OpenEmbedding: A Distributed Parameter Server for Deep Learning Recommendation Models using Persistent Memory

Cheng Chen*†, Yilin Wang*, Jun Yang*, Yiming Liu*, Mian Lu*, Zhao Zheng*, Bingsheng He†, Weng-Fai Wong†, Liang You‡, Penghao Sun‡, Yuping Zhao§, Fenghua Hu§, Andy Rudoff§

*4Paradigm Inc, †National University of Singapore, ‡Alibaba Group, §Intel Corporation

{chencheng, wangyilin, yangjun01, liuyiming, lumian, zhengzhao}@4paradigm.com, {hebs, wongwf}@comp.nus.edu.sg
{youliang.yl, penghao.sph}@alibaba-inc.com, {yuping.zhao, fenghua.hu, andy.rudoff}@intel.com

***Abstract*—In this paper, we present *OpenEmbedding*, a distributed parameter server system for deep learning recommendation models (DLRM) workloads. In order to support rapid growth in the number of features and the model size (Terabytes are common) of DLRM workloads, OpenEmbedding takes advantage of emerging persistent memory (PMem) to address scalability and reliability issues in training DLRMs. Compared to DRAM, PMem can have much lower per-GB cost, higher density, and non-volatility, while with slightly low access performance to DRAM. OpenEmbedding uses DRAM as cache and PMem as storage for the sparse features and develops a simple but effective pipeline processing approach to optimize the access latency of the sparse features in PMem. For reliability, we develop a lightweight synchronous checkpointing scheme that is specially co-designed with the pipelined cache to reduce the run-time overhead of checkpointing. Our evaluations on a real-world industry workload consisting of billions of parameters demonstrate 1) the effectiveness of our PMem-aware optimizations, 2) checkpointing mechanism with near-zero runtime overhead to the training performance and 3) fast recovery with up to 3.97× speedup compared to the state-of-the-art. OpenEmbedding has been deployed in hundreds of scenarios in industry within 4Paradigm, and is open-sourced[1].**

***Index Terms*—machine learning system, recommendation model, parameter server, persistent memory**

I. Introduction

Deep learning recommendation models (DLRM) [1] are now widely used in large-scale e-commerce and online applications in data centers such as personalized recommendation and click-through rate (CTR) prediction [2]–[5]. To achieve higher accuracy, DLRM relies on a large number of training inputs and features. Recent studies showed that the size of DLRM increases by more than 3× in the past two years [6]. In production environments, DLRMs need to be frequently retrained to capture the new data in order to deliver better quality of recommendations. Thus, improving the efficiency of training DLRMs is critical for the accuracy and up-to-date model quality in many e-commerce and online applications.

DLRMs consist of dense features (such as height and weight of each user) and sparse features (such as items purchased by users, where each user only purchases a small set of items). The sparse features often occupy more than 99% of the total size of a large-scale DLRM, ranging from hundreds of Gigabyte (GB) to tens of TB [5]. In DLRMs, training inputs are extremely sparse, because the number of features in each input is far smaller than the total number of features in the entire model. Many applications contain billions to trillions of features but each input only has a few non-zero features.

DLRM utilizes massive embedding tables to store the sparse features. On the one hand, such huge model sizes pose significant challenges in the access efficiency to sparse features. During the training process, there are a lot of random accesses to embedding entries. There have been some solutions for in-memory parameter server systems [7]–[9]. However, using only DRAM makes these solutions very costly due to the large size requirement of DLRMs. Although existing solutions that store embedding entries to solid state drives (SSDs) [4], [5] demonstrate promising results for scalability, the random accesses to SSDs are almost two orders of magnitude slower than DRAM and training large-scale DLRMs becomes prohibitively time-consuming [4].

On the other hand, a large-scale DLRM requires a long training process that may last for tens of hours or even a few days. In data centers, failures of long running training jobs caused by various hardware and software failures (e.g. out of memory) are noticeable [6], [10], [11]. Thus, training reliability is essential for training large-scale DLRMs. Existing methods with checkpoints to local SSDs or remote storage could lead to significant performance degradation in training [12].

In this paper, we present *OpenEmbedding*, a distributed parameter server system for training DLRMs. OpenEmbedding has been integrated with Tensorflow/Keras framework [13]. It supports synchronous DLRM training, which demonstrates higher accuracy and better model convergence according to the previous studies [14], [15].

Compared with other DLRM training systems [4], [5], a unique feature of OpenEmbedding is that it takes advantage of emerging persistent memory (PMem) to address scalability and

[1]https://github.com/4paradigm/OpenEmbedding

TABLE I
PERFORMANCE COMPARISON OF DIFFERENT DEVICES

Devices	**Bandwidth(R/W) (GB/s)**	**Latency(R/W) (ns)**
DRAM	115 / 79	81 / 86
PMem	39 / 14	305 / 94
Flash SSD	2∼3 / 1∼2	>10000

reliability issues in training DLRMs. Intel recently announced a persistent memory product named Optane™ Persistent Memory Module (PMem) [16] which has much lower per-GB cost, higher density and non-volatility in comparison with DRAM. Note that directly replacing DRAM with PMem in the parameter servers may slow down the training process because PMem has a lower bandwidth than DRAM. As shown in Table I, the read and write throughput of PMem is only one-third and one-fifth of that in DRAM respectively. Thus, we use DRAM as cache and store embedding entries into PMem.

The key challenge for synchronous DLRM training is that all GPU workers simultaneously initiate pull requests to the parameter servers at the beginning of each batch and update requests at the end of each batch. This generates a sharp and burst read/write I/O to PMem, thus significantly increase the parallelism overhead. Furthermore, PMem's byte-addressability allows us to develop fine-grained storage models for the embedding entries. Using a fine-grained cache reduces write amplification, but it also increases the granularity and bring additional overhead of multiprocess synchronization. Thus we propose a co-designed cache mechanism that tailored for DLRM training process. Specifically, we develop a simple but effective pipelined workflow that allows us to hide the expensive PMem reads and writes as well as cache replacement overhead behind the GPU training process.

For training reliability, the most recent work [6] from Facebook proposed to use checkpointing. Under the context of synchronous training, that approach pauses training, takes a snapshot of the DLRM and dumps them to remote persistent backup storage. They assume that the local storage is either too slow (e.g., hard disks or SSDs) or too small for the checkpoint. With PMem, it is possible to leverage PMem to reduce the checkpointing overhead. Thus the DLRM training components can perform checkpointing on the local storage in short periods, and then perform checkpointing on the remote storage in large periods. Thus, we start with applying the most recent work of DLRM checkpointing [6] on PMem, where the DLRM training and the checkpointing are two independent components. However, the bandwidth of PMem is already mostly occupied by the training system, and the additional checkpoint backup IOs interfere the training execution, which causes a significant slowdown in the entire training process. In order to further reduce the run-time overhead, we propose a lightweight checkpointing scheme that co-designs the cache replacement with the checkpointing process in order to reduce the overhead of checkpointing during the training process.

The contributions of this paper are as follows:

- We study a real-world and representative DLRM training workload from a customer of 4Paradigm, analyse and summarise its I/O access characteristics. Those findings are common in real workloads for DLRM, and we identify the challenges and opportunities in the efficiency of DLRM training.
- We present *OpenEmbedding*, a distributed parameter server system for training DLRMs. To our best knowledge, this is the first DLRM training system that is specially optimized for byte-addressable PMem.
- OpenEmbedding utilizes a co-designed cache mechanism that tailored for DLRM training process. Meanwhile, we propose a lightweight checkpointing scheme that co-designs the cache replacement with the checkpoint process to minimize the overhead of checkpointing.
- We integrate OpenEmbedding with Tensorflow/Keras framework. OpenEmbedding has been deployed in hundreds of scenarios in industry within 4Paradigm. The whole project is close to 60,000 lines and now is available on GitHub[2].
- OpenEmbedding takes advantage of the large capacity and low cost per GB of PMem to provide a much more cost-efficient and reliable parameter server system for DLRM training. The result shows that OpenEmbedding reduces training time by up to 53.8% compared to the state-of-the-art caching system, saves up to 42% storage cost compared with the pure DRAM solution. OpenEmbedding can make checkpoints during the training process without affecting the training performance, and speed up recovery by up to 3.97 times.

II. BACKGROUND AND RELATED WORK

In this section, we will introduce the background as well as related work on DLRM training and PMem.

A. DLRM Training

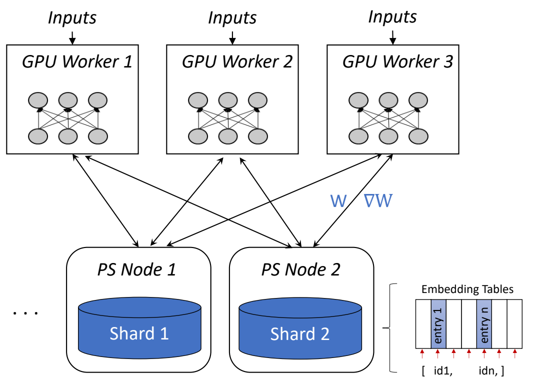

Fig. 1. The training process of the DLRM. embedding entries are partitioned into shards, which are usually stored in the DRAM of parameter server (PS) nodes. The training servers (i.e., GPU workers) lookup the embedding entries and write the gradients in each iteration.

[2]https://github.com/4paradigm/OpenEmbedding.

DLRMs are quite common in large-scale e-commerce and online applications. For example, when browsing shopping websites, such as Amazon or Taobao, we can see a series of recommended items under the product link we are visiting. There are more than hundreds of millions of products have been purchased by billions of users everyday, but an individual user only visits and purchases a few products. Therefore, the links between products and users are sparse. Typically, dense features (such as the user's height) have continuous values, while sparse features are encoded as multi-hot vectors that are categorical. For instance, a sparse feature may represent user's gender or the items viewed by user in the last 2 hours. Because a multi-hot vector may be used to represent limited number of interactions in a large domain with millions or billions dimensions, it must be first transformed into dense vectors (called embedding entries) using embedding table operations. In order to achieve a high accuracy, DLRMs require a large number of embedding entries.

As shown in Figure 1, the embedding entries are indexed by their identifiers (id). At the beginning of each training batch, GPU workers first lookup and read the embedding entries according to the id (e.g. id1 and idn) included in the training inputs of one batch. The training depends on the readiness of the embedding entries. After obtaining the embedding entries (e.g. entry 1 and entry n), those embedding entries are fed into a Multi-Layer Perceptron (MLP) layer on GPU server. The MLP layer is a dense model that consumes a lot of compute power but the size of the MLP parameters is much smaller than the previous sparse part [6]. The GPU workers calculate the gradients of those spare features and then write them back to the parameter server at the end of each batch. Due to the huge data footprint of the embedding entries, most of the current training systems deploy embedding entries across multiple parameter servers. The total amount of the embedding entries is huge, but only a limited number of embedding entries are accessed during each batch of training, the process of reading and writing embedding entries generates massive random I/O, which easily dominates the overall training latency.

There are two typical synchronization patterns in DLRM training [9]: 1) *Synchronous Training*: strictly synchronizes all updates after each worker completes its own training in each iteration. 2) *Asynchronous Training*: all workers keep running without synchronization. In this paper, we focus on synchronous training since the previous studies [14], [15] have reported that the synchronous training yields faster convergence with higher accuracy than asynchronous training.

Checkpoint in DLRMs: As mentioned in Introduction, checkpoint mechanisms have been commonly used for improving training reliability for the long-running DLRM training process. There are two kinds of checkpointing: asynchronous checkpointing and synchronous checkpointing. Different from asynchronous checkpointing that training resumes normally when a checkpoint is being taken, synchronous checkpointing pauses training when checkpoints are being taken. Due to the inconsistency caused by the entries updating during checkpointing, asynchronous checkpoint might affect the convergence of the model in an unexpected way. Based on our experience as well as that from Google and Facebook's teams working on DLRM training [17], synchronous checkpoints are the most widely used and the state-of-the-art method of checkpointing DLRM systems in the industry.

In synchronous checkpointing, a checkpoint is triggered at the end of a batch. There have been quite some works in performing checkpoints for traditional deep neural network [10], [11]. Relatively fewer studies have been conducted on checkpointing DLRM. [6] presents a scalable checkpointing system for training large DLRM in Facebook. The system applies incremental checkpointing and quantization techniques to reduce the checkpoint size. The checkpointing systems of the DLRM usually stores checkpoint file remotely, and their work is complementary to OpenEmbedding, where we focus on performing checkpoints to the local storage.

B. PMem

Intel Optane™ Persistent Memory Module (PMem) [16] is the first commercial product of the byte-addressable non-volatile memory technology now available in the market. Different from Intel's Optane SSD products which are used as block devices via PCI-E interface, PMem resides in the same DIMM slots as DRAM offering byte-addressability and much higher throughput via memory channels. Intel open-sourced a Persistent Memory Development Kit (PMDK) to help developers to implement applications that can fully utilize the performance and persistency of PMem without worrying about the details of PMem such as space management. In this study, we mainly use PMDK to develop our system. As shown in Table. I, although PMem is much faster than SSDs, there is still a performance gap between PMem and DRAM. Meanwhile, previous works [18]–[22] have shown that implementing data consistency in PMem is non-trivial and error-prone.

Cache design for PMem: In a hybrid memory system consisting of DRAM and PMem, many studies have used DRAM as cache [20], [23], [24]. [24] proposed a key-value store for point read, blind update, and read-modify-write operations. [23] places the index in fast DRAM and stores the data in a persistent log. The main target of this work is to boost the performance when the small-sized access pattern in key-value stores doesn't match with the persistence granularity in PMem. [20] also puts the index in DRAM and speeds up the overall write performance by leveraging a PMem-aware storage layout. [20] is the most recent work that shows the best performance among other DRAM-PMem key-value stores. However, its hybrid data structure only achieves about 2/3x of its DRAM-only version. When burst accesses occur frequently in workload, this performance gap will be even larger. Researchers have developed different cache replacement strategies to improve the cache performance. In contrast, we do not focus on improving the cache replacement policies. Instead, we aim at improving the entire I/O pipeline by considering caching and training in DLRM.

TABLE II
ACCESS PATTERN OF THE EMBEDDING ENTRIES IN DLRM WORKLOAD

% of Top Accessed Entries	Top 0.05%	Top 0.1%	Top 1%
% of Total Access	85.7%	89.5%	95.7%

A caching system works as a black box, that its internal process is transparent to the upper applications. The pair operations of pull and update on the same embedding entry within the same batch are considered to be two independent operations in cache. The extra reorder is unnecessary since it does not contribute to better capture the access pattern of the embedding entries. In this study, OpenEmbedding propose a different way of maintaining a fine-grained DRAM cache for batch-based training on PMem so that such software overhead can be minimized.

III. CASE STUDY & MOTIVATION

We start with analysing a real DLRM workload trace from a customer of 4Paradigm. The trace was from a top retail company that has more than 2,000 stores worldwide over a period of 147 days. This production DLRM is used to provide real-time recommendation services for customers visiting their online shop. The real-world DLRM has more than 2.1 billion embedding entries stored on the parameter server. The training data size is 3.4 TB. The embedding entry is represented as a vector of 64 floating numbers, and the total size of the model is more than 500GB. From our experience in working with other customers, this workload trace is a representative workload in large-scale DLRM. Also, it is larger than the commonly used benchmark (such as Criteo Terabyte dataset [25]). We will use this workload in the following motivational analysis and the later experiments.

A. Workload Analysis of a Real-world DLRM

We observe that the real-world DLRM workload has the following characteristics.

1) Highly Skewed Access: We analyse the access frequency of the embedding entries. As shown in Table II, some embedding entries have very high locality, i.e., some embedding entries are accessed with high intensity. For instance, the features such as the user's gender, click time belonging to what day of the week, etc. are high-frequency features that appear in almost every batch. We call these highly accessed features as hot. Some of them are accessed even during every batch of training. In contrast, most of the features appear only a few times during the whole training process. Since the access pattern of the DLRM is highly skewed, it provides us a chance to put those frequently accessed features into the fast-speed storage to boost the access performance.

2) Burst I/O in Pairs: To better understand the I/O access pattern of the parameter server, we record each accessed embedding entries and count how many access requests received per millisecond during the DLRM training process. We observe two phenomena. First, the same embedding entry

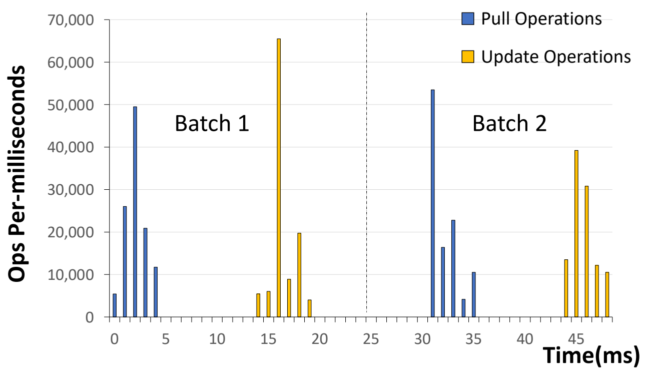

Fig. 2. Access pattern in two batches

is accessed twice in pairs within a batch. This is because DLRM training process always start from obtaining a group of embedding entries (e.g. A) at the beginning of the batch. When the GPU training process is completed, the gradient of A is passed back to the parameter server and the weights of A are updated to A'. This is why pull and update requests always appear in pairs within a batch on Figure 2, and the total amount is consistent.

Second, the access to the parameter server is not homogenized, and each batch starts and ends with an IO burst. For instance, there are clear burst at 2, 16, 31, and 45 milliseconds in Figure 2. Under the synchronous training mode, each worker must wait for the other workers to finish their updates on the parameter server before starting the next batch training. All workers in the synchronous mode will raise the pull requests simultaneously and complete the update of embedding entries at almost the same time. That results in the I/O bursts. Meanwhile, we observe that there is an obvious idle period between the pull and update operations on the parameter server.

B. Observations in using PMem

Our training system is motivated by the following two major observations and their design implications to DLRM systems in using PMem.

Observation 1: Training DLRM with a fine-grained DRAM-PMem hybrid parameter server can significantly degrade the training performance in comparison with training DLRM with DRAM-PS solutions. We build the DRAM-PMem hybrid key-value store by using the state-of-the-art concurrent hash map from Facebook [26] and list from C++ STL [27]. We then replace the original DRAM-based counterparts inside the parameter server, and evaluate the end-to-end training performance of the real-world DLRM workload. We also integrated a classic PMem-based concurrent hash map from Intel [28] as the storage engine of the parameter server (labeled as 'PMem-Hash') to show the basic performance of using an existing PMem work. The vertical axis of Figure 3 indicates the normalized training time. As shown in the Figure 3, when training the DLRM with four GPUs, the training time of DRAM-PMem hybrid system is 24% longer than the pure

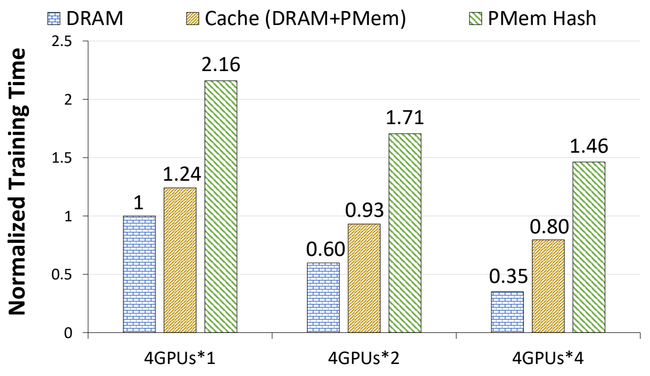

Fig. 3. Performance penalty when apply a fine-grained DRAM-PMem cache or a PMem aware structure on parameter servers (All values normalized to that of the DRAM parameter server on one 4GPU machine)

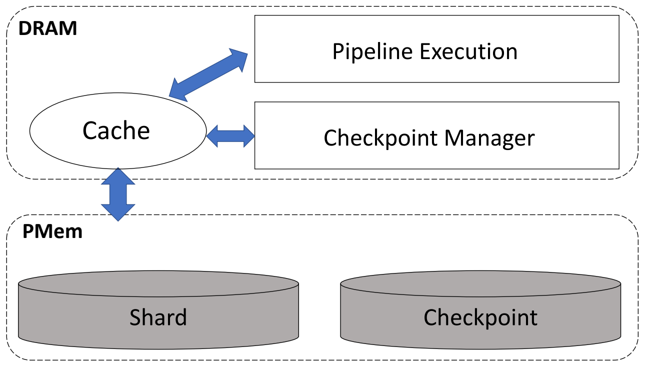

Fig. 4. The PS node design of OpenEmbedding

DRAM system. As we increase the GPU number to 8 and 16, the training time gap further increase to 55.8% and 1.27×, respectively. On the other hand, PMem Hash takes 1.16×, 1.85× and 3.17× longer training time than the DRAM-PS solution.

Design Implementation: The huge performance degradation is prohibitively unacceptable for directly running on PMem Hash. Therefore, we propose to take advantage of both DRAM and PMem to address the performance as well as the scalability challenges of DLRM workloads.

Observation 2: The batch-based DLRM training poses challenges in performance degradation and data consistency. Synchronous checkpointing slow-down the overall training performance even if we use the fast PMem as the checkpoint device. On the other hand, unlike holding the embedding entries in DRAM, the PMem-based parameter server already stores the embedding entries on persistent PMem. Using the traditional checkpoint backup model in such a case would incur unnecessary PMem write.

Design Implementation: Training data used in industrial-grade DLRM systems typically reaches terabytes or even petabytes in size. These training data are divided into multiple batches of equal size, which are then trained sequentially by the GPU workers. To ensure the model accuracy, the checkpoint and failure recovery of the parameter server needs to provide batch-based consistency, which means the atomicity is required for updating all the training batches between two checkpoints. Although the entries stored in PMem can still be accessed when recovering from failures, this alone is insufficient to guarantee correctness. Directly updating entries in PMem during training process cannot guarantee such batch-level consistency. Unfortunately, the state-of-the-art PMem-based data structures [19], [20], [29]–[33] do not support such batch-based atomicity. Thus, we develop a new workflow for updating PMem data to achieve batch-aware checkpointing.

IV. Design Overview of OpenEmbedding

OpenEmbedding is a high-performance cost-efficient parameter server system designed to store embedding tables for synchronous DLRM training using PMem. OpenEmbedding's bigger capacity, lower cost per GB and quick recovery also make it a good choice for large-scale DLRM applications. Embedding entries are partitioned into a a series of PS nodes. Openembedding identifies the correct PS node by hashing the entry's id, and then Openembedding sends the id to the corresponding node to perform embedding lookups and gradient pushes back accordingly. Figure 4 shows the PS node design of OpenEmbedding. Compared with the design of existing works [7]–[9] (as shown in Figure 1), our PS node design features the usage of PMem. Embedding entries and checkpoint snapshots are stored in PMem. DRAM is used as a cache for keeping the hot and recent embedding entries in DRAM. We acknowledge that there have been existing studies on design PMem-aware cache replacement policies [20], [23], [24], which can be used in our scenarios for improving the cache efficiency. On the other hand, we study the performance of DLRM in details, and find that we are able to develop a simple and effective pipeline processing approach to eliminate the latency in accessing the embedding entries in PMem. The key function is implemented in the component of *pipeline execution* (details in Section V-A).

As for checkpoint, we develop a checkpoint manager which is responsible for scheduling the checkpoint. We take advantage of the local persistence of PMem as well as its much faster I/O performance than SSDs. Specifically, by taking advantage of DRAM cache replacement, we develop a lightweight synchronous checkpointing scheme that is specially co-designed with the pipelined cache to reduce the checkpoint run-time overhead (details in Section V-B). We present how we implement and integrate OpenEmbedding into the two popular DLRM platforms, Tensorflow and Keras, in Section V-C.

V. Design and Implementation

A. Handling Pull Requests via DRAM Cache

Pipelining data preparation and model training is the key to the performance of the DLRM training system. Like most of the state-of-the-art training systems [2], [4], [5], OpenEmbedding adopts batch-based training. Furthermore, to address the performance degradation of the training system when directly

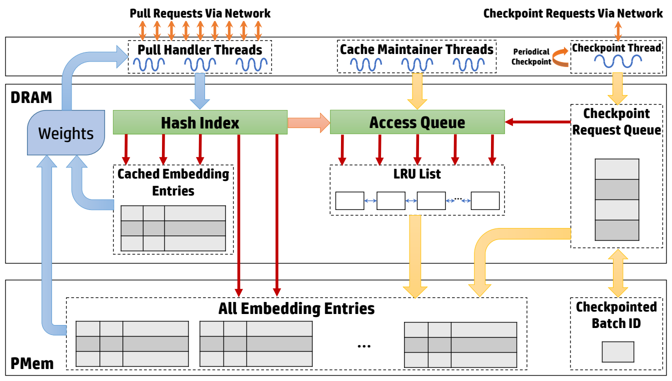

Fig. 5. Pipelined workflow of pull, cache management and checkpointing

Algorithm 1 Pull Weights

```
1:  Input: Keys, Current Batch Id $n$
2:  Output: Success
3:  $lock$.acquireRead()
4:  for $k_i$ in Keys do
5:      $entry \leftarrow$ find($k_i$)
6:      if $entry$ is New then
7:          $lock$.acquireWrite()
8:          $entry.key$ = $k_i$
9:          $entry.weight \leftarrow$ initializeInDRAM($entry$)
10:         $entry.version$ = $n$
11:         $lock$.releaseWrite()
12:     end if
13:     $entries \leftarrow$append($entry$)
14: end for
15: sendToWorker($entries.weight$)
16: $lock$.releaseRead()
17: $cacheReplacementThreadPool \leftarrow$asyncTask($entries$)
```

replacing DRAM with PMem, OpenEmbedding uses DRAM caches to accelerate the pull request of hot entries.

The left part of Figure 5 illustrates a typical workflow of handling pull requests in OpenEmbedding. Specifically, there are multiple threads pre-allocated to handle the concurrent pull requests coming from the network. Every thread looks in the DRAM-based *Hash Index* to locate the corresponding embedding entries in either DRAM or PMem. These pointers are implemented in the similar way as the smart pointers proposed in earlier work [21], which uses the lowest bit to indicate whether the target embedding entry is in DRAM or PMem. Based on the list of keys provided by the pull request, the corresponding weights of those embedding entries are copied from either DRAM or PMem to the network buffer. Once all the weights are ready, the response is immediately constructed and sent back to workers. At the same time, every accessed entry is also appended to the *Access Queue* which will be consumed by the *Cache Maintainer Thread* during cache replacement at a later stage.

The pseudo code of the entire procedure of handling pull requests is shown in Algorithm 1. Note that except for the first time an embedding entry is accessed (line 6-12) that happens only in the first epoch, the whole pull process is highly parallel and protected by a read lock. Therefore, processing the pull requests is very lightweight. This allows us to achieve a similar performance as the DRAM-only system as long as the cache holds most of the hot entries in DRAM.

B. Pipelined Cache Management and Light-weight Checkpointing

To achieve better performance, we use a LRU-like cache replacement policy to keep hot embedding entries in DRAM. However, different from traditional approaches in which the maintenance of the LRU list and necessary cache replacement are immediately triggered upon cache misses, we delay their execution to make it pipelined with the training phase. In addition, as the entries pulled before training and those updated after training in a single batch are always the same, we do not modify the LRU list even if the entries are updated after training. Thus, we can hide the maintenance overhead entirely without losing track of the access pattern for each batch.

Due to the volatility of DRAM, traditional checkpointing needs to pause all updates to the weights until the entire (or partial, if delta checkpointing is supported) model in DRAM is dumped to the underlying persistent storage such as HDDs/SSDs. In contrast, OpenEmbedding stores all embedding entries persistently in PMem. Therefore, we propose light-weight checkpointing to minimize its negative impact to the performance of the training system.

Specifically, we implement a novel checkpoint method which is co-designed with cache replacement. By carefully handling the batch ID of the on-going checkpoint and selectively flushing the cached embedding entries from DRAM and PMem, batch-aware checkpointing can be achieved efficiently as part of the pipelined cache's operation. Specifically, as shown in the right most part of Figure 5, checkpoint requests can be issued by either the users manually or the periodical checkpoint thread automatically. For each checkpoint request, the latest batch that completed training is appended to the *Checkpoint Request Queue*. Note that no other work needs to be done at this time. The batch ID of the latest completed checkpoint is stored in PMem and updated during the cache maintenance.

The middle part of Figure 5 illustrates the tightly coupled procedures of cache replacement and checkpointing. The pseudo code of the entire procedure is shown in Algorithm 2. OpenEmbedding uses multiple dedicated threads to do the cache replacement which are activated if and only if all pull requests for the current batch are completed (line 6-8). Each *cache maintainer thread* repeatedly pops the head of *Access Queue* (line 10-11) to retrieve every embedding entry accessed. Depending on where it resides, different workflows are executed to perform cache replacement and checkpointing at the same time:

- For the cached embedding entries (line 12-17), it starts with checking the batch ID of the on-going checkpoint and the cached entry. If former is larger, flushing the

Algorithm 2 Cache Replacement & Checkpoint

```
1:  **Input:** Current Batch Id $n$
2:  **Output:** Success
3:  **Thread**($i$) in $cacheReplacementThreadPool$
4:  $cp \leftarrow$ checkpointReqQueue.head()
5:  **while** $true$ **do**
6:    **if** Still has Pull ops not complete in batch $n$ **then**
7:      sleep & continue
8:    **end if**
9:    $lock$.acquireWrite()
10:   $entries \leftarrow$ getAsyncTask()
11:   **for** each $entry$ in $entries$ **do**
12:     **if** $entry\ in\ DRAM$ **then**
13:       **if** $entry.version <= cp$ **then**
14:         flushToPMem($entry$)
15:       **end if**
16:       $entry.version = n$
17:       reorder($entry$)
18:     **else**
19:       loadToDRAM($entry$)
20:       $entry.version = n$
21:       reorder($entry$)
22:       **if** cache is full **then**
23:         $erase\_entry \leftarrow$ findOldestEntry()
24:         **if** $erase\_entry.version > cp$ **then**
25:           **PMem**.atomicUpdateCheckpointId($cp$)
26:           checkpointReqQueue.pop()
27:           $cp \leftarrow$ checkpointReqQueue.head()
28:         **end if**
29:         flushToPMem($erase\_entry$)
30:         eraseFromDRAM($erase\_entry$)
31:       **end if**
32:     **end if**
33:   **end for**
34:   $lock$.releaseWrite()
35: **end while**
```

entry from DRAM to PMem is needed to keep this version of entry persistent in PMem. This is crucial to the correctness of checkpointing because this version of weights will be the correct one for the on-going checkpoint if there are no more updates afterwards before the batch of on-going checkpointing. Then the batch ID of the cached entry is updated to be that of the current batch. Finally, The *LRU List* is updated by moving the accessed entry to the head of the list.

- For entries not in the cache (line 18-30), a victim cached entry is selected to be evicted from the cache according to the *LRU List*. Due to the nature of LRU, the victim entry is always the one with the oldest batch ID in the cache. Therefore, if the batch ID of the selected entry is larger than that of the on-going checkpoint (the head of the *Checkpoint Request Queue*). The *Checkpointed Batch ID* on PMem will be set to batch ID of the on-going checkpoint persistently and the head of the *Checkpoint Request Queue* will be removed to indicate the on-going checkpoint is done. The evicted entry is then persistently written back to PMem to free up the cache entry so that the recently accessed entry in PMem can be read into the cache. Finally, the new cached entry is inserted as the head of the *LRU List*.

C. Other Implementation Details

Recovery: When a system failure occurs, Openembedding can restart training from the latest successful checkpoint instead of starting from scratch. To guarantee that the embedding entries belong to the latest successful checkpoint (*Checkpointed Batch ID* on PMem) is recoverable on a system failure, we rely on the underlying space manager of PMem to prevent them from being overwritten by the newer versions flushed to PMem. The space manager will recycle the space of these entries once the new checkpoint is done. Thus, the recovery can be done by (1) scanning all the embedding entries in PMem and discarding those with batch IDs larger than the *Checkpointed Batch ID*, (2) and then reconstruct the hash index in DRAM.

Integration with Tensorflow and Keras: OpenEmbedding has been integrated into the Tensorflow/Keras framework. We implement several Tensorflow operators including the function of 'PullWeights', 'PushGradients', 'UpdateWeights', etc. We have our own embedding class that inherits from Keras's embedding layer, and replace the embedding related operators with our own. These newly integrated Tensorflow operators use several high-performance communication technologies such as RDMA with low overhead RPC. OpenEmbedding utilizes these Tensorflow operators to transfer the embedding entries from GPU workers to the backend parameter servers.

VI. Experimental Results

A. Experiment Setup

TABLE III
Parameter Server Configurations

Name	In-memory Storage Engine	Checkpoint Configuration
DRAM-PS	DRAM-based hash @DRAM	Incremental Checkpoint
PMem-OE	Pipelined cache replacement @ DRAM + PMem	Proposed Checkpoint
Ori-Cache	Facebook concurrent hash+STL list @DRAM + PMem	Incremental Checkpoint

Comparison Approaches. The configurations of the parameter servers are listed in Table III. 'PMem-OE' is our proposed OpenEmbedding solution, and 'DRAM-PS' is a pure DRAM version of OpenEmbedding that is implemented according to the classic parameter server's standards. We also make it open-sourced within the same repository of the OpenEmbedding. 'Ori-Cache' is implemented by using Facebook's concurrent hash map [26] and STL's list [27]. Both PMem-OE and Ori-Cache use LRU as their caching algorithm. Due to the long training time of one epoch, DRAM-PS and Ori-Cache apply

TABLE IV
DETAILS OF CHECKPOINT VARIANTS

Name	Dense Features	Sparse Features
Proposed Checkpoint	Tensorflow's checkpoint	Batch-awre checkpoint
Incremental Checkpoint	Tensorflow's checkpoint	Incremental checkpoint from the most recent work [11]
No Checkpoint	No checkpoint	No checkpoint
Sparse Only	No checkpoint	Batch-aware checkpoint

the 'Incremental Checkpoint' by default, while PMem-OE applies our 'Proposed Checkpoint'. Table IV lists the detail of the checkpoint configurations. A DLRM consists of dense features in GPU and sparse features on parameter servers, and the dense features only occupy a very small portion (less than 1% of the total size in our case). 'Proposed Checkpoint' backs up the dense features by calling the checkpoint procedure that comes with Tensorflow and backs up the sparse features by using the proposed batch-aware checkpoint. 'Incremental Checkpoint' applies the same checkpointing strategy for the dense features, while applying the state-of-art incremental checkpointing published in the most recent work [11] for the sparse features. 'No Checkpoint' configuration does not execute any checkpoint during the training process. To better understand the overhead of the batch-aware checkpoint, 'Sparse Only' configuration removes the checkpointing overhead of the dense part.

For all checkpoint configurations, we choose the PMem as the checkpoint device for a fair comparison. According to Young's formula [34] and the mean time to failure reporting by Facebook [6], we set the checkpoint interval to be 20 minutes for our study. We studied the impact of the checkpoint intervals, and found similar performance comparison among baselines (results omitted due to space limits). We refer to this as the default setting and evaluate the end-to-end training performance.

We use an open-source framework [35] to execute the DeepFM [36] algorithm, applying horovod [37] on top of all the above solutions to execute training on multiple GPUs. The default batch size is 4096 and the embedding dimension is 64.

Hardware Setup. We conducted our experiments on Alibaba Cloud, and all the price mentioned in the paper is the "Pay-As-You-Go" price. Four GPU machines (instance type: ecs.gn6v-c8g1.8xlarge including 32 cores, 128GB DRAM, four NVIDIA V100 GPUs) are used as computation nodes. To store 500 GB DLRM in memory, several DRAM machines (instance type: ecs.r6e.13xlarge, 52 cores, 384GB DRAM) and PMem machines (instance type: ecs.re6p.13xlarge, 52 cores, 192 GB DRAM, 756 GB PMem) are used. The training data is stored in Alibaba's Extreme NAS file system. All the nodes, as well as the NAS servers, are connected through a 30 Gb intranet.

Experiment Outline. Firstly, we evaluate the end-to-end training performance using a large-scale real-world DLRM workload (Sec. VI-B). Then we study the impact of pipelined cache management (Sec. VI-C) and batch-aware checkpointing (Sec. VI-D) with other approaches in order to evaluate their impact on the training process. Additionally, we compare the

TABLE V
PRICE OF PARAMETER SERVERS

	DRAM-PS	PMem-OE	Ori-Cache
#of Machine	2 Large Dram Server	1 PMem Server	1 PMem Server
Instance Type	r6e.13xlarge	re6p.13xlarge	re6p.13xlarge
DRAM capacity per machine (GB)	384	192	192
PMem capacity per machine (GB)	0	756	756
Training time per epoch (Hours)	5.75	5.33	7.01
$ per Hour (PS only)	6.07	3.80	3.80
$ per Epoch (PS only)	34.9	20.3	26.6

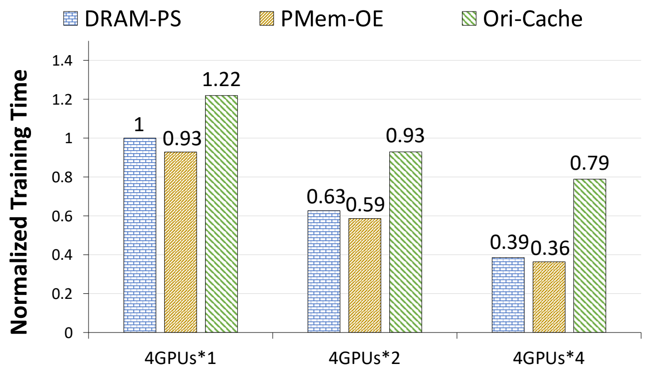

Fig. 6. Training time comparison (All values normalized to that of the DRAM-PS on 4 GPUs)

recovery time (Sec. VI-E). As a sanity check, we compare the performance of OpenEmbedding with a popular training system by Google (Tensorflow). As the Tensorflow's parameter server does not support synchronous training in the distributed setting, we are unable to deploy the DLRM on the Tensorflow embedding layer for the real-world DLRM model whose size exceeds the memory capacity of a single server. Thus, we use a smaller data set Criteo [38] in the last section (Sec. VI-F).

B. Overall Comparison

We conduct an overall comparison on cloud. The size of the DLRM is more than 500 GB. We first deploy the DRAM-PS on 2 DRAM servers with each has 384 GB DRAM, which is the most cost-effective solution we can choose on cloud according to the minimal memory capacity requirement. PMem-OE and Ori-Cache are deployed on two separate PMem servers with 768 GB PMem for each. We first use one GPU server with 4 V100 GPUs as the training workers. As shown in Table V, DRAM-PS, PMem-OE and Ori-Cache training take 5.75, 5.33, and 7.01 hours, respectively, when using 4 GPUs per machine. Thanks to PMem's lower per-GB price, PMem-OE saves 42% storage cost over DRAM-PS. Meanwhile, PMem-OE saves 24% storage cost compared with Ori-Cache, since PMem-OE significantly reduces the training time.

We further scale the number of machines to study the impact of training time. As shown in Figure 6, PMem-OE consistently outperform both DRAM-PS and Ori-Cache when the number of the GPU machines increased from 1 to 4. In particular, the training time of PMem-OE is 7.2% (4 GPUs), 6.4% (8 GPUs) and 5.6% (16 GPUs) shorter than DRAM-PS, and 23.8% (4 GPUs), 36.9% (8 GPUs) and 53.8% (16 GPUs) shorter than

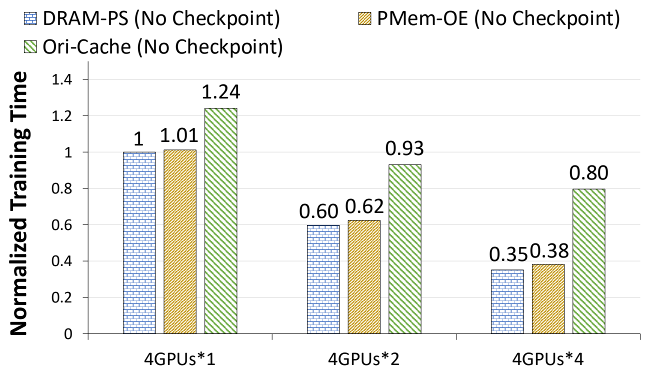

Fig. 7. Pipelined cache performance comparison
(All values normalized to that of the DRAM-PS on 4 GPUs)

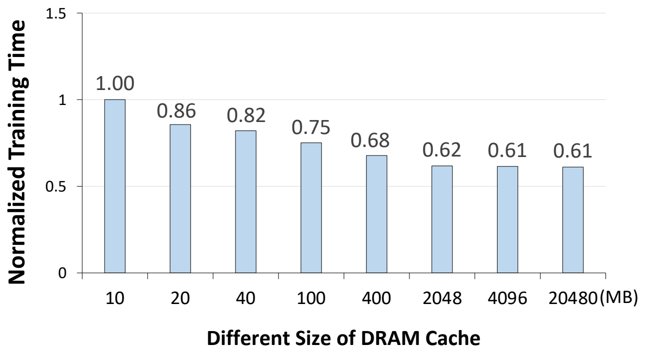

Fig. 8. Impact of DRAM cache size
(All values normalized to that of PMem-OE with 10 MB DRAM cache)

Ori-Cache. The pipelined cache management and batch-aware checkpoint together contribute to the reduction of training time, and we will discuss them separately in the following sections.

C. Evaluation on Pipelined Cache Management

This section does not execute checkpoints for all configurations in order to better study the individual impact of pipelined cache management.

1) Performance Comparison : As shown in Figure 7, the training time of the DRAM-PS reduces 40% and 65% when the number of the GPUs increases from 4 to 8 and 16, respectively. In contrast, Ori-Cache takes 1.24× (4 GPUs), 1.56× (8 GPUs) and 2.27× (16 GPUs) training time as long as the DRAM-PS. We can observe that the more GPUs involved, the slower Ori-Cache performs. As the number of GPUs increases, the parameter server has to perform exponentially more pull and update operations. These requests are concentrated at the beginning and end of each batch due to the synchronous training requirement of DLRM, which significantly rise the parallelism overhead. PMem-OE consistently outperforms Ori-Cache on all training configurations. In particular, the training time of PMem-OE is 18.4% (4 GPUs), 33% (8 GPUs) and 52.1% (16 GPUs) shorter than the Ori-Cache. Meanwhile, the training time gap between PMem-OE and performance upper bound (DRAM-PS) is only 8.7% when using 16 GPUs. Such gaps are further reduced to 4.3% and 1.2% when the number of the GPUs decreases to 8 and 4, respectively.

2) Impact of DRAM Cache Size: We fix the number of GPUs to 16, and study the impact of varying the size of the DRAM cache in PMem-OE. Figure 8 shows that training time reduces 14.4%, 18%, 24.9%, 32.2% and 38.2% when the cache size changes from 10 to 20, 40, 100, 400, 2048 MB, respectively. However, when the cache size is more than two GB, the improvement becomes much smaller. PMem-OE with a 20 GB DRAM is only 1% faster than that with a two GB DRAM cache. This is due to the skewed nature of the accesses. Similar phenomena can be observed in the previous study [2].

3) Dissecting Performance: We study the performance contribution of different optimizations in PMem-OE. Based on the findings in the last section, we fix the DRAM cache to two GB and test the combinations of enabling or disabling the cache and pipeline inside of the PMem-OE by using 16 GPUs.

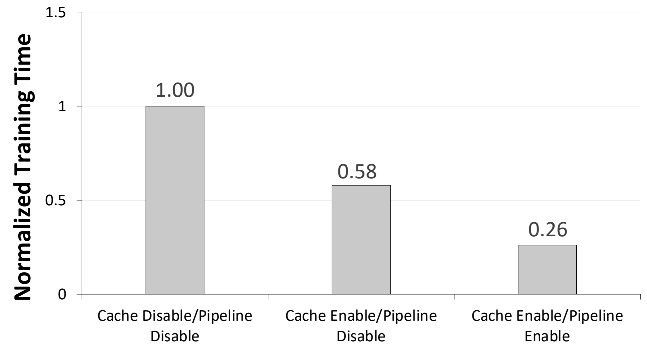

Fig. 9. Individual improvement of PMem-OE
(All values normalized to that of disable both cache and pipeline)

Compared with the first two columns in Figure 9, the cache mechanism reduces 42.1% training time, which follows a similar trend as the observation in the last section. If we only consider the effect of enabling and disabling the pipeline, it reduces 54.9% training time (comparing the training time of the second and the last columns). The combination of both cache and pipeline mechanism brings 73.9% training time reduction in total. The percentage of speedup from different components may vary according to the characteristics of the workload.

4) Performance under Different Distribution Skews: In Figure 10, we sorted features according to their access frequency. We can see that it follows an exponential distribution [39]. To study the impact of different workloads, we generated workloads of different skews by modifying the parameters of the exponential distribution while keeping the total amount of accesses the same. Compared to the original DLRM workload, we generated a more skew workload and a less skew workload whose distribution formulas are shown in Figure 10. We then execute end-to-end DLRM training using 16 GPUs, fixing the cache sizes of both PMem-OE and Ori-Cache to two GB.

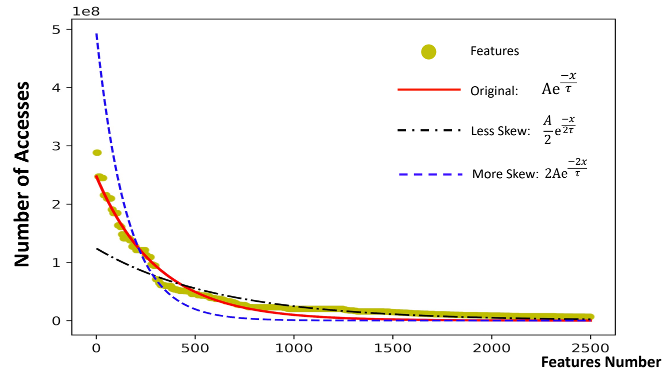

Fig. 10. Workload fitting and distribution adjustment

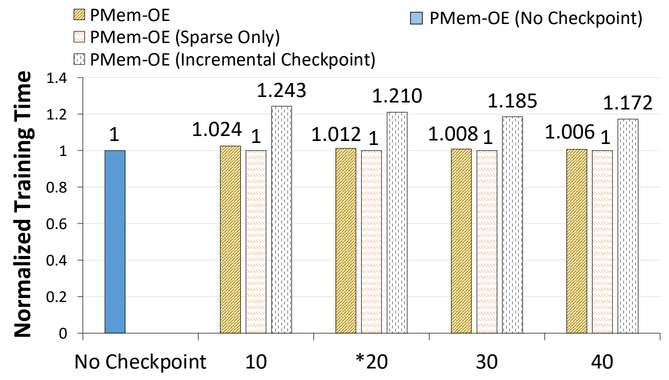

Fig. 12. Training time with different checkpoint interval
(All values normalized to that of training without checkpoint)

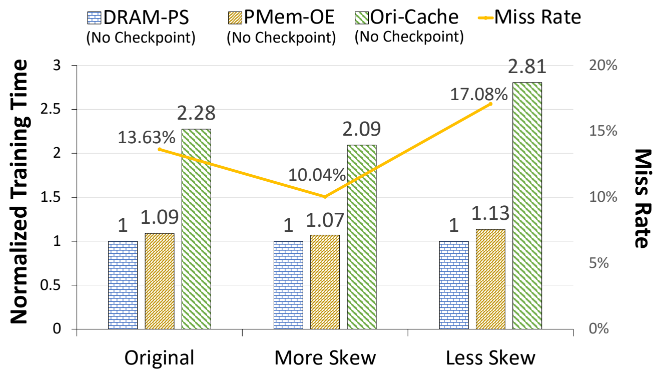

Fig. 11. Training time & miss rate under different distribution
(All values normalized to that of DRAM-PS)

PMem-OE and Ori-Cache have the same cache miss rate under the same distribution, since both of them use LRU as their cache replacement algorithm. Figure 11 shows that PMem-OE achieves good performance under the different workloads. As shown in the leftmost part of Figure 11, the miss rate is 13.63% when training the original DLRM workload. When the distribution becomes more skew, the miss rate reduces to 10.04% and the performance gap between DRAM-PS and PMem-OE is reduced from 9% to 7%. As shown in the rightmost part of Figure 11, cache miss rate increases to 17.08% when the workload becomes less skew. Compared with the original distribution, we observe that the training time with Ori-Cache increases by more than 20%, while that for PMem-OE only increases by less than 5% even though both uses the same cache algorithm. This is because that our proposed pipelined cache management hides the most expensive PMem write as well as the LRU maintenance overhead behind the GPU training process.

D. Evaluation on Batch-aware Checkpoint

As shown in Section VI-C, since the training time of a single epoch already lasts more than 5 hours, executing checkpoints periodically is the default option to avoid re-training from the very beginning upon a system failure. More frequent checkpointing reduces the re-training time (the time between the failure and the latest checkpoint) while bringing more extra overhead of checkpointing. As discussed in Section VI-A, we use 20 minutes as the default setting and discuss the impact of checkpoint frequency later in section VI-D1.

1) Checkpoint Overhead & Impact of Checkpointing Frequency: We use 4 GPU servers with a total of 16 GPUs to study the overhead of the batch-aware checkpoint, as well as the effect of doing checkpointing with different intervals. Figure 12 shows the training time of one epoch, and a lower value indicates a better result. PMem-OE(No Checkpoint) delegates the performance upper bound. As shown in Figure 12, more frequent checkpointing brings more extra overhead. Compared with PMem-OE(No Checkpoint), PMem-OE only brings 2.4% additional training time when the checkpoint executing every 10 minutes. This additional training time decreases to 0.6% when the checkpointing interval is set to 40 minutes. If we remove the checkpointing overhead of the dense part handled by Tensorflow, PMem-OE(Sparse Only) shows that our proposed batch-aware checkpointing does not impose any additional overhead with different checkpointing frequencies. Thanks to our lightweight checkpointing scheme that co-designs cache replacement with the checkpointing process, PMem-OE(Sparse Only) executes checkpointing smoothly with its overhead well hidden behind the GPU training period. On the other hand, compared with PMem-OE, PMem-OE(Incremental Checkpoint) generates 21.4%, 19.6%, 17.6%, and 16.5% higher overhead when the checkpoint is executed every 10, 20, 30, and 40 minutes, respectively. A large number of PMem writes are generated during checkpointing, which interfere with the PMem's read & write operations required by the training system.

2) Checkpoint Overhead with Different Amount of GPUs: We set the checkpoint interval as default (20 min) and adjust the number of GPUs to study the variant of the checkpoint overhead. Figure 13 shows that no matter how many GPUs are used for the training, PMem-OE has only a very low runtime overhead. In particular, compared with PMem-OE(No Checkpoint), PMem-OE consistently brings 1.2% additional checkpoint overhead when the number of the GPUs increases from 4 to 16. As discussed in Section VI-D1, PMem-OE(Sparse

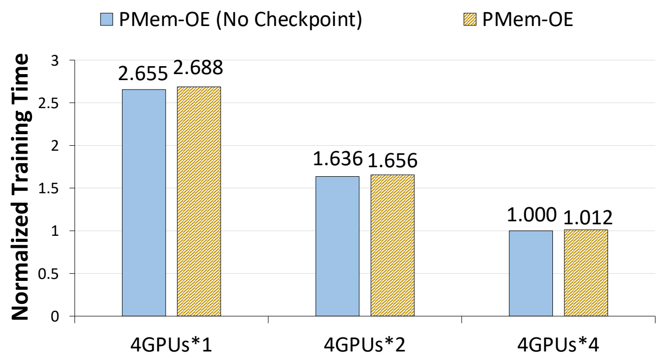

Fig. 13. Normalized training time with different number of GPUs
(All values normalized to that of training without checkpoint on 16 GPUs)

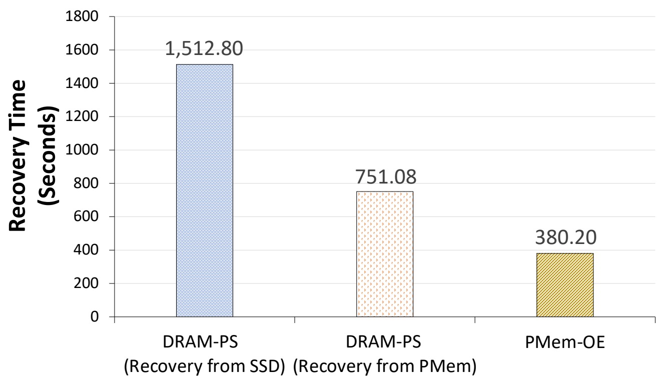

Fig. 14. Recovery time comparison

Only) does not bring extra overhead even with 16 GPUs. This means that the tiny extra training time comes from Tensorflow when checkpointing dense entries from GPU to local storage. As the weights of the dense model on multiple GPUs remain synchronized at the end of each batch, we only dump the dense model from any of the GPUs. Therefore, increasing the number of GPUs does not lead to additional checkpoint overhead.

E. Recovery Performance

The DRAM-PS needs to read the backup embedding entries from the persistent storage before inserting them into DRAM during the recovery process. This overhead dominates total data recovery time. As shown in Figure 14, DRAM-PS takes 1512.8 seconds to load the checkpoint file from SSD and restore the training environment in DRAM. When we store the checkpoint files on a fast PMem device, DRAM-PS still needs 751.08 seconds for recovery. PMem-OE only takes 380.2 seconds to recover since the large number of embedding entries is already stored in PMem. Recovery time in PMem-OE is dominated by the scanning of data in PMem and reconstruction of the hash index. We can further speed up the process by partitioning a single embedding table into several parameter server processes [7], thereby parallelizing both scanning and the rebuilding can be executed parallel on each part of the embedding tables.

F. Performance Comparison with Tensorflow

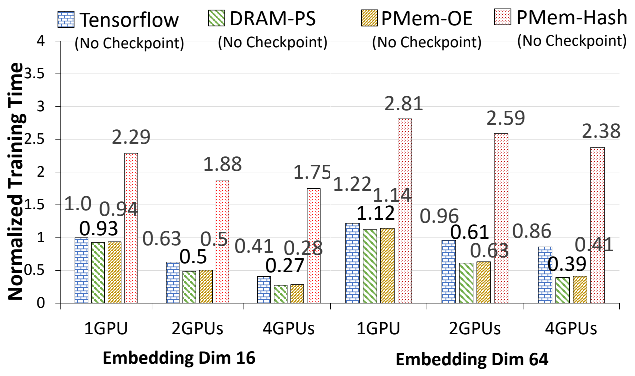

Fig. 15. Performance comparison on criteo
(All values normalized to that of Tensorflow with embedding dim 16 on 1 GPU)

In this section, we compare OpenEmbedding with Tensorflow on criteo kaggle dataset [38]. We do not execute checkpoints for all configurations in this section. Two embedding dimensions (16 and 64) of the sparse parameters are examined. All the results are normalized to that of Tensorflow with dimension 16 on 1 GPU. We configure 128MB DRAM as the cache of the PMem-OE, which is 6.4% and 1.6% of the total size of the embedding tables when embedding dimension is 16 and 64, respectively.

Figure 15 shows that PMem-OE consistently outperforms Tensorflow when the amount of the GPU increased from 1 to 4. In particular, PMem-OE reduces 6.3% (1 GPU), 19.5% (2 GPUs), and 30.1% (4 GPUs) of the training process when the embedding size is 16. The training time reduction further increase to 6.4% (1 GPU), 34.2% (2 GPUs) and 52% (4 GPUs) when embedding size set to 64, respectively. DRAM-PS achieves the best performance. Thanks to the caching and pipeline mechanism, the performance gaps between DRAM-PS and PMem-OE are all below 5%. Similar to Section III-B, the bar of PMem-Hash presents the performance if we directly replace the DRAM-based parameter server engine with the PMem-based data structure [28]. Because of the lower speed of PMem, PMem-Hash requires at most $4.3\times$ training time than Tensorflow. Such performance gap further increases up to $6.3\times$ compared to DRAM-PS.

VII. Conclusion

We propose a DLRM training system, OpenEmbedding, which takes advantage of Intel's Optane Persistent Memory Module (PMem). OpenEmbedding has two key designs based on PMem: an efficient training pipeline to reduce the I/O data access cost as well as a novel lightweight batch-aware checkpointing mechanism to improve training reliability. We have integrated OpenEmbedding into the Tensorflow/Keras framework and the project is now is available on GitHub.

References

[1] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[2] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair, "Accelerating recommendation system training by leveraging popular choices," *PROCEEDINGS OF THE VLDB ENDOWMENT*, vol. 15, no. 1, 2022.

[3] U. Gupta, S. Hsia, M. Wilkening, J. Pombra, H.-H. S. Lee, G.-Y. Wei, C.-J. Wu, D. Brooks *et al.*, "Recpipe: Co-designing models and hardware to jointly optimize recommendation quality and performance," *arXiv preprint arXiv:2105.08820*, 2021.

[4] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, "Aibox: Ctr prediction model training on a single node," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 319–328.

[5] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.

[6] "Check-n-run: a checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/eisenman

[7] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[8] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[9] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 566–579, 2018.

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

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

[12] E. Rojas, A. N. Kahira, E. Meneses, L. B. Gomez, and R. M. Badia, "A study of checkpointing in large scale training of deep neural networks," *arXiv preprint arXiv:2012.00825*, 2020.

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

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

[15] H. Rong, Y. Wang, F. Zhou, J. Zhai, H. Wu, R. Lan, F. Li, H. Zhang, Y. Yang, Z. Guo *et al.*, "Distributed equivalent substitution training for large-scale recommender systems," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020, pp. 911–920.

[16] Intel, "Intel® optane™ dc persistent memory," "https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2015, last accessed on 02-July-2020.

[17] K. Liu, "Checkpoint-free fault tolerance for recommendation system training via erasure coding," Ph.D. dissertation, Carnegie Mellon University Pittsburgh, PA, 2020.

[18] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, 2015, pp. 167–181.

[19] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *arXiv preprint arXiv:2003.07302*, 2020.

[20] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid pmem-dram key-value store," *PROCEEDINGS OF THE VLDB ENDOWMENT*, vol. 14, no. 9, pp. 1544–1556, 2021.

[21] C. Chen, J. Yang, M. Lu, T. Wang, Z. Zheng, Y. Chen, W. Dai, B. He, W.-F. Wong, G. Wu *et al.*, "Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 799–812, 2021.

[22] C. Wang, G. Brihadiswarn, X. Jiang, and S. Chattopadhyay, "Circ-tree: A b+-tree variant with circular design for persistent memory," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 296–308, 2021.

[23] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.

[24] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[25] C. Labs, "Terabyte click logs," https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/, 2015.

[26] Facebook, "Facebook concurrent hash map," https://github.com/facebook/folly/blob/main/folly/concurrency/, 2021.

[27] A. Stepanov, "std::list," 2019. [Online]. Available: https://en.cppreference.com/w/cpp/container/list

[28] Intel, "libpmemobj," 2019, https://github.com/pmem/libpmemobj-cpp/, Last accessed on 02-July-2020.

[29] G. Psaropoulos, I. Oukid, T. Legler, N. May, and A. Ailamaki, "Bridging the latency gap between nvm and dram for latency-bound operations," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, no. CONF. ACM, 2019.

[30] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. ACM, 2016, pp. 371–386.

[31] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "{WORT}: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 257–270.

[32] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[33] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, 2017, pp. 1753–1758. [Online]. Available: https://db.cs.cmu.edu/papers/2017/p1753-arulraj.pdf

[34] N. El-Sayed and B. Schroeder, "Checkpoint/restart in practice: When 'simple is better'," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 84–92.

[35] W. Shen, "Deepctr: Easy-to-use,modular and extendible package of deep-learning based ctr models," https://github.com/shenweichen/deepctr, 2017.

[36] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: a factorization-machine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.

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

[38] C. Labs, "Kaggle display advertising challenge dataset," https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/, 2014.

[39] Wikipedia contributors, "Exponential decay — Wikipedia,," 2004, [Online; accessed 22-April-2022]. [Online]. Available: https://en.wikipedia.org/wiki/Exponential_decay