Towards Redundancy-Free Recommendation Model Training via Reusable-aware Near-Memory Processing

Haifeng Liu[†], Long Zheng[†*], Yu Huang[†‡], Haoyan Huang[†], Xiaofei Liao[†], and Hai Jin[†]
[†]National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan, 430074, China
[‡]Zhejiang Lab, Hangzhou, 311121, China
{hfliu,longzh,yuh,vastrockh,xfliao,hjin}@hust.edu.cn

ABSTRACT

The memory-intensive embedding layer in recommendation model continues to be the performance bottleneck. While prior works have attempted to improve the embedding layer performance by exploiting the data locality to cache the frequently accessed embedding vectors and their partial sums. However, these solutions rely on the static cache, which is inapplicable in the embedding training scenario where the embedding vectors are updated frequently. To this end, this paper proposes ReFree, a redundancy-free *near-memory processing* (NMP) solution for recommendation model training. Specifically, ReFree identifies the reusable data in real-time for both embedding layer forward and backward stages and leverages a lightweight NMP architecture to enable redundancy-free near-memory acceleration of the entire embedding training process. Evaluation results on real-world datasets show that ReFree outperforms the state-of-the-art solutions by 10.9× and reduces 5.3× energy consumption on average.

1 INTRODUCTION

Deep learning-based recommendation models (DLRMs) [19] are widely deployed in the Internet industry to offer a better user experience. Specifically, DLRMs employ the embedding layer to capture the sparse features. The embedding layer usually consists of multiple embedding tables and performs sparse gathering and reduction operations. It exhibits large storage requirements, low compute intensity, and irregular memory access patterns, making it a typical memory-bounded application and the primary performance bottleneck of DLRM inference and training [1, 7].

In production-scale recommendation models, the embedding layer exhibits both gather and reduction locality [10]. The gather locality indicates that a small part of embedding vectors are frequently accessed, while most of the others are hardly visited, causing a highly skewed access distribution. The reduction locality indicates that the frequently accessed embedding vectors are more likely to be co-accessed simultaneously. However, due to the irregular memory access pattern over the large embedding tables, fully exploiting these two types of locality becomes challenging, and two types of redundancy arise in both inference and training, i.e., memory access (multiple memory accesses for the same vector) and computation redundancy (multiple reduction operations for the co-accessed vectors), more details are in §3.1.

Recently, several solutions [2, 10, 14, 22] have attempted to accelerate the embedding layer by exploiting these two types of locality. They identify and prepare the reusable data through a preprocessing stage and reuse them during the embedding query operation. Specifically, in the preprocessing stage, they first analyze the previous user-item access trace, identify the highly accessed items (gather locality) and popular co-accessed item combinations (reduction locality), and then cache or store the highly accessed vectors and partial sums, which are reused during the embedding operation to reduce memory accesses and computations. However, these solutions expose the following limitations. First, the embedding access pattern dynamically changes over time in the production environment, which defeats existing solutions that rely on a predefined cache of frequently accessed embedding vectors and their partial sums. Besides, prior works all focus on the DLRM inference stage and are inapplicable to the training process in which the embedding vectors are frequently updated (more details are in §3.2).

Prior works [4, 12, 15, 16, 20] employ the emerging *near-memory processing* (NMP) technology to accelerate the memory-intensive embedding layer. They integrate processing units inside the DIMM devices and offload the low compute-intensive embedding operations into the memory, which reduces the data movement and increases the internal memory bandwidth. However, the use of NMP does not reduce any memory access or computation, leaving much room for performance improvement. Some solutions [4, 16] are engaged in leveraging the gather locality to reduce the number of memory access, RecNMP [16] adds a small cache inside the NMP PE to cache the frequently accessed vectors, FAFNR [4] proposes a reduction tree among different ranks to reuse the accessed vectors and improve the reduction efficiency. However, these solutions only reduce memory access in the embedding inference stage, neglecting computation redundancy and are also inapplicable to the training process. Therefore, an efficient solution is urgently needed to achieve redundancy-free embedding layer training.

In this paper, we propose ReFree, a redundancy-free NMP solution for embedding training. The design goal of ReFree is to eliminate memory access and computation redundancy in both forward and backward stages of the embedding layer training. To this

*Long Zheng is the corresponding author. This work is supported by the National Key Research and Development Program of China under Grant No. 2023YFB4503400 and the National Natural Science Foundation of China under Grant No. 62322205 and 62072195.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656511

end, we first unify both embedding forward and backward stages with the *gather and reduce* (GnR) operation, allowing us to develop a generic solution for the entire embedding training process. Next, a data reusable-aware method is proposed to identify the reusable data during the training process in real-time and reuse both hot data and partial sums to eliminate memory access and computation redundancy. A generic NMP architecture is designed to implement this solution and enables redundancy-free near-memory acceleration for the entire embedding layer training process.

This paper makes the following contributions:

- We conduct a detailed analysis of the embedding data locality and reveal the redundancy in both forward and backpropagation of the DLRM embedding layer training process.
- We propose a reusable-aware solution to identify and reuse both the hot data and the partial sums in real-time to eliminate memory access and computation redundancy.
- We architect a reusable-aware NMP accelerator, ReFree, to accelerate the whole embedding training process with redundancy-free near-memory acceleration.
- Evaluation results show that ReFree significantly reduces redundancy and outperforms the state-of-the-art solutions.

2 BACKGROUND

2.1 Recommendation Model Training

Deep learning-based recommendation models (DLRMs) [19] are widely deployed in the Internet industry. The sparse embedding layer is the primary performance bottleneck of DLRM execution due to its high memory bandwidth requirements [1, 7]. The embedding layer typically comprises multiple embedding tables, each corresponding to a categorical feature. These tables can contain millions to billions of embedding vectors, each representing a unique item. The embedding layers can consume hundreds of GBs to even TBs of memory, occupying most of the model size [18].

Figure 1 shows the embedding layer training of the forward pass and backpropagation pass. The forward pass is equivalent to the DLRM inference process. As shown in Figure 1(a), an array of index IDs is provided to lookup the embedding table and access corresponding embedding rows, which are then reduced through element-wise operation into one embedding vector, which is then sent to the *fully connected* (FC) layer as input to make predictions, e.g., the click-through rate. DLRMs are usually trained in batches to improve the throughput, e.g., Figure 1 shows a batch size of 4. The backpropagation pass then generates the gradients of each query to update the embedding tables. As shown in Figure 1(b), each embedding vector that is gathered during the forward pass will be updated by the corresponding gradient. If one embedding vector is gathered multiple times in one batch, then multiple gradients will be used to update it in the backpropagation pass.

2.2 NMP for Embedding Layer Acceleration

As the embedding layer exhibits massive irregular memory access and low compute intensity, recent works [4, 12, 15, 16, 20] adopt the *near-memory processing* (NMP) technique to accelerate it by reducing the data movements and leveraging the memory-level parallelism. These works add processing units and offload the embedding operations into the memory, reducing the data movements between the memory and host. Besides, higher internal memory bandwidth is present by leveraging the memory-level parallelism, improving the embedding layer performance. However, these early efforts are applied only to improve the performance of the DLRM inference and are inapplicable to the embedding layer backward process. This paper explores the characteristics of the embedding forward and backward stages and targets accelerating the *whole* embedding layer training process with near-memory processing.

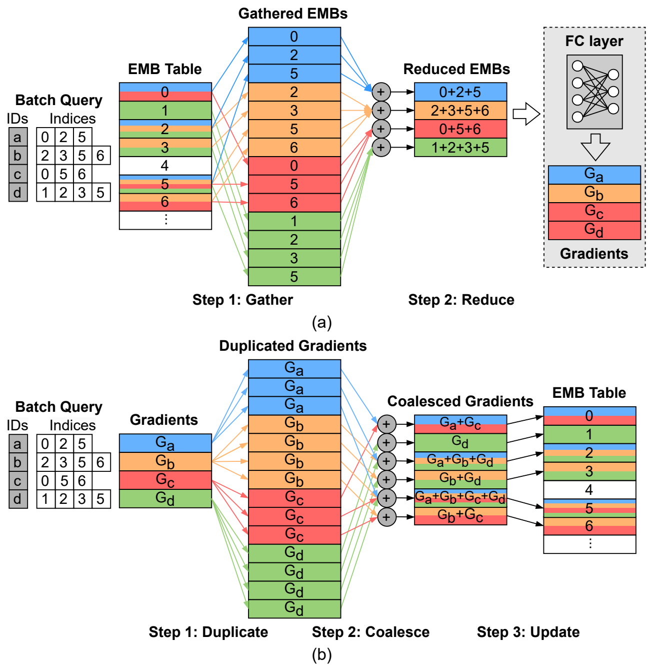


Figure 1: DLRM embedding layer training process of (a) forward stage and (b) backward stage

3 MOTIVATION

3.1 Redundancy in DLRM Embedding Layer

In production-scale recommendation models, the embedding layer exhibits *gather* and *reduction locality* [10]. The *gather locality* indicates that a small part of embedding vectors are frequently accessed, while most of the others are hardly visited, causing a highly skewed access distribution. This also corresponds to realistic scenarios, e.g., a few popular videos take the majority of watches and are more likely to be visited by more people. The *reduction locality* points out that the frequently accessed embedding vectors are more likely to be visited simultaneously, corresponding to the realistic scenarios that the popular items are more likely to be visited simultaneously by the same people. Two types of redundancy exist in the embedding layer due to these two types of locality.

Memory Access Redundancy. The memory access redundancy derives from the *gather locality*, i.e., if multiple queries contain the same embedding vector in the forward pass, this embedding vector will be accessed multiple times from memory. While in the backward pass, the memory access redundancy exists in the gradients duplication step that each gradient is duplicated as many times as the number of gathered embedding vectors to update them. For example, in Figure 1 (a), embedding vector 5 is gathered in

Table 1: Summary of Existing Solutions

Solution	Redundancy		Scenario	
	Mem	Com	FWD	BWP
FAE [2]	✓	✗	✓	✗
RecNMP [16]	✓	✗	✓	✗
FAFNIR [4]	✓	✗	✓	✗
SPACE [10]	✓	✓	✓	✗
MERCI [14]	✓	✓	✓	✗
GRACE [22]	✓	✓	✓	✗
Ours	✓	✓	✓	✓

all four queries, and it will be accessed four times in this batch, generating three redundant memory accesses. While in Figure 1 (b), gradient G_a is duplicated three times as query a contains three embedding vectors, also leading to two redundant memory accesses.

Computation Redundancy. The computation redundancy originates from the *reduction locality*, i.e., if multiple embedding vectors are co-occurrences in multiple queries in the forward pass, these vector combinations will be reduced multiple times. For example, in Figure 1, embedding vectors 2 and 5 co-occur in query a, b, and d, and are reduced three times in this batch, resulting in two redundant computations. Meanwhile, gradients G_a, G_b, and G_d are coalesced twice in the backward pass to update the vector 2 and 5, also yielding two redundant computations.

These two types of redundancy commonly occur as these two types of locality are widespread in realistic scenarios. Besides, higher locality leads to increased redundancy, highlighting the importance of this work in achieving redundancy-free embedding layer training.

3.2 Limitation of Prior Works

Table 1 summarizes existing studies that are proposed to reduce the redundancy in embedding layer. FAE [2] analyzes the item access frequency distribution and stores a portion of the highly accessed embedding vectors in the high-bandwidth GPU memory. RecNMP [16] is a rank-level NMP solution for embedding layer acceleration, and it adds a small cache inside the NMP PE to cache the hot vectors. These two solutions both use static cache to reduce partial memory accesses. FAFNIR [4] proposes a reduction tree inside the memory, and it reads only the unique vectors in a batch and reuses them through the reduction tree as many times as required without any caching mechanism. However, these solutions all have no contribution to reducing the computation redundancy.

SPACE pre-processes the user-item access trace and stores the highly accessed vectors and the partial sums of any two of them in the HBM. To this end, SPACE reduces both memory access and computation of the hot vectors by directly accessing them from the HBM. MERCI [14] does not limit the partial sums of two hot vectors. Instead, it first analyzes the popular co-accessed vectors and merges them into clusters, then stores all partial sums in these clusters in the main memory. Similarly, GRACE [22] proposes a graph-based method to find the popular item combinations, and stores their partial sums in a software-managed cache.

Despite their efforts, previous works still face two limitations.

(a) **Dynamic Query Unawareness.** Existing solutions rely on the predefined cache of the frequently accessed embedding vectors and their partial sums generated by costly analysis of the previous user traces. However, the embedding access pattern dynamically changes over time in the production environment, e.g., the frequently accessed vectors may get cold, and vice versa. As a result, when the predefined and stored hot vectors and their partial sums become less popular, they have no benefits to reduce memory access or computation, and another costly preprocessing is requested once the access pattern changes. Besides, storing the partial sums requires additional storage overhead that is even larger than the embedding data size, which is expensive and unacceptable.

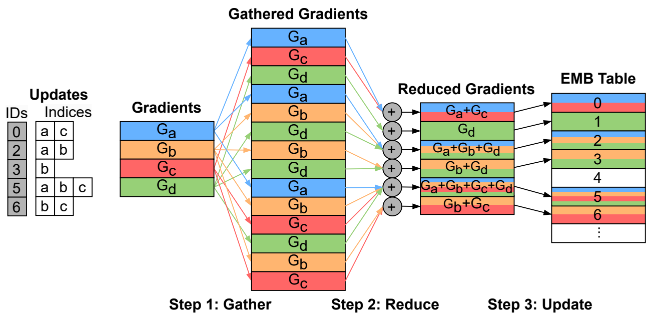


Figure 2: DLRM embedding backpropagation with gather-reduce (GnR) operation, unified with the forward pass

(b) **Embedding Backpropagation Unsupported.** Existing solutions require offline preparation of the hot embedding vectors and their partial sums and reuse them during the online querying process. However, the embedding access pattern dynamically changes in the training process, and the embedding data are updated at each training iteration. Therefore, the cached data of the previous iteration cannot be used in the next training iteration, making the existing solutions fail to support the training process. Besides, in the backward pass, the gradients are generated in real-time during each batch, making it impossible to analyze and generate the hot gradients and their partial sum in advance.

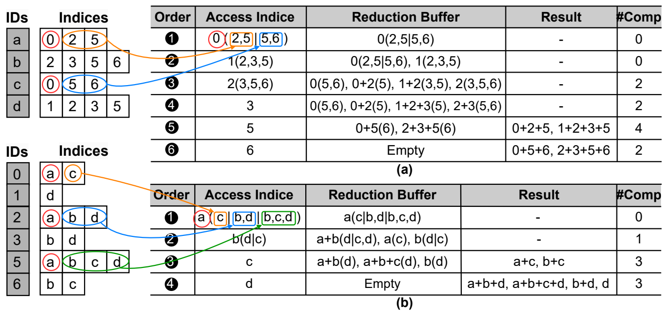
3.3 Our Goal: Redundancy-Free DLRM Training

Considering the above limitations, none of the existing solutions work for the embedding training. The goal of this paper is to build an efficient and low-overhead solution to eliminate memory access and computation redundancy during the embedding training process. In particular, our solution must meet the following expectations:

(a) **Real-time Reusability Awareness.** Existing works identify the reusable data by offline preprocessing, which does not work in the embedding training scenario. We argue that our solution must be able to recognize the reusable data in real-time, i.e., immediately identify the redundancy of every input training batch.

(b) **No Additional Storage Overhead.** Unlike the previous works that store hot vectors and partial sums in advance, which incur costly storage overheads in main memory or GPU memory, our solution that identifies and eliminates redundancy in real-time must be able to induce no additional storage overhead.

(c) **Applicable for Both Forward and Backward Stages.** In the training process, both the forward and backward stages contain memory access and computation redundancy, as we explained in § 3.1. Our solution must be able to identify and eliminate redundancy in both forward pass and backpropagation to enable redundancy-free DLRM embedding layer training.



IDs	Indices			
a	0	2	5	
b	2	3	5	6
c	0	5	6	
d	1	2	3	5

Order	Access Indice	Reduction Buffer	Result	#Comp
❶	0(2,5|5,6)	0(2,5|5,6)	-	0
❷	1(2,3,5)	0(2,5|5,6), 1(2,3,5)	-	0
❸	2(3,5,6)	0(5,6), 0+2(5), 1+2(3,5), 2(3,5,6)	-	2
❹	3	0(5,6), 0+2(5), 1+2+3(5), 2+3(5,6)	-	2
❺	5	0+5(6), 2+3+5(6)	0+2+5, 1+2+3+5	4
❻	6	Empty	0+5+6, 2+3+5+6	2

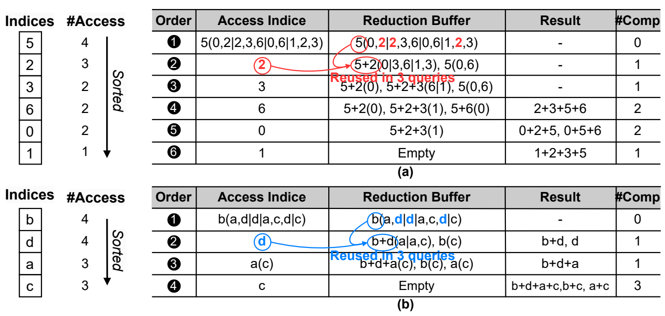
(a)

IDs	Indices			
0	a	c		
1	d			
2	a	b	d	
3	b	d		
5	a	b	c	d
6	b	c		

Order	Access Indice	Reduction Buffer	Result	#Comp
❶	a(c|b,d|b,c,d)	a(c|b,d|b,c,d)	-	0
❷	b(d|c)	a+b(d|c,d), a(c), b(d|c)	-	1
❸	c	a+b(d), a+b+c(d), b(d)	a+c, b+c	3
❹	d	Empty	a+b+d, a+b+c+d, b+d, d	3

(b)

Figure 3: Eliminating memory access redundancy in (a) forward (reduce from 14 to 6) and (b) backward (reduce from 14 to 4) stages by accessing and reusing the unique data



Indices	#Access
5	4
2	3
3	2
6	2
0	2
1	1

Order	Access Indice	Reduction Buffer	Result	#Comp
❶	5(0,2|2,3,6|0,6|1,2,3)	5(0,2|2,3,6|0,6|1,2,3)	-	0
❷	2	5+2(0|3,6|1,3), 5(0,6)	-	1
❸	3	5+2(0), 5+2+3(6|1), 5(0,6)	-	1
❹	6	5+2(0), 5+2+3(1), 5+6(0)	2+3+5+6	2
❺	0	5+2+3(1)	0+2+5, 0+5+6	2
❻	1	Empty	1+2+3+5	1

(a)

Indices	#Access
b	4
d	4
a	3
c	3

Order	Access Indice	Reduction Buffer	Result	#Comp
❶	b(a,d|d|a,c,d|c)	b(a,d|d|a,c,d|c)	-	0
❷	d	b+d(a|a,c), b(c)	b+d, d	1
❸	a(c)	b+d+a(c), b(c), a(c)	b+d+a	1
❹	c	Empty	b+d+a+c,b+c, a+c	3

(b)

Figure 4: Eliminating computation redundancy in (a) forward (reduce from 10 to 7) and (b) backward (reduce from 7 to 5) stages by prioritizing access hot data and reusing partial sums

4 REFREE DESIGN

This section details the ReFree designs, including the redundancy-free embedding training approach and the NMP architecture.

4.1 Redundancy-free Embedding Training

Unified Forward and Backward Process. The operations in the forward and backward stages are different, i.e., the embedding layer performs the gather-reduce operation in the forward pass while performing a duplicate-coalesce-update operation in the backpropagation, as illustrated in Figure 1. Fortunately, motivated by [13], the gradients coalescing and the embedding reduction perform the same arithmetic operations, i.e., both element-wise operations. Therefore, if we treat the gradients in each batch as a table, the duplicate-coalesce operation is identical to the gather-reduce operation on the gradient table. Figure 2 illustrates the backpropagation in the form of a gather-reduce-update operation, in which the query IDs are the indices of the yet to be updated embedding vectors, and the query indices are the gradients that need to be collected to update the corresponding embedding vector. Thus, the backpropagation operates the same way as forward stage (only with an additional update step), allowing us to build a unified solution for the entire training process.

Memory Access Redundancy Elimination. The key idea of our solution to eliminate memory access redundancy is to analyze the data reusability first, then access only the *unique* embedding vectors and reuse them as many times as they request in one batch. In the scenario of DLRM training, we can analyze the unique embedding vectors of each training batch in advance and investigate their reusability as the accessed indices of all batches can be known before training. Figure 3 illustrates an example of eliminating the memory access redundancy by only accessing the unique data. We first fetch the unique indices (6 in forward and 4 in backpropagation in this example) and their request indices of each related query. For example, as vector 0 is requested in queries *a* (0,2,5) and *c* (0,5,6), the request indices for vector 0 are (2,5) and (5,6), recorded as 0 (2,5|5,6). The vector value and the request indices are all efficiently stored in the reduction buffer. Crucially, each time we access new data, we ensure its reuse in all related queries. This way, all queries are processed in the process of accessing the unique data.

Computation Redundancy Elimination. An intuitive idea to eliminate redundant computation is to reuse the partial sums of these recurring combinations. However, significant overhead arises from analyzing recurring combinations as the number of combinations increases exponentially with the number of vectors. Therefore, existing methods [10, 14, 22] analyze and store the partial sums of these recurring combinations through a costly preprocessing, which is inapplicable in the training scenario with continuous updates on embedding vectors.

Under the scenario of training, we make the key observation that the reusable combinations are more likely to occur between popular data. Therefore, *prioritizing access to popular data allows the early generation of partial sums between popular data and maximizes opportunities for intermediate data reuse.* Figure 4 illustrates an example of eliminating the computation redundancy by prioritizing access to the popular data and reusing their partial sums. We first sort the number of vector accesses and then access embedding vectors in order of their access frequency. For example, when accessing the two most popular vectors 5 and 2, their partial result, 5+2, is used in three queries, eliminating two computations. Note that the sorting overhead is negligible as it is linear time complexity, and the access order can be analyzed before training.

4.2 ReFree Architecture

Figure 5 depicts the ReFree overall architecture. To support near-memory embedding operations, we add an NMP core dedicated to each rank inside the DIMM buffer. This NMP PE simultaneously supports both embedding forward and backward operations. As shown in Figure 5(c), there are two operation modes of this NMP PE, i.e., i) GnR mode and ii) update mode. The GnR mode performs the gather and reduce operations in both forward and backward stages, while the update mode works for updating the embedding vectors with the reduced gradients. Note that because forward and backward stages are performed sequentially during DLRM training, these two operation modes run on the same hardware in a time-sharing manner, improving NMP hardware utilization.

In the forward stage, the embedding vector is accessed from the memory (❶), and its vector ID is compared with the requested indices in the reduction buffer (❷). The reduction buffer stores the partial sums and the requested indices of each query. Figure 5 (d) illustrates the reduction buffer state after accessed the vector 2 in Figure 4 (a). Each accessed embedding vector and the corresponding partial sums are sent to the *computation unit* (CU) for reduction (❸), and the reduced result is written back to the reduction buffer (❹). If the embedding vector is the first vector accessed in the query,

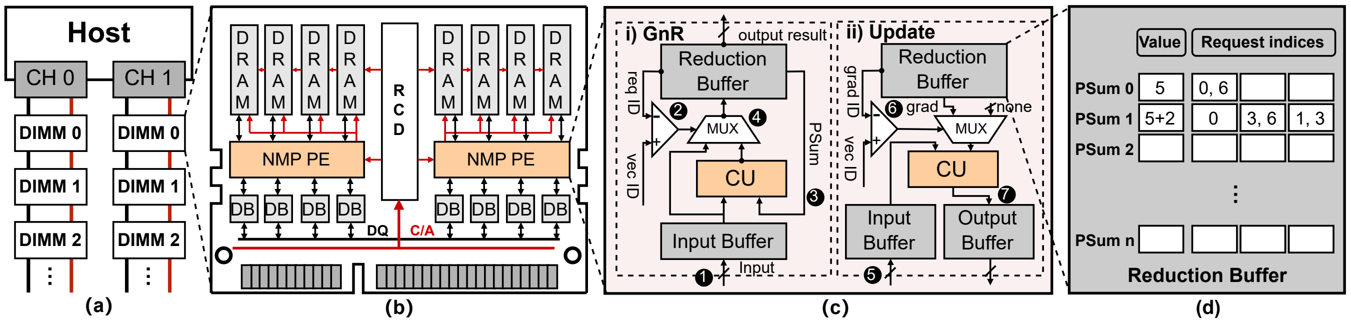


Figure 5: Detailed ReFree design of (a) overall architecture, (b) NMP-featured LRDIMM, (c) NMP PE, and (d) reduction buffer

Table 2: System Configurations

Memory Configuration
DDR5-4800 ×8 device, 4 Channels × 1 DIMM × 2 Ranks 64-entry RD/WR request queue, FR-FCFS scheduler
DRAM Timing Parameters
$tRCD$=40, tCL=40, tRP=40, $tRAS$=76, tRC=116, tBL=8, $tCCD_S$=8, $tCCD_L$=12, $tFAW$=32
ReFree NMP PE Specification
4KB Input/Output Buffer, 128KB Reduction Buffer, one 128 bit-width SIMD CU per PE, one PE per Rank

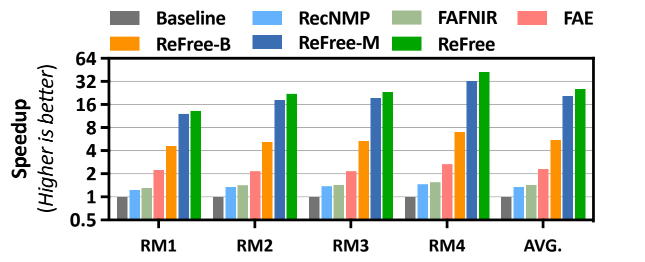


Figure 6: Normalized speedup of ReFree against the state-of-the-art solutions

it will be directly written to the reduction buffer. The embedding operation is finished when the request indices are empty.

The gradient GnR operation in the backpropagation is the same process as embedding GnR in the forward stage. When the gradients are all reduced, the NMP PE uses them to update the embedding vectors. The embedding vector is accessed from memory first (❺), and its vector ID is compared with the gradient indices in the reduction buffer to fetch its corresponding (reduced) gradient (❻). Then, the vector is updated with the gradient and written back to memory (❼). With this reusable-aware NMP architecture, ReFree reduces the memory access and computation redundancy in both embedding forward and backward stages.

5 EVALUATION

5.1 Experimental Setup

ReFree Setup. We implement a cycle-accurate simulation based on Ramulator [11] to evaluate the performance, energy consumption, and area overhead. Table 2 summarizes the ReFree configurations. Ramulator is modified and integrated with the rank NMP PEs to simulate the behavior of near-memory embedding operation. We synthesize the NMP PEs using Synopsys Design Compiler [21] with 32 nm technology to evaluate their timing, energy, and area. To emulate the end-to-end DLRM training performance, we export the per-batch training latency of the DNN layers from the NVIDIA A100 GPU and replay corresponding cycles. The energy consumption of the DRAM is taken from DRAMPower [17], and the energy consumption of the connection is obtained from CACTI-IO [8].

Datasets. We evaluate ReFree using four recommendation models, RM1-4, on four representative real-world datasets with diverse locality characteristics. RM1 is tested on Alibaba [3], which exhibits low locality. RM2 and RM3 are tested on Avazu [9] and Criteo Kaggle [6], respectively, both of which have medium locality. RM4 is tested on Criteo Terabyte [5] with a high locality. These four datasets (Alibaba, Avazu, Criteo Kaggle, and Criteo Terabyte) contain 5.1 million, 9.3 million, 33.8 million, and 266 million embedding rows, respectively, with embedding dimensions of 16 for the first three and 64 for the fourth. The default batch size used is 2048.

Baselines. We compare ReFree with the following state-of-the-art solutions: 1) `Baseline` [1]: a CPU-GPU system without any optimization. 2) `FAE` [2]: a CPU-GPU system with hot embedding vectors cached in the GPU memory. 3) `RecNMP` [16]: an NMP solution with hot embedding vectors cached in the NMP cache. 4) `FAFNIR` [4]: an NMP solution with a reduction tree architecture. Note that RecNMP and FAFNIR only accelerate the embedding forward without supporting the backpropagation stage. We also evaluate two variants of ReFree to evaluate the effectiveness of the two design points from our proposal: 5) `ReFree-B`: ReFree baseline without any redundancy elimination. 6) `ReFree-M`: ReFree with only memory access redundancy elimination. 7) `ReFree`: ReFree eliminates all memory access and computation redundancy.

5.2 Overall Results

Performance. Figure 6 shows the performance speedup of ReFree against the state-of-the-art solutions. Overall, ReFree outperforms the CPU-GPU baseline, RecNMP, FAFNIR, and FEA by 25.1×, 18.5×, 17.6×, and 10.9× on average, respectively. The performance speedup improves gradually as the dataset exhibits higher locality, primarily due to the fact that higher locality induces more redundancy in the embedding layer, which also means more data reusability opportunities are exposed to ReFree. Note that ReFree-B also achieves an average 5.5×, 4.1×, 3.8, and 2.4× speedup compared to the CPU-GPU baseline, RecNMP, FAFNIR, and FEA, respectively. This benefit arises from our unified NMP architecture for both embedding forward and backward stages, which reduces data movement (both embedding vectors and gradients) between the host and memory,

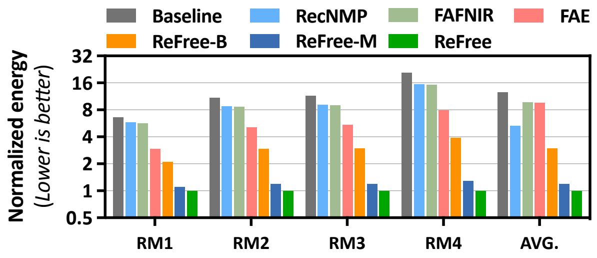


Figure 7: Normalized energy saving of ReFree against the state-of-the-art solutions

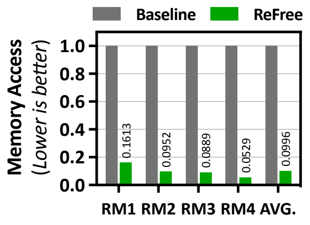


Figure 8: ReFree Memory access reduction

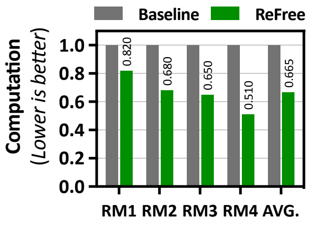


Figure 9: ReFree Computation reduction

significantly alleviating the performance bottleneck of the memory-intensive embedding layer. Combining with the redundancy elimination, ReFree further boosts the performance.

Energy Saving. Figure 7 shows the energy saving results. ReFree reduces an average of 12.5×, 9.8×, 9.6×, and 5.3× energy consumption compared to the CPU-GPU baseline, RecNMP, FAFNIR, and FEA, respectively. The reduction in energy consumption can be attributed to the shortened execution time and reduced data movements, resulting in lower execution and I/O energy consumption. Besides, the reduced memory access and computation naturally lead to lower energy consumption. Generally, the significant performance speedup ReFree achieved directly translates into energy-efficiency improvements.

5.3 Redundancy Reduction

Figure 8 and Figure 9 show the memory access and computation reduction of ReFree, respectively. ReFree reduces 9.65× memory access and 1.5× computation on average. Unlike the existing solutions, which mainly rely on the cache mechanism to reduce the memory access to the hot embedding vectors only, ReFree analyzes the reusable data in real-time, reducing the number of memory accesses to the unique embedding vectors in each batch and reducing the computation by reusing their partial sums. Thus, the higher the locality exhibits, the more memory access and computation can be reduced. Nonetheless, even under a low locality workload scenario, ReFree still reduces 5.5× and 1.2× memory access and computation, demonstrating its robustness.

5.4 Overhead Analysis

Power and Area. The only modification of ReFree is the added NMP PEs to the DIMM buffer chip. Each PE induces an additional area overhead of $0.26mm^2$ and a power overhead of $0.18W$. Considering a typical DIMM consumes over $13W$ and a buffer chip takes up $100mm^2$ [16], the induced area and energy overhead of ReFree is negligible even for one DIMM with four ranks, enabling seamless integration into existing commercial DIMMs.

6 CONCLUSION

In this paper, we propose ReFree, a redundancy-free NMP solution for DLRM training by eliminating both memory access and computation redundancy during the embedding layer training procedure. We first propose a generic solution to identify the reusable data in real-time for both forward and backward stages of the embedding layer training. Then, a lightweight NMP architecture is designed to enable redundancy-free near-memory acceleration of the entire embedding training process. Evaluation results show that ReFree significantly reduces memory access and computation redundancy, leading to 10.9× performance speedup and reducing 5.3× energy consumption compared to the state-of-the-art solutions.

REFERENCES

[1] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim M. Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *Proceedings of HPCA*.

[2] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. Accelerating Recommendation System Training by Leveraging Popular Choices. In *Proceedings of the VLDB Endowment*.

[3] Alibaba. 2023. User behavior data from taobao for recommendation. https://tianchi.aliyun.com/dataset/dataDetail?dataId=649&userId=1.

[4] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *Proceedings of HPCA*.

[5] CriteoLabs. 2013. Criteo AI Labs Ad Terabyte. https://labs.criteo.com/2013/12/download-terabyte-click-logs/.

[6] CriteoLabs. 2014. Criteo AI Labs Ad Kaggle. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset.

[7] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, and et al. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *Proceedings of HPCA*.

[8] Norman P. Jouppi, Andrew B. Kahng, Naveen Muralimanohar, and Vaishnav Srinivas. 2015. CACTI-IO: CACTI With OFF-Chip Power-Area-Timing Models. *VLSI* (2015).

[9] Kaggle. 2023. Avazu ads ctr. https://www.kaggle.com/c/avazu-ctr-prediction.

[10] Hongju Kal, Seokmin Lee, Gun Ko, and Won Woo Ro. 2021. SPACE: Locality-Aware Processing in Heterogeneous Memory for Personalized Recommendations. In *Proceedings of ISCA*.

[11] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* (2016).

[12] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of MICRO*.

[13] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2021. Tensor Casting: Co-Designing Algorithm-Architecture for Personalized Recommendation Training. In *Proceedings of HPCA*.

[14] 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 ASPLOS*.

[15] Haifeng Liu, Long Zheng, Yu Huang, Chaoqiang Liu, Xiangyu Ye, Jingrui Yuan, Xiaofei Liao, Hai Jin, and Jingling Xue. 2023. Accelerating Personalized Recommendation with Cross-level Near-Memory Processing. In *Proceedings of ISCA*.

[16] Ke Liu, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, and et al. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *Proceedings of ISCA*.

[17] Micron. 2017. Micron: System Power Calculator (DDR4).

[18] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, and et al. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of ISCA*.

[19] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, and et al. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019).

[20] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *Proceedings of MICRO*.

[21] Sysnopsys. 2023. Design Compiler. https://www.synopsys.com/.

[22] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. 2023. GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference. In *Proceedings of ASPLOS*.