A Cost-Efficient Failure-Tolerant Scheme for Distributed DNN Training

Menglei Chen, Yu Hua*, Rong Bai, Jianming Huang
WNLO, Huazhong University of Science and Technology, Wuhan, Hubei, China
*Corresponding author: Yu Hua
E-mail: {chenml, csyhua, bair, jmhuang}@hust.edu.cn

Abstract—**Distributed deep neural network (DNN) training is important to support artificial intelligence (AI) applications, such as image classification, natural language processing, and autonomous driving. Unfortunately, the distributed property makes the DNN training vulnerable to system failures. Checkpointing is generally used to support failure tolerance, which however suffers from high runtime overheads. In order to enable high-performance and low-latency checkpointing, we propose a lightweight checkpointing system for distributed DNN training, called LightCheck. To reduce the checkpointing overheads, we leverage fine-grained asynchronous checkpointing by pipelining checkpointing in a layer-wise way. To further decrease the checkpointing latency, we leverage the software-hardware co-design methodology by coalescing new hardware devices into our checkpointing system via a persistent memory (PM) manager. Experimental results on six representative real-world DNN models demonstrate that LightCheck offers more than 10× higher checkpointing frequency with lower runtime overheads than state-of-the-art checkpointing schemes. We have released the open-source codes for public use in https://github.com/LighT-chenml/LightCheck.git.**

Index Terms—**Computer systems, Checkpointing systems, Failure tolerance, Deep neural networks**

I. Introduction

Deep neural networks (DNNs) have been widely adopted in many domains, such as image classification [1], natural language processing [2], and autonomous driving [3]. In general, DNNs need to be frequently trained to achieve high accuracy. However, training DNN models is time-consuming and expensive, even if deploying models across multiple GPUs to accelerate DNN training (called *distributed DNN training*). For example, training a large language model (LLM) GPT-3 consumes up to thousands of NVIDIA A100 GPUs and several months, which spends more than 5 million dollars [2].

The time-consuming DNN training is vulnerable to system failures (e.g., infrastructure failure or software bug). When a failure occurs, the trained model states will be lost, thus causing significant waste of time and training resources. Unfortunately, failures are common during the long training process [4]. Studies from Microsoft [4], [5] show that the mean time between failures (MTBF) varies from a few minutes to several days when training DNNs in large-scale GPU clusters. Hence, the model states (i.e., model parameters and optimizer states) are regularly written to persistent storage for failure tolerance, which is termed *checkpointing*. By using checkpointing, when a failure occurs, only the training progress between two checkpoints will be lost, and others are efficiently saved. Except for failure tolerance, checkpointing is also critical for other scenarios in DNN training. For example, in the preemptive GPU cluster scheduling, the scheduler adopts a round-based scheduling scheme to share resources among multiple training jobs with optimization objectives, such as average job completion time (JCT) [6] or GPU resource utilization [7]. When switching jobs, the current training job is interrupted and its training states are checkpointed before loading the states of other training jobs. The interval between two switches may be only a few seconds. Such frequent switches in scheduling require fast checkpointing to achieve high performance.

Unfortunately, in DNN training, since the size of the model states is often larger than hundreds of megabytes (MB) or even gigabytes (GB), it is not cost-efficient to frequently checkpoint such large model states. Hence, existing DNN training frameworks usually perform checkpointing at the end of each epoch. However, such epoch-level checkpointing would lose more training progress after failures or interruptions. To achieve frequent checkpointing with low runtime overhead, prior designs focus on moving checkpointing out of the critical path in DNN training. DeepFreeze [8] and Check-N-Run [9] first copy the model states in memory and then asynchronously save the data copy into non-volatile storage via the background threads. However, these two checkpointing strategies need to block the training when copying the model states, so the overhead is still high compared with the millisecond-level iteration time. CheckFreq [10] pipelines the in-memory copy operation with computation. But CheckFreq focuses on the single-node DNN training, which fails to fully utilize the parallelism among computation, communication, and checkpointing in the distributed DNN training.

In recent years, a new class of storage media called persistent memory (PM), has received extensive attention [11]–[14]. PM enables byte-addressable access and large capacity with near-DRAM performance, which has been widely used in high-performance database [15], file system [16], and distributed transaction system [17]. PM provides an opportunity to achieve fast resilience for DNN training. Prior schemes [18], [19] leverage PM to provide byte-addressable persistence to GPU kernels, thus improving the performance of GPU appli-

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202, U22B2022 and 61821003.

cations. However, to fully utilize PM and PCIe (i.e., peripheral component interconnect express) bandwidth in checkpointing, the process of writing checkpoints needs to further consider the features of PM. Specifically, PM bandwidth is sensitive to the access pattern of running applications [20], [21], and GPU interacts with PM through the PCIe interface at 128-byte granularity.

In this work, we present a lightweight checkpointing system called LightCheck, which provides frequent checkpointing with low overheads for distributed DNN training. Specifically, LightCheck pipelines checkpointing with computation and communication in a layer-wise way, thus mitigating the impact of checkpointing on training performance. Besides, LightCheck efficiently coalesces the direct access (DAX) feature [22] of PM and the unified virtual addressing (UVA) technique to map PM into the GPU virtual address space, which allows direct access to PM from GPU. LightCheck further uses CUDA (i.e., compute unified device architecture) streams and events to overlap GPU-PM data transfer via GPU computation. To improve the PM write throughput, LightCheck separates the storage of tensor metadata and tensor data in PM, thus making tensor data accesses continuous for small writes and aligned for large writes. Furthermore, for ease of use, we extend the high-level training framework to facilitate the availability of LightCheck, users can easily use LightCheck to achieve fast checkpointing in the DNN training without any code modification to existing DNN training frameworks.

To show the efficiency of our LightCheck, we compare LightCheck with state-of-the-art checkpointing schemes on six popular DNN models. The experimental results show that LightCheck provides 10× higher checkpointing frequency with lower runtime overhead in distributed DNN training, compared with the state-of-the-art schemes. In addition, LightCheck significantly reduces the re-training time for failure recovery. The collection of GPU utilization statistics further indicates that LightCheck incurs no GPU resource utilization degradation when performing checkpointing. Besides, we demonstrate that LightCheck does not decrease the final accuracy of the trained model when resumed from the checkpoints. Moreover, the results also demonstrate the efficiency of our separated checkpoint data storage, which improves the PM write throughput by up to 2.8×. Through extensive analysis, LightCheck shows great abilities to offer efficient checkpointing for distributed DNN training.

II. Background and Related Work

A. DNN Training

To achieve high accuracy, DNNs need to be trained with massive training data. Due to the complex structures of DNN models and large volumes of training data, the training is often a time-consuming task. To improve the training performance, distributed DNN training has been widely used, which is divided into two types: data parallel and model parallel. In the data parallel training, different training nodes have the same parameters. The training data are partitioned into several non-overlapping parts and fed to the training nodes. In the model parallel training, the whole model is partitioned into several parts, and different training nodes are responsible for training different parts of the model. We focus on data parallel training.

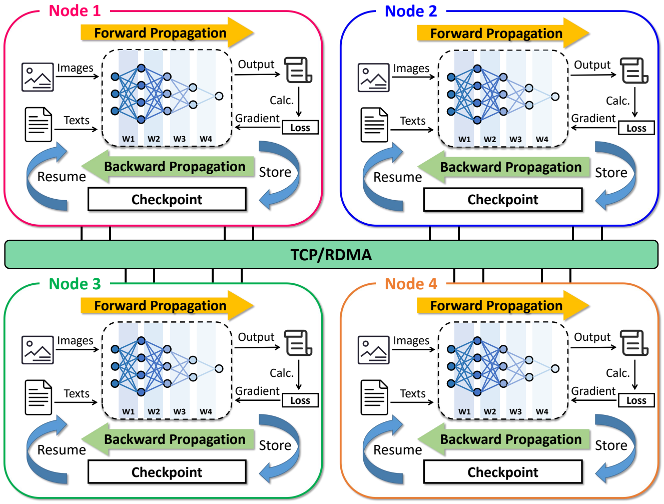

Fig. 1: The overview of the distributed DNN training.

As shown in Fig. 1, the model parameters (i.e., weights and bias) are replicated to all nodes, and the training data (e.g., images and texts) are uniformly distributed to all nodes. Training proceeds in iterations. Each iteration consists of forward propagation, backward propagation, communication, and updating parameters. Specifically, after processing the input data into the input tensors, the training system performs tensor operations on the input tensors and the model parameters to obtain the output tensors. This procedure is called *forward propagation*. Based on the output tensors and loss functions of DNN models, we calculate the gradients of the model parameters. This procedure is called *backward propagation*. Moreover, we collect the gradients of all nodes via TCP or RDMA (i.e., remote direct memory access) communication functions and calculate the global gradients. At the end of an iteration, each node leverages the global gradients to update the model parameters via the model optimizer. In general, we perform the training for multiple epochs. An epoch consists of several iterations and traverses the entire training dataset.

B. Checkpointing for DNN Training

Training a DNN model consumes high costs, including the training time and computing resources. The parameters of the DNN model are maintained in the volatile GPU memory during training. Any interruption to the training system, e.g., system crashes, tasks preemption, or job migration, causes the training failures. Upon a failure, the training states will be lost, the DNN needs to re-train from scratch, causing significant waste of resources. To address this problem, the model states are periodically checkpointed, i.e., the model states are persisted into the non-volatile storage. The training frameworks such as TensorFlow [23] and PyTorch [24] provide specific functions to save model states as files and load checkpoints. After interruptions, the system recovers the DNN training from the checkpoint. However, the intermediate

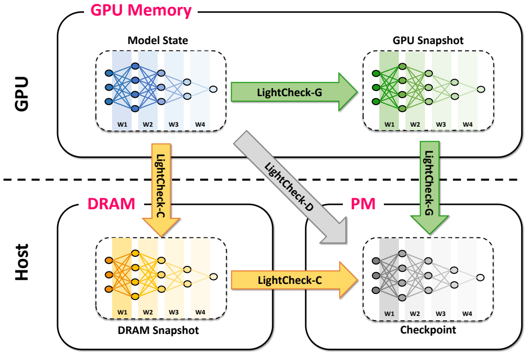

Fig. 2: Schematic diagram of different checkpointing strategies in LightCheck.

states/data between the current training point and checkpoint are lost. Such *training loss* can be mitigated by increasing the frequency of checkpoints, which unfortunately introduces training stall, and hence significantly decreasing the training performance.

To improve the performance of checkpointing, SCAR [25] leverages the behavior that machine learning models have the capability of tolerating perturbations to model parameters, thus proposes low-cost partial recovery for checkpointing. DeepFreeze [8] adopts the multi-level checkpointing that is previously applied to HPC for deep learning, and pipelines the serialization I/O with model training. It also utilizes tensor sharding to reduce I/O overhead. Check-N-Run [9] proposes the incremental checkpointing method based on the parameter update characteristics of recommendation models, and uses quantization technique to reduce checkpoint size. CheckFreq [10] pipelines both model state copy and serialization I/O with computation to enable iteration-level checkpointing, and the checkpoint frequency is auto-tuned to control the checkpoint overhead within a fixed threshold. However, above checkpointing systems cannot fully utilize the parallelism among computation, communication, and checkpointing in the distributed DNN training, thus failing to achieve low-cost frequent checkpointing.

III. The Design of LightCheck

We present LightCheck, which is a checkpointing system for facilitating frequent checkpointing with low overheads in distributed DNN training. LightCheck consists of two main components, including an efficient checkpointing scheme and a persistent memory manager. The checkpointing scheme asynchronously checkpoints the latest updated parts of model states based on the data dependency between model training and checkpointing in distributed training process. Moreover, the persistent memory manager enables effective data transfer between GPU memory and persistent memory by mapping PM into GPU virtual memory space and organizing the storage location of checkpoint data in PM. Besides, LightCheck is integrated into the high-level training framework, providing a transparent and automatic checkpointing approach to users. When interruptions happen, LightCheck loads the latest checkpoint from PM to resume training.

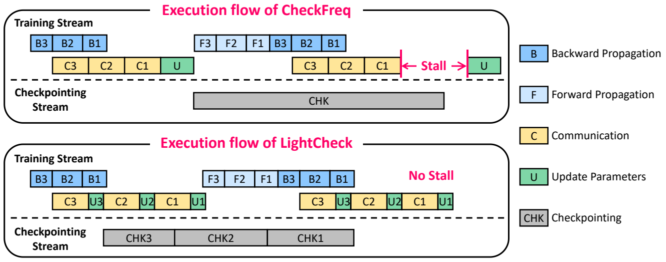

Fig. 3: The execution flows of LightCheck and CheckFreq.

A. *Checkpointing Strategies*

To efficiently perform checkpointing with minimum interference to the training process, we consider different trade-offs in LightCheck. There are three different checkpointing strategies in LightCheck, including LightCheck-G, LightCheck-C, and LightCheck-D (Fig. 2). Specifically, LightCheck-G constructs a copy of model states in GPU memory and writes the copy from GPU memory to PM. Since writing a copy in GPU memory is much faster than writing a copy in CPU memory or PM, LightCheck-G consumes minimal snapshot time. However, LightCheck-G suffers from high GPU memory consumption. Furthermore, LightCheck-C replicates the model states from GPU memory to CPU memory and then saves the CPU copy into PM. LightCheck-C can leverage existing GPU-CPU and CPU-PM data paths but may interfere with the running processes (e.g., preprocessing input images) on the CPU. LightCheck-D directly transfers the model states from GPU memory to PM in a layer-wise way. LightCheck-D does not need extra GPU and CPU resource consumption for copying model states. However, since PM has a lower bandwidth than DRAM and exhibits complex performance characteristics [21], for LightCheck-D, we carefully write checkpoint data to PM to fully utilize PM bandwidth.

B. *Asynchronous Layer-wise Checkpointing*

In general, the DNN training framework performs epoch-level checkpointing in a monolithic and synchronous way to guarantee the consistency of checkpoint data. If the parameters are updated during the execution of the checkpoint operation, the checkpoint data may be partially updated, which would corrupt the checkpoint file. Hence, during checkpointing, the training framework stalls the training and continues training after completing checkpointing. Such design ensures checkpoint data consistency since the model states remain unchanged during checkpointing, and thus the training can recover from the checkpoints. However, the synchronous checkpointing incurs high overheads, i.e., decreasing the computing resource utilization and increasing the overall training time.

Recent asynchronous DNN checkpointing schemes, such as CheckFreq [10], pipeline the checkpointing process with the forward and backward propagations to reduce training stall time incurred by checkpointing and performs iteration-level checkpointing. The top half in Fig. 3 shows the execution flow of CheckFreq when training a DNN model with three layers on multiple training nodes (C1 indicates the communication of layer 1). CheckFreq starts the checkpoint operation of

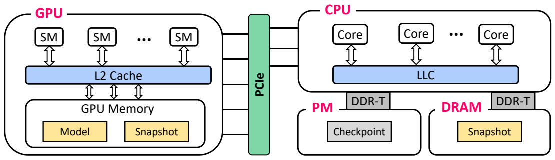

Fig. 4: The interconnections among GPU, CPU and PM.

iteration i after updating the parameters and further executes the training computation phases (i.e., forward and backward propagation) of iteration $i+1$ and checkpointing of iteration i in parallel. When the parameter update phase of iteration $i+1$ arrives, the training is blocked to wait for completing the checkpoint operation of iteration i. However, CheckFreq is sub-optimal due to the monolithic checkpointing process (i.e., the whole model states are continuously copied at once). In addition, CheckFreq does not fully explore the data dependency in the training process. Unlike CheckFreq, we explore and exploit the data dependency of distributed DNN training models to reduce the runtime overhead of checkpointing.

LightCheck introduces the asynchronous layer-wise checkpointing design for distributed DNN training models. We observe that the communication for model parameter synchronization often accounts for a large fraction of training time in the distributed DNN training [26], [27], which can be pipelined with checkpointing. Multiple nodes synchronize the model parameters layer-by-layer during communication. For example, in a communication scheduler with the FIFO order, when the backward propagation of layer $i+1$ is finished, the scheduler needs to wait until the parameter synchronization for layer i to finish and then starts the parameter transmission for layer $i+1$ through underlying communication stack.

Based on this observation, our LightCheck pipelines checkpointing with computation and communication to embed checkpointing into the training data flow. Since the checkpointing of iteration i needs to be completed before updating the corresponding parameters in iteration $i+1$, the checkpointing needs to be performed as soon as possible. When the gradient synchronization and parameters update of one layer are finished, the checkpointing of this layer is ready to start. We put the checkpoint operation into a FIFO queue and asynchronously execute it using a background thread. To guarantee the checkpoint data consistency, LightCheck starts the checkpointing of layer j in iteration i after the parameter synchronization of this layer, and the updates to layer j of iteration $i+1$ need to wait for ongoing checkpointing of layer j to complete. The bottom half in Fig. 3 shows the LightCheck's execution flows of model training and checkpointing. By executing computation and communication in parallel during checkpointing, LightCheck asynchronously schedules the layer-wise checkpointing, thus improving the training resources utilization in case of interruptions.

C. The Interconnection between GPU and PM

Currently, GPU-CPU [28] and CPU-PM [29] systems have been widely studied for a long time, but there are few discussions on how GPU accesses PM. CUDA library provides three techniques for developers to facilitate the data communication between host memory and GPU [30]. The first one is the direct memory access (DMA) technique, which utilizes a pinned buffer as a staging area for the data transfer between host memory and GPU memory. Since we can leverage the DAX feature of PM to directly map PM into CPU address space, this technique can be applied to persistent memory. We can transfer data between PM and GPU memory via cudaMemcpy API through the DMA data path. However, transferring data through the DMA data path still needs to go through the pinned buffer in the CPU memory, hindering the transfer performance. The second one is the unified memory (UM) technique, which further manages the device and host memory in the global memory address space, and automatically migrates memory pages between PM and GPU memory. This technique simplifies programming but still needs implicit page migration, which is hard to expand unified memory to include PM. The third one is the unified virtual addressing (UVA) technique, which enables zero-copy access over PCIe using the global memory address space. This technique allows GPU kernels to directly access PM after mapping PM into GPU virtual address space. Compared with DMA and UM, the UVA technique enables high performance and improves easy of use. Hence, we use the UVA technique to transfer data between PM and GPU memory in LightCheck. Specifically, LightCheck first maps PM into GPU virtual address space. LightCheck then disables data direct IO (DDIO) to control the destination of GPU writes via PCIe, since GPU moves the data into the last level cache (LLC) with DDIO enabled (Fig. 4). GPM [19] has revealed that disabling DDIO with a GPU fence instruction (i.e., __threadfence_system()) is capable of guaranteeing data persistence.

In tandem with the asynchronous layer-wise checkpointing design, LightCheck pipelines the data transfer between GPU and PM with GPU computation. The memory copies are performed on an extra CUDA stream by using the background thread. In addition, since the training needs to check whether the checkpointing process has finished, LightCheck monitors the progress of GPU-PM memory copies via CUDA events. After initiating a data transfer task, LightCheck records a CUDA event to mark a GPU stream execution. When the CUDA event is ready, it is guaranteed that all tasks that launch before the event have been completed.

D. Checkpoint Storage Management

For checkpointing, except for tensor metadata and some additional states (e.g., current epoch, current iteration, and training data index), almost all data are constructed in the tensor format. In general, the checkpoint data are organized in the form of a dictionary. It is important to replicate and maintain the data structure in PM to match the training framework interface for loading model states (i.e., load_state_dict()). However, since PM is sensitive to small random writes [20], the GPU-PM access via the PCIe interface needs to align with 128-byte granularity for better PCIe bandwidth utilization [30]. A misaligned access generates two separate PCIe requests, causing high PM write amplification. Therefore, to efficiently access PM from GPU, LightCheck divides the checkpoint data space into a data mapping region and a continuous tensor region. The data mapping consists of the addresses of tensor data in PM and the number of bytes of tensor data. The data mapping stored in PM is accessible for all GPUs and CPUs. Based on the UVA technique, each tensor in PM has a global virtual address for all GPUs and CPUs, and the processors can directly write tensors to the corresponding location in PM. During checkpointing file initialization, LightCheck allocates PM space and creates the data mapping for tensors. When allocating space in PM, LightCheck continuously stores small tensor data according to the access order, and sequentially allocates memory for tensors larger than 128 bytes at aligned PCIe granularity in the continuous tensor data region. Thanks to the separated checkpoint storage management, LightCheck can significantly alleviate the PM write amplification and reduce the number of required PCIe requests for writing checkpoints from GPU memory to PM.

In addition, although the checkpoints are frequently performed, LightCheck only maintains two checkpoint data mappings for a training model. Once one checkpoint is completed, the other checkpoint is obsoleted. Thus, LightCheck overwrites the obsoleted checkpoint with new checkpoint data, which reduces the PM space consumption and avoids memory allocation contention when multiple GPUs and training models share PM space. If a failure occurs, LightCheck guarantees that there exists at least one complete checkpoint data for recovery.

IV. Performance Evaluation

A. Experimental Setup

We conduct experiments on three machines, each of which is equipped with two 26-core Intel Xeon Gold 6230R CPUs, one Tesla V100 GPU, 192 GB DRAM, six interleaved 128 GB Intel Optane DC PM modules, 3.6 TB HDD (i.e., hard disk drive) and one 100 Gbps Mellanox ConnectX-5 InfiniBand RNIC. The three servers are connected via a 100Gbps Mellanox InfiniBand switch. All servers are installed with 64-bit Ubuntu 18.04. The CUDA version is 11.1. The PCIe interconnect is Gen 3.0×16.

We use six representative DNN models in our experiments, including four image classification models and two language processing models. The four image classification models (ResNet-18 [1], AlexNet [31], Inception-V3 [32], and VGG-16 [33]) use the Imagenet dataset [34], and the two language processing models (GPT-2 [35] and BERT [36]) use the WikiText-2 dataset [37]. We use Horovod with PyTorch 1.8.1 as the training framework, which wraps PyTorch optimizer with Horovod DistributedOptimizer and supports layer-wise all-reduce communication scheduling [38].

We compare three checkpointing strategies in LightCheck (i.e., LightCheck-G, LightCheck-C, LightCheck-D) with the state-of-the-art checkpointing system CheckFreq [10] and the torch.save() from PyTorch. Unless otherwise stated, LightCheck uses the LightCheck-D checkpointing strategy. Since CheckFreq uses disk storage to persist checkpoint data, we implement our asynchronous layer-wise checkpointing to write checkpoint data into disks, called LightCheck-disk, to facilitate fair comparisons. Besides, we evaluate CheckFreq and torch.save with the same checkpoint frequency as LightCheck-disk, i.e., CheckFreq-same and torch.save-same.

TABLE I: The checkpoint sizes and the intervals.

Models	Checkpoint Size (MB)	Checkpointing Interval (iterations)			
		LightCheck -G/C/D	*LightCheck -disk*	*CheckFreq*	*torch.save*
ResNet-18	90	1	7	20	102
VGG_16	1,056	6	64	146	904
Inception-V3	183	14	30	40	118
AlexNet	467	8	95	164	1,084
GPT-2	1,508	6	46	100	682
BERT	4,004	10	82	200	1,100

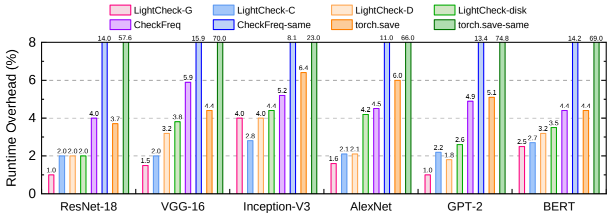

Fig. 5: The runtime overheads of different checkpointing systems.

B. Checkpointing Performance

Checkpointing Overhead. We compare the checkpointing frequency and runtime overhead of different checkpointing strategies with multiple DNN models. Table I shows the model sizes of different DNN models and the checkpointing intervals in different checkpointing strategies that limit runtime overhead within a threshold (i.e., 5%). For example, in ResNet-18, LightCheck-G/LightCheck-C/LightCheck-D/LightCheck-disk/CheckFreq save a checkpoint per 1/1/1/7/20 iterations. Note that the checkpointing frequency of CheckFreq follows its open-source code. Fig. 5 shows the runtime overheads of different checkpointing strategies in distributed DNN training.

From the experimental results, we obtain the following observations. First, compared with CheckFreq, LightCheck-G/LightCheck-C/LightCheck-D enables more than 10× higher checkpointing frequency with lower runtime overhead for most DNN models. For example, the runtime overhead of LightCheck is only up to 4%, i.e., LightCheck-D in Inception-V3. The reason is that LightCheck efficiently pipelines checkpointing with communication and computation in the training process. Moreover, these results show that LightCheck incurs negligible runtime overhead when performing frequent checkpointing, which is important for unstable training scenarios. In fact, checkpointing efficiency is critical to the systems that frequently perform transparent preemption and migration for training tasks in a GPU cluster. Second, our LightCheck-disk also outperforms CheckFreq in all evaluated models. Specifically, LightCheck-disk significantly reduces up to 10% runtime overhead compared with CheckFreq-same. The improvement stems from our asynchronous layer-wise checkpointing in LightCheck. Third, LightCheck efficiently exploits PM to improve checkpointing performance. LightCheck-disk not only consumes a long time to complete but also suffers from

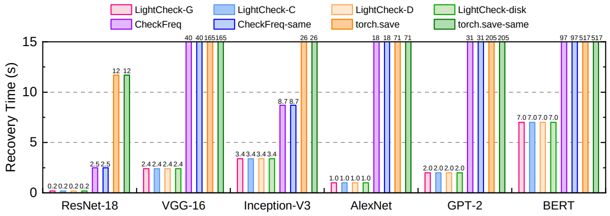

Fig. 6: The recovery times of different checkpointing systems.

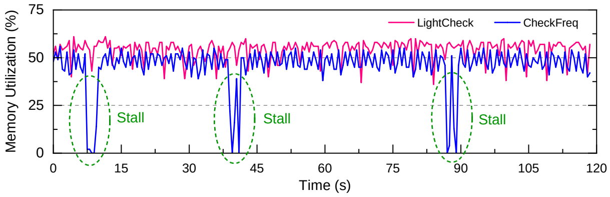

Fig. 7: The GPU memory utilizations of LightCheck and CheckFreq.

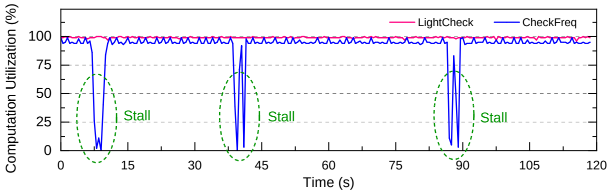

Fig. 8: The GPU computation utilizations of LightCheck and CheckFreq.

TABLE II: The total training time of different checkpointing systems.

Models	Total Training Time (h)			
	No Failure	*LightCheck*	*CheckFreq*	*torch.save*
ResNet-18	10.7	10.9	11.1	11.2
VGG_16	67.5	69.7	72.3	73.7
Inception-V3	79.7	83.0	84.1	85.4
AlexNet	6.0	6.1	6.3	6.5
GPT-2	161.7	164.6	171.1	179.7
BERT	501.3	514.8	537.5	598.6

the coarse-grained scheduling in the training system and the high startup overhead of store operation. On the other hand, LightCheck benefits from its fine-grained thread scheduling and the direct data transfer path with fast PM. Fourth, compared with LightCheck-G and LightCheck-C, LightCheck-D achieves comparable checkpointing performance without extra memory consumption, which demonstrates the effectiveness of direct GPU-PM data transfer. The reason is that LightCheck-D fully utilizes the PCIe bandwidth by organizing the storage location of checkpoint data in PM.

Recovery Time. We evaluate the time overhead of recovering the training models from checkpoints. The recovery time consists of the time to load the model states from the latest saved checkpoint and re-train the model to the state before the interruption. Hence, the recovery time depends on the checkpoint frequency. If the checkpoint frequency is high, only a few iterations of training progress are lost after the interruption, and thus the re-training time is short. As shown in Fig. 6, with LightCheck, training can be resumed within 7 seconds for all models. Compared with CheckFreq and torch.save, LightCheck significantly improves the recovery performance. The three checkpointing strategies of LightCheck have similar recovery performances. For example, they reduce the recovery time to 7 seconds when training the BERT model.

Resource Utilization. We monitor and record the GPU memory utilization and computation utilization through the NVIDIA system management interface every 50 ms when training VGG-16 model. As shown in Fig. 7, LightCheck improves the memory utilization up to 60% higher than that of CheckFreq during training. Besides, the memory utilization of CheckFreq drops to 0 when performing checkpointing, which significantly affects the training performance. Unlike CheckFreq, our LightCheck shows stable memory utilization, because the asynchronous layer-wise checkpointing is efficient to mitigate the waste of GPU resources during checkpoint saving. The computation utilization exhibits a similar trend as the memory utilization (Fig. 8).

Total Training Time. We evaluate the impact of different checkpointing strategies on total training time in the presence of failures. We inject failures into the training process with a fixed MTBF (i.e., one hour) and record the total training time of different checkpointing strategies. Table II shows the total training time of different checkpointing systems. Compared with CheckFreq and torch.save, LightCheck mitigates the impact of failures on total training time. We further demonstrate the impact of MTBF on total training time. Fig. 9 shows the total training time with different MTBFs when training BERT. The *ideal* line exhibits the total training time when training without failures. LightCheck provides lower total training time than CheckFreq and torch.save. Moreover, when failures frequently occur, the total training time of torch.save significantly increases, while LightCheck can maintain stable total training time.

C. The Benefits of Asynchronous Layer-wise Checkpointing

To better understand the performance benefit of the asynchronous layer-wise checkpointing scheme, we evaluate the runtime overheads of checkpointing when using persistent memory to save checkpoints for both LightCheck and CheckFreq. LightCheck directly copies the model states from GPU memory to PM in a layer-wise way. Here, CheckFreq-PM also copies the entire model state to PM after updating parameters, and does not need to persist the copy, since this copy has already existed in PM. When the corresponding parameter update phase of the next iteration arrives, if the checkpoint operation is not completed, the model training is blocked to guarantee checkpoint data consistency. We measure the three large DNN models. As shown in Fig. 10, LightCheck incurs up to 10% runtime overhead when performing checkpointing, while the runtime overhead of CheckFreq-PM is up to 41%. The experimental results show that the asynchronous layer-wise checkpointing scheme significantly reduces the runtime overhead.

D. Efficiency of Separated Checkpoint Storage Management

We measure the benefits of the optimized checkpoint data storage. When initializing the checkpoint data structure in

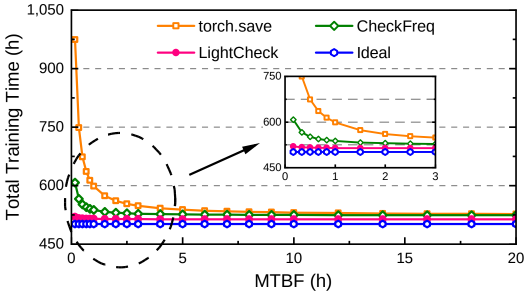

Fig. 9: The impact of MTBF on the total training time.

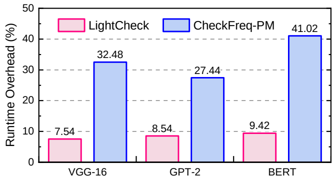

Fig. 10: The checkpointing overheads of LightCheck and CheckFreq-PM.

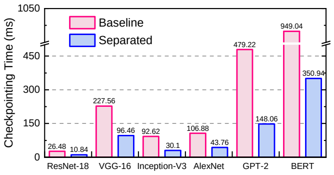

Fig. 11: The checkpointing times of different checkpoint storage managements.

the dictionary form, the *Baseline* does not distinguish the checkpoint data attributes (e.g., tensor metadata, tensor data, or some additional states) and stores them together in PM. Therefore, when saving checkpoints, tensor data will be written out-of-order, causing high write amplification. In contrast, the *Separated* storage management continuously writes tensor data, keeping the write amplification at 1× and reducing the number of PCIe requests for transferring the same amount of data. Fig. 11 shows the checkpointing time for various models in PM. The separated storage achieves 2.1-2.8× speedup compared with the Baseline, demonstrating its effectiveness in fully utilizing PM and PCIe bandwidth.

E. The Impact of Data Direct IO (DDIO) on Checkpointing

We observe that DDIO has a great performance impact on the data transfer between GPU memory and PM. As shown in Fig. 12, the throughput of GPU sequential writes only reaches 3.5 GB/s with DDIO enabled. On the other hand, the write throughput achieves 11.5 GB/s with DDIO disabled, which is close to the maximum bandwidth of PCIe 3.0. The reason is that when DDIO is enabled, GPU sequential writes are first cached in the last level cache (LLC), and then the LLC randomly evicts data to PM at the cache-line granularity. The GPU sequential writes are changed to random writes, thus causing high write amplification. We further measure PM write amplification when running the sequential access benchmark via ipmctl tool. The results show that enabling DDIO incurs about 2.6-3.0× write amplification for sequential GPU writes under various payload sizes (Fig. 13). In contrast, disabling DDIO keeps the write amplification around 1×. Overall, disabling DDIO provides better checkpointing performance.

F. Checkpointing in GPU Cluster Scheduling

We evaluate the effect of checkpointing in frequent preemptive scheduling scenarios. We adopt the Gavel [6] scheduler to run the least attained service (LAS) policy [39] with physical and simulated experiments on two types of traces: 1) Static. All jobs arrive when starting execution. 2) Continuous. Jobs are continuously added during execution. Gavel sets the round duration to 6 minutes, which means that it computes job allocation according to LAS and switches tasks to be executed in every round. By default, the state of preempted jobs is saved via torch.save(). For efficient comparisons, we use our LightCheck to checkpoint the preempted jobs. The two traces in the physical experiments run 25 jobs on 3 machines, and the job types are uniformly sampled from the six DNN models. As shown in Table III, compared with torch.save, LightCheck reduces average job completion time by 8% and 17% for the static and continuous traces, respectively. In the simulation, we inject preemption overheads measured by running specific models on actual GPUs. The results show that the difference between physical evaluation and simulation is small. Furthermore, we simulate a cluster with 90 NVIDIA V100 GPUs, the trace has varied input job rates, and the completion time of jobs with ID 4,000-5,000 is measured. The results show that LightCheck reduces average JCT by 12% at the low load rate, and supports the higher load rate than torch.save (Fig. 14).

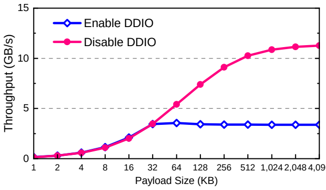

Fig. 12: The write throughputs of checkpointing with and without DDIO.

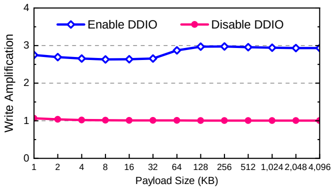

Fig. 13: The write amplifications of checkpointing with and without DDIO.

TABLE III: The effects of different checkpointing systems on GPU cluster scheduling.

Checkpointing Systems	Trace	Metrics	Physical (h)	Simulation (h)
torch.save	static	average JCT	2.38	2.33
LightCheck	static	average JCT	2.20	2.09
torch.save	continuous	average JCT	1.67	1.60
LightCheck	continuous	average JCT	1.38	1.37

G. Model Accuracy

We demonstrate the impact of checkpointing on accuracy. We train the ResNet-18 model to the target accuracy (i.e., 96%) using the mini-imagenet dataset. We train ResNet-18 in two different scenarios: 1) No interruption. The model is trained without interruption until its completion. LightCheck does not perform checkpointing in this scenario. 2) Resumed from checkpoints. In this scenario, LightCheck performs checkpointing during model training. The training process is interrupted at the fixed interval (one epoch), then resumes from the latest checkpoint and continues to train the model. We present the top-1 validation accuracy against the cumulative training time in Fig. 15. The final model accuracy of the second training scenario is similar to that of the model

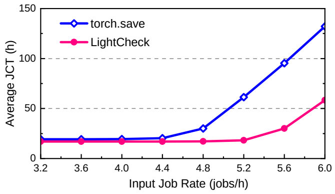

Fig. 14: The simulated average JCT with different input job rates under different checkpointing systems in GPU cluster scheduling.

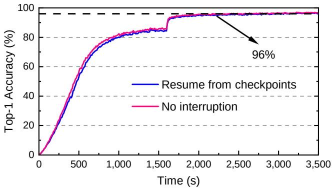

Fig. 15: The top-1 validation accuracy against the cumulative training time when training ResNet-18 with LightCheck.

without interruption, which indicates that the checkpointing in LightCheck has a negligible impact on training accuracy.

V. Conclusion

Distributed DNN training is important for AI applications in many domains. While distributed DNN training requires checkpointing for failure tolerance, it is a challenge to provide frequent checkpointing with low runtime overhead. In this work, we present LightCheck, a lightweight checkpointing system for distributed DNN training. We propose an efficient checkpointing scheme and a persistent memory manager for LightCheck to achieve fast checkpointing. The checkpointing scheme achieves fine-grained asynchronous checkpointing by pipelining checkpointing with computation and communication in a layer-wise way, which reduces the checkpointing overhead in DNN training. Moreover, the persistent memory manager enables effective data access to PM from GPU by mapping PM into GPU virtual memory space and separating the storage of tensor metadata and tensor data in PM. Our experimental results show that LightCheck reduces the runtime overhead of checkpointing while providing 10× higher checkpointing frequency in distributed DNN training.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[2] T. B. Brown, B. Mann *et al.*, "Language models are few-shot learners," in *NIPS*, 2020.

[3] S. Casas, A. Sadat, and R. Urtasun, "MP3: A unified model to map, perceive, predict and plan," in *CVPR*, 2021.

[4] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *ICSE*, 2020.

[5] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *USENIX ATC*, 2019.

[6] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *OSDI*, 2020.

[7] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *OSDI*, 2018.

[8] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *CCGRID*, 2020.

[9] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "Check-n-run: a checkpointing system for training deep learning recommendation models," in *NSDI*, 2022.

[10] J. Mohan, A. Phanishayee, and V. Chidambaram, "Checkfreq: Frequent, fine-grained dnn checkpointing," in *FAST*, 2021.

[11] C. Ruan, Y. Zhang, C. Bi, X. Ma, H. Chen, F. Li, X. Yang, C. Li, A. Aboulnaga, and Y. Xu, "Persistent memory disaggregation for cloud-native relational databases," in *ASPLOS*, 2023.

[12] H. Bae, M. Kwon, D. Gouk, S. Han, S. Koh, C. Lee, D. Park, and M. Jung, "Empirical guide to use of persistent memory for large-scale in-memory graph analysis," in *ICCD*, 2021.

[13] X. Li, H. Cui, and L. Liu, "NRHI: A concurrent non-rehashing hash index for persistent memory," in *ICCD*, 2021.

[14] X. Liu, Y. Hua, and R. Bai, "Consistent rdma-friendly hashing on remote persistent memory," in *ICCD*, 2021.

[15] W. Kim, C. Park, D. Kim, H. Park, Y. Choi, A. Sussman, and B. Nam, "Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory," in *OSDI*, 2022.

[16] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST*, 2016.

[17] M. Zhang, Y. Hua, P. Zuo, and L. Liu, "FORD: fast one-sided rdma-based distributed transactions for disaggregated persistent memory," in *FAST*, 2022.

[18] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "Dragon: breaking gpu memory capacity limits with direct nvm access," in *SC*, 2018.

[19] S. Pandey, A. K. Kamath, and A. Basu, "Gpm: leveraging persistent memory from a gpu," in *ASPLOS*, 2022.

[20] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," in *VLDB*, 2020.

[21] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *FAST*, 2020.

[22] Y. Chen, J. Shu, J. Ou, and Y. Lu, "Hinfs: A persistent memory file system with both buffering and direct-access," in *TOS*, 2018.

[23] M. Abadi, P. Barham *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[24] A. Paszke, S. Gross *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019.

[25] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *ICML*, 2019.

[26] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *USENIX ATC*, 2017.

[27] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *SOSP*, 2019.

[28] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale model training," in *USENIX ATC*, 2021.

[29] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," in *VLDB*, 2019.

[30] S. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W. W. Hwu, "Large graph convolutional network training with gpu-oriented data communication architecture," in *VLDB*, 2021.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[32] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[34] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," in *NIPS*, 2016.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," in *OpenAI blog*, 2019.

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

[37] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *arXiv preprint arXiv:1609.07843*, 2016.

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

[39] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *NSDI*, 2019.