Portus: Efficient DNN Checkpointing to Persistent Memory with Zero-Copy

Yuanhao Li[†*], Tianyuan Wu[†*], Guancheng Li[†], Yanjie Song[†], and Shu Yin[†‡§]
{liyh12022, wuty, ligch2022, songyj, yinshu}@shanghaitech.edu.cn
[†]Shanghaitech University, China
[‡] Shanghai Engineering Research Center of Intelligent Vision and Imaging, China

2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-8605-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDCS60910.2024.00015

***Abstract*—We introduce Portus, an efficient checkpointing system for DNN models. The core of Portus is a three-level index structure and a direct RDMA datapath that enables fast checkpoints between GPUs and persistent memory in a serialization-free way. Portus offers a zero-copy approach between GPU and persistent memory without involving main memory and kernel crossings to underlying file systems. Portus also applies an asynchronous mechanism to hide the checkpointing overhead in the model training procedures. We integrated a Portus prototype into a high-performance AI cluster with NVIDIA®V100 and A40 GPUs and Intel®Optane™persistent memory, then evaluated its performance in both single-GPU and multi-GPU large model training scenarios. Experiment results show that compared to a state-of-the-art checkpointing system, Portus achieves up to 9.23× and 7.0× speedup in checkpointing and restoring, respectively. Portus achieves up to 2.6× higher throughput and 8× faster checkpointing operation on a large language model, GPT-22B.**

***Index Terms*—Checkpointing, Persistence Memory, RDMA, Systems for AI**

I. Introduction

Deep Neural Networks (DNNs) gained popularity in various machine learning domains, including translation [1], image recognition (e.g., ResNet [2], vision transformers [3], and VGG [4]), speech recognition [5], [6], and data mining [7]. To achieve better performance on these tasks, there has been an explosive growth in the number of model parameters and the scale of training datasets [8]. This trend has become particularly notable in the era of large models like GPT, which now dominate modern deep learning research. The demands of effectively training these large models that cannot be accommodated on a single GPU led to the widespread adoption of model parallelism techniques, which partition and distribute the model across multiple GPU nodes [9].

Efficiently training larger DNN models presents various challenges for deep learning systems, particularly in fault tolerance. Large model training often involves a large number of GPUs, and a single GPU failure can result in a system-wide crash, leading to a considerably higher frequency of crashes compared to traditional DNN training [10]. As a common approach, the checkpointing technique allows DNN models to save parameters and optimizer states as files for restoring after a crash. Although checkpoints prevent DNN models from redundant re-training from the beginning, it is time-consuming when models contain enormous parameters. In model parallel training, checkpointing performance becomes worse. This is because each node saves independent checkpoint files to a shared file system for collaborations with other nodes (as detailed in §II-A), which incurs the I/O contention and synchronization overhead. The inefficiency of checkpointing systems prolongs large DNN model training time because the new training steps will not commence until all the node-wide checkpoints are complete. The more GPU nodes involved, the longer the training steps wait.

Researchers attempt to address the efficiency of DNN checkpointing issues via scheduling algorithms and systemic techniques [11]–[13]. Some studies focus on reducing checkpoint file size via data compression (e.g., Deepsz [14]) or incremental checkpoints (e.g., Check-N-Run [15]), while others try to eliminate checkpoint stalls by overlapping I/O with computing (e.g., CheckFreq [16]). We conducted a breakdown analysis of the checkpointing operation of BERT models to a storage node equipped with persistent memory (a.k.a. PMEM) and 100Gbps InfiniBand network. The results imply that the DNN checkpointing inefficiency comes from serialization from GPU to main memory and redundant data copies from main memory to file systems (detailed in §II-B).

Studies also indicate that DNN model training requires higher checkpointing frequency to tolerate execution failures. Eisenman *et al.* reported that the failure intervals of over 60% of failed DNN training jobs on Facebook's cluster are less than one hour [15]. Evaluation analysis from Oobleck [17] and Bamboo [18] implied that a failure usually occurs every 10 minutes. To address this issue, researchers suggested the need for finer-grained checkpoint mechanisms [16] with mathematical analyses [19].

This dilemma leaves developers questioning whether to apply frequent checkpoints. Decreasing the checkpointing frequency can eliminate the overhead of data persistence but leads to longer total training time because the models have to restore more iterations after a failure. Increasing the checkpointing frequency can reduce the restore overhead but it still leads to longer training time because more checkpoint writes block the training for the next step.

We propose Portus to address the inefficiency of DNN

[*] These authors contributed equally to this work.
[§] Shu Yin is the corresponding author.

2575-8411/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDCS60910.2024.00015

checkpointing and support finer-grained checkpoints for DNN model training. The core of Portus is a three-level index structure and a direct datapath between GPU and PMEM that eliminates serialization to main memory and redundant kernel crossings to file systems. Portus also decouples the checkpointing and computing with an asynchronous mechanism that ensures multiple concurrent checkpointing tasks of multi-tenant or distributed large models training.

The main contributions in this paper include:

- We propose Portus, an efficient DNN checkpointing system with minimal overhead. With a three-level index structure and a peer-to-peer datapath, Portus achieves zero-copy between GPUs and PMEM.
- We implement a prototype of Portus and integrate it into an AI cluster that equips Intel®Optane™persistent memories, NVIDIA®V100 and A40 GPUs, and Mellanox®100Gbps InfiniBand network.
- We conduct comprehensive experiments to evaluate the efficacy of Portus on conventional DNN models trained on a single GPU and an LLM model on multiple nodes. The results demonstrate a significant reduction in checkpointing and total training time.
- Portus can be seamlessly integrated into common Deep Learning frameworks (e.g., PyTorch) and provides an efficient and user-friendly solution for DNN checkpointing and restoring.

The rest of the paper is organized as follows. Section II provides the background and motivation of this research. Design and implementation details of Portus are proposed in Section III and Section IV, which is followed by an evaluation shown in Section V. While Section VI discusses some lessons learned from this research, and Section VII summarizes the related works. Finally, Section VIII concludes this paper.

II. Background and Motivation

In this section, we present a brief overview of the DNN training process and discuss the challenges of checkpoint structures with increasing model scales.

A. Why Checkpoint for DNN models?

As the scale of DNN models continues to increase, the number of parameters is getting enormous. For example, LLaMa has 65 billion parameters [20], GPT-3 features 175 billion ones [21], and PaLM pushes the number to 540 billion [22]. It makes fault tolerance challenging because days or weeks of model training escalate the failure rate of software and hardware. A study shows that software failures occur every 14 to 30 hours in deep recommendation systems, which introduce *43%* slowdown to handle training restarts [19]. The Meta AI team encountered more than 110 hardware failures during the training of OPT-175B [23]. Hardware failure is critical because it necessitates system-level reboots for every malfunction incident.

Although some studies attempt to use main memory for checkpoints [10] or exchange compute with storage [18], storing model and optimizer states as a checkpoint file to disks is prevalent for fault-tolerance. Nonetheless, persistent checkpoint files of large models are more challenging, especially when modern frameworks like Megatron [9] and DeepSpeed [24] employ parallelism techniques to train a DNN model across multiple GPUs.

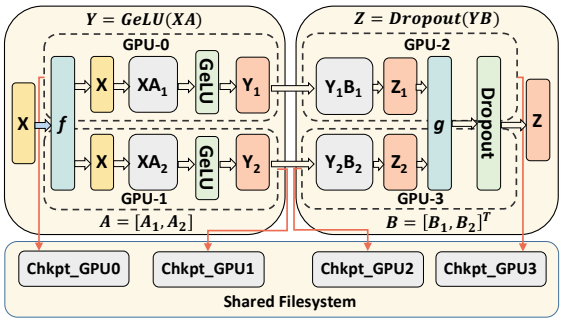


Fig. 1. An example of tensor parallel and pipeline parallel

Figure 1 demonstrates an example of training a Multi-Layer Perceptron (MLP) model across four GPUs. The pipeline parallelism divides the model into two layers, the first of which computes $Y = GeLU(XA)$ and the second processes $Z = Dropout(YB)$. The tensor parallelism then partitions them in each layer, with A into $[A_1, A_2]$ and B into $[B_1, B_2]^T$. The model is finally distributed across four GPUs. Tensor and pipeline parallelism that partition and distribute a large model across multiple GPUs make every processor generate a checkpoint file of its own. The up-scaling of a model leads to the increase of both the file size and file quantities, which causes the explosive growth of a checkpoint volume. Besides, all the checkpoint files must be retrieved and aggregated to facilitate a restart from a failure, which is time-consuming for not only I/O time but also synchronization overhead to form a complete checkpoint instance.

Motivation1: Large DNN models introduce additional requirements for checkpointing when multiple GPU nodes are involved. Existing solutions have limits in handling concurrent checkpoint files of a distributed DNN model that runs across multiple GPUs. Therefore, there is a need to design a structure to manage multiple shared checkpoint files of a DNN training for both checkpointing and restoring.

B. Why Users Do NOT Checkpoint?

Although checkpointing is feasible for the DNN model's fault tolerance, users prefer retraining the model over checkpoints because of the time-consuming data storage and retrieval.

We ran an experiment to study the performance overhead of checkpointing under existing frameworks, leveraging cutting-edge hardware and software. The experiment dumped DNN checkpoints from NVIDIA V100 GPUs to a storage server equipped with Intel®Optane™Persistent Memory, which offers non-volatile byte-addressable data access with ultra-low latency and high throughput I/O. The network connection is with Mellanox®InfiniBand, which supports up to 100Gbps RDMA bandwidth. We chose three popular DNN models to

generate checkpoint files with the frequency according to the state-of-the-art CheckFreq [16] (i.e., one checkpoint every 83 iterations for VIT, one checkpoint per 100 iterations for GPT-10B and GPT-22.4B).

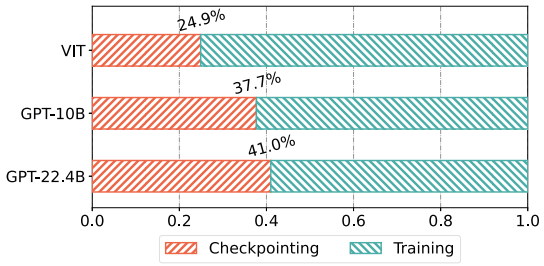


Fig. 2. DNN checkpointing time costs (normalized)

We learned from the results that a checkpointing operation weighs at least 24.9% of the total time (shown in Figure 2). As the model goes upscaling, the checkpointing overhead becomes more significant (up to 41%). Our further profiling studies reveal that the kernel reason behind the inefficiency of checkpointing is the serialization from GPU memory to main memory and the redundant data movements from main memory to file systems (shown in Figure 3). Figure 3 demonstrates the DNN checkpointing datapath from GPU memory to a PMEM-enabled BeeGFS file system in our profiling studies. Step 1, A DNN model copies the layered parameters and optimizers from GPUs to main memory. Step 2, the DNN training framework (e.g., PyTorch) adds metadata headers to the tensors in each layer, serializes them, and packs them into a checkpoint file, then triggers syscall to write the file to BeeGFS Client on the compute node. Step 3, BeeGFS Client transfers the file to BeeGFS Daemon, which runs in the main memory of a storage server via RDMA. Step 4, BeeGFS Daemon performs direct access (DAX) write to persist the file on the underlying file system.

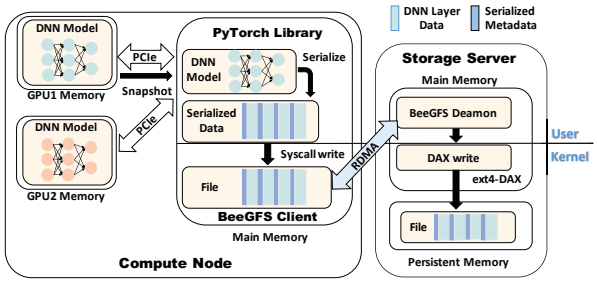


Fig. 3. Traditional distributed checkpointing datapath

This results in at least three redundant data copies: one from GPU memory to the main memory on the compute node, one from the main memory on the compute node to the main memory on the storage node, and one from the main memory on the storage node to PMEM. It also causes three crossings between user and kernel mode: one serializes data to a checkpoint file on BeeGFS Client, the second transfers it from BeeGFS Client to BeeGFS Daemon, and the third writes the checkpoint to PMEM.

TABLE I
DNN CHECKPOINTING OVERHEAD

Operation	Percentage (%)
GPU to Main Memory	**15.5%**
Serialization	**41.7%**
Transmission (RDMA)	30.0%
Server DAX write	12.8%

We conducted a breakdown analysis of the checkpointing process to determine the time cost of each operation. As depicted in Table I, copying the DNN model from GPU to the compute node's main memory and model serialization account for over 57.2% of the checkpointing time. In contrast, RDMA and DAX write operations in the storage node's kernel consume the remaining 42.8% of time. This breakdown analysis provides insight into the inefficiency of checkpointing I/O and suggests optimization objects by eliminating serializations and redundant copies.

Motivation2: Existing checkpoint approaches suffer inefficient datapath from serialization and redundant copies for DNN models. There is a need to design a prompt checkpoint datapath between GPUs and persistent storage systems to support up-scaled DNN model complexity and prolonged training time.

III. DESIGN

A. Design Goals

First of all, Portus should provide a fast peer-to-peer datapath to transfer checkpoint files between GPUs and PMEMs. Second, it should manage the checkpoint data structure to enhance finer-grained checkpointing without serialization and redundant data copies. Third, Portus should be efficient and flexible for large DNN models with distributed model parallel training.

B. Architecture Overview

Figure 4 illustrates the architecture of Portus, which consists of a client library as an extension of distributed deep learning frameworks (e.g., PyTorch, DeepSpeed, Megatron-LM) and a user-space daemon to manage the metadata processing and data transmission of checkpoints. Portus Client runs in the main memory on a compute node, and Portus Daemon is hosted in the main memory on the storage server.

Upon a new DNN model training job, Portus Client collects pointers to each tensor on the pre-allocated GPU memory by the DNN framework. Portus Client then registers the GPU address space for each tensor as an RDMA memory region (a.k.a. MR) using NVIDIA Peer Memory [25]. Every MR holds a unique remote key. After the registration, Portus Client gets remote keys for MRs and aggregates them with the metadata of layers one-to-one correspondingly into a packet to describe a DNN model. Every metadata consists of the layer name, data type, and shape. Finally, Portus Client sends the

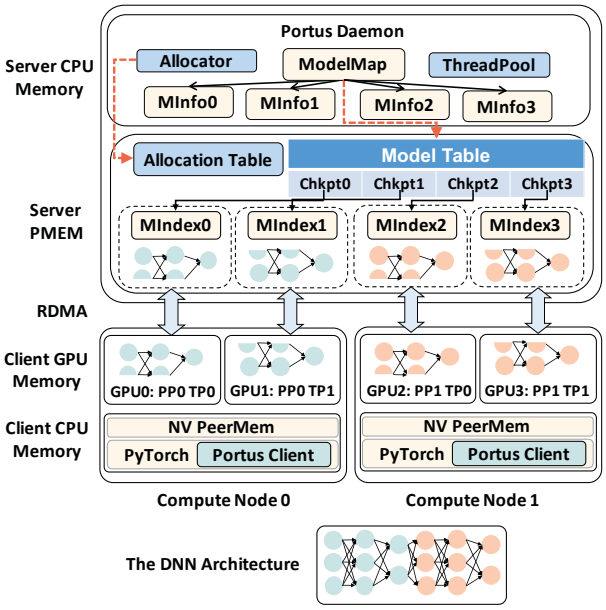


Fig. 4. Architecture Overview

packet to the Portus storage server by TCP socket via IPoIB protocol.

With the packet, Portus Daemon on the server dispatches it to an available worker from ThreadPool and maintains a three-level index structure (a.k.a. ModelTable-MIndex-TensorData). The worker inserts the DNN model name into ModelMap for lookups and creates a set of indexes (i.e., MIndex) for PMEM to map to the DNN checkpoint structure under the guidance of the received packet. The worker then allocates PMEM regions for each DNN tensor. The Allocator records the allocation status of each PMEM region in AllocTable. For *single GPU models*, we map each MIndex to an individual model, and its corresponding TensorData contains all parameters for restoration. For *large models* trained with tensor and pipeline parallellism across multiple nodes and GPUs (as illustrated in Figure 4), we map each MIndex to a model shard on a specific GPU. The collection of all these shards forms the whole model states, which can be loaded together for restoration. At last, Portus is ready for peer-to-peer RDMA DNN checkpointing between GPU memory and PMEM. Please refer to detailed discussions in §III-D.

Upon persistent checkpoint files, Portus operates data transfer opposite to the ordinary checkpointing where the DNN model writes data to the server. Instead, Portus Daemon reads the checkpoint file proactively from the GPU memory to PMEM. Similarly, Portus Daemon writes a checkpoint file to the remote GPU memory for restoration instead of letting the DNN model read the file from the server. We first explain the details of Portus Datapath in §III-C.

C. Portus Datapath

Portus is to maintain a simple yet efficient transmission datapath, to reduce the number of data copies and user-kernel context switches. We discovered that the DNN model properties (i.e., #layers, tensor shapes, data types, GPU addresses) are pre-determined and fixed during training. The structure of checkpoint files is also pre-determined hence can be constructed on the storage server before the beginning of a training iteration. In this way, Portus can map the checkpoint structure on the PMEM to the DNN layers on the GPU memory by RDMA memory regions. And then, Portus performs peer-to-peer transmission between GPU memory and PMEM in user space via RDMA verbs and NVIDIA®PeerMem kernel module.

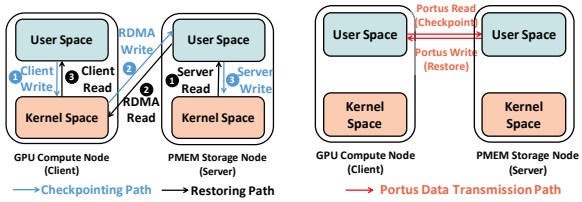
Figure 5(b) demonstrates the Portus datapath in checkpointing and restoration of a DNN model. Upon a DNN checkpointing, Portus Client informs Portus Daemon with the word "`DO_CHECKPOINT`" via a TCP socket. The server then performs an *one-sided RDMA read* operation to fetch each tensor from the remote GPU memory to PMEM. Portus Daemon notifies the completion of pulling to Portus Client via the same TCP socket. Similarly, Portus writes a checkpoint file to the remote GPU memory for restoration instead of letting the DNN model read the file from the server.

We demonstrate the traditional datapath on the bases of BeeGFS as a reference in Figure 5(a). Compared to the straightforward datapath of Portus, there are at least three kernel crossings in a traditional checkpointing– 1) Client DNN framework writes the serialized DNN model to BeeGFS via syscall write, 2) BeeGFS client module dispatches this write request out of client's kernel as RDMA writes to the storage server, 3) the file is persistent from user space to the kernel space on PMEM via DAX write. The restoring operates an inverse datapath that also has three kernel crossings.

Portus has two advantages: 1. it simplifies the datapath because pulling does not trigger kernel crossings to VFS; and 2. it decouples checkpointing with training, thereby no need to suspend the training process for the completion of a checkpoint. Besides the advantages of the random access performance of PMEM via RDMA, Portus implements an efficient index structure to store a checkpoint file without a dedicated serialization process.

D. Data Management on Persistent Memory

With the help of the straightforward peer-to-peer datapath, Portus can support rapid checkpointing, thereby making finer-grained multi-tenant model training foreseeable. We first



(a) Checkpointing and Restoring datapath with distributed filesystems (b) Optimized checkpointing and Restoring datapath with Portus

Fig. 5. Checkpointing and Restoring datapath comparison between distributed filesystems and Portus

present the three-level index structure in Portus for fast, finer-grained, and multi-tenant checkpointing then discuss the two common data issues, namely, crash consistency and garbage collections.

1) Index Structure of Persistent DNN Model: We design a three-level index structure in PMEM (i.e., ModelTable-MIndex-TensorData) to manage multiple checkpoints that belong to diverse DNN models (shown in PMEM in Figure 4). The root level *ModelTable* is a sorted array that stores the mapping relationship between a DNN model name (`model_name`) and the location to the second level index *MIndex* (`info_offset`). Such the array turns into a red-black tree structure in the main memory called *ModelMap* (shown in Portus Daemon in Figure 4). ModelMap is to quickly look up and locate the target model, each entry of which is a pair of key-value, where the key is `model_name`, and the value is `info_offset`. The dashed arrows between ModelMap and ModelTable represent persistent pointers from main memory to PMEM. Note that the Portus server manages PMEMs in `devdax` mode, meaning that users can perform direct access (i.e., DAX) to PMEM via `mmap` and detour kernel file systems [26]. Portus only maintains ModelMap and MInfo in the main memory, and keeps TensorData on PMEM for peer-to-peer transmission without additional data copies.

The value of the second level index is a MIndex record that contains metadata of a DNN model TensorData. A MIndex record consists of layer quantities of a model, the name of each layer, data type, tensor shape, size of each tensor, and address of each tensor on PMEM (i.e., persist pointers to data blocks). For example, MIndex for a BERT-large model is:

```
{ layers=397, tensor1:(name=Bert.embedding,
dtype=float32, shape=(512, 1024),
size=524288, paddr=0xffff0000),
tensor2:..., tensor3:... }.
```

The value of the last level index is a contiguous PMEM region representing TensorData, which is registered as an RDMA MR for data transmissions. Upon checkpointing, Portus Daemon pulls all tensors of a DNN model via Portus Datapath from GPU memory as a set of TensorData accordingly to PMEM, which is interpreted as a checkpoint file. Note that only ModelTable is shared while each worker thread is independent in Portus, meaning that a worker holds its own MIndex record and TensorData. We apply the *compare&swap* intrinsic to ensure the lock-free of the whole system for high concurrency.

2) Consistency and Space Management: The conventional approach to handle crash consistency is to write a checkpoint in a new file and replace the old checkpoint when the latest one is done. But it is not efficient for Portus because each time it needs to allocate space on PMEM and initializes a new RDMA connection. Inspired by the copy-on-write technique, we design a double mapping mechanism to address the crash consistency problem in Portus (demonstrated in Figure 6).

Upon starting a new DNN training job, the Portus Daemon creates two identical-structured checkpoints on PMEM to hold the last two versions of the DNN model. To avoid frequent

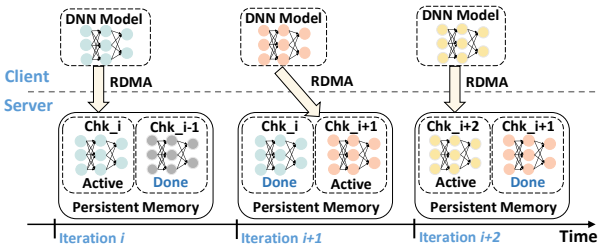


Fig. 6. Data consistency of checkpoints

operations on PMEM and RDMA connections, Portus assigns an `active` flag to the checkpoint whose data transmission has not been completed yet. After successful transmission, it sets the `active` flag to `done`, indicating that this checkpoint version is ready for restoration. This technique ensures that there is at least one valid checkpoint version present on PMEM for recovery, thereby guaranteeing fault tolerance efficiently.

To prevent potential waste of space on PMEM due to having two checkpoint versions, we have designed a repacking tool to address the issue. There are two possible scenarios that can result in invalid checkpoints: (1) when a training job completes, only the last checkpoint version is valid, while the other is outdated, and (2) when a training job crashes during checkpointing, the last checkpoint version is still active, but its data is incomplete or collapsed. To address this, the repacking tool aggregates the valid checkpoints and frees up more space on PMEM, as illustrated in Figure 7. This tool is not required to be used frequently, as the capacity of PMEM is typically in terabytes, which is sufficient to store thousands of models' checkpoints. Moreover, the repacking overhead can be negligible as it is only triggered when the available space on PMEM is low and takes only a few minutes to complete, which can be done in the background with an overlap of training.

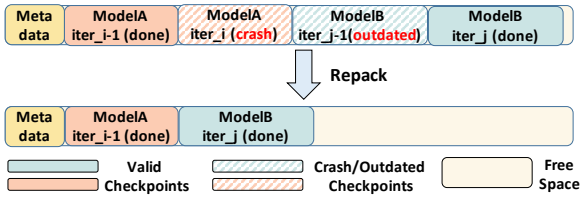


Fig. 7. Persistent memory repacking

E. Asynchronous Checkpointing

Figure 8 illustrates a typical DNN training iteration procedure that consists of three phases: forward pass (**F**), back-propagation (**B**), and parameter updates (**U**). The parameters in each tensor are fixed in the forward and backward propagation and only changes in the update phase. Hence, if the checkpoint can be persisted before the parameter updates, the training process will no longer need to wait for I/O. Portus can further reduce the checkpointing overhead by operating it in an asynchronous way, meaning that Portus can decouple the checkpointing from training iterations so that there is no need to suspend any iteration for checkpointing.

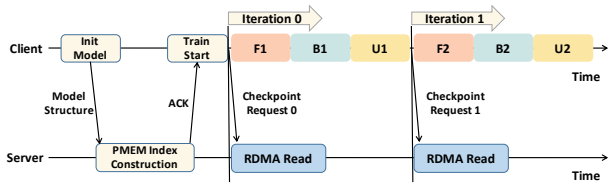


Fig. 8. Async Server-Client Communication

Figure 9 demonstrates training timeline differences amongst the ordinary PyTorch built-in checkpointing policy, a state-of-the-art CheckFreq policy [16], and two Portus policies. We can see that Portus achieves much better training efficiency due to the eliminated serialization for snapshots and much less persistent time costs in both synchronous and asynchronous ways (shown in Figure 9(c) and (d)). The asynchronous version in Figure 9(d) introduces minimal training stalls that optimize training efficiency in the single-tenant scenario and support higher concurrency in the multi-tenant case.

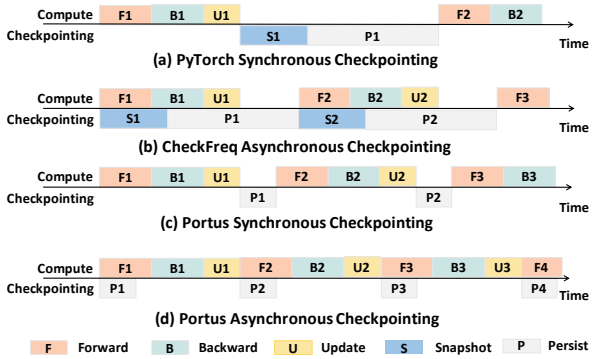


Fig. 9. Training Timeline Comparison

F. Model Recovery

Original PyTorch uses a naive recovery mechanism where users read checkpoints from storage using `torch.load()` and then restore the DNN model from this checkpoint. However, this approach is suboptimal due to redundant data copies and high model reconstruction overhead. Although GPU Direct storage enables loading checkpoints directly from storage to GPU memory, the deserialization overhead of structured files to DNN models still makes restoring inefficient. In contrast, Portus provides an efficient restoration solution that is similar to checkpointing, but the data flow for recovery is to write a valid checkpoint file from the Portus PMEM server directly to the GPU memory in the compute node.

First, Portus Client initializes an ”empty” model on GPU without tensor weights transferred. Similar to checkpointing, Portus Client then registers the GPU memory regions as RDMA MRs and informs the server with `rkeys` for restoring. Portus Client sends a restore request to Portus Daemon on the storage server. Portus Daemon writes the requested file to the GPU memory on the compute node via the peer-to-peer *one-sided RDMA writes* and notifies Portus Client upon the completion of data transmission.

IV. Implementation

a) Portable Implementation.: Portus Server is written in 3000 LOCs of C++ and built using CMake. The only dependency of Portus Daemon is the RDMA-supported NIC. Our implementation requires Infiniband, but users can use RoCE for RDMA on existing network infrastructure. Additionally, upon the absence of PMEM detected on the server, Portus can use DRAM as alternatives. Portus Client is written in Python and C++ (in additional 1,000 LOCs) as an extension of PyTorch. Users can download the source code and run `python setup.py install` to install it. After installation, users can use the user-friendly Python interfaces as any other Python package. We plan to release this extension to Python package managers like `pip` or `Anaconda` [27] for better portability.

b) Easy Sharing.: Modern AI researchers often share DNN model epochs (or checkpoints) with other community users after training, requiring these shared models to be stored in general formats like pickle or HDF5. To meet this demand, we implement a convenient command line tool `Portusctl` for Portus users to manage and share DNN models on the persistent memory. Users can use `Portusctl view DEVICE` to view all models stored on a PMEM device. They can select and dump desired model checkpoints out of PMEM as general DNN checkpoint formats using `Portusctl dump CHECKPOINT FILENAME`. The dumping is also efficient because of the well-indexed data structure and high I/O throughput of PMEM. In this way, Portus is compatible with other existing deep learning frameworks like PyTorch, TensorFlow, and Caffe [28].

V. Evaluation

In this section, we first analyze the characteristics of Portus Datapath between different devices (i.e., GPU memory-PMEM, main memory-PMEM). We then evaluate the performance and efficacy of Portus on checkpointing and restoring with widely-used DNN models.

We seek to answer the following questions: 1) Is the brand-new Portus Datapath effective and efficient in transmitting data between compute and storage nodes? (§V-B); 2) How well does Portus optimize the checkpointing and restoring operation time? (§V-C); 3) How does Portus achieve more efficient checkpointing? (§V-D); 4) How much does Portus enhance the distributed large language model (LLM) training with a model parallel on a real-world AI cluster? (§V-E).

A. Experimental Setup

We evaluated the performance of Portus using an AI cluster as clients, and a storage node with PMEM as server.

Compute Node (denoted as *Client*): The cluster contains two types of clients, namely *Client-Volta* and *Client-Ampere*.

Client-Volta equips two 64-Core AMD®EPYC™7742 @2.25GHz, $32 \times 32GB$ DDR4-3200MHz DRAM, and four

NVIDIA®V100 GPUs and a Mellanox®ConnectX-5 100Gbps Infiniband RNIC via PCIe4.0 interfaces.

Client-Ampere equips two 64-Core Intel®Xeon®Gold 5318Y @2.1GHz, $24 \times 32GB$ DDR4-3200MHz DRAM, and eight NVIDIA®A40 GPUs and a Mellanox®ConnectX-6 100Gbps Infiniband RNIC via PCIe4.0 interfaces.

We configure a BeeGFS client (ver. 7.3.2) on Clients to enable a remote BeeGFS file system. Each Client is with RDMA and NVIDIA®GPU-Direct Storage enabled. Each client also equips NVMe SSDs with an ext4 filesystem for local storage.

Storage Node (denoted as *Server*): The Server is an AEP storage server, which features two 36-core Intel®Xeon®Gold 6240L @ 2.6GHz, $6 \times 32GB$ DDR4-2933MHz DRAM, $6 \times 256GB$ Intel®Optane™DC PMEM (1.5TB in total), a Mellanox®ConnectX-5 100Gbps Infiniband RNIC, running Linux 5.15.0. We configure three PMEMs to one namespace in *fsdax* mode and formatted an ext4-DAX file system on it. We stack a BeeGFS server (ver. 7.3.2) on the ext4-DAX with RDMA and NVIDIA®GPU-Direct Storage enabled (denote as BeeGFS-PMEM). We configure the other half of PMEMs to *devdax* mode to enable Portus server to direct access PMEM bypassing the kernel and file systems (denote as Portus). The connection between Clients and Server is a Mellanox®MSB7800 100Gbps switch.

TABLE II
DNN MODELS SPECIFICATIONS

Model	Layers	Params	Size
Alexnet	16	61.1M	233MiB
ConvNeXt_base	344	88.6M	338MiB
ResNet50	161	25.6M	97MiB
Swin_b	329	87.8M	335MiB
VGG19_bn	70	143.7M	548MiB
VIT_l_32	296	306.5M	1169MiB
BERT	396	336.2M	1282MiB

Models: We evaluate 76 DNN models that are widely used in computer vision and natural language processing domains, with their default batch size and other hyper-parameters. Due to the page limits, we only present experimental results of seven representative models in the paper (i.e., ResNet50 [2], Convnext_base [29], Alexnet [30], VGG19_bn [4], VIT_l_32 [3], and Swin_b [31] on Imagenet [32], and Bert-Large-Uncased [1] on CLOTH-high [33]). Table II summarizes the model types, the number of model layers, the number of parameters, and the total size of parameters.

We also evaluate a large language model, GPT, using the Megatron model parallel framework on *Client-Ampere*. The number of total parameters of GPT ranges from 1.5 billion to 22.4 billion, which checkpoint size ranges from 6GB to 89.6GB. Evaluation results of the all the models are presented in the Appendix.

B. Analysis of Portus Datapath

The bandwidth and latency of the Portus Datapath between the *Client-Volta* and Server with different devices are presented in Figure 10. Specifically, Figure 10(a) and (b) demonstrate the read performance of four Portus Datapaths: the one between the main memory on the Server (denoted as *Server DRAM*) with the main memory on the Client (denoted as *Client DRAM*), the one between Server DRAM with GPU memory on the Client (denoted as *Client GPU*), the one between PMEM on the Server (denoted as *Server PMEM*) with Client DRAM, and between Server PMEM with Client GPU. On the other hand, Figure 10(c) and (d) show the write performance of the four datapaths, respectively. It is important to note that Figure 10(b) and (d) represent the two new datapaths for checkpointing only under the support of Portus. Before Portus, no servers could read data from the GPU without the intervention of DRAM for the checkpointing perspective.

We observe from the figures that DRAM or PMEM on Server as the storage target will not affect the checkpoint performance with Portus. It is because DRAM and PMEM are much faster than the network connection, even with the RDMA enabled. We want to highlight the efficacy of the Portus Datapath between Server and Client GPU. The development of this new datapath enables Portus to transmit checkpoint data with minimal overhead. Note that the peak bandwidth to access GPU memory is 5.8GB/s, which is 30% less than DRAM. This is because all the address mappings for GPU read are managed by a dedicated unit called base address register (BAR), which disables prefetching for GPUs. The intervention of BAR becomes a performance bottleneck to read GPU memory. We argue that even though the peak bandwidth to read GPU memory is not as fast as that to DRAM, it is still much faster than reading from state-of-the-art PCIe®4.0 NVMe SSDs [34] (which have a maximum sequential write bandwidth of 2.7GB/s). Furthermore, Figure 10(d) indicates that BAR does not affect writes.

We also observe that Portus maintains its peak bandwidth closer to the RNIC limit when the data transmission package size exceeds 512KB. In other words, Portus can take full advantage of RDMA network bandwidth to transmit data larger than 512KB. Recall the size of the seven popular DNN models in Table II that the average size of a model layer is around 2.5MiB, implying that Portus fits the DNN models checkpointing and restoring scenarios.

C. Checkpointing and Restoring Operations

To evaluate Portus's performance, we measured the checkpointing and restoring operation time for various models running on *Client-Volta*. We compare Portus's performance with not only the shared BeeGFS on PMEM (denote as BeeGFS-PMEM), but also the local ext4 filesystem on NVMe SSDs without networking overheads (denote as ext4-NVMe). Our results demonstrate that Portus significantly reduces the cost of checkpointing and restoring.

1) Checkpointing Time: Figure 11 illustrates the checkpointing time of various DNN models with different storage options, where Portus is 8.49x faster than remote BeeGFS-PMEM and 8.18x faster than local ext4-NVMe on average. This is due to the reduction in serialization overhead and

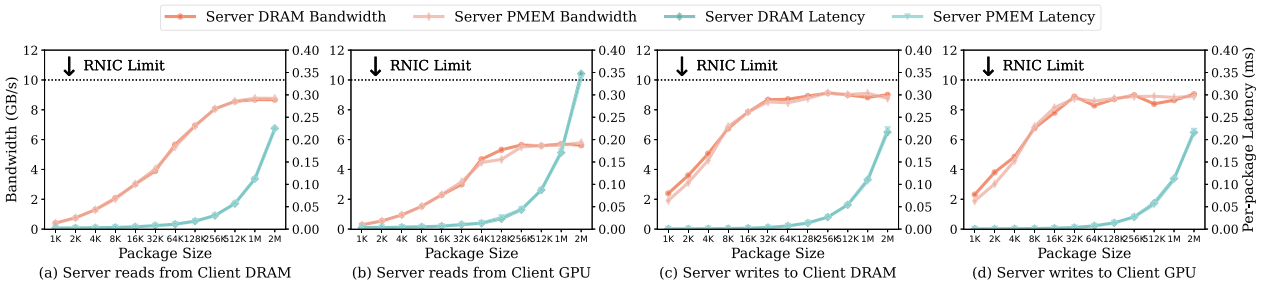


Fig. 10. Bandwidth and latency of Portus Datapath between different storage devices

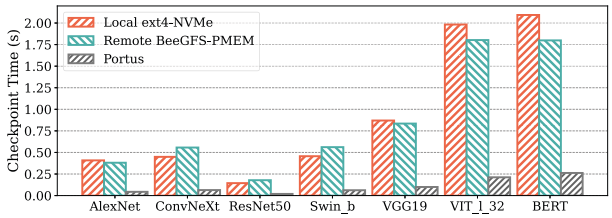


Fig. 11. Checkpointing time of different models

data copy time between different devices. Especially, Portus performs up to 9.23x faster than BeeGFS-PMEM in ResNet50 due to its higher metadata operation overhead introduced by small file writes (e.g., path resolution and permission check).

2) Restoring Time: We then evaluated the restoring performance of Portus and observed that it consistently outperformed BeeGFS-PMEM and ext4-NVMe. Figure 12 illustrates that, on average, Portus achieves 5.15x and 3.83x faster restoring time than BeeGFS-PMEM and ext4-NVMe, respectively. For ResNet50, Portus achieves an impressive 7.0x speedup compared to BeeGFS-PMEM. The improved performance is mainly attributed to the reduction in serialization and data copy overheads. However, it is worth noting that the performance gain of Portus on restoring is relatively lower than checkpointing because GPU-Direct Storage allows loading checkpoint files from storage devices to GPU memory without involving the main memory.

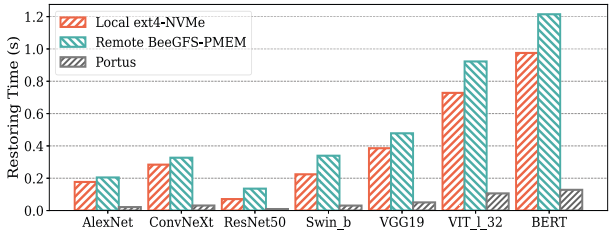


Fig. 12. Restoring time of different models

D. Checkpointing Breakdown

We perform a checkpointing time breakdown analysis of a Bert model to study the advantages of Portus in detail. We use `NVProf` to record GPU overheads, `strace` to monitor syscalls, and `ibdump` to keep track of RDMA requests. Figure 13 shows that Portus costs much less time compared to ext4-NVMe and BeeGFS-PMEM. We find that the RDMA data transmission dominates the Portus checkpointing time while a constant time for serialization and cuMemcpy contributes 46.5% and 57.2% of the total time to ext4-NVMe and BeeGFS-PMEM, accordingly. We also observe that the local ext4-NMMe is much slower than Portus because a local file system takes 53.7% of time to interact with block devices via kernel crossings. Besides, Portus costs less RDMA time than BeeGFS-PMEM because GPU-RDMA that Portus adopts is a time-efficient one-sided protocol while BeeGFS-PMEM uses a more time-consuming two-sided protocol RPCoRDMA [35].

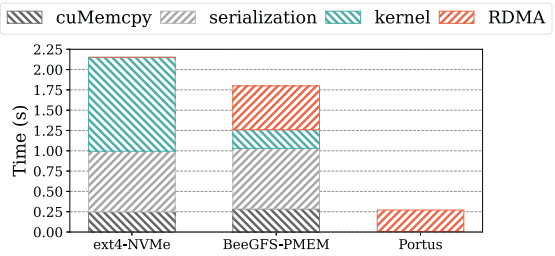


Fig. 13. Breakdown Analysis of Bert Checkpointing Time

E. Distributed Large Model Training

We evaluate Portus's performance in large model checkpointing with distributed model parallel training, which generates highly concurrent checkpointing requests with complex checkpoint structures. We integrate Portus into Megatron [9], a widely-used model parallelism framework for training large language models (LLMs). The GPT model is trained on two *Client-Ampere* nodes with 16 NVIDIA A40 GPUs. To demonstrate the scalability of Portus towards exascale models, we scaled the parameter size from 1.5 Billion to 22.4 Billion. Figure 14 shows that dumping a checkpoint file of a GPT model with 22.4 billion parameters using the `torch.save()` interface (used by traditional checkpointing and CheckFreq) to the shared BeeGFS storage takes more than 120 seconds, slowing down the overall training throughput and preventing finer-grained checkpointing policies. In contrast, Portus achieves an average speedup of 8.18× and takes only 15 seconds to dump the model with 22.4 billion parameters (89.6GB of data), highlighting its significant performance gain.

Note that it is the time improvement for one checkpoint instant, the performance gain of Portus is accumulated when the training keeps proceeding with a rapid checkpoint frequency. We foresee that Portus saves more than 1.5 hours for training if the model does checkpoints every half an hour and runs for 24 hours. The time saved from Portus becomes 10.7 and 23 hours if the model runs for one week and one month, respectively.

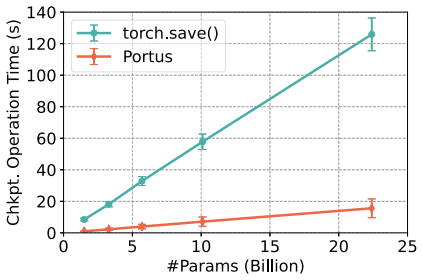


Fig. 14. Operation time comparison of dumping a checkpoint of GPT model via Portus and torch.save()

Portus's efficient checkpointing datapath and management structure contribute to higher training throughput and finer-grained fault-tolerance guarantees. As illustrated in Figure 15, Portus improves the training throughput of the GPT model with 22.4 billion parameters by 2.6×. We also foresee Portus supporting 14,400 more iterations than the state-of-the-art CheckFreq if the model runs for 24 hours. This is due to the dominance of checkpointing costs when applying finer-grained checkpointing policies. Portus reduces this cost significantly with its efficient three-level structure and zero-copy, serialization-free datapath.

Figure 16 implies that Portus also enriches GPU utilization because the checkpointing optimization reduces stalls for I/O. According to the 500-second GPU utilization profiling trace of training the GPT-22.4B model in the figure, we can see that Portus assists in achieving the average utilization of 76.4%, compared with less than 43% by CheckFreq.

VI. Lessons Learned

In this section, we summarize three lessons that we learned from this research.

Checkpointing data path should be condensed as much as possible: Traditional checkpointing data path from GPU to

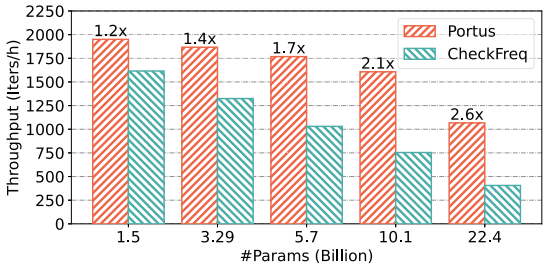


Fig. 15. Overall training time comparison of GPT model via Portus and CheckFreq

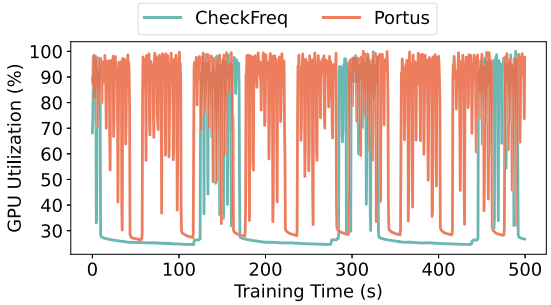


Fig. 16. GPU utilization of training the GPT model with 22.4 billion parameters with Portus and CheckFreq

a storage server slows down applications because of multiple data copies and context switches between user and kernel mode. Therefore, systems can benefit from fast hardware and protocols (i.e., PMEM, InfiniBand, and RDMA) when the data path between compute and storage nodes shortens as much as possible by reducing unnecessary data copies. Portus optimizes the checkpointing efficiency because it offers a peer-to-peer data path between GPU memory and PMEM. Portus avoids data copies between user and kernel spaces on both compute and storage nodes via an inverse data access pattern (i.e., PMEM reads data from GPU memory instead of GPU memory writes to PMEM).

Redundant serializations should be eliminated as much as possible: Data serialization is a common time-consuming process in any application involving checkpointing to save the state of applications to a file and recover them when needed. We observe that serialization costs over 30% of a DNN checkpointing time. We also find that a set of serializations becomes redundant once an application generates a sequence of checkpoints, where only the last version is effective for recovery. In other words, as soon as an application creates a new version of a checkpoint, it can abandon all the previous versions, leaving the labor for the serializations for every predecessor checkpoint in vain. Therefore, we can reduce a significant amount of time in finer-grained checkpointing systems if we can eliminate serializations for intermediate checkpoints. Portus can boost the checkpointing performance because the state of GPU memory is dumped to a three-level index structure on PMEM without serialization. Portus will perform serialization only upon an archive of a checkpoint to file systems such as Lustre. Note that serialization in Portus does not affect training efficiency because it operates asynchronously.

Efficient checkpointing and management are crucial for large models: As the trend toward training large models with hundreds of billions of parameters continues, distributed model parallel training becomes increasingly important. This new training methodology poses challenges for checkpointing systems, as the large model is partitioned across many GPUs, with each GPU dumping its own checkpoints. Aggregating checkpoints from all GPUs to restore the model on failures introduces increasing checkpoint management complexity and

requires an efficient methodology to store multiple checkpoints in shared storage with minimal stalls. Portus is natively designed for distributed training scenarios with complex checkpoint structures and has good potential for deployment on larger-scale production systems.

VII. Related Works

DNN checkpointing and datapath optimization are discussed in several prior works. However, to the best of our knowledge, this research is the first work that can do iteration-based fine-grained checkpointing with almost zero overhead. In this section, we first briefly summarize some optimizations on DNN checkpointing, then we introduce several data transmission path optimizations and finally propose some persistent memory management works.

DNN Checkpointing: With the growth of DNN model complexity and data size, traditional epoch-based DNN checkpointing methods fail to persist real-time model states and waste computation to restart when the training job stops unexpectedly. To do fine-grained checkpointing with low overhead, CheckFreq [16] introduced an asynchronous checkpointing method to cover I/O by computation. However, it does not optimize data transmission itself so the overhead is noticeable for multi-tenant training workloads. For algorithms, Check-N-run [15] and Qiao et al. [13]'s method introduced incremental checkpointing. For devices, Wood et al. [36] proposed checkpointing using PMEM, but did less optimization. And for multi-tenant checkpointing, Jeon et al. [37] analyzed the overheads of multi-tenant DNN training jobs. However, Portus is different from above, we focus on the fine-grained checkpointing in multi-tenet workload.

Datapath Optimization: There are several existing works concentrating on optimizing the data transmission path between GPU and other devices. The datapath between SSD and GPU can be highly optimized by NVIDIA GPU direct storage [38], Assise [39], and P2PSSD-GPU [40]. However, there is still an I/O performance gap between NVMe SSDs and persistent memories [41]. GPM [42] was the first work to access persistent memory by GPU threads directly but need to compute nodes equipping both specific GPU and persistent memory, which is very rare because commercial Optane Persistent Memory requires specific motherboards and CPU. Also, there is heavy labor to rewrite the low-level application code to fit GPM even we have new hardware. In the data transmission domain, Wei et al.'s work [43] gave invaluable advice on accessing PMEM via RDMA. Ekko [44] proposed a peer-to-peer transmission protocol for deep learning recommenders, enabling parameter passing from the training cluster to the inference cluster without intermediate storage. Different from these existing works, however, Portus provides a more efficient zero-copy datapath with almost no code modification and better portability.

Persistent Memory Utilization: Efficient persistent memory management plays an important role in Portus. NVALLOC [45], and Intel PMDK [46] proposed novel allocation and management methods for PMEM. ctFS [47] introduced a hierarchical structure to manage files on PMEM under `devdax` mode. Besides, there is an abundance of general-propose filesystems for PMEM including PMFS [48], NOVA [49], and WineFS [50]. However, skewing to generality may lose the performance gain on specialized workloads. So Portus choose a different approach and designed a lighter-weight, application-specialized index structure to manage DNN checkpoints on PMEM efficiently.

VIII. Conclusion

The rapid evolution of DNN model complexity and trends in multi-tenant training make it more challenging to develop prompt finer-grain checkpointing mechanisms. To solve this challenge, we proposed Portus, which builds a dedicated peer-to-peer datapath and a three-level index structure between GPU memory and PMEMs for checkpointing without serialization and kernel crossings.We developed a Portus prototype and integrated it into a production AI cluster. We evaluated it using 76 individual DNN training jobs. We further tested Portus under the distributed multi-node model checkpointing using a large language model, GPT. The experimental results show that Portus improves the performance of DNN models checkpointing and restoring by up to 9.23x and 7.0x. Reducing checkpointing time implies that Portus can reduce the overall training time.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[3] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[5] Hossein Hadian, Hossein Sameti, Daniel Povey, and Sanjeev Khudanpur. End-to-end speech recognition using lattice-free mmi. In *Interspeech*, pages 12–16, 2018.

[6] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[7] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[8] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[9] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *SC20:Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[10] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1677–1689, 2020.

[12] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 637–651, 2021.

[13] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. In *International Conference on Machine Learning*, pages 5220–5230. PMLR, 2019.

[14] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*, pages 159–170, 2019.

[15] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. {Check-N-Run}: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[16] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained} {DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[17] Jang Insu, Yang Zhenning, Zhang Zhen, Jin Xin, and Chowdhury Mosharaf. Oobleck: Resilient distributed training of large models using pipeline templates. 2023.

[18] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[19] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems*, 3:637–651, 2021.

[20] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[21] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

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

[24] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[25] NVIDIA. NVIDIA Peer Memory. https://github.com/Mellanox/nv_peer_memory. Accessed: 2022-11-13.

[26] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.

[27] Anaconda Inc. Anaconda. https://www.anaconda.com/. Accessed: 2022-11-14.

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[29] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[31] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

[32] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[33] Qizhe Xie, Guokun Lai, Zihang Dai, and Eduard Hovy. Large-scale cloze test dataset designed by teachers. *arXiv preprint arXiv:1711.03225*, 2017.

[34] Samsung. Samsung NVMe SSDs. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/. Accessed: 2022-11-19.

[35] Khaled Z Ibrahim, Paul H Hargrove, Costin Iancu, and Katherine Yelick. An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect. In *2014 IEEE 28th international parallel and distributed processing symposium*, pages 1115–1125. IEEE, 2014.

[36] Andrew Wood, Moshik Hershcovitch, Ilias Ennmouri, Weiyu Zong, Saurav Chennuri, Sarel Cohen, Swaminathan Sundararaman, Daniel Waddington, and Peter Chin. Towards fast crash-consistent cluster checkpointing. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2022.

[37] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[38] NVIDIA. NVIDIA GPU Direct Storage. https://developer.nvidia.com/blog/gpudirect-storage/. Accessed: 2022-11-14.

[39] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: performance and availability via nvm colocation in a distributed file system. *arXiv preprint arXiv:1910.05106*, 2019.

[40] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless operating system integration of Peer-to-Peer DMA between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 167–179, Santa Clara, CA, July 2017. USENIX Association.

[41] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[42] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. Gpm: leveraging persistent memory from a gpu. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 142–156, 2022.

[43] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and optimizing remote persistent memory with {RDMA} and {NVM}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 523–536, 2021.

[44] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, et al. Ekko: A {Large-Scale} deep learning recommender system with {Low-Latency} model update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 821–839, 2022.

[45] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. Nvalloc: rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–127, 2022.

[46] Intel. PMDK: Persistent Memory Development Kit. https://pmem.io/. Accessed: 2022-11-14.

[47] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. {ctFS}: Replacing file indexing with hardware memory

translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 35–50, 2022.

[48] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15, 2014.

[49] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[50] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 804–818, 2021.