Efficient Large Scale DLRM Implementation on Heterogeneous Memory Systems

Mark Hildebrand*[0000−0001−6105−1643], Jason Lowe-Power[0000−0002−8880−8703], and Venkatesh Akella[0000−0003−3014−5326]

University of California, Davis
{mhildebrand, jlowepower, akella}@ucdavis.edu

Abstract. We propose a new data structure called *CachedEmbeddings* for training large scale deep learning recommendation models (DLRM) efficiently on heterogeneous (DRAM + non-volatile) memory platforms. *CachedEmbeddings* implements an implicit software-managed cache and data movement optimization that is integrated with the Julia programming framework to optimize the implementation of large scale DLRM implementations with multiple sparse embedded tables operations. In particular we show an implementation that is 1.4X to 2X better than the best known Intel CPU based implementations on state-of-the-art DLRM benchmarks on a real heterogeneous memory platform from Intel, and 1.32X to 1.45X improvement over Intel's 2LM implementation that treats the DRAM as a hardware managed cache.

1 Introduction

Deep Learning Recommendation Models (DLRM) are state of the art AI/ML workloads underlying large scale ML-based applications [16]. These models require hundreds of gigabytes of memory and thousands of sparse embedding table operations [16], which makes them challenging to implement on current computer systems. As shown in Figure 1, DLRM model operates on a collection of dense features and sparse features. Dense features are processed by a standard Multi-Level Perceptron (MLP) network. The sparse features, on the other hand, are used to index into embedding tables to extract dense features. Sparse features can encode information such as a user id, product id, etc. The outputs of the individual embedding table lookups are concatenated together and combined with the output of the bottom MLP using various feature interaction techniques. Post interaction tensors are processed by a final top MLP before yielding a final result. The architectural implications of these networks has been investigated in depth in the literature [7]. Embedding table lookup and update operations are memory bandwidth intensive while the dense MLP layers, on the other hand, are compute intensive. This combination stresses many architecture subsystems. Further complicating matters is the size of these embedding tables, which can occupy tens to hundreds of gigabytes and are expected to grow [7, 12].

Emerging heterogeneous memory based platforms that combine terabytes of non-volatile RAM such as 3DXpoint [11] and hundreds of gigabytes of DRAM

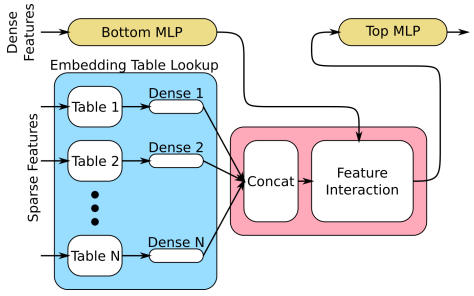


Fig. 1: *Generalized DLRM architecture*

are naturally attractive as they meet the memory demands of DLRM workloads at a reasonable power/cost [2, 4]. When moving to these heterogeneous memory systems, we must manage the data movement and placement smartly to achieve the best performance. There are three classes of techniques to move data in these heterogeneous memory systems: hardware (usually at a 64-byte block granularity), operating system (usually at a page granularity), or directly by the application (at any granularity). Unfortunately, each of these techniques come with significant downsides. Hardware-based data movement wastes memory bandwidth by requiring up to four *extra* memory accesses on every demand request and can lead to poor performance [9]. OS-based data movement is not always timely and can be wasteful for applications with sparse memory access patterns [13, 22]. Finally, requiring the application developer to manually move data is burdensome and requires modifying the algorithm and deep application changes [3, 20].

Naive methods for heterogeneous memory embedding table management may fall short for several reasons. First, just placing the tables in non-volatile memory will not yield good performance due to the significantly lower performance of non-volatile memory technologies such as 3DXpoint when compared with DRAM. Next, the reuse pattern of entries within an embedding table can vary significantly from essentially random highly local and can change over time [5]. This suggests the need for a dynamic policy that is capable of meeting these different requirements. Further, while researchers have investigated using heterogeneous memory to store portions of these embedding tables [4], these works tend to focus on using NVMe SSDs for their tiered storage. The main issue with simple caching is that embedding table are sparsely accessed and lookups have little spatial locality and varying temporal locality.

In this paper, we focus on deep learning recommendation workloads with very large sparse embedding tables. We will show the data use/reuse patterns with sparse embedding tables is complex and there are complex interactions between due to the sparsity of the tables, batch size, features size, number of tables accessed, number of accesses, and parallelization techniques for lookup/update operation which cause poor performance for hardware caches. To decrease the burden on programmers but get the performance benefits of manually data movement, we introduce *CachedEmbeddings* which is a new runtime-optimized data

structure for the embedding tables of the Deep Learning Recommendation Model (DLRM) workload. Specifically, we will target data movement during the embedding table lookup and gradient descent update operations. In this paper, we refer to a system with DRAM and non-volatile memory as a *heterogeneous memory* system and Intel's Optane Persistent Memory which is based on 3DXpoint non-volatile memory technology as **PM** (**P**ersistent **M**emory).

The novel contributions of this work are two fold. First, almost all the prior work in this area has focused on optimizing the embedding table operations on a homogeneous memory platform (CPU or GPU) taking advantage of the statistical distribution of the embedding table rows with clever partitioning and data layout techniques. In this work, we first perform benchmarking and analysis of a heterogeneous memory platform and show that it introduces a different set of tradeoffs (Section 3). Second, the core contribution of this work is a data tiering framework (or algorithm) that is centered around a new data structure (called *CachedEmbeddings*) and an API to implement different platform-specific data movement optimizations integrated with Julia programming framework. So, the proposed framework can be used for future workloads that may have different access patterns, model capacities, and statistical distributions, and more importantly different hardware platforms. The proposed data tiering framework goes beyond a traditional software-managed cache in terms of providing a comprehensive mechanism for memory allocation and deallocation, prefetching, and moving data at larger granularity that is closer to the semantics of the data. This makes it easier for the programmer to use the proposed API.

We evaluate our implementation of DLRM based on *CachedEmbeddings*and find it is is 1.4X to 2X better than the best known Intel CPU based implementations on state-of-the-art DLRM benchmarks on a real heterogeneous memory platform from Intel [12], and 1.32X to 1.45X improvement over Intel's 2LM implementation that treats the DRAM as a hardware managed cache for the non-volatile memory.

2 Related Work

Bandana [5] aims to reduce the amount DRAM required for DLRM inference workloads on CPU clusters by using a combination of DRAM and SSDs, using heuristics to determine how to cache embedding vectors in DRAM. Like our work, Bandana also caches hot vectors in DRAM. However, Bandana needs to overcome the coarse read granularity of SSDs and must use hypergraph partitioning to group vectors with spatial locality to the same sector within the SSD. Persistent memory does not have this limitation, so this work investigates fine-grained vector caching while still maintaining high read and write bandwidth to PM. A performance model for DLRM training on GPUs is presented in [14] and using heterogeneous memory for DLRM inference to lower power consumption and cost is presented in [2] and DLRM inference on CPU cluster is presented in [8]. There are two state-of-the-art implementations of DLRM training in recent literature. Facebook's NEO [16] is software/hardware codesign of large scale

DLRM models on a custom GPU-based hardware platform called ZionEX. It uses a customized 32-way set-associative software cache with LRU and LFU cache replacement policies and enables fine grain control of caching and replacement. Though NEO is focused on the GPU ecosystem, it provides motivation for the need of software managed caches to deal with large embedding tables. Intel's DLRM implementation [12] focuses on efficient parallelization across multiple CPU and a novel implementation of the SGD optimizer targeting mix-precision training. We extend this work by proposing a scale-up solution taking advantage of heterogeneous memory. Recently there has been work [1, 19, 21] in identifying and storing "hot" vectors in faster memory. Further, recent work [6] proposed a software caching idea similar to ours for GPU-based DLRM training, though with a different implementation mechanism.

To the best of our knowledge this is the first work on implementation and optimization of large scale DLRM *training* on a system with DRAM and nonvolatile RAM (Intel's Optane Persistent Memory). In addition, this work introduces a generic data management API for optimizing embedded table implementations on heterogeneous memory systems that is useful beyond just DLRM workloads. This work goes beyond just *caching* frequently used vectors to providing a mechanism to the programmer to tailor the movement of data algorithmically to meet the unique constraints/features of the underlying hardware platform.

3 Implementing Embedding Tables in Heterogeneous Memory Systems

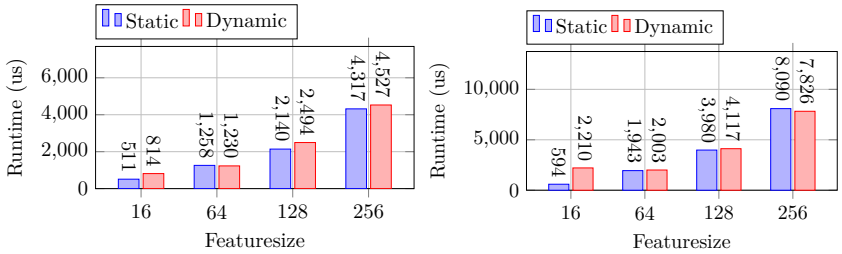
As noted in prior works [1, 16, 17, 19] embedding table operations have high bandwidth demands and low computation intensity, and moreover, one size doest not fit all. So, it is a challenge even on a homogeneous memory system like a CPU or GPU. Heterogeneous memory introduces new challenges. Performance of embedding tables depends on a variety of parameters such as number of threads, whether the feature size is fixed as a compiler-time parameter or dynamic, which means known at runtime, the feature size (we sweep from 16 to 256), the number of accesses, number of tables (we vary from 10 to 80), the location of the tables, whether they are in PM or DRAM, number of worker threads, direct vs indirect lookup (one memory access to retrieve the pointer to the vector and one more access to retrieve the vector) standard vs non-temporal stores for conducting the final write operation of an embedding table update. Non-temporal stores hint to the hardware that the associated data is not intended to be used in the near future, enabling CPU cache optimizations.

Methodology Experiments were conducted on a single socket, with one thread per core on a 2-socket 56 core (112 thread) Intel Xeon Platinum 8276L running Ubuntu 21.10 with 192 GiB (6x32 GiB) DRAM and 1.5 TB (6x256 GiB) Optane DC NVRAM (3DXpoint-based PM) per socket. We used an embedding table library written in Julia[1] to decouple embedding table operations from data struc-

[1] https://github.com/darchr/EmbeddingTables.jl

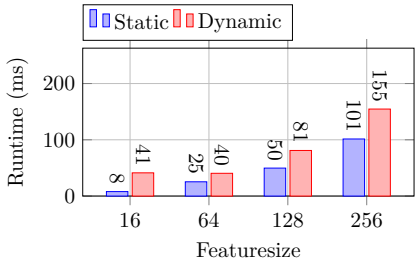
ture implement. For deep learning primitives, we wrote a Julia wrapper around Intel's oneDNN library[2].

The experiments consisted of running the kernel of interest multiple times until 20-seconds of wall-clock time had elapsed, the execution time for each invocation was logged. For each invocation of the kernel, new lookup/update indices where generated randomly from a uniform distribution. Execution time for the gradient descent update kernels includes the time for reindexing. In addition to execution time, hardware performance counters for DRAM and PM read and write traffic were also collected, sampled at the beginning and end of each kernel invocation. All experiments used a large batchsize of 16384. Embedding tables were sized to occupy a memory footprint between 1 GiB and 80 GiB to minimize the effect of the L3 cache.

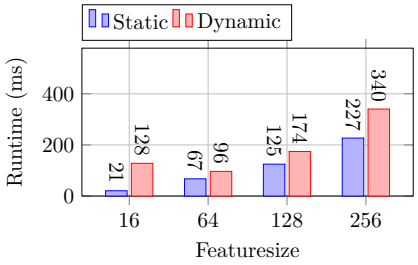


(a) Non-reducing lookup using DRAM.

(b) Non-reducing lookup using PM.



(c) Reducing lookup using DRAM.



(d) Reducing lookup using PM.

Fig. 2: *Comparing the execution time of static versus dynamic feature-sizes for a single embedding table lookup operation using a single thread. Figures cover the range of non-reducing and reducing (accesses* $= 40$*) operations with the embedding table in either DRAM or Optane PM. Within each regime, a range of feature sizes is explored. All runs used single-precision floating point with a batchsize of* $16,384$ *and nvectors* $= 10,000,000$

Systems equipped with Optane can run in two modes, an *app direct* mode where memory is explicitly allocated on PM with loads and stores going directly

[2] https://github.com/hildebrandmw/OneDNN.jl

to the devices and a *2LM* cache mode where DRAM acts as a transparent direct-mapped cache for PM [11]. Unless otherwise specified, all of our experiments were conducted in *app direct* mode. Next, we present relevant and interesting results from the large number of experiments conducted.

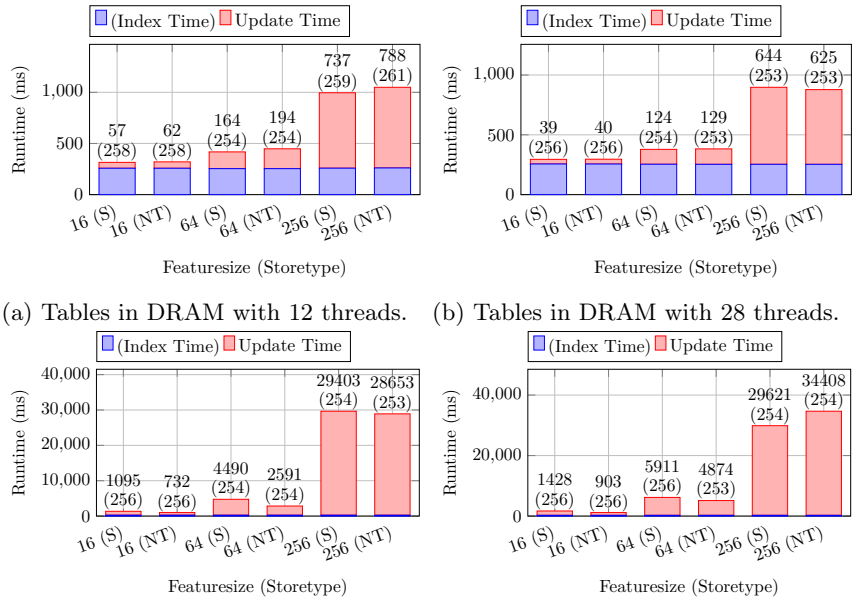
Static and Dynamic Featuresize First, we investigate the trade-off between dynamic and static feature size definitions for both reducing and non-reducing lookups. Figure 2 compares the execution time of static versus dynamic features sizes for a single embedding table lookup using a single thread across the combinations of reducing ($accesses = 40$) and non-reducing lookups with the embedding table in DRAM and Optane PM. In different situations, embedding table definitions may or may not know *a priori* the size of the embedding table entries, which leads to different code generation and different performance. With static feature sizes, the compiler can specialize the embedding table lookup code for a single feature size. In the dynamic case, the compiler cannot optimize the embedding table accesses. Additionally, when feeding the embedding table lookup results into the dense MLP in DLRM, sometimes a single embedding table entry is used (non-reducing) and other times many entries from the embedding table are *reduced* into a single value which is sent the MLP. We find that for non-reducing accesses there is little difference in performance, but when multiple lookups are required for each output, the static implementation outperforms the dynamic one in the reducing case. Finally, the performance of PM in these applications is on the order of 2× slower than DRAM showing that even for a single thread, memory location matters. This demonstrates that kernel implementation matters and knowledge of the underlying hardware is key to achieving high performance for these types of workloads.

To show that the lookup implementation is highly performant, we demonstrate that the implementation achieves close to the theoretical bandwidth of the platform. When the tables are located in DRAM, we achieve close to 100 GB/s of read bandwidth. This is close to the theoretical bandwidth of 110 GB/s. THe PM bandwidth achieved during ensemble lookup is between 10 GB/s (featuresize 16) and 25 GB/s (featuresize 256), which tracks well with the expected random-access read-only bandwidth for these devices [11].

SGD Update Performance - Worker Threads and Nontemporal Stores Figure 3 shows an example ensemble gradient update performance broken down between DRAM and PM, number of worker threads, and usage of standard versus non-temporal stores. The performance of DRAM (Figures 3a and 3b) increases with the number of threads with little performance difference between standard and non-temporal stores during the update phase. However, for DRAM, the indexing time to create the new CSR array for gradient updates dominates the total update time except for the largest embedding element sizes.

Persistent memory (Figures 3c and 3d) exhibits more nuanced behavior. Because the write bandwidth to PM is much lower, reindexing time is less of a bottleneck than it is for DRAM. Furthermore, non-temporal stores tend to perform significantly better, especially for larger feature sizes. This is likely because

non-temporal stores evict the corresponding cachelines from the cache. This causes the writes to appear at the memory controller as a group allowing for write-combining within the Optane memory controller (this generation of Optane DIMMs have a 256 B access granularity). Without non-temporal stores, the corresponding cache lines only arrive at the memory controller when evicted from the L3 cache, leading to lower spatial locality.



(a) Tables in DRAM with 12 threads.

(b) Tables in DRAM with 28 threads.

(c) Tables in PM with 12 threads.

(d) Tables in PM with 28 threads.

Fig. 3: *Execution time for embedding table SGD application comparing the use of non-temporal (NT) and standard (S) stores. 40 independent tables were used with 1 million vectors each, 40 tables accesses per output, batchsize 16384. Times to perform the update (no parentheses) and the indexing procedure (in parentheses) are given above each bar.*

For these experiments, the time taken by the reindexing procedure is mostly constant and takes a large fraction of the overall execution time when the embedding tables are in DRAM. This is largely because the reindexing procedure is largely targeted for situations where the number of unique indices accessed is relatively small compared to the number of vectors in the table. A choice of data structures and reindexing operation targeted more specifically at this "high density" situation may reduce the this time.

Note that in all cases exhibit a sharp increase in execution time when moving from a feature size of 64 (256 bytes) to 256 (1024 bytes). This is because the contiguous memory accesses of 1024 bytes are sufficient to trigger the stream-

ing prefetcher, which fetches more than just the necessary cache lines causing bandwidth bloat. This phenomenon goes away when the streaming prefetcher is disabled in the system BIOS. In our experimental data is reported with the prefetcher enabled as we expect this to be a more common scenario.

Design Space Exploration Summary Through our experiments, we make the following conclusions. First, placing the tables in PM results in lower performing lookup and update operations than DRAM. Further, this highlights the need to perform some kind of heterogeneous memory management to get the capacity advantage of PM without paying the full performance price. Second, higher performance implementations of embedding table operations requires cooperation with and understanding of the underlying hardware and the best implementation can change depending on the particular operation. For example, the use of non-temporal stores for update operations is beneficial for performance when embedding tables are in PM, but makes little difference when DRAM is used. Finally, in the context of multithreaded ensemble lookups and updates, an extra level of indirection can be tolerated limited performance penalty (about a $2\times$ overhead for featuresize 16 down to about 10% for a feature size of 128). This is the main idea behind our idea of memory management for these tables which will be presented in the next section. Adding this indirection allows individual vectors to be stored in either PM or DRAM. With careful selection, we should be able to move frequently accessed vectors into DRAM while leaving infrequently accessed ones in PM, providing most of the performance of an all DRAM with the capacity of PM.

4 Cached Embeddings

In this section, we discuss how to apply the framework of heterogeneous memory management to embedding table lookups and updates into an approach called *CachedEmbeddings*.[3] Key aspects to keep in mind are that (1) access to each embedding table is performed on the granularity of feature vectors, (2) there is no reason to expect accesses to exhibit spatial locality, and (3) accesses *may* exhibit temporal locality. The key insight of *CachedEmbeddings* is to add an extra level of indirection to each feature vector access, allowing individual feature vectors to be cached in DRAM while stored in PM.

Figure 4 shows an overview of our approach. Base data for the embedding table is located in PM (beginning at address `0x1000` in the example). Each embedding table maintains a cache in DRAM that vectors can be migrated to. Internally, the embedding table maintains a vector of pointers, one for each row, pointing to where the primary region for that row is. Since embedding table rows are relatively large ($> 64\,B$), these pointers have unused lower order bits. We use the least significant bit (LSB) to encode whether the corresponding row is in the base data or in a cache page. The second LSB is used as a lock-bit. A thread wanting to move a row uses an atomic compare-and-swap to gain ownership of

[3] https://github.com/darchr/CachedEmbeddings.jl

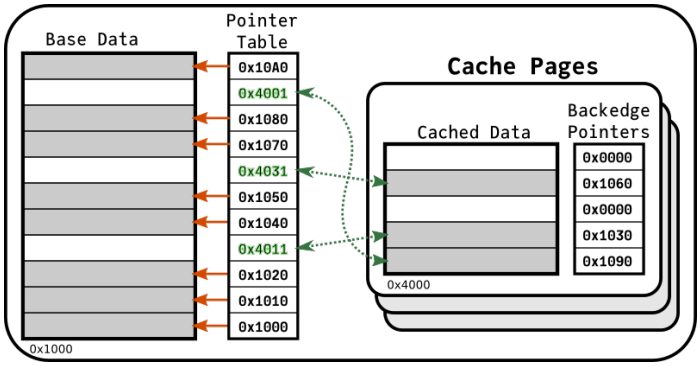


Fig. 4: Overview of *CachedEmbeddings*. *Base data lives in PM, (with a base address of* `0x1000` *as an example). In this example, each feature vector occupies 16 bytes. A pointer table tracks the actual location of each vector with the least significant bit indicating whether it's cached. Upon a lookup access, vectors are moved into cache pages. Each page contains backedges, which indicates whether the corresponding slot is filled and if so, the vectors original location.*

the row. If ownership is acquired, the thread is free to move the row into the cache and unlock the row.

To support multithreaded access, the cache is composed of multiple cache pages with synchronization for allocation. If the most recent cache page is full, then the thread must acquire a lock for the table in order to allocate new cache page. The cache has a configurable maximum size, beyond which no more feature vectors can be migrated until the cache is flushed. Each cache page also maintains a vector of backedge pointers to each cached row's original location (or null if the slot is empty) to facilitate this flushing. The cache is flushed one page at a time. If the cache page is entirely clean (in the case that only lookups were performed with no update operations), flushing a cache page simply involves updating the *pointer table* back to each vector's original location and then deleting the cache page. If the vectors are dirty (e.g., the table was used during training) then the vectors within the cache page must also be written back to their original location.

The size of the cache is determined by two parameters. The parameter `cachelower` is a soft lower bound for the size of the cache. When the cache is flushed, pages will be sequentially flushed until the size of the cache is less than `cachelower`. The parameter `cacheslack` is flexible space to allow the cache to grow. New vectors can be cached until the total size of the cache exceeds `cachelower + cacheslack`. Thus, the size of the DRAM cache for each table can fluctuate between `cachelower` and `cachelower + cacheslack`.

Table 1 outline the API for a CachedEmbeddingTable. At a high level, the functions `access_and_cache` and `access` provides methods for retrieving feature vectors while optionally migrating vectors into the table's DRAM cache. Setters `set_cachelower` and `set_cacheslack` are used to modify their corresponding

Operation	Description
`access_and_cache`	Get the pointer for the requested feature vector, caching it in DRAM if (1) the cache is not full, (2) the vector is not already cached, and (3) ownership of the row is acquired.
`access`	Get the pointer for the requested feature vector without caching. This function is connected to the `rowpointer` function for all other access contexts besides `Forward`.
`set_cachelower`	Set the `cachelower` variable.
`set_cacheslack`	Set the `cacheslack` variable.
`isfull`	Return `true` if the cache is full. Otherwise, return `false`.
`flush_clean`	Purge the oldest cache pages until the size of the cache is less than `cachelower`. Do not write back data from cache pages to the base array.
`flush_dirty`	Purge the oldest cache pages until the size of the cache is less than `cachelower`. Do write back data from cache pages to the base array.

Table 1: API for a CachedEmbeddingTable.

cache size parameter variables. Finally, `flush_clean` and `flush_dirty` provide methods for reducing the size of the cache to enable future vector accesses to be cached. With this API, we can simply extend *CachedEmbeddings* to new memory architectures (e.g., CXL) by modifying the backend implementations of these functions. On the user-facing side of the API, there will be no changes required to port the application to a new memory technology.

We evaluated the performance impact of a level of indirection and found it does not have a significant impact on the embedding table lookup time. An extra level of pointer chasing causes a slight slowdown when embedding tables are in DRAM, but it has roughly performance parity when the tables are in PM. In this bandwidth constrained environment with a large number of threads, the overhead introduced by an extra level of pointer chasing is negligible. Thus, we can add a level of indirection, allowing individual feature vectors to be located in either DRAM or PM, without a large sacrifice in performance.

4.1 *CachedEmbeddings* Performance

In this section, we perform experiments to determine the performance of the *CachedEmbeddings*.

Methodology When comparing the performance of *CachedEmbeddings* to standard embedding tables, we focus on the lookup operation performance. This is because, in the context of DLRM training, feature vectors will be cached in DRAM during the lookup operation and simply accessed during the gradient descent operation. The performance of this update operation and subsequent cache flushing is harder to micro-benchmark for a couple of reasons. First, in

the context of DLRM training, we would expect all embedding tables entries accessed during the update phase to already be cached. Second, the frequency of a flush operation is dependent on the input index distribution and thus doesn't necessarily occur on every training iteration. Consequently, we will examine update performance when we study then end-to-end performance of DLRM with *CachedEmbeddings* in Section 6.

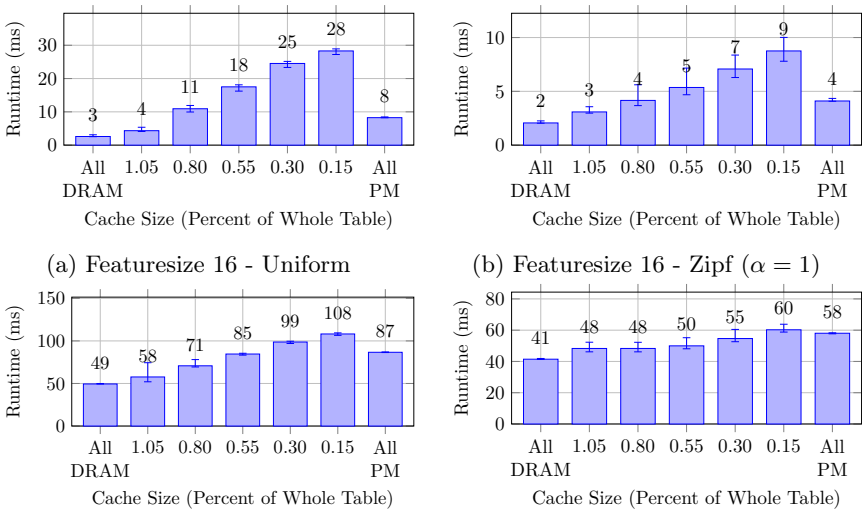
For our benchmarks, we want to target conditions where a mix of DRAM and PM makes sense (i.e., the total memory footprint is high). To that end, we investigate ensemble lookups with 80 tables and 28 threads with featuresizes of 16 and 256 and accesses of 1 and 40. Furthermore, each table consisted of 1 million vectors and a batch size of 16384 was used. To investigate the effects of cache size, we set `cacheslack` to be 5% and `cachelower` to 10%, 25%, 50%, 75% and 100% of each table's total memory footprint.

To investigate the effects of temporal locality (e.g., users frequently returning to a web application), the lookup indices for each table are drawn from either a uniform distribution (which has low temporal locality) or a Zipf [18] distribution with $\alpha = 1$ (which has high temporal locality). In order to avoid spatial locality introduced by the Zipf distribution, the index sampling is followed by a maximum length linear feedback shift register (LFSR) using a different seed for each table.

For comparison points, the same experiments were run for standard embedding table with either all data stored in DRAM or PM and no indirection in the lookup accesses. As before, each lookup operation is invoked multiple time with different indices until the total benchmark runtime exceeds 20 seconds. For the experiments conducted using *CachedEmbeddings*, the `flush_clean` operation is run after each invocation.

Results Figure 5 shows the results for a non-reducing embedding table ensemble lookup. The left-most and right-most bars in each figure show the performance of a standard embedding table with all DRAM and PM respectively. In between is shown the performance of a *CachedEmbeddings*, with the label giving the sum of `cachelower` and `cacheslack` as a percent of the ensemble's total memory footprint. For a featuresize of 16 (Figure 5a and 5b), the overhead of cache management overheads dominates resulting in significant slowdown over the all PM simple table. Even with the larger featuresize of 256, *CachedEmbeddings* requires a fairly large cache size to outperform the all PM standard table.

There are a number of reasons for this. First, non-reducing lookups are essentially a memory copy from either DRAM to DRAM or PM to DRAM. This higher DRAM write traffic can, to some extent, help mitigate the lower read bandwidth of PM which we can see with the $2\times$ lower performance of the PM based simple tables than the DRAM based ones for the uniform distribution. Second, because `flush_noclean` is called after every invocation and only at most 16384 are accessed on each lookup (around 1.6% of the embedding table) the table never reaches the state where the cache is full (recall that `cacheslack` was set to 5% of the overall table size). This means that the *CachedEmbeddings* table



(a) Featuresize 16 - Uniform

(b) Featuresize 16 - Zipf ($\alpha = 1$)

(c) Featuresize 256 - Uniform

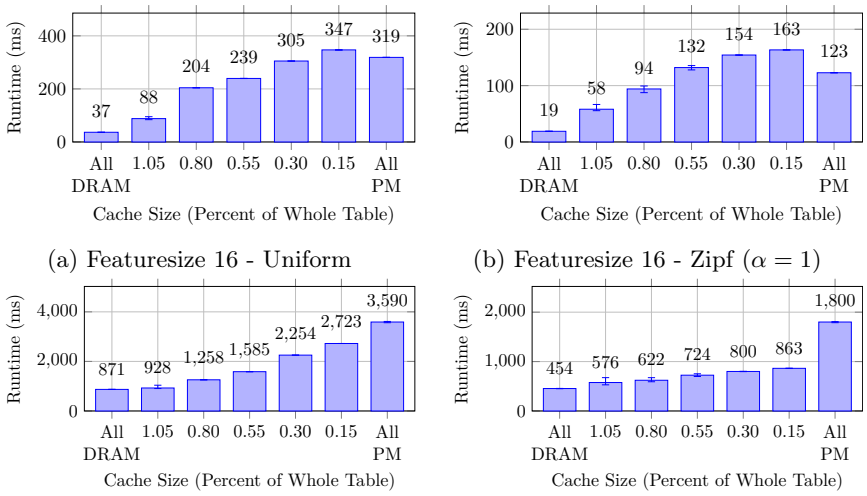
(d) Featuresize 256 - Zipf ($\alpha = 1$)

Fig. 5: *Comparison of* CachedEmbeddings *with standard embedding tables located in DRAM or PM for* nonreducing *lookups for uniform and zipf distributions. Runs were conducted with 80 embedding tables and 28 worker threads.*

is always doing extra work and cannot necessarily take advantage of preexisting cached vectors.

Figure 6 shows the performance of *CachedEmbeddings* for reducing lookups (with $accesses = 40$). Again, the smaller feature sizes yield poorer performance advantages (or even performance regressions at smaller cache sizes) because the time spent moving data around is so low enough that the extra steps required by *CachedEmbeddings* can dominate. However, for larger feature sizes like 64 and 256, the performance of *CachedEmbeddings* nearly interpolates linearly between the performance of all DRAM and all PM. This is because with a batchsize of 16384 and 40 accesses per batch, a large portion of each embedding table is accessed on each lookup operation, resulting in the each embedding table's cache staying "full" for a large portion of the lookup operation. When full, the extra level of indirection for the embedding tables is amortized by the large number of worker threads, providing a performance benefit over all PM when an accessed vector is in DRAM with little overhead when it is not. This effect is magnified with the Zipf distribution which yields a very high DRAM hit rate with only a modest cache size.

Discussion There are several regimes where this approach of fine-grained heterogeneous memory management can be effective. When the hit rate into the managed DRAM cache is sufficiently high (in the case of the Zipf index distribution) and the feature size is large enough to amortize the overhead of adding



(a) Featuresize 16 - Uniform

(b) Featuresize 16 - Zipf ($\alpha = 1$)

(c) Featuresize 256 - Uniform

(d) Featuresize 256 - Zipf ($\alpha = 1$)

Fig. 6: *Comparison of* CachedEmbeddings *with standard embedding tables located in DRAM or PM for* reducing *lookups with 40 Runs were conducted with 80 embedding tables and 28 worker threads using the preallocation strategy.*

indirection to vector access, then *CachedEmbeddings* can outperform all PM with a relatively small amount of DRAM. Even in cases where the hit rate is not particularly high (the case of the uniform index distribution), *CachedEmbeddings* can still achieve a level of performance between all DRAM and all PM provided the cache becomes full and the amount extra work involved on each access decreases. At this operating point, each vector access just adds a level of indirection, sometimes hitting in DRAM and sometimes hitting in PM. Those accesses to DRAM are accelerated while those to PM have little penalty over the all PM case.

This suggests another use strategy for *CachedEmbeddings* called the *static* approach. If the input distribution is known to have little locality *or* if hot entries in the distribution are known *a priori*, then an appropriate subset of the table can be preemptively moved to DRAM (using `access_and_cache`) until the table's cache is full. At this point, further accesses will only fetch and not move feature vectors. This approach will not respond dynamically to changes in the input distribution, but as we pointed out, may be appropriate is some situations.

5 DLRM Implementation Methodology

We implemented the DLRM model in Julia[4], and to verify our model performance, we compared our DLRM implementation Intel's optimized PyTorch [12]

[4] https://github.com/darchr/DLRM.jl

	Small Mode	**Large Model**
Featuresize	16	128
Num Embeddings Tables	26	26
Embedding Table Sizes	min = 3, max = $8.9e6$, $\mu \approx 1.2e6$, $\sigma = 2.6e6$	
Bottom MLP	512-256-64-16	512-256-128
Top MLP	512-256-1	1024-1024-512-256-1
Batchsize	8192	32768

Table 2: Model hyperparameters used for DLRM PyTorch comparison.

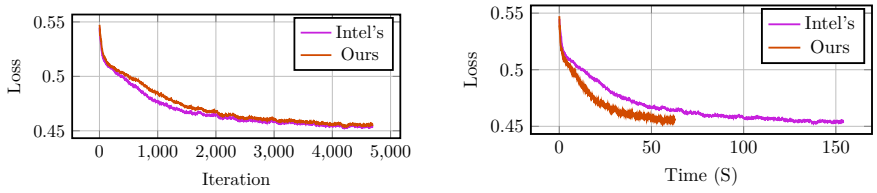
submission to MLPerf [15]. This reference model using custom PyTorch extensions to enable BFloat16 for high performance dense network computations. We were able to acquire temporary access to an Intel Cooperlake server, a generation equipped with vector instructions for BFloat16 based dot products. Since our implementation is build on top of oneDNN (which supports the BFloat16 datatype), we incorporated the BFloat16 data type into our model as well.

We used two models for comparison, a small model used as Facebook's official DLRM sample model and the model used in MLPerf 2019 training [15]. The hyper parameters for these tables is shown in Table 2. The optimized PyTorch implementation used *split SGD* [12] for their BFloat16 weights. With this optimizer, MLP and embedding table weights are kept in BFloat16, and each weight array is associated with a similar sized array filled with 16-bit integers. During the weight update phase of training, these BFloat16 variables are concatenated with their respective 16-bit integer in their sibling array to create a full 32-bit float. The gradient update is applied to this 32-bit value, which is the decomposed back into a BFloat16 and 16-bit "mantissa". Using this strategy, the authors keep a full 32 bits of precision for training while using 16 bits of precision for inference. Importantly, this technique *does not* decrease the memory requirement of the embedding tables. Consequently, we implement the split SGD trick for the MLP layers of our implementation, but keep our embedding tables in full Float32.

Training data came from the Kaggle Display Advertising Challenge dataset. Both small and large models were run for a single epoch of training on the dataset, iterating over the data in the same order. Further, both our model and the PyTorch model began with the same initial weights.

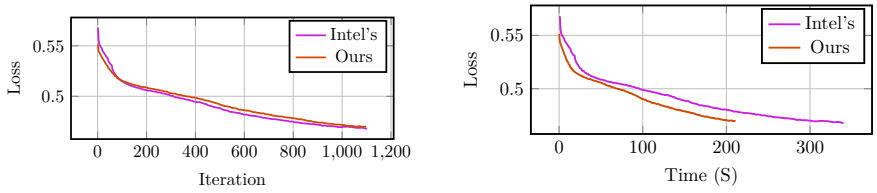
Figure 7 shows the loss progression of our model and the optimized PyTorch model for the small and large networks. Figures 7a and 7c show loss as a function of iteration number while Figures 7b and 7d show loss as a function of time.

We found that our model has slightly higher (worse) loss per iteration, implying our treatment of BFloat16 is not quite as precise as the PyTorch. However, our model has a significant less in loss over time because each iteration is processed much more quickly. When comparing end-to-end performance for training DLRM, our Julia model is slightly *faster* than the optimized PyTorch demon-



(a) Small model, training loss per iteration.

(b) Small model, training loss over time.



(c) Large model, training loss per iteration.

(d) Large model, training loss over time.

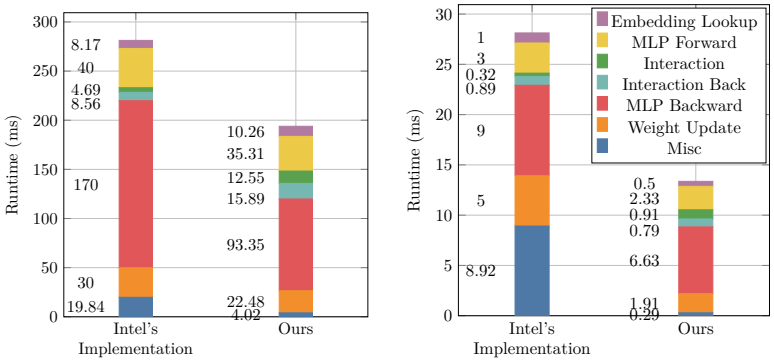
Fig. 7: Convergence comparison between the PyTorch optimized DLRM and ours. Our model has a slightly *higher* loss per iteration, but *lower* loss per wall clock time.

strating that we have a high-performant implementation of DLRM to investigate the impacts of different embedding table lookup algorithms.

Figure 8 shows the time breakdown of each iteration for both implementations and models. Our performance benefit comes from three major areas. First, our MLP backward pass is much faster. This is because we are using an up to date version of oneDNN to compute our backward pass kernels while the PyTorch model at the time was using libxsmm. It should be noted that Intel's extensions for PyTorch have since switched to using oneDNN. Second, our implementation has a faster embedding table and weight update through our parallel embedding table update and parallel weight update strategies. Note that even though the wall-clock time for the large network embedding lookup is slightly larger than PyTorch, we're moving twice the amount of data because our tables were kept in `Float32` while PyTorch used `BFloat16`. Finally, our implementation has less miscellaneous overhead, a factor especially apparent for the small network where PyTorch.

6 End-to-End DLRM Performance

In this section, we investigate the performance of *CachedEmbeddings* for full DLRM training. We investigate several different management schemes built on top of *CachedEmbeddings* and compare their performance with Intel's built-in 2LM hardware managed DRAM cache.



(a) Large Network (b) Small Network

Fig. 8: *Timing breakdown of key layers in our DLRM comparison.*

Policies We implemented three simple policies on top of *CachedEmbeddings*. The *simple* policy leaves all embedding vectors in PM, using DRAM to store the results of an embedding table lookup and intermediate data for the dense computations. This policy uses a simple embedding table without the level of indirection required for a CachedEmbedding table. The *static* policy allocates a specified amount of memory in DRAM as cache pages, fills these cache pages with random rows, then disables all dynamic row caching. At run time, a row access will either be serviced from DRAM (if one of the rows that was cached ahead of time) or from PM. The *dynamic* policy involves dynamically moves feature vectors into cache pages in DRAM. During lookup of a particular row, the current thread checks if the accessed row is cached and if so directly returns a pointer. If the row is not cached, the thread attempts to dynamically cache the row using the mechanism described above before returning the pointer. If the row fails to obtain ownership of the row, a pointer to the base data is used.

Over time, the *dynamic* policy will increase the footprint of the cache pages as more rows are moved into DRAM. In order to compare fairly with *memory mode* (which has access to all of DRAM), we need a per-table cache size small enough to fit in DRAM along side all memory used by the dense computations but large enough to achieve high utilization of the available DRAM. Thus, we set a cache size limit of 2 GiB for each table for a total memory footprint of 128 GiB across the ensemble. Cache pages are sized to be a fraction of this limit and when the limit is reached, the oldest cache page is cleaned up.

If the sparse input distributions are known, then policies can be updated on a per-table basis, (e.g., changing the amount of cache allowed for a table).

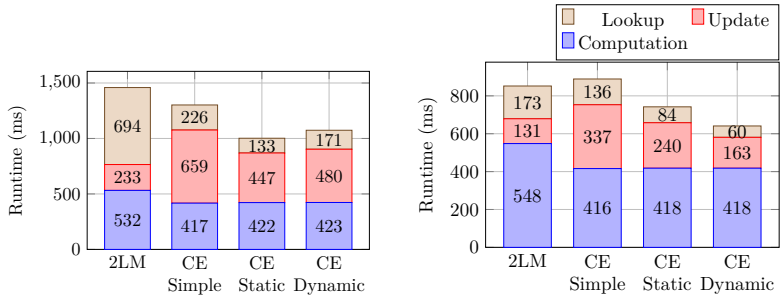
Methodology To test *CachedEmbeddings*, we used a very large DLRM with the hyper parameters shown in Table 3. This model has large and deep MLPs and a memory footprint of around 393 GB for its embedding tables. For this large model, both embedding table operations and dense computations take a sig-

Parameter	Value	Parameter	Value
Number of Tables	64	Rows per Table	6000000
Featuresize	256	Lookups per Output	100
Bottom MLP Length	8	Bottom MLP Width	2048
Top MLP Length	16	Top MLP Width	4096
Batchsize	512		

Table 3: Parameters for the large DLRM model used for benchmarking.

nificant fraction of overall training iteration time. Models with smaller dense networks will be more bottlenecked on embedding table operations, and models with fewer tables or with fewer lookups per output will be more compute bound.

The input distributions for embedding tables used in industry are proprietary, though literature suggest that there is at least some temporal locality. In this work, we chose to select two extremes. First, we use a uniform random input distribution for all tables. This is nearly the worst case for caching as there is limited reuse. Second, we use a Zipf [18] distribution with $\alpha = 1$ for each table, scrambling the input for each table using a maximum length LFSR starting at a random phase. This distribution has significant temporal locality. Dense inputs were generated using a normal distribution.



(a) Uniform.

(b) Zipf ($\alpha = 1.0$)

Fig. 9: *Performance with different sparse input distributions. Operations "Lookup" and "Update" refer to embedding table lookup and update respectively. All other operations are grouped into "Computation". Abbreviation "CE" stands for "CachedEmbeddings" and "2LM" stands for Intel's default hardware cache.*

Results The results for our large DLRM model are shown in Figure 9. Figure 9a shows performance when a uniform distribution is used to drive sparse accesses while Figure 9b demonstrates the same model for the Zipf distribution. The *baseline* that we compare to is "2LM" or using the DRAM as a hardware-based cache for the PM. Since the embedding table size greatly exceeds the DRAM size, only a small part cache be cached in DRAM at any time, and during training all of these entries will be updated and must be written back to PM when new entires are moved into the DRAM (i.e., it is a writeback cache).

These writebacks mostly occur during the lookup which is why that portion of the bar in Figure 9a is dominate. In the case with more locality (Figure 9b) the writebacks mostly occur during the MLP computation. By explicitly managing the memory movement in software, we avoid these hardware cache actions.

For the *CachedEmbeddings* runs, the performance of the dense layers is nearly the same. This is expected since now all dense computations are performed with memory in DRAM. The *simple* case is capable of achieving nearly the whole bandwidth of the PM devices. However, since embedding table updates must be done directly into PM, we see a performance degradation due to the low PM write bandwidth. The *static* policy performs the best. In this mode, embedding table lookup and update operations are serviced from both DRAM and PM. Thus, there is a performance benefit if for accessing rows in DRAM over the *simple* policy without a performance loss if the vector is in PM. The *dynamic* policy is able to perform a little better than the *simple* one because all embedding table updates go to DRAM. However, it is slower then *static* for embedding table lookups because the eager caching of embedding table vectors incurring more DRAM write bandwidth, competing with PM reads. Further more, *dynamic* incurs a slightly higher update penalty due to cache management (writing back dirty rows from old cache pages).

When switching from a uniform distribution (low reuse) to a Zipf distribution ($\alpha = 1$, high reuse), we observe speedups in embedding table and lookup performance across the board. Several factors are at play here. First, with this level of reuse, CPU caches become effective, reducing overall memory traffic. The embedding table update sees further performance increases due to our gradient aggregation strategy where the entire gradient for each embedding table vector is accumulated before applying the optimizer. With higher reuse, there are fewer indices per lookup and lower write traffic to PM.

Finally, we can see the effect of 2LM and *CachedEmbeddings* based caching mechanisms. The lookup performance of 2LM increases by 4× as the DRAM cache stops experiencing such a high miss rate. Further, the performance of *dynamic* improves by 2.85× compared to with the uniform distribution, surpassing the static strategy since it is able to correctly cache the hot vectors in DRAM. Indeed, we observe that there is a slight performance regression of *simple* when compared to 2LM as there is enough locality in the accessed vectors to overcome some of the issues associated with the hardware managed DRAM cache.

We again see the benefit of adding knowledge of program behavior to the memory management policy. When the sparse input distribution is uniform, our cache is too small to have a high enough hit rate to offset the overhead of moving vectors into the cache. In this case, a static partition of the data structures results in better utilization of the multiple levels of memory. However, when there *is* enough temporal locality in the input distribution for caching to be effective, fine grained memory management is exactly what we need. Tailoring of policy to the specifics of hardware and runtime situation is essential for performance.

7 Conclusions and Future Work

In this work, we presented the design space exploration of implementing multiple large and sparse embedding table operations on a heterogeneous memory platform using a new data structure called *CachedEmbeddings*. The main technique presented in this paper works best at larger feature sizes where the effort required to maintain the embedding table is out-weighed by the cost of the embedding table operation itself. Nevertheless, the existence of a caching mechanism for embedding table entries allows for custom policies to be implemented, tailored to the observed distribution in embedding table accesses.

Large and sparse embedding tables are not unique to DLRM workloads but also are useful in other ML workloads such as Transformers [10]. As a software-only technique, *CachedEmbeddings* can be adapted to future disaggregated memory systems, for instance CXL-based fabric-attached memory platforms.

References

1. Adnan, M., Maboud, Y.E., Mahajan, D., Nair, P.J.: Accelerating recommendation system training by leveraging popular choices. Proc. VLDB Endow. **15**(1), 127–140 (sep 2021), https://doi.org/10.14778/3485450.3485462
2. Ardestani, E.K., et al.: Supporting massive DLRM inference through software defined memory. CoRR **abs/2110.11489** (2021), https://arxiv.org/abs/2110.11489
3. Dhulipala, L., McGuffey, C., Kang, H., Gu, Y., Blelloch, G.E., Gibbons, P.B., Shun, J.: Sage: Parallel semi-asymmetric graph algorithms for nvrams. Proc. VLDB Endow. **13**(9), 1598–1613 (May 2020), https://doi.org/10.14778/3397230.3397251
4. Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K.M., Petersen, C., Cidon, A., Katti, S.: Reducing DRAM footprint with NVM in facebook. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018. pp. 42:1–42:13 (2018), https://doi.org/10.1145/3190508.3190524
5. Eisenman, A., Naumov, M., Gardner, D., Smelyanskiy, M., Pupyrev, S., Hazelwood, K.M., Cidon, A., Katti, S.: Bandana: Using non-volatile memory for storing deep learning models. CoRR **abs/1811.05922** (2018), http://arxiv.org/abs/1811.05922
6. Fang, J., Zhang, G., Han, J., Li, S., Bian, Z., Li, Y., Liu, J., You, Y.: A frequency-aware software cache for large recommendation system embeddings (2022), https://arxiv.org/abs/2208.05321
7. Gupta, U., et al.: The architectural implications of facebook's dnn-based personalized recommendation. CoRR **abs/1906.03109** (2019), https://arxiv.org/abs/1906.03109
8. Gupta, U., et al.: Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 982–995. IEEE (2020)
9. Hildebrand, M., Angeles, J.T., Lowe-Power, J., Akella, V.: A case against hardware managed dram caches for nvram based systems. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 194–204 (2021)

10. Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., Hoefler, T.: Data movement is all you need: A case study on optimizing transformers. Proceedings of Machine Learning and Systems **3**, 711–732 (2021)
11. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., Zhao, J., Swanson, S.: Basic performance measurements of the intel optane DC persistent memory module. CoRR **abs/1903.05714** (2019), http://arxiv.org/abs/1903.05714
12. Kalamkar, D., Georganas, E., Srinivasan, S., Chen, J., Shiryaev, M., Heinecke, A.: Optimizing deep learning recommender systems training on cpu cluster architectures. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15. IEEE (2020)
13. Kim, J., Choe, W., Ahn, J.: Exploring the design space of page management for multi-tiered memory systems. In: 2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21). pp. 715–728 (2021)
14. Lin, Z., et al.: Building a performance model for deep learning recommendation model training on gpus (2022), https://arxiv.org/abs/2201.07821
15. Mattson, P., et al.: Mlperf training benchmark (2019)
16. Mudigere, D., et al.: Software-hardware co-design for fast and scalable training of deep learning recommendation models. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 993–1011. ISCA '22, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3470496.3533727
17. Naumov, M., et al.: Deep learning recommendation model for personalization and recommendation systems. CoRR **abs/1906.00091** (2019), http://arxiv.org/abs/1906.00091
18. Powers, D.M.W.: Applications and explanations of Zipf's law. In: New Methods in Language Processing and Computational Natural Language Learning (1998)
19. Sethi, G., Acun, B., Agarwal, N., Kozyrakis, C., Trippel, C., Wu, C.J.: Recshard: Statistical feature-based memory optimization for industry-scale neural recommendation. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 344–358. ASPLOS '22, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3503222.3507777
20. Shanbhag, A., Tatbul, N., Cohen, D., Madden, S.: Large-scale in-memory analytics on intel® optane™ dc persistent memory. In: Proceedings of the 16th International Workshop on Data Management on New Hardware. DaMoN '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3399666.3399933
21. Xie, M., Lu, Y., Lin, J., Wang, Q., Gao, J., Ren, K., Shu, J.: Fleche: An efficient gpu embedding cache for personalized recommendations. In: Proceedings of the Seventeenth European Conference on Computer Systems. p. 402–416. EuroSys '22, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3492321.3519554
22. Yan, Z., Lustig, D., Nellans, D., Bhattacharjee, A.: Nimble page management for tiered memory systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019. pp. 331–345 (2019), https://doi.org/10.1145/3297858.3304024