NVREC: A High-Performance Recoverable Recommendation System for Non-Volatile Memory

PING CHEN, The State Key Laboratory of Blockchain and Data Security at Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou City, China
SHUIBING HE*, Zhejiang University, Hangzhou, China and Zhejiang Key Laboratory of Big Data Intelligent Computing, Hangzhou, China
PEIYI HONG, Zhejiang University, Hangzhou, China
XUECHEN ZHANG, Kennesaw State University, Marietta Campus, Marietta, United States
SILING YANG, Zhejiang University, Hangzhou, China
TONG WU, Zhejiang University, Hangzhou, China
GANG CHEN, Zhejiang University, Hangzhou, China
XIAN-HE SUN, Illinois Institute of Technology, Chicago, United States

Current recommendation systems are designed with the assumption that computer memory is volatile. Thus, they periodically execute checkpointing operations to save model weights and embeddings to slower solid-state drives (SSDs) for failure recovery. Such designs cannot leverage the unique characteristics of non-volatile memory (NVM). This paper proposes a new recommendation model training framework called NVrec, which uses a hybrid memory system consisting of DRAM and NVM to support computing and in-memory checkpointing. NVrec proposes three recommendation-oriented techniques. (1) It partitions embeddings so that only popular vectors are placed in DRAM to hide NVM-induced latency. (2) It maintains a dual-version scheme for in-memory checkpointing: each vector in NVM alternates between a persistent version for crash recovery and a runtime version for training updates. (3) It employs fine-grained parallelism to hide the checkpointing overhead of model parameters in DRAM and GPU. Our experiments demonstrate that NVrec improves recommendation-model training throughput by 1.05×-6.85× and the checkpointing process by more than 5× compared to existing models when evaluated on the synthetic and real-world datasets.

Additional Key Words and Phrases: Recommendation System, DNN Training, Non-volatile Memory, Checkpointing

*Corresponding author.

Authors' Contact Information: Ping Chen, The State Key Laboratory of Blockchain and Data Security at Zhejiang University, Hangzhou, Zhejiang Province, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou City, Zhejiang Province, China; e-mail: zjuchenping@zju.edu.cn; Shuibing He, Zhejiang University, Hangzhou, Zhejiang Province, China and Zhejiang Key Laboratory of Big Data Intelligent Computing, Hangzhou, Zhejiang Province, China; e-mail: heshuibing@zju.edu.cn; Peiyi Hong, Zhejiang University, Hangzhou, Zhejiang Province, China; e-mail: hongpeiyi@zju.edu.cn; Xuechen Zhang, Kennesaw State University, Marietta Campus, Marietta, Georgia, United States; e-mail: xuechen.zhang@wsu.edu; Siling Yang, Zhejiang University, Hangzhou, Zhejiang Province, China; e-mail: slingzjunet@zju.edu.cn; Tong Wu, Zhejiang University, Hangzhou, Zhejiang Province, China; e-mail: wu.tong@zju.edu.cn; Gang Chen, Zhejiang University, Hangzhou, Zhejiang Province, China; e-mail: cg@zju.edu.cn; Xian-He Sun, Illinois Institute of Technology, Chicago, Illinois, United States; e-mail: sun@iit.edu.

ACM 1553-3093/2026/7-ART
https://doi.org/10.1145/3829361

Table 1. Device Characteristics – Comparison of key characteristics of DRAM [43], NVM (Optane DC Persistent Memory [28]), and SSD (Optane DC P4800X [29]).

	DRAM	NVM	SSD
Latency			
Idle Sequential Read Latency	75 ns	170 ns	10 μs
Idle Random Read Latency	80 ns	320 ns	12 μs
Bandwidth (6 DRAM and 6 NVM modules)			
Sequential Read	180 GB/s	91.2 GB/s	2.6 GB/s
Random Read	180 GB/s	28.8 GB/s	2.4 GB/s
Sequential Write	180 GB/s	27.6 GB/s	2.4 GB/s
Random Write	180 GB/s	6 GB/s	2.3 GB/s
Other Key Attributes			
Capacity (per device)	~64–256 GB	~128–512 GB	~0.5–2 TB
Price ($/GB)	10	4.5	2.8
Addressability	Byte	Byte	Block
Media Access Granularity	64 B	256 B	16 KB
Persistent	No	Yes	Yes

1 Introduction

Recommendation systems are widely used in various fields, including e-commerce, online search, social media, and entertainment [21, 67, 68]. These systems commonly use embeddings, which are floating-point vectors, to represent user and item features. Real-world models used in large commercial marketplaces, such as Facebook, Amazon, and Baidu, require billions of embeddings, resulting in the need for hundreds of gigabytes, and even terabytes, of storage space to store model features [18, 33, 66].

Embeddings are usually placed in DRAM close to CPUs or in high bandwidth memory (HBM) on GPUs to minimize access latency [18]. Nonetheless, the memory needs for storing this data could far exceed the capacity of DRAM/HBM. Prior research uses either scale-out or scale-up approaches to overcome this limitation. Scale-out approaches [3] expand available memory space through the use of distributed systems. However, distributed servers are more expensive and their performance is constrained by network bandwidth. In contrast, the scale-up approaches [9, 32, 33] store embeddings on secondary storage devices, such as SSDs. However, such methods' access performance is hindered by the limited I/O bandwidth of storage devices [41] (as discussed in §2.3).

The emerging non-volatile memory (NVM) technology provides a new opportunity to address this issue [26]. First, NVM is byte-addressable and offers higher capacity than DRAM. Second, it provides persistence like SSDs while typically delivering lower-latency and higher-bandwidth access than SSDs [40, 55, 69], as shown in Table 1. In this paper, we present a new recommendation model training framework called NVrec. It uses a hybrid memory system consisting of NVM and DRAM for both computing and in-memory checkpointing. NVrec has the following desired properties.

Hiding the latency of NVM. NVM reads and writes may exhibit higher latency compared to DRAM [15, 26, 41]. The longer latency may limit the training performance of recommendation systems if all embeddings are stored in NVM. Our study finds that training throughput drops by 28%[1] when all embeddings are placed in NVM instead of DRAM (Figure 2(c)). Importantly, recommendation systems exhibit highly non-uniform embedding accesses. A natural strategy is to partition embeddings so that popular ones are placed in DRAM and the rest in NVM.

[1]This decline differs from the devices' peak-performance gap because embedding accesses account for only part of the end-to-end runtime, while the remaining compute and system overheads (e.g., feature interactions and MLP layers) remain unchanged.

Existing approaches, such as dynamic caching [8, 56, 57] and coarse-grained static profiling [33], attempt to realize this idea by identifying and caching popular embeddings in DRAM. Unfortunately, none of these techniques achieves a high hit ratio when DRAM size is limited. In fact, frequency-based placement is a natural solution, but recommendation workloads require a finer vector-level formulation with remapping and retraining support. To address this challenge, we develop an offline, fine-grained, vector-oriented placement approach that exploits embedding access patterns for high-hit-ratio placement, and an efficient vector rebuild technique to support retraining as user preferences evolve.

In-memory checkpointing for embeddings in NVM. Checkpointing techniques have been widely used by recommendation systems to achieve both crash recovery and consistency [17, 42, 44]. As shown in Table 1, NVMs[2] have superior read and write performance compared to SSDs[3] [69]. NVMs may be used as block devices to expedite the recovery and checkpointing process by storing these checkpoints in NVM-based file systems. Unfortunately, the byte-addressability of NVM is not utilized to its highest potential with this approach because these checkpointing data stored in NVM are only accessed when failures occur, and not during an application's training period [8]. An alternative approach is to use NVM as the main memory extension, whereby the stored embeddings in NVMs are updated in place. Unfortunately, this approach does not ensure crash consistency, because if a crash occurs during writes, embedding tables may be left in a partially updated and inconsistent state, making recovery impossible. To address these limitations, NVrec utilizes a dual-version technique which uses in-memory checkpoints for both training and failure recovery. Thus, no extra writes are needed to maintain these checkpoints.

Fine-grained pipelined checkpointing for other data. Besides embeddings, NVrec also needs to save other objects (e.g., metadata in DRAM, embeddings in DRAM, and weights in HBM) to persistent NVM for data consistency. Synchronously checkpointing such data to NVM can be expensive due to the longer write latency of NVM compared to DRAM. To tackle this issue, we propose a fine-grained pipelined checkpointing method to persist such memory data to NVM. Specifically, the metadata and embeddings in DRAM are asynchronously written from DRAM to NVM via a DRAM buffer. Concurrently, the weights on GPU are checkpointed directly to NVM. Once DRAM buffer writes and weights writes are completed, the subsequent training resumes. This approach overlaps the training process with the persistence process to NVM, leading to a reduction in the end-to-end latency of recommendation systems.

In summary, we made the following contributions:

- We present NVrec, a hybrid-memory framework for recommendation system training that co-designs embedding placement, checkpointing, and recovery, with adaptations tailored to the characteristics of recommendation workloads, by leveraging the capacity and non-volatility of NVM.
- We develop a vector-oriented offline placement mechanism for recommendation workloads to reduce NVM-induced access latency and the number of writes to NVM. We further integrate in-memory checkpointing for embeddings in NVM with fine-grained pipelined checkpointing for other training states.
- We implement a software prototype of NVrec and conduct extensive experiments based on real-world and synthetic datasets. Experimental results show that NVrec speeds up the training performance by 1.05×-6.85× and reduces the checkpointing time by more than 5× compared to the state-of-the-art counterparts.

2 Background and Motivation

2.1 Deep Learning Recommendation Systems

Deep learning recommendation models (DLRMs) account for 79% of AI inference cycles at production-scale data centers because of their high prediction accuracy [21]. DLRMs employ multi-layer perceptrons (MLPs) to process

[2]We use "NVM" to denote *byte-addressable persistent memory* [28].
[3]We use "SSD" to denote *block-addressable NAND-based storage* (e.g., SATA/NVMe SSDs [29]).

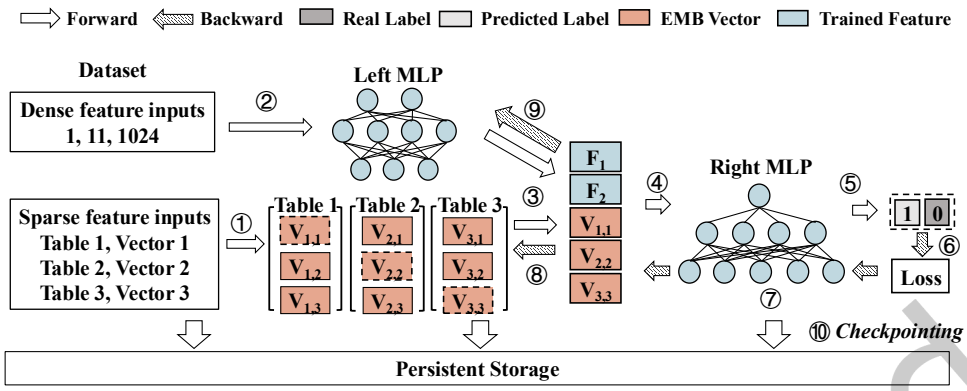


Fig. 1. Illustration of DLRM training workflows.

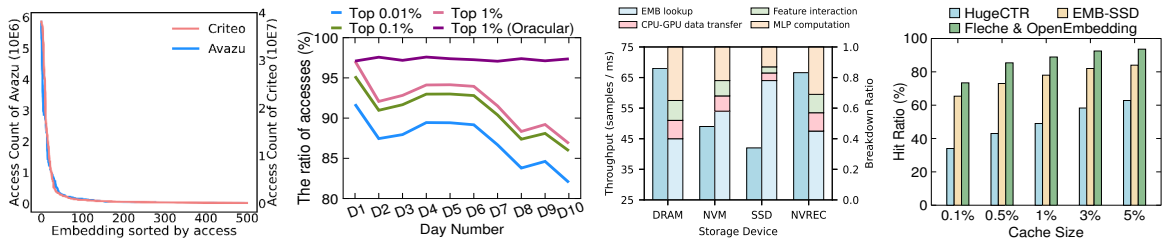
continuous dense features (e.g., age of users) and embedding tables to transform sparse features (e.g., the indices of goods purchased by the users) to low-dimensional embedding vectors. Each row of embedding tables is a vector encoding of a particular sparse feature value. Generally the embedding vectors are stored in the DRAM close to CPUs and the MLPs are stored in the HBM on GPUs.

Figure 1 illustrates the DLRM training workflow, which includes the forward (Steps 1-5), backward propagation and gradient update (Steps 6-9) and checkpointing (Steps 10). When training begins, the system first reads table IDs and vector IDs in batches① as sparse feature inputs. Meanwhile, it reads dense feature inputs for training the MLP (left) on GPU②. After that, the output of looking up embedding vectors using table IDs and vector IDs are used for executing feature interaction on CPU③. It then feeds the interaction features into the MLP (right) on GPU④. The model then produces the prediction⑤. For backward propagation, it computes the loss gradient by comparing the predicted and ground-truth labels⑥ and computes gradients and updates MLP (right) weights⑦. Finally, it computes gradients and updates embedding vectors⑧ and MLP (left) weights⑨ at their original locations, respectively. After multiple training iterations, the system needs to save all model states (Left MLP, Vectors, and Right MLP) in persistent storage to ensure crash consistency⑩ (checkpointing).

Recommendation systems have unique data computing and access characteristics. First, they are trained using datasets which record the accessed embedding vectors in real-world applications. We can analyze the datasets and understand the access patterns during training [35]. Second, recommendation systems frequently access numerous embedding tables, each containing millions of vectors composed of several (e.g., 512) floats [17, 18, 33]. Therefore, systems need vast memory space to store embeddings [17, 18]. Third, recommendation systems exhibit non-uniform vector access patterns as users typically focus on a small subset of items, as shown in Figure 2(a). Fourth, the popularity of embeddings changes over time with the arrival of new requests. For instance, Figure 2(b) illustrates a decrease in the ratio of accesses of the most popular vector on Day 1 to the total accesses over the subsequent 10 days. Hence, upon gathering new daily or weekly training datasets, systems must facilitate model retraining to prevent inaccurate recommendations [45].

2.2 Checkpointing and Recovery

To have a model with good prediction accuracy, we need to iteratively train DLRMs. Due to the long training time, the intermediate training states (e.g., embedding states and MLP weights in DRAMs and HBMs) need to be periodically written to persistent storage as checkpointing files for fault tolerance [16]. Otherwise, an interruption to the training can wipe out the runtime model states and a re-training of the model from the beginning state after failures will be very expensive.

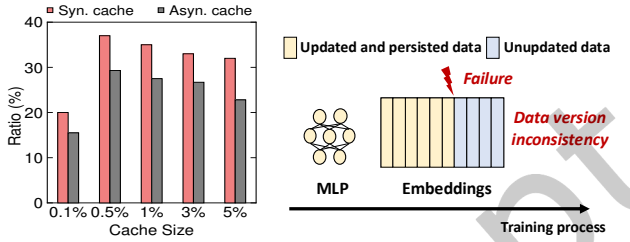


(a) The non-uniform access frequency distribution.

(b) Popularity changes in Avazu.

(c) Model training throughput and runtime breakdown across different storage devices.

(d) The cache hit ratio.



(e) The performance loss due to cache thrashing.

(f) Illustration of directly persisting data in NVM.

Fig. 2. (a) The highly skewed accesses of embedding vectors. (b) Top 1% (Oracular) represents the ratio of accesses to popular embeddings if we only use the top 1% popular embeddings observed on a particular day. (c) The training throughput and runtime breakdown when storing embedding tables on DRAM, NVM, and NVMe SSD, respectively. (d) The hit ratio of the DRAM cache for different embedding placement schemes. (e) Fleche/OpenEmbedding training performance loss under different cache sizes, comparing synchronous cache eviction (red, write-back/eviction on the critical path) and asynchronous cache eviction (gray, write-back overlapped with training). (f) Treating NVM as memory devices and persisting data on NVM during training updates.

Traditionally, the checkpointing mechanism needs at least 1× storage space to save model states in slow block devices of local or remote file system and executes at a decent frequency to avoid long re-training time. Upon failures, the system loads the latest checkpoint for recovery. The checkpointing and recovery operations need to write and read a few hundred GBs of data, resulting in the non-negligible performance and resource loss in training [17, 62]. In Meta, checkpoint-related overhead accounts for 12% of the total job time. It is a dire implication because the production-scale training cluster wastes up to 1156 machine-year worth of computation from 17000 training jobs [42]. Based on the rental price of a cloud server (an 8-GPU server at $40 per hour) [7], checkpoint-related overhead could result in a monetary loss of around 400 million dollars per year.

To reduce the checkpointing overhead, recent works [17, 44] propose asynchronous DNN checkpointing with pipelining to reduce end-to-end model training latency. They create a copy of all parameters and write it to file systems in parallel with training. However, these approaches may incur high space and time overhead, especially when copying large amounts of data, which is inefficient for recommendation system training, as shown in the evaluation in §5.2. Besides, some works use compression to reduce checkpoint size [17, 36, 44]. However, these methods may cause non-negligible model accuracy loss [17], which is not acceptable in recommendation systems because any loss of accuracy may lead to significant engagement and revenue declines [4].

2.3 New Opportunity with NVM

NVM has become available as a standard byte-addressable DDR4 DIMM memory device on the memory bus [14, 59]. As a memory device, NVM's performance is comparable to that of DRAM and much superior to that of block devices, such as SSDs. Moreover, it has a larger storage capability and comes at a much lower cost than DRAM [69]. In this paper, we adopt NVM as a memory device and configure NVM to work in the App Direct Mode. Because the Memory Mode is not persistent, to ensure the persistence of NVM-resident data, we use *clwb* and *sfence* instructions to flush data from CPU cache to NVM.

To leverage NVM in recommendation systems, we can simply use a non-hybrid approach, which stores all embedding tables in NVMs. We study the performance of this approach using the Criteo dataset. We set the embedding dimension to 512 as described in §5. Figure 2(c) shows that the training throughput of the DLRM systems is 68, 49, and 42 samples/ms when the embedding tables are entirely stored in the DRAM, NVM, and SSD, respectively. The system with SSD (i.e., scale-up approach) achieves the worst throughput because of its relatively low I/O bandwidth. The system throughput with NVM is 28% lower than that with DRAM, owing to NVM-induced latency. In contrast, *an approach using hybrid memory* can hide NVM-induced latency because it stores frequently accessed embedding tables in DRAM. For example, our approach NVrec achieves 98% of system throughput compared to storing all embeddings in DRAM.

To understand this performance gap, we further break down the training pipeline into embedding lookup, CPU–GPU data transfer, feature interaction, and MLP computation, as also shown in Figure 2(c). We find that the difference across storage devices mainly comes from embedding lookup. Specifically, embedding lookup accounts for about 40%, 58%, and 78% of the end-to-end runtime when embeddings are stored in DRAM, NVM, and SSD, respectively, whereas the relative contributions of CPU–GPU data transfer, feature interaction, and MLP computation change much less. This result indicates that embedding access constitutes a significant portion of the storage-sensitive behavior in recommendation training. With hybrid placement, NVrec reduces the embedding-lookup fraction to about 45%, close to the DRAM case, thereby recovering most of the lost throughput.

2.4 Design Challenges and Opportunities with Hybrid Memory

How to place embedding vectors in memory? With hybrid memory, we need to determine where to place embedding tables. Prior studies propose dynamic caching and static profiling approaches. In dynamic caching, such as HugeCTR [2], Fleche [57], and OpenEmbedding [8], the system will utilize a caching policy to select embedding vectors stored in the fast access device (i.e., DRAM) for efficient vector reads and writes. However, dynamic caching has two issues when used with NVM. (1) It increases software overhead due to frequent data movement between NVM and DRAM for embedding reads/updates. (2) With large embeddings and highly random accesses, caching can suffer from a reduced effective hit ratio, as quantified in §5.3: compared to the ideal case, the cache hit ratio drops by 5%–20%. Together, these effects degrade end-to-end training throughput by more than 20%, as shown in Figure 2(e).

Some prior work uses asynchronous write-back to overlap computation with data movement. However, this strategy can still be inefficient for large-scale recommendation training. In particular, MLP computation is relatively lightweight, whereas embedding lookups and updates are bandwidth-bound and account for a major portion of end-to-end runtime [63]. Consequently, asynchronous DRAM–NVM flushing has only limited slack to hide behind MLP execution. More importantly, when flushing overlaps with embedding lookup/update, it contends for the same memory bandwidth and can directly slow down those embedding operations, thereby reducing the potential benefit of asynchrony.

Moreover, because embedding accesses are inherently random and repeatedly revisit recently used vectors, asynchronous eviction may flush vectors that are accessed again shortly afterward. This behavior introduces extra NVM-to-DRAM reloads and can lead to write–read thrashing, wasting bandwidth and increasing stall time.

To quantify this effect, in Figure 2(e), we implement an asynchronous eviction baseline that moves cache eviction to a background thread during training. The baseline evicts low-priority (cold) vectors first, progressively flushing them from DRAM to NVM to proactively free space for future accesses. As shown in Figure 2(e), even with asynchronous eviction, Fleche/OpenEmbedding still incur a 16%–29% loss in training throughput.

Static profiling attempts to select the popular embeddings offline according to the access patterns of datasets [33]. Using these patterns, it can store the popular embedding tables in fast devices to avoid frequent embedding movements between DRAM and NVM. For example, the scheme used in EMB-SSD [33] classifies embedding tables into three types including Type-1: high locality with small vector size; Type-2: low locality with small vector size; Type-3: moderate locality with large vector size. It only selectively caches all vectors belonging to Type-1 in the fast devices. However, since each embedding table contains both popular and non-popular vectors, using EMB-SSD's coarse-grained vector classification may store much non-popular data in DRAM, resulting in low cache efficiency.

To illustrate the limitations of existing schemes, we evaluate their cache performance using the Criteo [30] dataset. Figure 2(d) shows HugeCTR achieves the lowest hit ratio because it admits vectors without a global view. EMB-SSD performs better than HugeCTR, but it still has 12.4% lower cache hit ratio than Fleche and OpenEmbedding. This is because EMB-SSD caches vectors at the granularity of the embedding table, which may cause less-frequently accessed vectors loaded in the fast device. Although Fleche and OpenEmbedding achieve the highest hit ratio, it incurs 14.3% embedding movement overhead between NVM and DRAM due to cache thrashing, resulting in 37% training performance slowdown, as shown in Figure 2(e).

In contrast, recommendation model training offers a unique advantage: the entire dataset is known before training begins [37]. This allows us to collect access traces in advance and construct a placement strategy from a global view, which achieves the optimal allocation of embeddings between DRAM and NVM. Moreover, since embedding accesses in recommendation systems naturally occur at the vector level, this observation motivates a placement strategy at the same fine granularity. By leveraging these properties, these observations motivate our fine-grained placement strategy, which avoids the thrashing overhead of dynamic caching while achieving a high hit ratio without online migration.

How to adapt to changes of embedding access patterns? The access patterns of recommendation systems change with the arrival of daily new requests, as depicted in Figure 2(b). It is necessary to retrain the models with new datasets collected daily or weekly to ensure accurate recommendations [45, 65]. Dynamic caching is capable of incremental training with new datasets [2, 57, 66]. However, the excessive data movements between DRAM and NVM during retraining negatively impact its performance. In contrast, static profiling techniques can eliminate data movement overhead. But they are inherently unsupportive of incremental training because embedding vectors are stored at fixed memory locations [33]. Alterations in data access patterns can undermine the efficiency of dynamic caching and static profiling-based approaches. Hence, our design should dynamically adapt to changes in data access patterns while preserving high cache hit ratios and access correctness.

How to provide crash-consistency on NVMs for DLRM training? Crash consistency is the recoverability of persistent data from NVM in a consistent state after system failures. We can store checkpoints of DLRM states in NVM to achieve high performance while still enforcing crash consistency. The first approach is using NVMs as block devices. Specifically, we can store checkpoints on NVM-based file systems. However, it incurs training performance degradation because the system still needs to write the checkpointing data to NVMs. The second approach is using NVMs as memory devices. Embeddings and MLP weights can be directly persisted on NVMs upon updates, eliminating the need for additional checkpoint writes. However, as shown in Figure 2(f), it does not support crash consistency for two reasons. (1) *Partial updates of embedding tables.* If a failure occurs before completing all updates to the billions of embedding vectors in the tables, the embedding tables will be partially updated after the failure, leading to data inconsistency in NVM. (2) *Inconsistent updates of correlated data.* In each DLRM training iteration, both MLP weights and embedding tables need updating (refer to Figure 1). If a failure

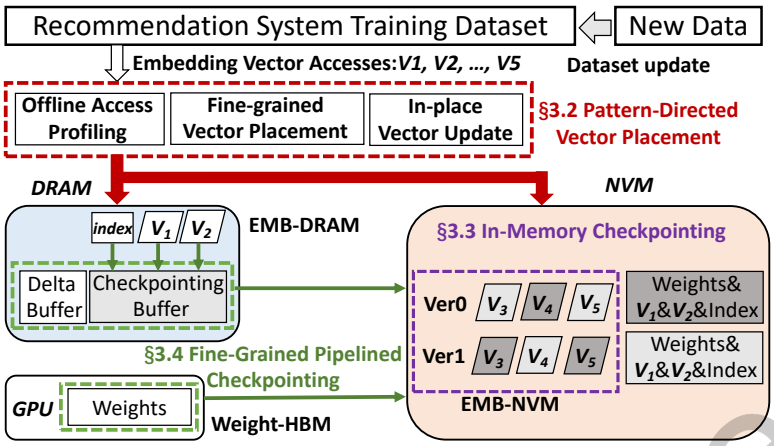


Fig. 3. The system overview of NVrec.

happens after writing MLP weights but before finishing writing embedding tables, the recommendation system may observe stale embedding vectors, causing data inconsistency between MLP weights and embedding vectors. Inconsistent data prevents recommendation systems from directly recovering memory states, even if they are stored in NVM.

3 Design of NVrec

The design of NVrec is guided by three goals including maximizing cache hit ratios for data in DRAM, implementing a dual-version mechanism to enable in-memory checkpointing and computation for data in NVM, and supporting pipelined checkpointing for data in DRAM and HBM.

3.1 System Overview

Figure 3 shows the overview of NVrec. It consists of three key modules. For a training job, *the pattern-directed placement module* analyzes the embedding access patterns offline, making the proper vector placement policy on DRAM and NVM (§3.2). During subsequent job runs, the system utilizes hybrid data placement to enable quick runtime vector accesses. Additionally, it leverages *the in-memory checkpointing mechanism* that uses a dual-version scheme for fast checkpointing (§3.3). To further reduce the checkpointing time, *the pipelined checkpointing* hides the checkpointing time behind the job training time (§3.4). Our contribution lies in co-designing these ideas into an effective hybrid-memory framework by exploiting key characteristics of recommendation training, rather than in proposing the underlying ideas as individually new. We describe the proposed three modules in detail in the following sections.

3.2 Pattern-Directed Vector Placement

NVrec partitions and stores embedding vectors in both DRAM and NVM. We utilize embedding locality to determine storage locations, leveraging the skewed distribution of the embedding vectors. Consequently, the vectors representing popular items are placed in DRAM to mitigate the long latency of NVM and leverage DRAM's high bandwidth, while the rest data are stored in NVM. The mechanisms described here capture the core design. Other practical adaptations to specific deployment settings (e.g., distributed training) are discussed in §5.9.

Pattern-directed vector placement. Previous techniques, such as dynamic caching and static profiling, often suffer from high data movement overhead or poor DRAM utilization (discussed in §2.4). Although frequency-based

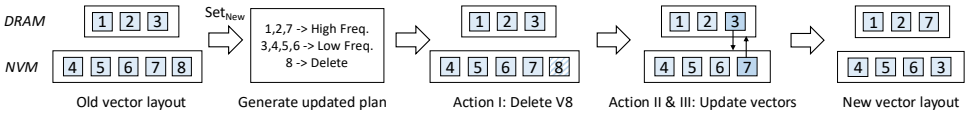


Fig. 4. The illustration of in-place vector updates. The numbers correspond to embedding IDs.

placement is an intuitive idea, NVrec makes it practical for recommendation workloads by applying it at the fine-grained vector level. This design is non-trivial, as it requires handling billions of vectors with highly skewed popularity distributions while keeping the offline placement construction efficient. To address these issues, we design a fine-grained vector-oriented static profiling and placement scheme to store only popular vectors in DRAM. For a recommendation system training job, we need to collect its vector access trace at the first time when the job is submitted. Each line in the trace records vector IDs and table IDs accessed by each user's request.

The vector-oriented profiling and placement scheme has six steps. (1) We analyze the access pattern of embedding vectors using the recorded traces offline without affecting ongoing training cycles. (2) We assign a universal vector number to each embedding vector for the convenience of indexing. (3) We scan the traces to collect the access frequency of each vector. (4) Given the DRAM size, NVrec places the popular vectors in DRAM until it is full and places the rest of them in NVM. (5) We assign each vector a new ID based on its access frequency. And (6) based on the vector ID remapping, we create a new dataset trace for the subsequent training. For example, vector 5 can be renamed as vector 1 if the vector has the highest access frequency. The largest ID of vectors in DRAM is named the threshold ID. In this way, it is more efficient for NVrec to determine the locations of embedding vectors at runtime by comparing the vector ID to the threshold ID.

Complexity and cost analysis. We summarize the algorithmic complexity and scalability as follows. Assume that A denotes the number of access records in the trace, V denotes the number of unique embedding vectors (after assigning universal IDs), and K denotes the number of vectors that fit in DRAM. In Steps (2)–(3), we build the ID index and count per-vector frequencies via a linear scan of the trace, taking $O(A)$ time and $O(V)$ space for per-vector counters. Selecting the Top-K popular vectors in Step (4) takes $O(V \log V)$ time with full sorting. In Steps (5)–(6), we construct the old-to-new ID map in $O(V)$ time/space and rewrite the trace in $O(A)$ time. Overall, the offline time complexity is $O(A + V \log V)$ and the space complexity is $O(V)$. As V grows, the cost is dominated by a one-time $O(V \log V)$ ranking. The remaining work consists of two linear trace scans ($O(A)$). The $O(V)$ space is used for counters and the ID map. The measured overhead is reported in §5.8. At the same time, due to the complexity of real-world systems, our analysis does not capture the full operational cost in large-scale production environments, such as trace replication and concurrent access from multiple jobs. The proposed vector placement policy provides a reasonably efficient preprocessing scheme than a naive placement approach.

However, in production-scale hyperscaler deployments, the same dataset may be concurrently accessed by multiple users or applications and replicated across nodes or even datacenters. In such cases, vector renaming may require non-trivial consistency, versioning, and coordination mechanisms, making it too costly or complex. Therefore, NVrec focuses on scenarios where vector renaming can be performed with manageable overhead.

In-place vector updates for model retraining. Given the evolving nature of user preference, it is imperative to periodically retrain the recommendation model [45]. NVrec takes advantage of this inherent property of recommendation systems, where periodic retraining naturally provides updated access traces that can guide vector relocation. In our target deployment scenarios, vector relocation is performed as an offline preprocessing step before launching a retraining job. Each retraining job uses its own remapped trace and embedding layout throughout training. NVrec does not target scenarios where multiple independent training jobs concurrently share and modify the same embedding layout. However, adapting to changes by rebuilding embeddings is costly, since a straightforward approach often allocates a large buffer for the new layout and migrates all data into it, leading to substantial movement and memory allocation overhead between DRAM and NVM. To solve this

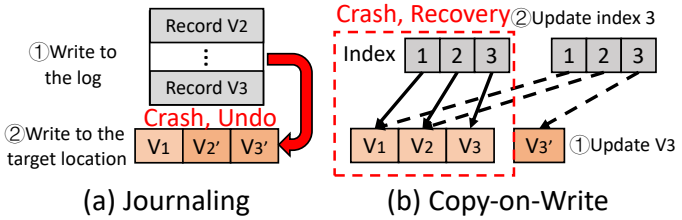


Fig. 5. Crash consistency: journaling and copy-on-write. The embedding vectors are all stored on NVM in the figure.

challenge, we use in-place vector updates that only move the changed states in the original array to avoid the costly allocation and deallocation of vectors in the memory. Upon the updated training data, the in-place vector update requires the following steps, as shown in Figure 4. For clarity, Set_{old} and Set_{new} serve as the datasets' old and new versions, respectively. (1) Vector placement analysis: An offline vector-oriented profiling of embedding vectors in Set_{new} is conducted before retraining. (2) Creating an updated plan: We execute one of the three procedures (Action I, II, or III) for each vector in $Set_{old} \bigcup Set_{new}$. Action I entails deleting vectors that are no longer in use from memory, Action II involves moving low-frequency vectors from DRAM to NVM, and Action III entails moving high-frequency vectors from NVM to DRAM. We maintain three queues, i.e., Queue I, II, and III, to record the vectors for each action. (3) Plan execution: We execute the plan in the order of Queue I, Queue II, and Queue III. Our experimental results indicate that in-place updating incurs only 0.3% time overhead (as in §5.8), while the training throughput is increased by 5%.

3.3 In-Memory Checkpointing for Data in NVM

During training periods, NVRec needs to periodically save *a portion of embedding tables in DRAM* (EMB-DRAM), *the other portion of embedding tables in NVM* (EMB-NVM), *MLP weights in GPU HBM* (Weight-HBM), and *other metadata in DRAM* to persistent devices for failure recovery or retraining, as shown in Figure 3. We design two checkpointing mechanisms to persist the training state for failure recovery and retraining: in-memory checkpointing for EMB-NVM (described in this section) and pipelined checkpointing for the remaining data (§3.4). We apply two different approaches for checkpointing EMB-NVM and the rest of training states. This is because EMB-NVM already resides in persistent memory. For EMB-NVM, our optimization focuses on efficiently capturing a recoverable snapshot. For the rest of the training states, checkpointing operations must copy them from DRAM/HBM to persistent devices. Therefore, we use pipelined data movement for optimization.

For saving EMB-NVM, previous approaches adopt journaling [10] or copy-on-write (CoW) [12] to ensure crash consistency. The journaling approach maintains alternative copies of original embeddings in a log, which stores new data updates (redo logging) or old data values (undo logging). Upon failures, the system replays the logs to restore. However, journaling requires writing data twice, incurring a significant loss of performance. In addition, log replay increases the recovery time, reducing the fast recovery benefit of NVM [52].

CoW creates a new copy of the vector upon a write request rather than updating the data in-place. For data consistency, it needs to update and persist the corresponding index periodically. For failure recovery, the system loads the index and restores old vectors accordingly. However, because recommendation systems use a large number of vectors, the time and space overhead is high for managing the indexes. For example, an embedding dataset of 320 GB with vector dimensions of 32 needs 10 GB indexes.

To tackle these problems, inspired by previous work [11, 51], NVRec designs a recommendation-oriented dual-version checkpointing mechanism for in-memory checkpointing. It maintains an untainted persistent vector for recovery and a runtime vector that accepts direct updates on NVM. Once the runtime version is updated, it can naturally serve as the next checkpoint, thereby eliminating additional checkpoint writes to persistent devices.

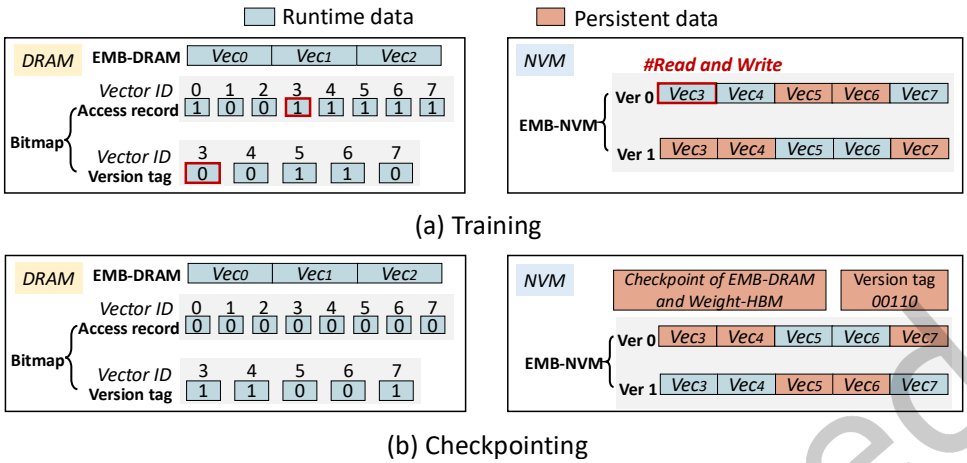


Fig. 6. (a) Memory states after training with NVrec; (b) Memory states after checkpointing with NVrec. In this example, we assume that vectors 3-7 are updated after training.

Furthermore, all vectors are stored in pre-assigned locations in NVM, which avoids the significant index overhead caused by dynamic allocation. Specifically, as illustrated in Figure 6, we store two kinds of EMB-NVM in NVM: *persistent data* and *runtime data*. The persistent data remains read-only between checkpointing operations and guarantees failure recovery, while the runtime data is continuously updated during training. NVrec maintains two versions of EMB-NVM (version 0 and version 1), with each version consisting of both persistent data and runtime data, and lightweight bitmaps are used to track which version is currently valid.

While dual-version designs have been explored in prior systems [11, 51], recommendation training poses different challenges. First, the embedding access pattern in recommendation training makes dual-version indexing more complex. Specifically, within a checkpoint interval, the same embedding may be accessed and updated multiple times. After the first update, the "read" copy is no longer fresh, so subsequent reads and writes must be routed to access the correct version (persistent vs. runtime). The training may read stale or inconsistent values if the index is not correctly updated. In contrast, prior works typically read and write each item once per interval, so a single version switch or index update is sufficient. NVrecSecond, recommendation training involves orders of magnitude more state data than the application states targeted by prior designs [11]. This scale makes per-item tracking unavoidable and demands extremely low space and time overhead.

To address these challenges, NVrec proposes an access-aware lightweight dual-version design. It uses two DRAM-resident lightweight bitmaps to track where each embedding's persistent and runtime copies reside across version 0 and version 1, enabling correct routing of accesses while minimizing metadata overhead.

The first one is a bitmap of the *version tag*. Its bit indicates the location of the runtime data in NVM. For example, in Figure 6(a), we assume the embedding vectors 0-2 are EMB-DRAM and vectors 3-7 are EMB-NVM. We also assume that embedding vectors 3-7 need to be updated in training. The version tag is [0, 0, 1, 1, 0], indicating that the runtime data of vectors 3, 4, 5, 6, and 7 are stored in versions 0, 0, 1, 1, and 0, respectively, in NVM. The second one is a bitmap of the *access record*. Its bit indicates whether an embedding vector has been updated after the previous checkpoint. For example, in the same figure, the *access record* is [1, 0, 0, 1, 1, 1, 1, 1], indicating that the vectors 0, 3, 4, 5, 6, and 7 have been updated. Here, we design the bitmap for tracking vectors to save space. Specifically, two bitmaps in our design only consume negligible 0.01% memory footprint for the 512-dimension vector (§5.8). Moreover, lightweight indexes are accessed rapidly due to their small size, making the access time negligible compared to vector accesses.

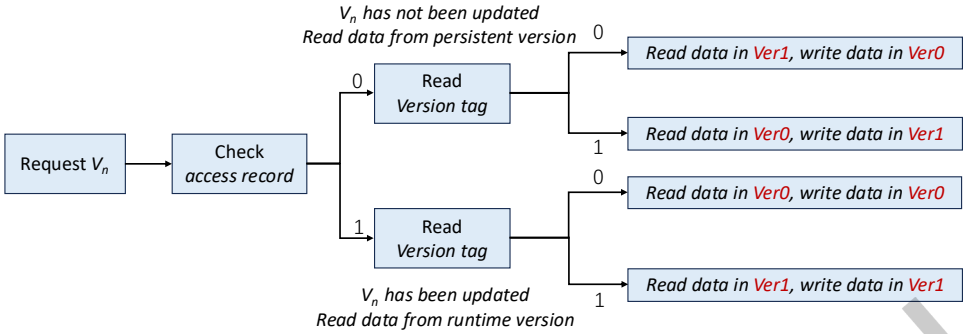


Fig. 7. Details of data access in NVrec training process.

Training. In training, requests may access either EMB-DRAM or EMB-NVM. For the requests accessing EMB-DRAM, we read and update embeddings directly. We incrementally save checkpoints of EMB-DRAM on NVMs. Its optimization is described in §3.4. For requests reading EMB-NVM, the system must determine where the most up-to-date value resides. If the embedding vector has been updated since the last checkpoint, as indicated by the *bitmap of the access record*, its runtime data is used to serve the request; otherwise, the persistent version already holds the latest value and is accessed directly. For writes, we always use the runtime data to serve the requests. The detailed workflow is shown in Figure 7.

Training example. We continue using the example in Figure 6(a) for illustration. When vector 3 is accessed, NVrec finds its corresponding access record marked as "1", indicating that it has already been updated since the last checkpoint and thus its latest value resides in the runtime data. Furthermore, because the version tag of vector 3 is equal to 0, meaning that its runtime location is in version 0, the system reads its value from version 0 and writes the updated data back to the same place.

Checkpointing and recovery of EMB-NVM. In the checkpointing process, NVrec needs to save training superparameters, including the random seed for the dataset, training epochs, the number of training iterations, and other parameters. For EMB-NVM, the system simply saves the current version tag to NVM and, for embeddings marked as updated in the access record, reverses their corresponding version tag in DRAM. In the end, it will reset the access record to "0" for all the vectors. Figure 6(b) shows the memory states in NVM after checkpointing in the above example.

For recovering EMB-NVM upon failures, NVrec needs to map version 0 and version 1 to process address space. Then, it reads training superparameters. After that, it reads persisted version tag from NVM to DRAM and initializes the access record with "0". Finally, NVrec can continue training where the system is left off.

Checkpointing and recovery of EMB-DRAM. To save memory bandwidth, we use the incremental checkpointing technique for saving EMB-DRAM. Specifically, NVrec will save the whole EMB-DRAM in the first checkpoint as the full baseline. In the subsequent checkpointing, it scans the access record and only the embedding vectors marked as "1" in the bitmap are asynchronously saved to NVM, as in §3.4. To reduce space consumption and recovery overhead, the baseline and incremental vectors are periodically merged into a new baseline. For failure recovery, we can rebuild EMB-DRAM using the recent baseline checkpoint and delta values saved in the following checkpoints.

Checkpointing and recovery of Weight-HBM in GPU. To save Weight-HBM, we create a buffer in DRAM. The Weight-HBM is copied to the buffer and saved to NVM, as in §3.4. For failure recovery, we read the saved Weight-HBM from NVM to GPU to rebuild the networks.

3.4 Fine-Grained Pipelined Checkpointing for Other Data

The dual-version checkpointing can achieve in-NVM checkpointing with low overhead for EMB-NVM. However, NVrec still needs to save Weight-HBM, EMB-DRAM, and other metadata structures (e.g., version tag, random seed, and training iteration) to NVM. A direct method is to perform non-pipelined (synchronous) checkpointing, where the checkpointing operations and the subsequent model training are executed serially, as shown in Figure 8(a). However, this approach can be expensive due to the longer write latency of NVM as compared to DRAM. Since these training states reside in different memory domains (GPU HBM, DRAM, and NVM), efficiently persisting them requires a recommendation-specific adaptation of pipelined checkpointing rather than a generic asynchronous scheme. In this section, we describe how NVrec adapts pipelined checkpointing to hide the persistence cost of such data behind training time.

Naive pipeline mechanism. As shown in Figure 8(b), we first design a naive pipeline mechanism to transform the non-pipelined checkpointing to a pipelined (asynchronous) process by introducing an in-DRAM buffer. When pipelined checkpointing starts, NVrec copies the updated Weight-HBM, EMB-DRAM, and metadata to the checkpointing buffer in DRAM. Then, it asynchronously moves these data from the checkpointing buffer to NVM. Meanwhile, NVrec can resume model training in parallel with the asynchronous data transfer, as [17, 44] do. However, the naive pipeline needs to first transfer the Weight-HBM from GPU to DRAM and then to NVM, leading to long data movement latency.

Fine-grained pipeline mechanism. To seek opportunities for further parallelism, we persist the Weight-HBM to NVM bypassing the DRAM buffer. As shown in Figure 8(c), we overlap Weight-HBM writes with the copying of EMB-DRAM and metadata to reduce the checkpointing overhead. To achieve a high GPU-NVM access performance, we coalesce the direct access (DAX) feature of NVM and the unified virtual addressing (UVA) technique to map NVM into the GPU virtual address space. We disable data direct IO (DDIO) to control the destination of GPU writes via PCIe and use GPU fence instruction (i.e., threadfence) to guarantee data persistence [48]. Our experimental results show that the fine-grained pipelined checkpointing can further reduce 40% and 20% checkpointing time compared to the synchronous and naive pipeline solutions (§5.4).

Challenges. We aim to overlap model training with asynchronous data transfer. However, this introduces a challenge. During checkpointing, requests that access EMB-DRAM can be served by in-place updates, since their checkpoints have already been copied to the checkpoint buffer. In contrast, requests accessing EMB-NVM cannot directly update vectors on NVM, because there is no additional indexed location to accommodate the new data, as shown in Figure 10(a). For example, assume that a new write request arrives, we cannot overwrite it until version T1's checkpoint buffer is fully written to NVM. Otherwise, since T1 is not yet recoverable, we cannot write to its location; however, overwriting vectors of T0 version would remove the only valid checkpoint and risk data loss.

To address this issue, NVrec serves write requests to EMB-NVM using a *delta buffer* in DRAM, as shown in Figure 10(b). We write the new data to the delta buffer instead of the NVM. To ensure that subsequent accesses observe the most up-to-date state before these buffered updates are persisted, NVrec serves read requests from the delta buffer first. If it misses in the buffer, the requested data will be read from NVM. Our experimental results show that 99% of vectors can be found directly from the delta buffer. Moreover, since the delta buffer is small, the overhead of searching the buffer is negligible (§5.8).

Fine-grained synchronization mechanism. When checkpointing data transfer is completed, NVrec transfers the updates from the delta buffer to EMB-NVM in the background. At this point, write conflicts may occur between the model training process and the delta-buffer transfer thread when they access the same embedding vector, potentially pausing model training until the entire delta buffer has been written back, as illustrated in Figure 9(a). Specifically, suppose the delta buffer contains three vectors Vec_0, Vec_1, and Vec_2, and the training thread issues a new write to Vec_1 while the buffer is being transferred to NVM. Under buffer-level synchronization,

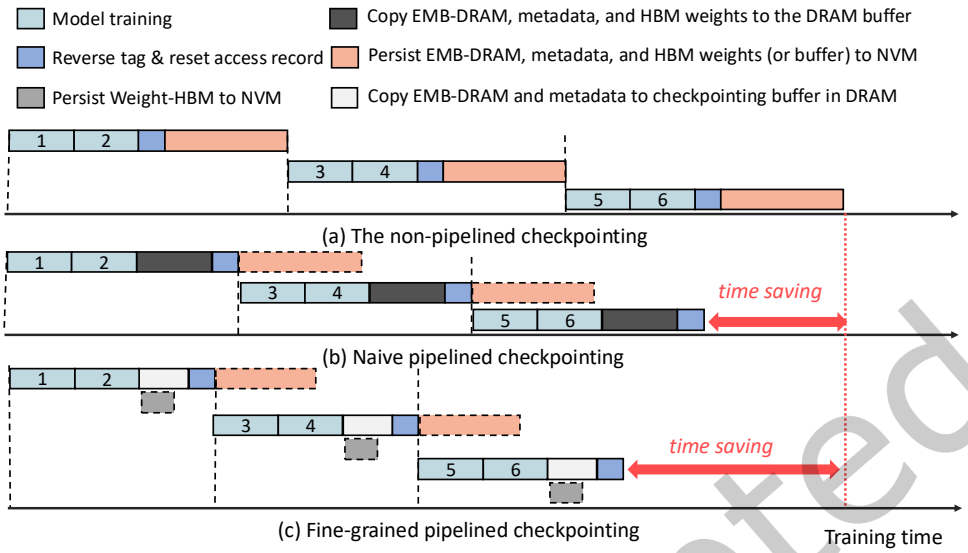


Fig. 8. The non-pipelined, naive, and fine-grained pipelined checkpointing approaches. The dashed rectangles represent parallel processes.

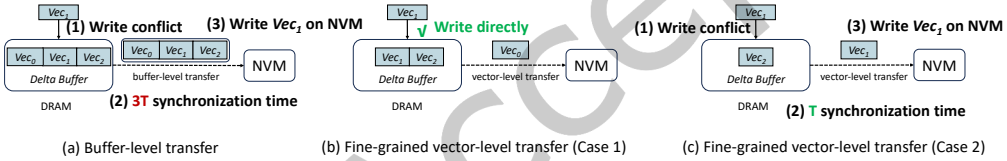


Fig. 9. Comparison between buffer-level and fine-grained vector-level synchronization.

the training thread must wait for the entire buffer transfer to complete before serving the write to Vec_1, leading to a synchronization delay of $3T$, where T is the transfer time of one vector.

To reduce this synchronization overhead, we design a fine-grained vector-level synchronization mechanism that provides two benefits: (1) it ensures correctness while avoiding unnecessary waiting by synchronizing only on the conflicting vector. As shown in Figure 9(b), if the training thread writes a vector that is not currently being transferred (e.g., Vec_1 while Vec_0 is being flushed), the new update can be served directly in the delta buffer without waiting. (2) If the write conflicts with the vector currently under transfer, as shown in Figure 9(c), the training thread only waits until that specific vector finishes transferring to NVM, and then serves the new write. In this case, the synchronization delay is reduced from $3T$ to T. In our evaluated setting on Criteo, fine-grained vector-level synchronization reduces the latency of a checkpoint operation by about 5% compared with coarse-grained buffer-level synchronization, with the experimental setup described in §5.1.

Crash analysis during delta-buffer asynchronous transfer. If the system crashes while the delta buffer in volatile DRAM is being written to NVM, crash recovery is still guaranteed. Specifically, recovery rolls the system back to the most recent committed checkpoint. For example, in Figure 10(b), once the T_2 delta buffer (the blue DRAM entry) starts being flushed from DRAM to NVM, it implies that all checkpoint data of version T_1 have already been durably persisted in NVM. Therefore, T_1 is the latest committed checkpoint, and the old T_0 location can be reused for future writes. If a crash occurs while the T_2 delta buffer is still being flushed, recovery rolls back to checkpoint version T_1. In this case, the lost training progress is limited to the updates generated after T_1,

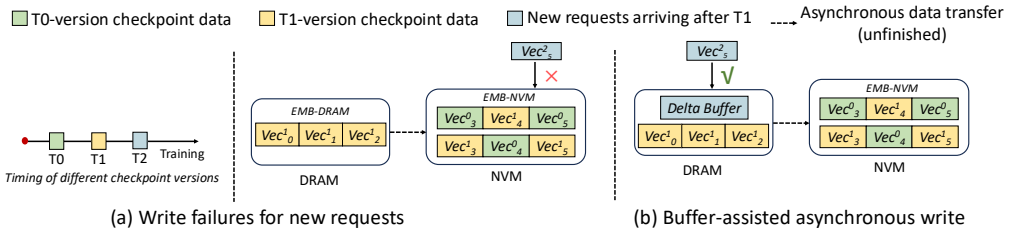


Fig. 10. The challenge and solution of fine-grained pipelined checkpointing. Vec_b^a denotes the vector with ID b in checkpoint version T_a.

including the uncommitted T_2 delta-buffered updates that have not yet become durable. After recovery, training resumes from the recovered T_1 state. We further validate this recovery semantics in §5.5 through crash-injection experiments at different stages of asynchronous transfer.

4 Implementation

We implement NVrec in C++. Overall, our implementation consists of three components: (i) a DLRM training stack on GPU, (ii) embedding storage and access on a DAX-enabled NVM space, and (iii) checkpointing and persistence mechanisms for different platforms (ADR/eADR).

(i) DLRM training stack. Similar to PyTorch [50], we use cuDNN [47] and cuBLAS [13] to build DLRM. Specifically, we define the DLRM network architecture and its dependencies across layers, and instantiate the associated training states, including model parameters and optimizer states.

(ii) Embeddings on NVM. We deploy NVM in App Direct mode with a DAX-aware file system (e.g., ext4). We create an embedding file, allocate space via `posix_fallocate()`, and use STL vectors to support dynamic array management.

(iii) Data movement and GPU-to-NVM support. For general training, we transfer data between NVM and DRAM using `memcpy()`, and move data between host memory and GPU HBM using `cudaMemcpy()`, consistent with common implementations in PyTorch/TensorFlow. For pipelined checkpointing, our design can additionally leverage a direct GPU-to-NVM path by combining DAX with GPU unified virtual addressing (UVA) to map NVM into the GPU virtual address space. We disable DDIO to steer PCIe writes to NVM and use GPU fences (e.g., threadfence) to enforce persistence ordering [48]. This GPU-to-NVM path assumes a UVA-capable NVIDIA GPU and an Intel server platform where DDIO can be configured. In CUDA, UVA is available in 64-bit processes on NVIDIA GPUs with compute capability ≥ 2.0 (i.e., Fermi and newer GPU architectures [46]). DDIO is supported on Intel Xeon server platforms (e.g., Xeon E5/E7 v2 [23] and modern Xeon Scalable CPUs such as Xeon Gold 5218R [25]), typically configurable via BIOS [22].

(iv) Persistence on ADR/eADR platforms. On ADR platforms, we use `_mm_clwb` to flush cache lines to NVM and `_mm_sfence` to enforce ordering at the end of checkpointing. On eADR platforms, persistence is guaranteed once data reaches the CPU LLC; thus we omit explicit flush/fence to simulate eADR execution [27, 39].

5 Evaluation

5.1 Experimental Setup

Experimental platforms. We run NVrec on a server with 2×Intel Xeon Gold 5218R processors, 64G RDIMM DDR4 DRAM, 4×128GB Intel Optane NVM, and a NVIDIA A100 GPU with CUDA 11.6.

Datasets and models. We use the real-world datasets and synthetic datasets to evaluate the performance of NVrec. The real-world datasets include the open-source Avazu [1] (10 days) and Criteo [30] (7 days). We set their embedding dimension as 512. Avazu has 22 tables and a total of 9 million embeddings with a memory footprint

of 18 GB. Criteo has 26 tables and a total of 33 million embeddings which has a memory footprint of 64 GB. For synthetic datasets, we generate embedding vectors subjecting to a power law distribution and set the α as -2 to cater to the characteristics of real-world datasets for 10 days [57]. Besides, its embedding dimension is 512 and its embedding table count is 26 with 79 million embeddings, which needs 150 GB memory. We randomly assign each embedding vector to an embedding table. Moreover, we evaluate NVRec on DLRM [16] and set the dimension of the left and right MLPs as 512-256-2 and 512-256-64-16, respectively.

Compared systems. We implement several recommendation systems for comparison. For embedding placement, *HugeCTR* maintains an LRU cache for each embedding table to dynamically cache its hot vectors [2]. *Fleche* and *OpenEmbedding* build one global LRU cache to capture the global hotspots of all tables [8, 57]. *EMB-SSD* (ES) takes a static approach to identify popular tables and places them in DRAM [33].

For checkpointing, *Check-N-Run* uses incremental checkpointing [17]. Specifically, the Check-N-Run follows a two-step process. Firstly, it saves all model states as a complete baseline checkpoint in NVM. Secondly, it copies any updated embedding vectors and weights to a delta buffer in DRAM and asynchronously transfers the buffer to NVM during checkpointing. When recovering the model, Check-N-Run reads the baseline and merges consecutive incremental checkpoints from NVM. If not specified, HugeCTR, Fleche, and EMB-SSD use this Check-N-Run mechanism as their checkpointing design.

NV-Checkpoint (NVCkpt) [12] utilizes Copy-on-Write and maintains two sets of indexes for EMB-NVM: one in DRAM for training and the other persisted in NVM for recovery. When writing data, it creates a new copy of the updated embedding in NVM and updates the DRAM index. During checkpointing, it persists the EMB-NVM index and asynchronously persists the incremental EMB-DRAM from DRAM to NVM.

OpenEmbedding (OpenEMB) uses DRAM as a cache and NVM as storage, employing a co-design approach for checkpointing and cache replacement. Volatile vectors in DRAM are actively flushed to NVM when being accessed, while the remaining vectors are passively flushed periodically [8]. Checkpointing is considered complete only after all DRAM checkpoint vectors are written to NVM.

For model retraining, we implement *NVRec-Online*. It incrementally trains the model using the in-place vector update (§3.2) with the trace collected from each day. In contrast, for NVRec and other systems, we merge all the traces from each day and train the model using the merged trace. Thus, we do not need to update embedding vectors for NVRec.

Other configurations. In recommendation systems, checkpointing frequency is often determined by a fixed amount of training progress, such as a given number of data samples [16], or derived using failure-aware models such as the Young–Daly formula based on the system failure rate [53]. Accordingly, our evaluation covers both settings. By default, all systems checkpoint after making the same amount of training progress (i.e., a fixed number of iterations as in [44], implemented as every 5% of data samples) to ensure an identical number of checkpoint operations across methods for fair comparison. We also evaluate multiple checkpoint frequencies in §5.7 to show how checkpoint interval affects system throughput. As a complementary analysis, we further use the Young–Daly model to derive near-optimal checkpoint intervals under different failure assumptions, and present the details in §5.4. If not specified, the cache size for counterparts or the size of DRAM space for NVRec is set to 1% of the size of embedding vectors, and the batch size is 2048. Besides, we also evaluate the performance of NVRec under different cache sizes in §5.7.

5.2 Overall Performance

We compare the whole training throughput of NVRec against the five counterparts across three datasets. We omit the comparisons of HugeCRT+NVCkpt and Fleche+NVCkpt because NVCkpt cannot support dynamic caching.

As shown in Figure 11(a)-(c), NVRec achieves 1.05-1.47×, 1.12-2.13×, and 2.21-6.30× speedup over counterparts on the Avazu, Criteo, and Synthetic datasets, respectively. There are three reasons for NVRec's performance

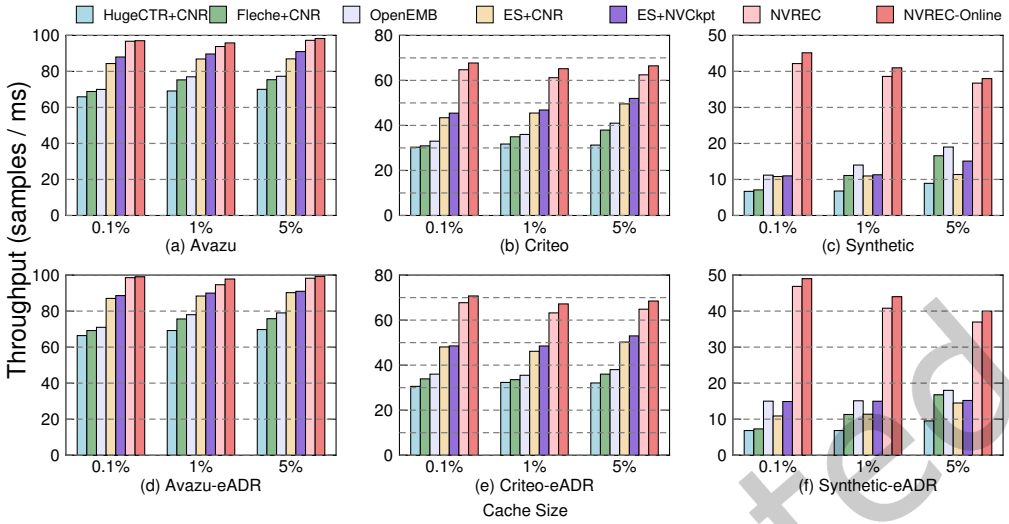


Fig. 11. Overall training throughput with the cache size (size of embeddings in DRAM) increased from 0.1% to 5% (based on the DRAM-to-NVM ratio of modern servers [58]) across the three datasets. Figures (a)-(c) and Figures (d)-(f) are conducted with NVM in the ADR and eADR modes. The throughput improvement of NVrec and NVrec-online comes from the combined effect of embedding placement and checkpointing optimizations.

improvements. First, HugeCTR+CNR, Fleche+CNR, and OpenEmbedding utilize cache to store the popular embeddings in DRAMs. If a cache miss happens, they need to read embeddings from NVM to DRAM and then to the CPU cache, thus increasing the end-to-end latency significantly. Second, they achieve 24.6%, 16.4%, and 16.4% lower cache hit ratio than NVrec, respectively, as shown in Figure 12(b). Third, they have a longer checkpointing time than NVrec. The first two factors mainly reflect the benefit of NVrec's placement in reducing embedding-access cost, while the third factor captures the additional gain from faster checkpointing, as shown in Figure 12(a). Detailed analyses are provided in §5.3 and §5.4. checkcheck

Besides, NVrec observes 1.08-3.89× and 1.05-3.72× throughput improvement across the three datasets compared to ES+CNR and ES+NVCkpt, respectively. The main reasons are higher cache hit ratios as shown in Figure 12 and better checkpointing performance.

Another observation is NVrec achieves better performance on larger datasets because the recommendation systems spend more time on checkpointing. Consequently, in-memory checkpointing in NVrec has more potential to reduce the checkpointing time. The other observation is that the performance of NVrec does not always increase as more DRAM space is allocated. For example, on Criteo, the throughput of NVrec decreases from 64.7 to 61.2 samples/ms when the cache size is increased from 0.1% to 1%. Then, it is increased to 62.4 samples/ms at 5% cache size. The reason is that the throughput is affected by two factors, including embedding access time and checkpointing time. Allocating more DRAM space increases the cache hit ratio thus reduces the embedding access time. At the same time, it increases the checkpointing time because NVrec needs to transfer more embeddings from DRAM to NVM.

We also conduct experiments to verify the effectiveness of the model retraining when vector access patterns change. The experimental results show that NVrec-Online can achieve averaged 1.04× performance improvement compared to NVrec. It is because NVrec-Online adapts to new data access patterns through vector rebuilding, which increases the cache hit ratio, as shown in Figure 12(b).

Analyzing NVrec on eADR. Intel announces enhanced ADR (eADR) for Optane DC Persistent Memory. eADR feature (extended ADR) will drain the entire CPU cache contents to NVM after power failures which

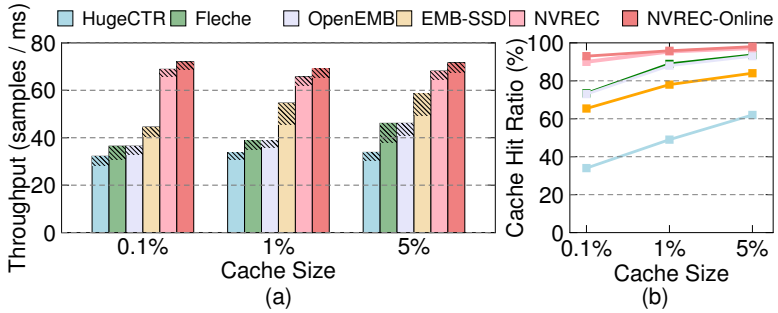


Fig. 12. (a) Training throughput with checkpointing enabled vs. disabled and (b) hit ratio under different cache sizes. The shaded region indicates the throughput degradation introduced by enabling checkpointing (i.e., the checkpoint overhead).

will obviate the need to flush caches from the CPU [27]. We also investigate how eADR will impact NVRec. Specifically, NVRec outperforms HugeCTR+CNR, Fleche+CNR, OpenEmbedding, ES+CNR, and ES+NVCkpt by 1.37-6.85×, 1.25-6.41×, 1.21-3.12×, 1.07-4.30×, and 1.05-3.14×, respectively. NVRec on eADR achieves 1.05× higher training throughput than that on ADR because NVRec on eADR does not need to use *clwb* and *fence*.

5.3 Impact of Vector-Oriented Placement

We analyze the benefit of vector-oriented data placement policy here. We train models on Criteo disabling the checkpointing operations. Figure 12(a) shows the training throughput across 0.1%, 1%, and 5% DRAM cache sizes, respectively. NVRec outperforms HugeCTR, Fleche, OpenEmbedding (OpenEMB), and EMB-SSD by 2.02×, 1.75×, 1.75×, and 1.28× respectively. NVRec achieves the best throughput because it has the highest hit ratio among all cache sizes as shown in Figure 12(b). The hit ratios with the vector-oriented data placement are up to 56%, 16%, 16%, and 24.6% higher than those of HugeCTR, Fleche, OpenEmbedding, and EMB-SSD, respectively. This result indicates the efficiency of the vector-oriented placement scheme of NVRec. Moreover, NVRec-Online improves 1.05× performance compared to NVRec.

5.4 Study of Checkpointing Performance

First, we compare the overall checkpointing performance of NVRec to the state-of-the-art checkpointing techniques including *Full-CKPT-SSD* in which all embeddings are stored in SSDs for checkpointing, *Full-CKPT-NVM* in which all embeddings are stored in NVM for checkpointing, Check-N-Run [17], NV-Checkpoint [12], and OpenEmbedding [8]. Full-CKPT-SSD is the default checkpointing approach used in PyTorch [49] and TensorFlow [54]. For a fair comparison, we use the vector-oriented data placement policy together with all the checkpointing techniques.

From Figure 13, we have six observations. (1) NVRec improves the checkpointing performance by 449-3332×, 250-1830×, 6.3-58×, 5.6-21×, and 5-12.7× compared to Full-CKPT-SSD, Full-CKPT-NVM, Check-N-Run, OpenEmbedding, and NV-Checkpoint, respectively. (2) Full-CKPT-SSD and Full-CKPT-NVM have long checkpointing latency because they need to write the whole embeddings and MLP weights to checkpoints. (3) Compared to Check-N-Run that only persists the updated embeddings, NVRec improves the checkpointing performance by 25.5× on average. The reason for NVRec's advantage is that it eliminates the persistence overhead for embeddings in NVM. (4) NVRec outperforms OpenEmbedding in checkpointing for two reasons. First, OpenEmbedding incurs performance degradation by forcibly writing EMB-DRAM back to NVM on the critical path during checkpointing. Second, OpenEmbedding requires a two-step process: first writing Weight-HBM in DRAM, then persisting it in NVM. In contrast, NVRec mitigates the persistence cost of EMB-DRAM and Weight-HBM by introducing

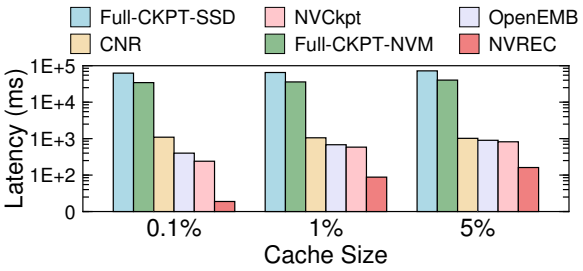


Fig. 13. The average latency per checkpointing. 1E+N denotes 10^N.

a fine-grained pipeline mechanism, effectively hiding these costs. (5) NVrec achieves better checkpointing performance than NV-Checkpointing for two reasons. First, more index data in NV-Checkpointing need to be persisted during checkpointing. Specifically, it consumes 4 bytes per vector for indexing, while NVrec uses only 1 bit tag version. Second, NV-Checkpointing needs to release numerous old embeddings on the NVM after the new index is persisted, which incurs additional time overhead. (6) NVrec can further reduce checkpointing overhead when storing more embeddings in NVM. For example, the checkpointing time decreases from 161 ms to 19 ms when the ratio of embeddings in NVM changes from 95% to 99.9%.

Next, we evaluate the impact of individual checkpointing optimizations. We compare NVrec against four checkpointing approaches: *CNR* (Check-N-Run), *NV-CKPT* (NV-Checkpoint), *+Dual*, which uses dual-version checkpointing for EMB-NVM and non-pipelined checkpointing for other data, *+Pipe*, which first transfers Weight-HBM, EMB-DRAM, and metadata to a DRAM buffer and then asynchronously persists them to NVM, and *+All*, which combines dual-version checkpointing with the fine-grained pipelined checkpointing described in §3, where Weight-HBM is persisted directly to NVM by bypassing the DRAM buffer. We also include *Oracular*, which disables checkpointing to provide the best-case training performance. Figure 15(a) shows the impact of these techniques on training throughput on Criteo. We make three observations. (1) Compared to CNR and NV-CKPT, +All improves training throughput by 1.13× and 1.11×, respectively. (2) +All further improves training throughput by 5.6% and 2% compared to +Dual and +Pipe, respectively, because fine-grained pipelined checkpointing hides the checkpointing latency of embeddings in DRAM and MLP weights in HBM. (3) +All incurs only 5% performance loss compared to Oracular.

Checkpoint-interval sensitivity under the Young–Daly model. We further evaluate all systems using checkpoint intervals derived from the Young–Daly formula. For each system, we measure its checkpoint cost C and compute the near-optimal checkpointing interval as $T^* \approx \sqrt{2CM}$, where M is the Mean-Time-to-Failure (MTTF). To improve the generality of the evaluation, we consider four representative MTTF settings ranging from frequent to infrequent failures (5, 15, 30, and 60 minutes), and report the resulting end-to-end training throughput. Figure 14 shows the resulting training throughput. We make three observations. First, Full SSD (Full-CKPT-SSD) and Full NVM (Full-CKPT-NVM) achieve relatively low throughput because they checkpoint the entire training state, and the large volume of checkpointed data causes substantial training stalls. Second, NVrec consistently achieves the highest throughput across all MTTF settings, benefiting from both its efficient vector placement and its lightweight checkpoint design. Third, when fixing the same vector placement strategy, the gap among NVREC, NVrec (+NV-CKPT), and NVrec (+CNR) is larger under smaller MTTF. This is because checkpointing becomes more frequent in this regime, making the advantage of NVrec's checkpoint mechanism more pronounced. As MTTF increases, checkpointing becomes less frequent, and the gap correspondingly narrows.

5.5 Recovery Evaluation

Comparisons of recovery time. We next evaluate the recovery time of different checkpointing approaches on Criteo. We force training to stop halfway to simulate a system crash and immediately recover the model state. The

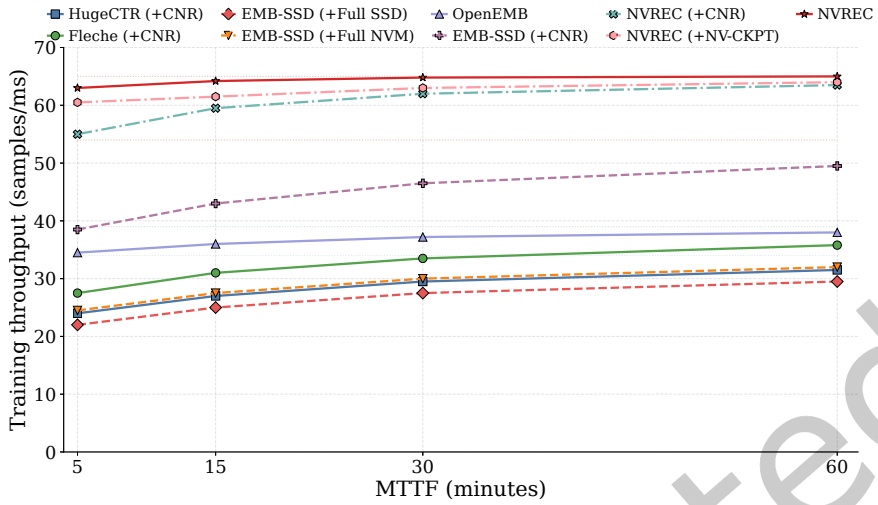


Fig. 14. Training throughput under near-optimal checkpoint intervals derived from the Young–Daly model, where $T^* \approx \sqrt{2CM}$, C is the average checkpoint cost, and M is the MTTF. Results are shown at 1% cache size on Criteo for MTTF values of 5, 15, 30, and 60 minutes. Methods are denoted as "Base (+Checkpoint)", where the prefix indicates the vector placement used in this work, and the term in parentheses specifies the checkpointing mechanism.

recovery process includes reading checkpoint data from persistent devices and restoring them to their runtime locations. As shown in Figure 15(b), NVrec reduces recovery time by 23×, 17×, 16×, 3×, and 1.2× compared with SSD-CKPT (Full-CKPT-SSD), NVM-CKPT (Full-CKPT-NVM), CNR (Check-N-Run), OpenEMB (OpenEmbedding), and NV-CKPT (NV-Checkpoint), respectively. NVrec achieves this reduction because CNR must rebuild memory states from incremental checkpoints, OpenEMB spends substantial time scanning all embedding entries in NVM and discarding non-checkpoint data, and NV-CKPT requires loading an additional index. In contrast, NVrec only needs to read the lightweight version tag in NVM to locate persistent embeddings for recovery, without triggering a large amount of I/O from storage devices.

Recovery analysis under asynchronous-transfer crashes. To further evaluate recovery behavior in the most vulnerable failure window, we inject crashes at different stages of asynchronous transfer, including before asynchronous transfer, at the start of asynchronous transfer, in the middle of asynchronous transfer, and after asynchronous transfer. Table 2 reports the rollback version, recovery time, lost training iterations, and whether the recovered state is valid. The results show that when a crash occurs before or during asynchronous transfer, the system rolls back to the latest committed checkpoint T_0, because checkpoint version T_1 has not yet become fully durable. In contrast, when a crash occurs after asynchronous transfer, recovery rolls back to checkpoint version T_1, since its checkpoint data have already been persisted to NVM. Across all crash points, the recovery time remains stable, ranging from 3.36 s to 3.44 s. The number of lost training iterations increases from 12 to 24 and 39 as the crash point moves later within the uncommitted transfer window, because more training progress has accumulated after checkpoint version T_0 while T_1 is still not recoverable. After asynchronous transfer, the lost iterations decrease to 5, since the system can already recover from checkpoint version T_1 and only a small amount of progress after T_1 is lost. In all tested cases, the recovered state is valid and training can resume normally.

5.6 Analysis of Checkpointing Frequency

Impact of checkpointing frequency. To study the impact, we reduce the checkpointing frequency from checkpointing after training every 5% of data samples to checkpointing after training every 20% of data samples.

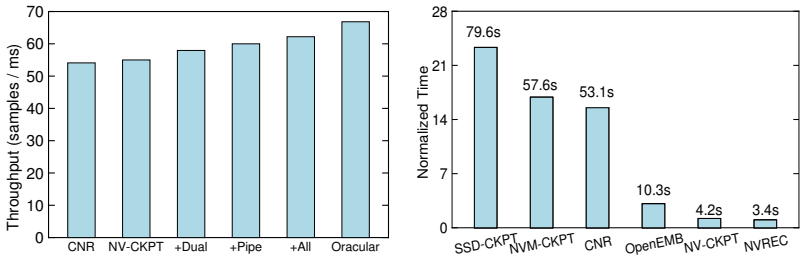


Fig. 15. (a) Impact of individual optimization. (b) Recovery time with different approaches. The recovery time (s) is normalized to that of NVrec. Wall-clock time is shown on the top of each bar.

Table 2. Recovery behavior under crashes at different stages of asynchronous transfer of the non-NVM portion of checkpoint version T_1 (e.g., data in DRAM and HBM) in Figure 10.

Crash point	Rollback version	Recovery time	Lost iterations	Recovered state valid
Before asynchronous transfer	T_0	3.38 s	12	Yes
At the start of asynchronous transfer	T_0	3.41 s	24	Yes
In the middle of asynchronous transfer	T_0	3.44 s	39	Yes
After asynchronous transfer	T_1	3.36 s	5	Yes

We use the vector-oriented data placement in compared systems for fair comparison expect for OpenEmbedding because it is designed for dynamic caching. Figure 17(a) shows that NVrec outperforms all counterparts. NVrec has increased benefit as the checkpointing time is increased.

5.7 Sensitivity Analysis

Impact of real-time drift in user–item popularity. We evaluate NVrec under real-time drift in user–item popularity to quantify the impact of not being able to perform timely profiling and embedding placement re-optimization. On the 7-day Criteo dataset, NVrec is profiled only on Day 1 dataset and keeps the Embedding vector placement fixed thereafter. Figure 16 shows that NVrec's throughput slightly decreases over days but remains the highest among all counterparts, while HugeCTR, Fleche, and OpenEMB achieve lower yet stable throughput, because they incur additional overhead from cache thrashing. NVrec-Online re-profiles on the current training data and serves as an oracle upper bound. NVrec matches NVrec-Online on Day 1 and gradually diverges as drift increases, but the gap remains within 9% overall.

In summary, NVrec is particularly well suited to periodic retraining workflows widely adopted in industry (e.g., Amazon [5], Alibaba [19, 64], and Tencent [70]), making NVrec (NVrec-Online) a practical choice for production deployment since profiling can be run before each retraining cycle. Moreover, when hot-data migration is relatively slow, NVrec does not require frequent profiling: moderate popularity drift has limited impact on training performance, and in practice the profiling interval can be tuned to match workload dynamics and system constraints.

Impact of embedding dimension and batch size . Figure 17(a) demonstrates that NVrec outperforms all counterparts across various embedding dimensions and batch sizes. Additionally, increasing embedding dimensions slows down training performance as larger vectors are accessed, bottlenecking memory bandwidth.

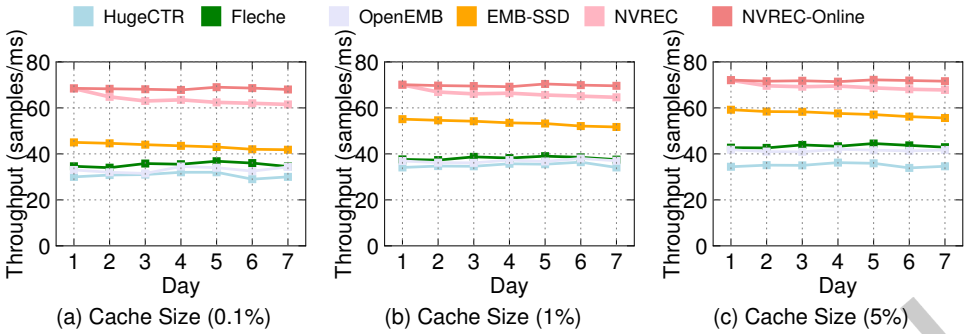


Fig. 16. Throughput of recommendation systems on the 7-day Criteo dataset under real-time drift in user–item popularity. The x-axis denotes the day index (models are trained on the union of dataset from the first N days). For NVrec and EMB-SSD, profiling is performed only on the Day 1 dataset and its overhead is included in the reported throughput; NVrec-Online is an oracle variant that profiles on the most recent training dataset. Other systems do not require profiling. Results are shown for cache sizes of 0.1%, 1%, and 5%.

Impact of large cache size. We conduct the evaluation with large cache sizes to verify NVrec's performance at various DRAM-to-NVM ratios, simulating different system configurations in real-world environments. We observe that NVrec still outperforms all the best counterpart by up to 1.6× when the cache size is increased from 20% to 80%.

5.8 Overhead Analysis

DRAM space cost. NVrec maintains two bitmaps: *version tag* and *access record*. The *version tag* indicates the locations of the persistent and runtime data for each embedding vector. The *access record* keeps the embedding update history. Each bitmap needs at most 9 MB memory for the largest synthetic dataset in our experiments. We also introduce a *delta buffer* to store the updated embeddings during checkpointing. In our experiments, the size of the delta buffer is up to 190 MB on the largest synthetic dataset, which is significantly small compared to the size of embedding tables (i.e., 150 GB).

NVM space cost. As shown in Table 3, PyTorch, TensorFlow, and OpenEmbedding need additional 1× NVM space to store the checkpointing of model parameters. Check-N-Run consumes more than 1× NVM space to save model states, including a full baseline checkpoint (i.e., all EMB-NVM) and the delta embeddings. NV-Checkpoint needs to pre-allocate a memory pool as large as possible to reduce data allocation and release time, which may consume up to 1× more NVM space for the best training performance. In contrast, NVrec maintains a full copy of the model parameters together with lightweight metadata (e.g., a 1-bit version tag per vector). Even in the extreme case of a 1-dimensional vector, which requires 32 bits, the additional space overhead of the version tag is below 3%. For higher-dimensional vectors, this fraction quickly diminishes, making the space cost practically negligible.

Offline profiling time. NVrec needs to offline profile the popularity of embedding vectors and update the vector placement before training, as described in §3.2. In our experiments, we spend less than 30 seconds profiling the Criteo. Moreover, NVrec can finish the profiling within 9 minutes for TB-scale embeddings (Figure 17(b)) with multi-threading, which is negligible compared to the long time model training (i.e., less than 1%).

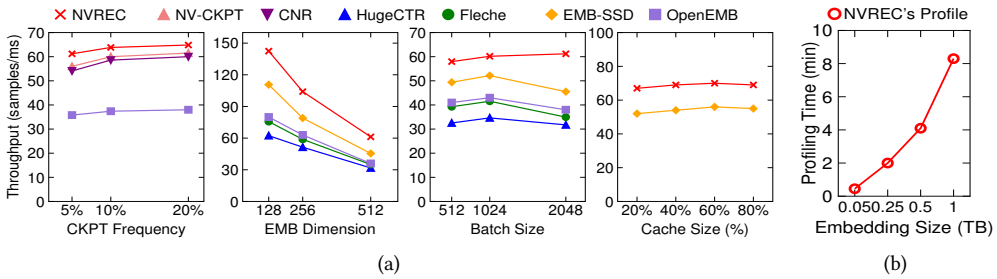


Fig. 17. (a) Sensitivity analysis on the Criteo dataset. (b) Profiling time.

5.9 Applicability Discussion

Applicability to distributed training. We currently implement and evaluate NVrec on a stand-alone server. We also outline how the design could be extended to distributed deployments. In a typical distributed setup, the MLP networks are replicated across GPUs, while the embedding tables are partitioned across servers and stored in hybrid memory [4]. In this setting, NVrec can apply vector-oriented placement on each server, keeping popular vectors in DRAM while balancing load across servers, and can use in-memory and pipelined checkpointing to persist each server's memory states. However, achieving a well-optimized solution in large-scale production systems would require additional engineering support and customization to the characteristics of the target system.

Specifically, supporting distributed profiling and in-place vector updates requires additional engineering adaptation and introduces some communication overhead. When training data is sharded across storage nodes and streamed by CPU data workers, NVrec can parallelize the access profiling in Section 3.2. Specifically, each worker scans its local shard and sends a per-ID summary to an aggregator node (elected among training nodes), which merges summaries to derive the popularity for Set_{new}. Given these statistics, each embedding server generates an *in-place* update plan only for its local embedding partition (Queue I/II/III in §3.2) and executes it locally, moving changed vectors. The resulting metadata (placement/ID mapping) is broadcasted for consistent lookup.

We aggregate $O(WK)$ summary entries and broadcast $O(C)$ messages recording the remapping/placement data. If remapping migrates vectors across nodes, we additionally transfer $O(C_{\text{mig}})$ vectors. Here, W is the number of CPU data workers, K is the number of IDs reported per worker, C is the number of vectors whose placement/ID mapping changes, and C_{mig} is the number of migrated vectors. On modern high-bandwidth interconnects, the transfer time can be upper-bounded by $T_{\text{comm}} \lesssim (WK \cdot b + C \cdot b_m + C_{\text{mig}} \cdot S_{\text{vec}})/B_{\text{net}}$, where b and b_m are the bytes per summary and per remapping entry, S_{vec} is the bytes per vector, and B_{net} is the effective network bandwidth.

However, NVrec does not aim to cover all distributed deployment scenarios. For deployments in which the same dataset or embedding table is concurrently shared by multiple users, applications, or long-running training jobs, vector renaming may require non-trivial consistency mechanisms. Therefore, NVrec focuses on scenarios where vector renaming can be performed offline before retraining with manageable overhead. Specifically, a retraining job uses its own remapped trace and embedding layout throughout training, rather than concurrently sharing and modifying the same embedding layout with other independent training jobs.

Applicability to different failures. There are different types of failures in production clusters [17] for model training. For power outages, NVrec can restart the application after reboot and recover training states from the persisted data in NVM. For network issues and non-NVM hardware failures, NVrec can migrate devices to new servers and recover model states from NVMs [24].

Table 3. Additional NVM space consumption of different checkpointing mechanisms.

Scheme	Extra NVM Space	Notes
PyTorch	1×	One full copy of embeddings for checkpointing
TensorFlow	1×	One full copy of embeddings for checkpointing
OpenEmbedding	1×	One full copy of embeddings for checkpointing
Check-N-Run	1×-N×	Full baseline copy + delta embeddings
NV-Checkpoint	1×-2×	Pre-allocated memory pool
NVREC	1×-1.03×	Dual-version (EMB-NVM) + pipelined checkpointing (others) + version tag

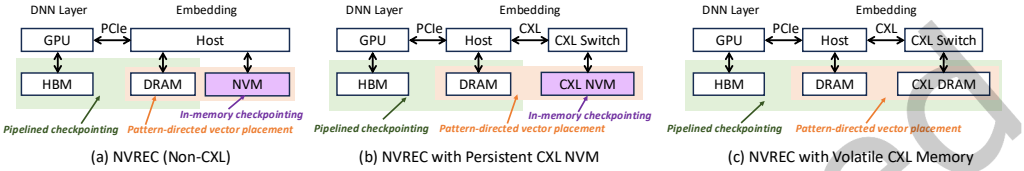


Fig. 18. NVREC applicability across memory architectures. (a) Hybrid memory with local DRAM+NVM. It does not use CXL. (b) CXL-based hybrid memory where the CXL-attached tier is persistent. (c) CXL-based hybrid memory where the CXL-attached tier is volatile.

NVREC needs to ensure cross-node consistency. Specifically, during training, NVREC writes updates only to the runtime version, while the persistent version remains read-only as the last committed checkpoint. If a node failure is detected early (e.g., via heartbeat monitoring) before a checkpoint is committed, recovery simply reloads the persistent version. When a checkpoint is triggered, all processes synchronize at a global barrier, and the checkpoint becomes recoverable only after every node reaches the barrier and commits the same checkpoint version. Then, each node atomically flips its version bitmap to promote the runtime version to the new persistent version.

If any node fails before the commit, the entire checkpoint is canceled: no node flips versions, and recovery always rolls back to the last committed persistent checkpoint. This barrier-and-commit rule ensures all nodes recover from the same version and avoids cross-node inconsistency. In HPC-like environments where node-local NVM may become inaccessible after a crash, NVREC can asynchronously transfer incremental checkpoint data to a remote/shared file system at a configurable (typically less frequent) interval to limit overhead and preserve performance, enabling recovery without relying on local NVM.

Applicability to different architectures. In this paper, we do not place embeddings on GPUs. However, NVREC supports storing popular embeddings in the HBM of GPUs and DRAM of CPUs, ensuring efficient vector access and correct checkpointing by transferring these embeddings to NVM for persistence.

Applicability to variants of persistent memory. In this paper, we evaluate NVREC using Intel Optane DC Persistent Memory. Although Intel Optane has been discontinued, we use it as a representative platform to study system design for memory-semantic tiered memory. The techniques in NVREC are not tied to Optane-specific features (e.g., its 256 B access granularity). Instead, they build on a general tiered-memory abstraction shared by emerging memory-semantic technologies [20, 34, 60, 61], including load/store access, non-volatility, larger capacity than DRAM, and lower access latency than SSD. Therefore, we design NVREC based on a general tiered-memory abstraction, which makes it potentially applicable to future memory-semantic NVM tiers with similar characteristics.

Applicability to the CXL-based architecture. We view CXL-like tiered-memory systems as a promising future direction for extending this design. In particular, NVREC can be applied to emerging CXL-based memory hierarchies where CXL Type-3 devices provide a memory-semantic, byte-addressable expansion tier, which helps

ensure the long-term relevance of our contribution beyond a specific Optane product [31]. If the CXL-attached memory is *persistent*, NVrec applies directly: since accessing local DRAM is more efficient than accessing a CXL-attached device [38], NVrec places hot embeddings in local DRAM and cold embeddings in the CXL-attached persistent tier via pattern-directed vector placement, while our in-memory checkpointing and fine-grained pipelined checkpointing remain applicable to accelerate checkpointing (Figure 18(b)). If the CXL-attached memory is *not* inherently persistent, NVrec still improves training via fine-grained pattern-directed vector placement to speed up embedding accesses, but our in-memory checkpointing design no longer applies as-is, and the system falls back to conventional pipelined incremental checkpointing to a persistent storage backend (Figure 18(c)). Since we do not experimentally evaluate NVrec on a CXL platform in this work, this discussion is intended as a forward-looking design consideration rather than an experimentally validated deployment claim.

6 Related Work

Embedding placement for recommendation systems. (1) *Dynamic caching*. Prior works [2, 4, 8, 18, 57, 66] implement data partition and dynamic caching for reducing latency using fast memory devices. While, they may suffer a low cache hit ratio and incur high software overhead. In contrast, NVrec employs a vector-oriented data placement scheme in a hybrid memory system to enhance the cache hit ratio. It accesses NVM directly, eliminating the overhead of transferring data from NVM to DRAM. (2) *Static profiling*. EMB-SSD [33] proposes a table-oriented policy for vector placements, which leads to a low cache hit ratio. PetPS [58] offloads all embeddings to NVMs and uses the hash function to locate embeddings. In contrast, NVrec utilizes a hybrid DRAM+NVM system to store embeddings, leveraging small fast DRAM to mitigate NVM-induced latency. Additionally, NVrec employs arrays for indexing embeddings, as we observe that additional hash calculations and collisions result in a 10% indexing overhead. (3) *Embedding prefetching* has been used to expedite data reading and mask CPU-GPU communication latency [4, 6, 35]. Due to the high proportion of vector-related I/O access, data prefetching's enhancement is limited. We do not consider this technique in this work.

Checkpointing approaches for recommendation systems. Recent works [17, 44] use asynchronous DNN checkpointing for pipelining to reduce end-to-end model training latency, which is similar to the naive pipelined checkpointing in §3.4. However, these approaches incur high overhead and under-utilize NVM, as checkpoint data is accessed only during system failures. OpenEmbedding co-designs the cache replacement with the checkpointing process. It needs to flush the checkpointed data in critical path, resulting in training performance loss. Unlike previous works, NVrec uses in-memory checkpointing for data in NVM (i.e., EMB-NVM) without extra writes, and fine-grained pipelined checkpointing for data in DRAM and HBM to further reduce checkpointing time.

7 Conclusion

In this paper, we present NVrec, a framework that leverages the unique characteristics of the hybrid DRAM and NVM to accelerate the training of recommendation systems. To achieve this goal, we design three key techniques including the fine-grained vector-oriented data placement approach to partition embeddings on DRAM and NVM for efficient data accesses, the dual-version checkpointing mechanism to support in-memory checkpointing and runtime data access, and the pipelined checkpointing to hide checkpointing time behind training time. We have implemented a software prototype of NVrec and tested it with real-world and synthetic datasets. Evaluations show that NVrec can improve the recommendation model training throughput by up to 6.85×, promote the performance of checkpointing over 5×, and reduces recovery time by up to 96%, compared to counterparts.

8 Acknowledgments

This work was supported by the National Science Foundation of China (U25A20423), Major Projects of Zhejiang Province (LD24F020012), Hangzhou Chengxi Sci-Tech Innovation Corridor Special Development Fund (K20241957), Key Research Project of Zhejiang Lab (2025SSYS0005), and the Science and Technology Program of Zhejiang Province (2026SDXT012).

References

[1] 2015. Avazu Dataset. https://www.kaggle.com/c/avazu-ctr-prediction.

[2] 2022. HugeCTR. https://github.com/NVIDIA-Merlin/HugeCTR.

[3] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. 2021. High-Performance Training by Exploiting Hot-Embeddings in Recommendation Systems. *arXiv:2103.00686* (2021).

[4] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. 2022. BagPipe: Accelerating Deep Recommendation Model Training. *arXiv:2202.12429* (2022).

[5] Amazon Web Services. 2026. CreateSolution — Amazon Personalize. https://docs.aws.amazon.com/personalize/latest/dg/API_CreateSolution.html.

[6] Keshav Balasubramanian, Abdulla Alshabanah, Joshua D Choe, and Murali Annavaram. 2021. cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models. In *Proceedings of the 15th ACM Conference on Recommender Systems*.

[7] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, Ennan Zhai, and Alibaba Cloud. 2024. Crux: GPU-Efficient Communication Scheduling for Deep Learning Training. In *Proceedings of the Special Interest Group on Data Communication*.

[8] Cheng Chen, Yilin Wang, Jun Yang, Yiming Liu, Mian Lu, Zhao Zheng, Bingsheng He, Weng-Fai Wong, Liang You, Penghao Sun, et al. 2023. OpenEmbedding: A Distributed Parameter Server for Deep Learning Recommendation Models using Persistent Memory. In *Proceedings of the 2023 IEEE 39th International Conference on Data Engineering*.

[9] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*.

[10] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-volatile Memories. *ACM SIGARCH Computer Architecture News* (2011).

[11] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[12] Tyler Coy, Shuibing He, Bin Ren, and Xuechen Zhang. 2020. Compiler Aided Checkpointing Using Crash-Consistent Data Structures in NVMM Systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*.

[13] cuBLAS. 2022. Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas.

[14] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[15] Zheng Dang, Shuibing He, Xuechen Zhang, Peiyi Hong, Zhenxin Li, Xinyu Chen, Haozhe Song, Xian-He Sun, and Gang Chen. 2024. PMAlloc: A Holistic Approach to Improving Persistent Memory Allocation. *ACM Transactions on Computer Systems* (2024).

[16] DLRM. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. https://github.com/facebookresearch/dlrm.

[17] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*.

[18] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of Machine Learning and Systems*.

[19] Yufei Feng, Fuyu Lv, Weichen Shen, Menghan Wang, Fei Sun, Yu Zhu, and Keping Yang. 2019. Deep Session Interest Network for Click-Through Rate Prediction. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*.

[20] Scott W Fong, Christopher M Neumann, and H-S Philip Wong. 2017. Phase-Change Memory-Towards a Storage-Class Memory. *IEEE Transactions on Electron Devices* (2017).

[21] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*.

[22] Intel. 2012. Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[23] Intel. 2014. Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual. https://www.intel.com/content/www/us/en/content-details/671360/intel-xeon-processor-e5-v2-and-e7-v2-product-families-uncore-performance-monitoring-reference-manual.html.

[24] Intel. 2019. Defining the Future of In-Memory Database Computing. https://www.mouser.com/pdfDocs/accenture-partner-whitepaper.pdf.

[25] Intel. 2019. Intel Xeon Gold 5218R Processor. https://www.intel.com/content/www/us/en/products/sku/199342/intel-xeon-gold-5218r-processor-27-5m-cache-2-10-ghz/specifications.html.
[26] Intel. 2019. Pmem is a Revolutionary Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.
[27] Intel. 2021. New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr_new_opportunities_for_persistent_memory_applications.html.
[28] Intel. 2022. Optane DC Persistent Memory. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.
[29] Intel. 2022. Optane SSD DC P4800X. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-mdt-brief.pdf.
[30] Kaggle-Criteo. 2022. Criteo Dataset. https://www.kaggle.com/c/criteo-display-ad-challenge/data.
[31] Senior Manager DRAM Product Planning Kapil Sethi. 2022. Expanding the Limits of Memory Bandwidth and Density: Samsung's CXL Memory Expander. https://semiconductor.samsung.com/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/.
[32] Hiwot Tadese Kassa, Paul Johnson, Jason Akers, Mrinmoy Ghosh, Andrew Tulloch, Dheevatsa Mudigere, Jongsoo Park, Xing Liu, Ronald Dreslinski, and Ehsan K Ardestani. 2023. MTrainS: Improving DLRM Training Efficiency using Heterogeneous Memories. *arXiv:2305.01515* (2023).
[33] Minsub Kim and Sungjin Lee. 2020. Reducing Tail Latency of DNN-based Recommender Systems using In-storage Processing. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*.
[34] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an Energy-efficient Main Memory Alternative. In *Proceedings of 2013 IEEE International Symposium on Performance Analysis of Systems and Software*.
[35] Youngeun Kwon and Minsoo Rhu. 2022. Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward not Backwards. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.
[36] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, et al. 2023. AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*.
[37] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W Lee, and Tae Jun Ham. 2021. MERCI: Efficient Embedding Reduction on Commodity Hardware via Sub-Query Memoization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[38] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.
[39] Zhenxin Li, Shuibing He, Zheng Dang, Peiyi Hong, Xuechen Zhang, Rui Wang, and Fei Wu. 2024. CCL-BTree: A Crash-Consistent Locality-Aware B+-Tree for Reducing Xpbuffer-Induced Write Amplification in Persistent Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*.
[40] Zhenxin Li, Bing Jiao, Shuibing He, and Weikuan Yu. 2022. PHAST: Hierarchical Concurrent Log-Free Skip List for Persistent Memory. *IEEE Transactions on Parallel and Distributed Systems* (2022).
[41] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2019. Enforcing Crash Consistency of Evolving Network Analytics in Non-volatile Main Memory Systems. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*.
[42] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. 2021. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *Proceedings of Machine Learning and Systems*.
[43] Micron. 2022. DDR5 DRAM. https://www.micron.com/products/memory/dram-components/ddr5-sdram?srsltid=AfmBOoo-d4GZq-UmUepAQDkvjknUUja2iajFH6Vg_31rktFGCDGwJCgJ.
[44] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*.
[45] Yabo Ni, Dan Ou, Shichen Liu, Xiang Li, Wenwu Ou, Anxiang Zeng, and Luo Si. 2018. Perceive Your Users in Depth: Learning Universal User Representations from Multiple E-Commerce Tasks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
[46] NVIDIA. 2014. Unified Addressing [CUDA Driver API]. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/html/group__CUDA__UNIFIED.html.
[47] NVIDIA. 2022. cuDNN Frontend API. https://github.com/NVIDIA/cudnn-frontend.
[48] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: Leveraging Persistent Memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[49] PyTorch. 2022. Saving and Loading a General Checkpoint in PyTorch. https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html.
[50] PyTorch. 2024. PyTorch. https://pytorch.org.
[51] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-Tree vs. LSM-Tree Write Amplification Gap on Modern Storage Hardware with Built-In Transparent Compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies*.
[52] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*.
[53] Nirmal Raj Saxena, Saurabh Hukerikar, Mikolaj Blaz, and Swapna Raj. 2024. Optimal Checkpoint Interval with Availability as an Objective Function. *arXiv:2410.18124* (2024).
[54] TensorFlow. 2022. TensorFlow Checkpointing. https://www.tensorflow.org/guide/checkpoint.
[55] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture*.
[56] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive based Recommendation Inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[57] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. 2022. Fleche: an Efficient GPU Embedding Cache for Personalized Recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*.
[58] Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, and Jiwu Shu. 2023. PetPS: Supporting Huge Embedding Models with Persistent Memory. In *Proceedings of the Very Large Data Base Endowment*.
[59] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*.
[60] Siling Yang, Shuibing He, Hexiao Duan, Weijian Chen, Xuechen Zhang, Tong Wu, and Yanlong Yin. 2023. APQ: Automated DNN Pruning and Quantization for ReRAM-based Accelerators. *IEEE Transactions on Parallel and Distributed Systems* (2023).
[61] Siling Yang, Shuibing He, Wenjiong Wang, Yanlong Yin, Tong Wu, Weijian Chen, Xuechen Zhang, Xian-He Sun, and Dan Feng. 2025. GOPIM: GCN-oriented pipeline optimization for PIM accelerators. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture*.
[62] Chenxuan Yao, Feifan Liu, Yuchong Hu, Zhengyu Liu, Xinjue Zheng, and Wenxiang Zhou. 2025. LowDiff: Efficient Frequent Checkpointing via Low-Cost Differential for High-Performance Distributed Training Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
[63] Chaoliang Zeng, Xudong Liao, Xiaodian Cheng, Han Tian, Xinchen Wan, Hao Wang, and Kai Chen. 2024. Accelerating Neural Recommendation Training with Embedding scheduling. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*.
[64] Yujing Zhang, Zhangming Chan, Shuhao Xu, Weijie Bian, Shuguang Han, Hongbo Deng, and Bo Zheng. 2022. KEEP: An Industrial Pre-Training Framework for Online Recommendation via Knowledge Extraction and Plugging. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*.
[65] Yang Zhang, Fuli Feng, Chenxu Wang, Xiangnan He, Meng Wang, Yan Li, and Yongdong Zhang. 2020. How to Retrain Recommender System? A Sequential Meta-learning Method. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*.
[66] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*.
[67] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending What Video to Watch Next: a Multitask Ranking System. In *Proceedings of the 13th ACM Conference on Recommender Systems*.
[68] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.
[69] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*.
[70] Yuanhang Zou, Zhihao Ding, Jieming Shi, Shuting Guo, Chunchen Su, and Yafei Zhang. 2023. EmbedX: A Versatile, Efficient and Scalable Platform to Embed Both Graphs and High-Dimensional Sparse Data. In *Proceedings of the Very Large Data Base Endowment*.

Received 5 October 2025; revised 19 June 2026; accepted 19 June 2026