroup, composed of FP32 optimizer states (parameters, momentum, and variance), becomes even larger because of FP32 gradients, each fetch operation is slower than in the case when the subgroup only holds the optimizer state. On the other hand, FP16 to FP32 gradient conversions on a modern CPU have a high throughput (65 GB/s on Testbed-1 in Table 1) that is an order of magnitude larger than the fetch throughput. Thus, we propose an alternative strategy: during the backward pass, we simply store the FP16 gradients on the host memory– which anyway needs to reserve enough room for the FP16 gradients of all subgroups to enable gradient accumulation. Then, during the update phase, unlike state-of-the-art approaches, we fetch the subgroup (without FP32 gradients) and instead convert the FP16 gradients to their FP32 variant on-the-fly (using the same standardized numeric primitives as DeepSpeed [31, 43]). Thus, we accelerate both the backward pass (as we eliminate large asynchronous FP32 gradient flushes that can potentially delay the backward pass if they do not fully overlap with the computations) and the update phase (because the overhead of in-place FP16 to FP32 gradient conversion is typically negligible compared with the I/O overhead of fetching FP32 gradients from slow tiers).

3.3 Performance Model for Subgroup Allocation

To enable load balancing for virtual tiers that can leverage multiple alternative storage options by aggregating multi-path I/O, we propose an approach that assigns model subgroups proportional to the I/O bandwidth of each alternative storage. Specifically, we assume each worker splits its model shard into M equally sized subgroups, which is typically the case to achieve computational load balancing. Furthermore, we assume a virtual tier composed of N storage tiers P_i, where $0 \leq i < N$, each with respective I/O bandwidth of B_i– the minimum of read or write throughput. Then, the number of subgroups T_i allocated to each storage tier P_i can be represented as:

$$T_i = \left\lceil \frac{M \cdot B_i}{\sum_{i=0}^{N} B_i} \right\rceil, \text{adjusted such that} \sum_{i=0}^{N} T_i = M \qquad (1)$$

The intuition behind Equation 1 is to allocate to each alternative storage a number of subgroups roughly equal to the contribution of its I/O bandwidth to the total aggregated I/O bandwidth. This results in parallel fetches and flushes of subgroups from different alternative storage that finish at roughly the same time. Therefore, this will reduce the likelihood of computational stalls due to a straggling alternative storage while the others remain idle.

Initially, B_i for each alternative storage is measured using microbenchmarks. Then, after the first iteration, B_i is adjusted based on the average observed I/O bandwidth for subgroup flushes and fetches. This ensures that our approach adapts to any potential shifts in I/O bandwidth trends that may affect some of the alternative storage options. For example, a local SSD exclusively owned during a batch job reservation will not experience I/O bandwidth shifts. However, a parallel file system may be under I/O pressure from different batch jobs owned by different users, in which case an updated B_i can modify the value of T_i to repartition the subgroups across different virtual tiers based on their I/O bandwidths.

In addition to faster backward and update phases, the virtual storage tiers in *MLP-Offload* also accelerate the checkpointing process by pre-staging a fraction of optimizer states to persistent storage. This can be leveraged by multi-tier asynchronous checkpointing engines such as DataStates-LLM [24] to flush the remainder of model and optimizer states from the GPU memory, host memory, and the non-persistent storage tiers, such as local-disk, during the immutable forward and backward passes.

3.4 System Composition of *MLP-Offload*

We illustrate how to combine the design principles mentioned above in Figure 6, showing the end-to-end iteration execution of DeepSpeed's ZeRO-3 vs our approach. Specifically, we assume a simplified LLM architecture where each model shard on a GPU is composed of four subgroups denoted by $S^1 \dots S^4$. The same pattern happens in parallel on the other processes when using model parallelism. DeepSpeed ZeRO-3 computes the FP16 gradients for each subgroup during the backward pass, and then flushes them to the host memory in the background. On the host, the FP16 gradients are converted to FP32 and flushed to the NVMe. Then, during the update phase, each subgroup (composed of FP32 parameters, momentum, variance, and gradients) is asynchronously fetched from the NVMe using a pipeline: as soon as S^1 is available in the host memory, a corresponding parallel multi-core CPU computation is triggered to update the model and optimizer states. Meanwhile, S^2 is fetched to the host memory. When the computation is complete, the subgroup's new parameters are fetched on GPU and the optimizer state is flushed to the SSD after discarding FP32 gradients. The fetches, updates, and flushes overlap throughout the update phase. At the next iteration, the same sequence of patterns repeats.

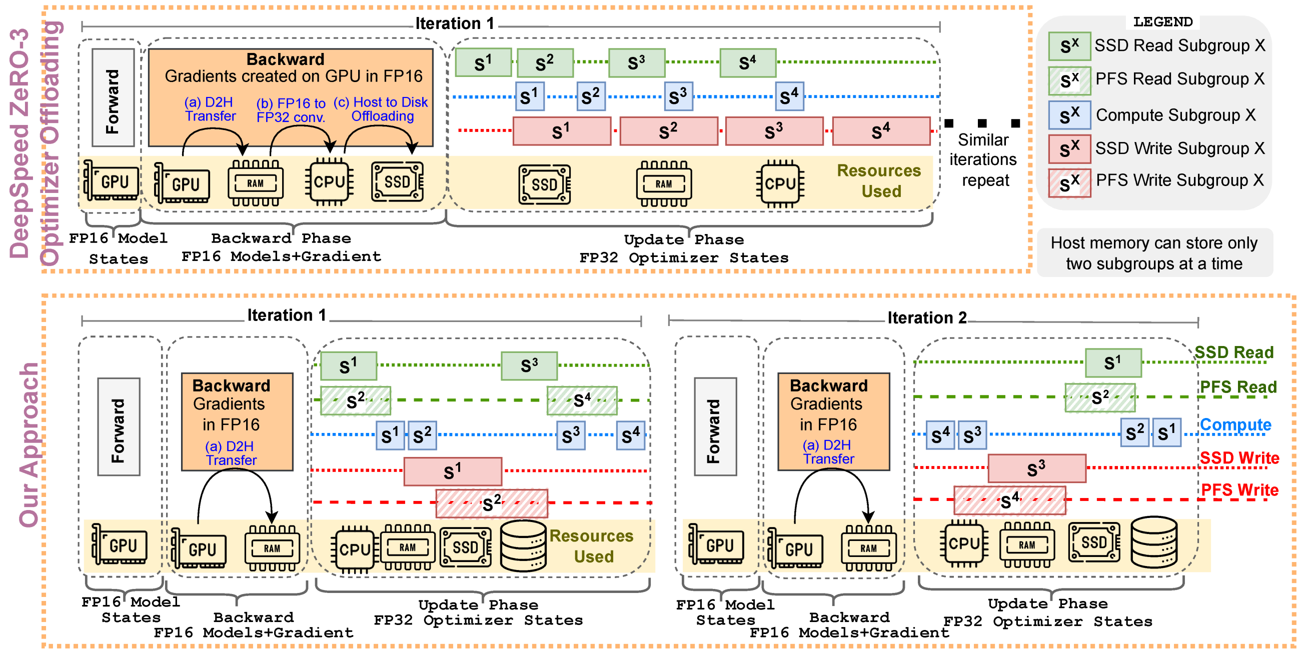

Figure 6: Illustrative example: application of the design principles proposed by *MLP-Offload* vs. state-of-art (DeepSpeed ZeRO-3).

Algorithm 1: Optimizer Multi-Tier Offloading and Updates

```
Input: iter: training iteration number; ⟨S⟩: Optimizer subgroups;
       ⟨B⟩: I/O bandwidth of local and remote tiers. f2h, h2f, and
       h2d, denote file-to-host, host-to-file (for NVMe and PFS),
       and host-to-device (GPU) transfers, respectively.
1  Function run_update(iter, ⟨S⟩, ⟨B⟩):
2      update_order ← {0, 1, . . . |S| − 1} or {|S| − 1, |S| − 2, . . . 0}
           // reversed at each update phase to maximize host memory hits
3      for i ∈ update_order do
4          pt ← storage_tier(i)
5          s ← f2h_prefetch_wait_subgrp(i)
6          s.grad ← host_grad_accum_FP16[i].to(FP32)
7          cpu_update_kernel(s)
8          async_h2d_transfer(s.params.to(FP16))
9          t ← assign_storage_tier(i, B)
10         async_h2f_flush(s, t)
11         async_f2h_prefetch(next_subgroup(i), pt)
```

In the case of our approach, we create a virtual third-level tier that combines the local NVMe with a parallel file system (PFS). Initially, the subgroups are created on the host memory and flushed to either the NVMe or PFS (according to the performance model discussed in § 3.3). The FP16 parameters are copied to the GPU. Then, the training iterations can start. Unlike ZeRO-3, the backward pass does not upscale and flush gradients to storage tiers. Algorithm 1 provides a high-level pseudo-code description, complementing Figure 6. Specifically, during the update phase, S^1 and S^2 are fetched in parallel from the NVMe and PFS through separate I/O paths. As soon as S^1 completes (Line#5), the corresponding gradients on the host accumulation buffer are upscaled from FP16 to FP32 (Line#6), after which the update computation starts (Line#7). Then, an asynchronous host-to-device transfer of the downscaled FP16 model parameters is initiated (Line#8) after which the new storage tier t for S^1 is determined using equation 1 in Line#9. Subsequently, the asynchronous flush of S^1 to its new tier t and asynchronous prefetch of the next group on the previous tier pt of S^1 is initiated (Lines #10-11). As seen in Figure 6, since the fetch of S^2 finishes before the update phase of S^1, the update computation of S^2 can start immediately after the update computation of S^1. Meanwhile, the flush of S^1 and prefetch of S^4 can progress independently, again on separate I/O paths. While the asynchronous prefetch and flush operations can be enqueued, they might get deferred due to the tier-exclusive concurrency control mechanism (§ 3.2) when competing with other node-local processes. Similarly, S^3 and S^4 have an opportunity to leverage separate I/O paths for fetches. However, unlike ZeRO-3, note that we do not need to flush S^3 and S^4 to the virtual tier, because we can switch the order in the next update phase (Line #2) to directly use S^4 and S^3 from the host memory.

3.5 Implementation and Integration

We implement the proposed design principles in *MLP-Offload* [1] as a standalone open-source library integrated with DeepSpeed ZeRO-3. Specifically, the integration extends DeepSpeed's DeepNVMe offloading engine, which leverages `libaio` [8], a kernel-accelerated asynchronous I/O library, to support multi-path parallel read/write operations across processes (each mapped to GPUs) and storage tiers. In addition to benefiting from kernel-accelerated I/O, DeepNVMe's C++-based implementation avoids inefficiencies introduced by Python's Global Interpreter Lock (GIL) and PyTorch's pooled memory management. Building on this foundation, *MLP-Offload*

[1]https://github.com/DataStates/artifacts/blob/main/MLP-Offload

orchestrates efficient host buffer management through explicit pool-based allocations for asynchronous fetch/flush operations, enabling fine-grained concurrency and reducing I/O contention when processes share a storage tier.

MLP-Offload can be enabled and configured via two JSON key-value pairs in the DeepSpeed runtime configuration. During ZeRO-3 initialization, we instantiate multiple offloading engine objects per process, corresponding to the number of storage tiers. Each offloading object is assigned a dedicated host buffer to facilitate asynchronous prefetching and lazy flushing. The host buffer size is configurable based on available host memory, the number of processes per node, and the number of storage tiers. Additionally, *MLP-Offload* allows user-specified distribution of subgroups across storage tiers, guided by the performance model (§ 3.3). For example, a 2:1 split between `/local/` and `/remote/` directories ensures that for every subgroup stored remotely, two are offloaded to the local disk. Beyond managing multiple offloading engines, *MLP-Offload* dynamically predicts subgroup prefetch order and where to lazily flush updated subgroups based on the offloading ratio. While designed for Megatron-DeepSpeed with ZeRO-3, the core principles of *MLP-Offload* make it extensible to other training runtimes, such as TensorNVMe [37] in Colossal-AI [19] by specifying multiple `DiskOffloader` objects to create the virtual third-level tier, on each of which the corresponding subgroups dictated by our performance model can be consequently offloaded.

4 Performance Evaluation

4.1 Methodology

Experimental Setup. We conducted our experiments on Testbed-1 (ANL JLSE) [2] and Testbed-2 (ALCF Polaris) [3], consisting of 4×H100-80GB and 4×A100-40GB GPUs per node, respectively, outlined in Table 1. The ratio of host memory to aggregated GPU memory for the Testbed-1 and Testbed-2 platforms are 1.6:1 (similar to AWS `p4de.24xlarge`) and 3.2:1 respectively (similar to AWS `p5.48xlarge`), neither adequate enough to hold the 8:1 optimizer to model state memory ratio described in ZeRO-Inifinity [30], compelling NVMe offloading. Both platforms consist of 2× RAID-mounted 1.6 TB NVMe M2 SSDs for local storage, the read and write throughputs of which are listed in Table 1. The Testbed-1 nodes feature 2× Intel(R) Xeon(R) Platinum 8468, consisting of 96 CPU cores, while the Testbed-2 nodes feature 1× AMD EPYC 7543P, consisting of 32 CPU cores. We use Testbed-1 for small-scale experiments and Testbed-2 for scalability experiments. In terms of external storage use as alternative offloading tiers, Testbed-1 features a VAST [42] parallel file system of 1 PB capacity using 4 DNodes, whose read/write throughputs roughly correspond to speeds of AWS's advanced FSX Lustre FS [1]. Testbed-2 is composed of 100 PB storage using HPE ClusterStor E1000 platform through 160 (OSTs), with read/write throughputs described in Table 1.

Compared Approaches. We compare ***MLP-Offload***, illustrated in Figure 6 (bottom) with ***DeepSpeed ZeRO-3***, which is a prominent implementation of the latest state-of-art in LLM training and illustrated in Figure 6 (top). Specifically, DeepSpeed ZeRO-3 supports NVMe offloading of the optimizer states, similar to Colossal-AI [19]. We use its asynchronous offloading engine, i.e., DeepNVMe, which

Table 1: Testbed configurations.

Feature ↓ | Testbed →	Testbed-1 [2]	Testbed-2 [3]
GPUs	4× H100-80GB	4× A100-40GB
Pinned D↔H B/W (GB/s)	55	25
Number of CPUs cores	96	32
Per node host memory (GB)	512	512
NVMe Read | Write thruput (GB/s)	6.9 | 5.3	13.5 | 4.8
Parallel File System (PFS)	VAST FS	Lustre FS
PFS Read | Write throughput (GB/s)	3.6 | 3.6	6.9 | 13.7

Table 2: Models used for evaluations. N_L: Number of layers; D_H: Hidden dimensions; AH: Attention heads.

Model	40B[44]	52B[20]	70B[40]	100B[44]	120B[35]	130B[48]	280B[28]
N_L	128	64	80	124	96	70	72
D_H	5120	8192	8192	8192	10240	12288	16384
AH	40	64	64	64	80	96	128

overlaps all three operations (fetch, update, flush) to accelerate the update phase at the cost of additional host memory required for asynchronous data movement. Given the fact that the host memory is exhausted while holding a fraction of subgroups, we consider the advanced asynchronous engine as a representative baseline.

Models and Dataset. The configurations of models used in our evaluations, which are based on real-world LLM training scenarios, are summarized in Table 2. We do not consider models smaller than 40B because their optimizer states are small enough to fit in the host memory (512 GB). We use a subset of the OSCAR-en dataset consisting of 79K records, included in the repository of the Bloom model [45], and use the default LLaMA2 [39] tokenizer for preprocessing the dataset into tokens. Unless otherwise noted, similar to the OPT training configuration [49], we set the default sequence length to 2048 and microbatch size to 1 to avoid OOM errors in any configuration.

Runtime Configurations. As discussed in Section 2, DeepSpeed does not support pipeline parallelism in combination with ZeRO-3, which is responsible for the sharding of the model and optimizer states. Therefore, for single-node experiments, we use data-parallelism with ZeRO-3, which shards the model parameters, gradients, and optimizer states to fit in the GPU memory. For weak scalability experiments, we use a combination of tensor-parallelism (intra-node) and data-parallelism (inter-node) approaches to maximize performance and memory savings [30, 33].

In each experiment, all the GPUs on the selected node(s) are utilized, and each GPU is associated uniquely with a single process. For all models, local NVMe offloading is enabled, based on the asynchronous approach for overlapping fetch, flush, and update operations of subgroups. Our approach has additional access to the PFS. To facilitate prefetching and lazy-flushing of subgroups for asynchronous offloading, a configurable number of pinned host buffers are pre-allocated such that the host memory is utilized to the maximum extent (>90% memory utilization) for all compared approaches. The size of this host buffer for asynchronous offloading varies between different models because each model reserves a

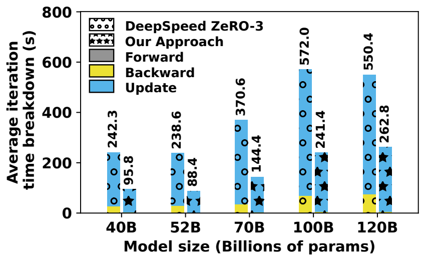

Figure 7: Average iteration time breakdown on scaling model sizes.

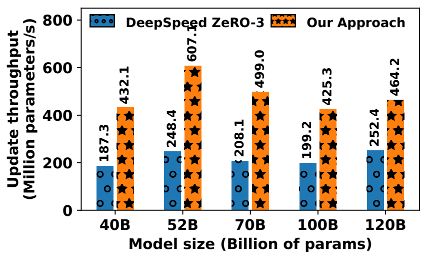

Figure 8: Average update throughput observed when scaling model sizes.

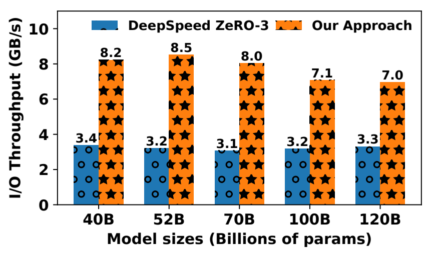

Figure 9: Effective I/O throughputs for different model sizes.

different amount of runtime-level buffers, such as for gradient accumulation, all-reduce, etc. We refer the reader to ZeRO-Offload [32], ZeRO-Infinity [30], and DeepSpeed memory estimator [7] for the breakdown of GPU/host memory consumed by the runtime.

Throughout our evaluations, we ensure that the aggregated GPU memory is sufficient to store the following: (1) FP16 model parameters; (2) activation checkpoints generated by the forward pass; and (3) FP16 gradients of one at least subgroup generated during the backward pass which is flushed asynchronously to the gradient accumulation buffer residing on the host memory. We also ensure that the host memory is large enough to hold runtime-level buffers (e.g., gradient accumulation, all-reduce buckets, etc. [7]), and a minimum of three subgroups to facilitate asynchronous updates: the previous subgroup being lazily flushed to disk, the current subgroup being updated, and the next subgroup being prefetched from local NVMe.

Although the subgroup sizes do not impact the iteration duration, convergence, or accuracy as mentioned in [14, 26], smaller subgroups achieve better I/O and compute overlap of offloaded subgroups. Therefore, for all approaches, we use a subgroup size of 100 million trainable parameters as opposed to DeepSpeed's default size of 1 billion parameters per subgroup, which allows better load balancing for our approach. Similar to Turing-NLG 17.2B, GPT-3 175B, BLOOM-176B [45], we used activation checkpointing to reduce the GPU memory utilization at the expense of 33% additional recomputations during the backward pass [30]. This is a popular choice for training with scarce GPU memory. Nevertheless, our approach is complementary to activation checkpointing and would produce similar results without activation checkpointing as well.

Key Performance Metrics. We use the following metrics for evaluating the aforementioned approaches: (1) average time to run a single training iteration (broken down by the duration of the forward pass, backward pass, and update phase); (2) update throughput (expressed as millions of parameters updated per second); (3) effective read/write throughput observed while fetching and flushing disk-offloaded subgroups; and (4) distribution of the optimizer states across different tiers. These metrics are important to understand the end-to-end performance and scalability of LLM training using our approach vs. state of art, as well as to highlight important intermediate steps that influence the end-to-end results.

4.2 Results: Model Size Scalability

We first measure the iteration time, broken down by forward, backward, and update phase durations for increasing model sizes as listed in Table 2 on a single 4×H100 node of Testbed-1 (Table 1). Each experiment runs for 10 iterations, of which the first 2 are warmups, and the average from 8 iterations is reported. We vary the model size between 40B and 120B such that all FP16 parameters and gradients for a single subgroup fit within the aggregated 320 GB of GPU memory. With increasing model size, the intensity of offloading increases as well, thus highlighting the effectiveness of the compared approaches as the offloading pressure increases (at 120B parameters, the optimizer state reaches 1.8 TB).

As observed in Figure 7, the iteration duration follows an increasing trend for increasing model sizes (with slight exceptions for 52B vs. 40B and 120B vs. 100B, as they have fewer transformer layers but more hidden dimensions). As expected, due to offloading, the update phase is the longest, while the forward pass is almost negligible in comparison. For DeepSpeed ZeRO-3, the backward pass begins to be noticeable, while our approach reduces it to a negligible level. We also observe that our approach accelerates the update phase by up to 2.4×, leading to iterations that are overall 2.7× faster compared with DeepSpeed ZeRO-3.

To further explain these results, we depict in Figure 8 the update throughput, which for reference is ~40000M params/s on the GPUs and ~ 8000M params/s on the CPUs when the model parameters and optimizer states are fully available in the GPU and host memory, respectively. With offloading the update throughput drops by an order of magnitude, even on the CPUs, which confirms that the bottleneck is not on the compute side, but rather due to the slow I/O to the NVMe and/or PFS. Furthermore, the update throughput stays relatively stable for all model sizes (the update throughput per subgroup remains unchanged, only the number of subgroups changes for different models). Overall, *MLP-Offload* achieves an update throughput 1.8×–2.4× higher than DeepSpeed ZeRO-3.

4.3 Results: I/O and Storage Tier Load

As mentioned previously, the update throughput is subject to I/O bottlenecks. Therefore, we discuss next the I/O throughput sustained by *MLP-Offload* vs. DeepSpeed ZeRO-3. As opposed to the microbenchmarks discussed in Figure 4, running an end-to-end training overlaps I/O (asynchronous prefetch and flush) with computations, which introduces additional overheads. Specifically, the I/O throughput is computed as $2 \times subgroup\_size\_bytes/(read\_time + write\_time)$, averaged over all subgroups. The size is doubled because every subgroup needs to be both read and written. Figure 9 depicts the aggregated I/O throughput of all subgroups for an increasing model size. Interestingly, the DeepSpeed ZeRO-3 approach

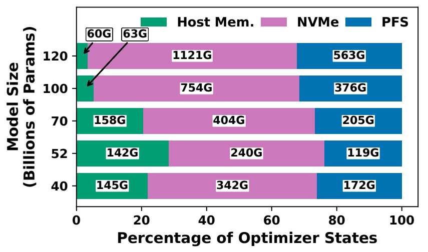

Figure 10: Distribution of optimizer states across different tiers.

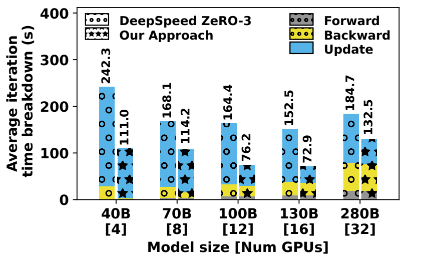

Figure 11: Weak scaling: Iter. time for increasing model sizes with # GPUs.

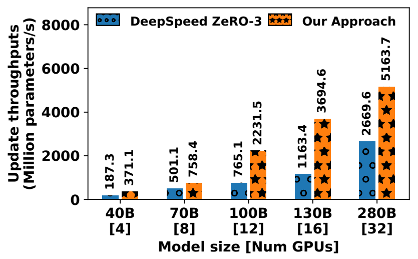

Figure 12: Weak scaling: Update thru. for increasing model sizes with # GPUs.

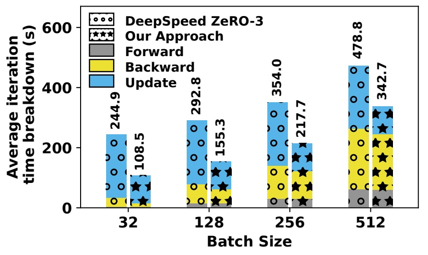

Figure 13: Average iteration time of different batch sizes for the 40B model.

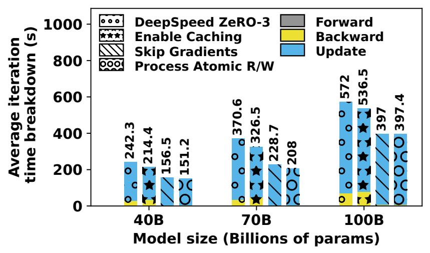

Figure 14: Performance ablation on node-local NVMe.

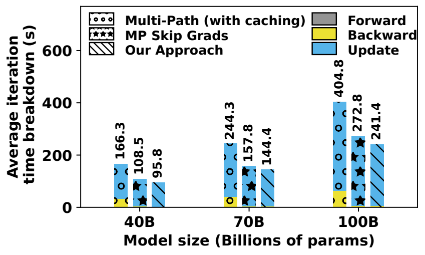

Figure 15: Performance ablation on node-local NVMe and PFS.

demonstrates I/O throughput of ∼ 3.2 GB/s, which is much lower than the peak write speed (5.3 GB/s) of the NVMe on Testbed-1 (Table 1). This is because parallel multi-threaded reads and writes from all processes create contention on the CPU-NVMe interconnect (which is PCIe in this case) and on the NVMe storage subsystem. Conversely, *MLP-Offload* alleviates the pressure on the local NVMe by using a PFS, thus benefiting from both multi-tier offloading and better utilization of individual tiers, ultimately being 2.6× faster. The effective I/O throughput with *MLP-Offload* decreases slightly with increasing model size because smaller fractions of the optimizer states can be cached on the host memory, thereby reducing the effectiveness of caching. However, *MLP-Offload* still achieves ∼2× I/O speedup for larger models compared to the DeepSpeed ZeRO-3 approach.

To explain the impact of multi-tier offloading, we highlight the distribution of the optimizer states across the different storage tiers. A major chunk of the host memory is consumed by the DeepSpeed runtime for setting up the ZeRO-3 specific data structures that consume 250-350 GB of host memory, proportional to the model size, as reported by DeepSpeed's memory estimator [7]. The remaining host memory is used for caching subgroups and asynchronous I/O operations. Figure 10 depicts the percentage of optimizer states distributed across the host memory, local NVMe, and the PFS at every iteration. Unlike *MLP-Offload*, DeepSpeed ZeRO-3 experiences cache thrashing on the host buffers due to sequential-ordered subgroup updates and does not utilize the PFS, forcing all optimizer states to be read and written back to local NVMe in each iteration. The fraction of the optimizer states distributed across the local NVMe and the PFS confirm the effectiveness of our performance model (§ 3.3), showing a 2:1 NVMe to PFS offloading that is consistent with the read and write throughputs in Table 1.

4.4 Results: Weak Scalability

Next, we study the weak scalability of *MLP-Offload* by varying the model sizes proportionally to an increasing number of nodes. As discussed in § 2, DeepSpeed does not implement pipeline parallelism in combination with the sharding of model parameters and optimizer state. Therefore, we use tensor parallelism across the four co-located GPUs on the same compute node, and data parallelism between the compute nodes.

ZeRO-3's parameter sharding across data-parallel ranks requires frequent scatter collectives leading to higher communication costs during forward and backward passes at the expense of memory savings. Therefore, this scalability study is important to understand if higher communication costs offset the gains achieved by *MLP-Offload* in backward and update phases at scale.

For this experiment, we scale up to 8 nodes (32× A100-40GB GPUs) on Testbed-2 (Table 1). The largest model that fits within the aggregated GPU memory in FP16 format is selected from Table 2, ensuring a proportional increase in model size with node count: 40B (1 node), 70B (2 nodes), 100B (3 nodes), 130B (4 nodes), and 280B (8 nodes). Figure 11 shows that iteration time decreases for an increasing number of GPUs. In this case, the communication overheads in forward and backward passes are not as significant compared to the I/O bottlenecks of offloaded optimizer state updates due to the fast interconnect between the compute nodes (e.g., Slingshot, Infiniband) that are typically available on HPC infrastructures. Furthermore, an increasing node count enables independent I/O to the local NVMe, which accelerates the subgroup updates. Consequently, *MLP-Offload* achieves up to 2× faster iterations than DeepSpeed ZeRO-3, even at scale. To explain this trend, we analyze the update throughputs in Figure 12. As expected, update throughput scales with increasing resources (I/O bandwidth and CPUs);

and when correlated with the iteration duration this confirms that I/O is still the bottleneck.

In addition to demonstrating *MLP-Offload*'s scalability gains over ZeRO-3 with increasing model sizes, the weak scalability experiments also highlight the cost-effectiveness of NVMe-offloading with *MLP-Offload* as compared to the GPU-only training scenario. For instance, training the 70B model without offloading requires the aggregated memory of ~80 A100-40GB GPUs [24] and runs a single iteration in 24s. Conversely, as shown in Figure 11, with NVMe-based optimizer offloading, ZeRO-3 can run using 10× fewer GPUs, but takes 168s per iteration, i.e., 7× slower. In contrast, *MLP-Offload* is only 4.8× slower, thereby achieving a 5× slowdown while using 10× fewer GPUs– yielding a 2× improvement in cost-effectiveness compared to GPU-only training.

4.5 Results: Gradient Accumulation Scalability

We next study the impact of using gradient accumulation, which is a popular technique [45] to reduce the number of update phases (and thus the impact of offloading) by running multiple forward and backward passes before each update phase. While the number of iterations does not decrease, it is equivalent to running the training with larger mini-batches (an alternative that is not possible when the GPU memory is scarce). The goal is to show that despite the reduced frequency of the update phase, our approach still delivers a significant end-to-end speedup compared with the state-of-the-art.

We experiment with the 40B model running on Testbed-1 (4×H100 GPUs), which can accommodate a mini-batch size of 8 samples, beyond which we encounter out-of-memory on GPUs. Consequently, when running across 4 GPUs in a data-parallel fashion, when we run an update phase for and increasing number of forward and backward passes (1–16), the equivalent batch size increases in the range of 32–512. As observed in Figure 13, even when gradient accumulation is used to amortize the cost of expensive update phases, *MLP-Offload* still outperforms DeepSpeed ZeRO-3 by at least 40%.

4.6 Results: Ablation Study

In the last set of experiments, we perform ablation studies to understand the impact of each optimization proposed as part of our design principles (§ 3). We consider three models, i.e., 40B, 70B, and 100B, which represent small-scale, medium-scale, and large-scale models, respectively, on Testbed-1. Figure 14 depicts the accumulated impact (i.e. progressive activation) of each optimization when the optimizer state is only offloaded to the node-local NVMe. The approaches labeled as *Enable Caching*, *Skip Gradients*, and *Process Atomic R/W* correspond to the design principles discussed in § 3.2: cache-friendly subgroup reordering, delayed in-place mixed-precision gradient conversion, and optimized virtual tier concurrency control, respectively. As can be observed, progressive activation of each optimization further reduces the iteration duration, which means each optimization individually contributes to the speedup, resulting in up to 1.6× speedup vs. DeepSpeed even without a PFS. Figure 15 depicts the same accumulation of optimizations but with the PFS active (multi-path). In this case, activating all three optimizations is equivalent to our approach. Compared with Figure 14, the multi-path parallel I/O further speeds up the iteration by 1.6×, resulting in 2.5× faster iterations compared with DeepSpeed ZeRO-3.

5 Conclusions

In this paper, we present a novel technique, *MLP-Offload*, and its implementation as a library that can be integrated with state-of-the-art LLM runtimes that enable scalable training and fine-tuning. Specifically, we target the offloading of the optimizer state to a multi-level, multi-path memory and storage hierarchy to accelerate the training of large LLMs under GPU memory constraints. In this context, state-of-art approaches suffer from significant I/O bottlenecks with optimizer state offloading to storage tiers due to the large size of the full-precision optimizer states (8× larger than FP16 parameters), which spill beyond the capacity of the host memory (that is typically only 2× larger than the aggregated GPU memory) and therefore need to be offloaded to tertiary storage tiers (e.g., node-local NVMe devices), whose I/O bandwidth is orders of magnitude lower. To reduce these I/O bottlenecks, *MLP-Offload* proposes several design principles, such as multi-level multi-path asynchronous offloading, concurrency control for multi-path I/O, cache-friendly subgroup update reordering, and dynamic in-place mixed-precision gradient conversion. The design principles are implemented as a modular extension to DeepSpeed's offloading engine. Extensive evaluations on 40B–280B parameter models demonstrate 2.5× faster training iterations as compared to DeepSpeed ZeRO-3 for different configurations at scale. Encouraged by these results, we next plan to explore parallel I/O paths for next-generation Compute-Express Link (CXL) memory pools and the integration of *MLP-Offload* with other offloading runtimes, frameworks, and accelerators. Of particular interest is a deeper study on the behavior of globally shared alternative storage tiers under I/O competition, which is the case of parallel file systems and object stores. In this case, we plan to explore how to mitigate predictable fluctuations in I/O bandwidth.

Acknowledgments

This work is supported in part by the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research (ASCR) under contract DEAC02–06CH11357/0F–60169 and the National Science Foundation (NSF) under award no. 2411386/2411387, 2106635. Results presented in this paper are obtained using Argonne's HPC systems– ALCF Polaris, Joint Laboratory for System Evaluation (JLSE), and NSF's CloudLab and Chameleon testbeds.

References

[1] Amazon Web Services. 2025. Amazon FSx for Lustre Pricing. https://aws.amazon.com/fsx/lustre/pricing/. Accessed: 2025-02-05.
[2] Argonne Joint Laboratory for System Evaluation (JLSE). 2023. NVIDIA H100. https://www.jlse.anl.gov/nvidia-h100. Accessed: 2025-02-04.
[3] Argonne Leadership Computing Facility. 2025. Polaris. https://www.alcf.anl.gov/polaris.
[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[5] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2021. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1967–1981.
[6] Xiaoyu Chu, Daniel Hofstätter, Shashikant Ilager, Sacheendra Talluri, Duncan Kampert, Damian Podareanu, Dmitry Duplyakin, Ivona Brandic, and Alexandru Iosup. 2024. Generic and ML Workloads in an HPC Datacenter: Node Energy, Job Failures, and Node-Job Analysis. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 710–719.
[7] DeepSpeed Team. n.d.. Memory Requirements- DeepSpeed Documentation. https://deepspeed.readthedocs.io/en/latest/memory.html. Accessed: 2025-04-09.

[8] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Intl. Conference on Systems and Storage (SYSTOR)*. ACM, 120–127.

[9] 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. 2024. Efficient Training of Large Language Models on Distributed Infrastructures: A Survey. arXiv:2407.20018 [cs.DC] https://arxiv.org/abs/2407.20018

[10] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. AI and memory wall. *Micro* 44, 3 (2024), 33–39.

[11] Mikaila J. Gossman, Bogdan Nicolae, and Jon C. Calhoun. 2024. Scalable I/O aggregation for asynchronous multi-level checkpointing. *Future Generation Computer Systems* 160 (2024), 420–432.

[12] Matt Hamblen. 2023. Update: ChatGPT runs 10K Nvidia training GPUs with potential for thousands more. https://www.fierceelectronics.com/sensors/chatgpt-runs-10k-nvidia-training-gpus-potential-thousands-more.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. *GPipe: efficient training of giant neural networks using pipeline parallelism.* Curran Associates Inc., Red Hook, NY, USA.

[14] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. 2024. 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, 345–360.

[15] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2024. NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference. arXiv:2411.01142 [cs.DC] https://arxiv.org/abs/2411.01142

[16] Yao Jiang, Xinyu Yan, Ge-Peng Ji, Keren Fu, Meijun Sun, Huan Xiong, Deng-Ping Fan, and Fahad Shahbaz Khan. 2024. Effectiveness assessment of recent large vision-language models. *Visual Intelligence* 2, 1 (2024), 17.

[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[18] Joonhyung Lee, Jeongin Bae, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. 2024. To FP8 and Back Again: Quantifying the Effects of Reducing Precision on LLM Training Stability. arXiv:2405.18710 [cs.LG] https://arxiv.org/abs/2405.18710

[19] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*. 766–775.

[20] Xiang Li, Yiqun Yao, Xin Jiang, Xuezhi Fang, Chao Wang, Xinzhang Liu, Zihan Wang, Yu Zhao, Xin Wang, Yuyao Huang, Shuangyong Song, Yongxiang Li, Zheng Zhang, Bo Zhao, Aixin Sun, Yequan Wang, Zhongjiang He, Zhongyuan Wang, Xuelong Li, and Tiejun Huang. 2024. Tele-FLM Technical Report. arXiv:2404.16645

[21] Changyue Liao, Mo Sun, Zihan Yang, Kaiqi Chen, Binhang Yuan, Fei Wu, and Zeke Wang. 2024. Adding NVMe SSDs to Enable and Accelerate 100B Model Fine-tuning on a Single GPU. *arXiv preprint arXiv:2403.06504* (2024).

[22] Changyue Liao, Mo Sun, Zihan Yang, Jun Xie, Kaiqi Chen, Binhang Yuan, Fei Wu, and Zeke Wang. 2024. LoHan: Low-Cost High-Performance Framework to Fine-Tune 100B Model on a Consumer GPU. arXiv:2403.06504 [cs.DC] https://arxiv.org/abs/2403.06504

[23] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[24] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In *Proc. of the International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC'24)*. 13 pages.

[25] Avinash Maurya, Jue Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Breaking the Memory Wall: A Study of I/O Patterns and GPU Memory Utilization for Hybrid CPU-GPU Offloaded Optimizers. In *FlexScience'24: Workshop on AI & Scientific Computing at Scale using Flexible Comp. Infrastructures*.

[26] Avinash Maurya, Jie Ye, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Deep optimizer states: Towards scalable training of transformer models using interleaved offloading. In *Proceedings of the 25th International Middleware Conference*. 404–416.

[27] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. arXiv:1710.03740 [cs.AI] https://arxiv.org/abs/1710.03740

[28] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, et al. 2022. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. arXiv:2112.11446 [cs.CL] https://arxiv.org/abs/2112.11446

[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[30] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *SC'21: The 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis, USA, Article 59, 14 pages.

[31] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[32] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[34] Shuaiwen Leon Song, Bonnie Kruft, Minjia Zhang, Conglong Li, Shiyang Chen, et al. 2023. DeepSpeed4Science Initiative: Enabling Large-Scale Scientific Discovery through Sophisticated AI System Technologies. arXiv:2310.04610 [cs]

[35] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. Galactica: A Large Language Model for Science. arXiv:2211.09085 [cs.CL] https://arxiv.org/abs/2211.09085

[36] DeepSpeed Team. 2022. GitHub Issue: Delayed parameter update in ZeRO-Offload. https://github.com/deepspeedai/DeepSpeed/discussions/2461.

[37] HPCAI Tech. 2025. TensorNVMe: A Python library for transferring PyTorch tensors between CPU and NVMe. https://github.com/hpcaitech/TensorNVMe. Accessed: 2025-02-06.

[38] Alan Thompson. [n. d.]. Inside Large Language Models. https://lifearchitect.ai/models. Accessed: 2025-04-12.

[39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs]

[40] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288

[41] Jonathan Vanian and Kif Leswing. [n. d.]. ChatGPT and generative AI are booming, but the costs can be extraordinary. https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html. Accessed: 2023-03-15.

[42] VAST Data. 2023. The VAST Data Platform Whitepaper. https://www.vastdata.com/whitepaper. Accessed: 2025-04-12.

[43] Guanhua Wang, Masahiro Tanaka, Xiaoxia Wu, Lok Chand Koppaka, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2023. DeepSpeed ZeRO-Offload++: 6x Higher Training Throughput via Collaborative CPU/GPU Twin-Flow.

[44] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 364–381.

[45] BigScience Workshop. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100 [cs]

[46] Gang Xian, Wenxiang Yang, Yusong Tan, Jinghua Feng, Yuqi Li, Jian Zhang, and Jie Yu. 2024. Mobilizing underutilized storage nodes via job path: A job-aware file striping approach. *Parallel Comput.* 121 (2024), 103095.

[47] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414* (2022).

[48] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. GLM-130B: An Open Bilingual Pre-trained Model. arXiv:2210.02414 [cs.CL] https://arxiv.org/abs/2210.02414

[49] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[50] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM computing surveys (CSUR)* (2019).

[51] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[52] Xia Zhao, Limin Wang, Yufei Zhang, Xuming Han, Muhammet Deveci, and Milan Parmar. 2024. A review of convolutional neural networks in computer vision. *Artificial Intelligence Review* 57, 4 (2024), 99.

[53] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860.