PIMPR: PIM-based Personalized Recommendation with Heterogeneous Memory Hierarchy

Tao Yang[1,2], Hui Ma[1], Yilong Zhao[1], Fangxin Liu[1], Zhezhi He[1], Xiaoli Sun[4] and Li Jiang[1,2,3]
[1]Shanghai Jiao Tong University, Shanghai, China, [2]Shanghai Qi Zhi Institute, Shanghai, China
[3]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University
[4]Zhejiang Institute of Science and Technology Information

***Abstract*—Deep learning-based personalized recommendation models (DLRMs) are dominating AI tasks in data centers. The performance bottleneck of typical DLRMs mainly lies in the memory-bounded embedding layers. Resistive Random Access Memory (ReRAM)-based Processing-in-memory (PIM) architecture is a natural fit for DLRMs thanks to its in-situ computation and high computational density. However, it remains two challenges before DLRMs fully embrace ReRAM-based PIM architectures: 1) The size of DLRM's embedding tables can reach tens of GBs, far beyond the memory capacity of typical ReRAM chips. 2) The irregular sparsity conveyed in the embedding layers is difficult to exploit in ReRAM crossbars architecture.**

In this paper, we present a PIM-based DLRM accelerator named PIMPR. PIMPR has a heterogeneous memory hierarchy—ReRAM crossbar-based PIM modules serve as the computing caches with high computing parallelism, while DIMM modules are able to hold the entire embedding table—leveraging the data locality of DLRM's embedding layers. Moreover, we propose a runtime strategy to skip the useless calculation induced by the sparsity and an offline strategy to balance the workload of each ReRAM crossbar. Compared to the state-of-the-art DLRM accelerator SPACE and TRiM, PIMPR achieves on average 2.02× and 1.79× speedup, 5.6×, and 5.1× energy reduction, respectively.

***Index Terms*—Recommendation System; PIM; Embedding; Acceleration; Architecture Design.**

I. INTRODUCTION

Deep learning-based recommendation models (DLRMs) are broadly used in industry (e.g., Facebook, Netflix, Youtube, etc.) [1], [2]. In these DLRMs, *Embedding layers* and fully connected layers (*FC layers*) occupy the primary inference time [3], [4]. As the advancement of DNN acceleration, embedding layers become the bottleneck: **1)** The embedding layers include embedding tables consisting of item feature vectors stored in a dense data structure. These embedding tables (e.g., Facebook, Youtube, Amazon) range from tens of MBs to several GBs (e.g., LastFM [4]: 78.2MB; Amazon TV & Movie [5]: 1.81GB). This large amount of data put pressure on both memory capacity and memory bandwidth [3], [4]. **2)** The embedding operation in embedding layers, including lookup (gather) and pooling (reduction), exhibit a sparse and irregular computation pattern, which is not friendly to typical neural network accelerators.

This work was partially supported by the National Key Reserch and Development Program of China (2018YFB1403400), National Natural Science Foundation of China (Grant No. 61834006). The author Tao Yang is supported by Wu Wen Jun Honorary Doctoral Scholarship, AI Institute, Shanghai Jiao Tong University. Corresponding author: Li Jiang.

Near-Data Processing (NDP) architectures are proposed to accelerate embedding operations and reduce the data movement on traditional memory architecture [4], [6], [7]. TensorDIMM [6] and RecNMP [7] deploy near-memory processing elements (NMPEs) around specialized DIMMs to scale the accelerator's memory space up to tens of GBs. TRiM [8] proposes to accelerate the embedding operation further by exploiting the tree-like interconnect topology of DIMM-based memory. These NDP solutions, however, still need to transfer a large amount of data between the memories and the NMPEs, limiting the overall computing parallelism and processing throughput [4], [6], [8]. Moreover, energy consumption is always constrained by the Power Usage Effectiveness (PUE) requirement in data centers.

As the countermeasure of the well-known "memory wall", ReRAM crossbar [9] emerges as a promising solution, owing to its Computing-In-Memory (CIM) capability that has the potential to solve the problem of bandwidth limitation completely. To the best of our knowledge, one ReRAM-based accelerator design for Personalized Recommendation has been proposed and named REREC [10]. However, two main limitations become the hurdle for REREC from being widely used in real scenes:

- REREC can only work with small workloads. The embedding tables of the production recommendation models can reach tens of GBs, which is far beyond the memory capacity of a typical ReRAM chip (e.g., 400 MBs of ReRAM in REREC).
- REREC only supports the specific two recommendation systems [11], [12] with inner-product operation as the feature interaction method in embedding operations. However, for embedding operation in mainstream DLRMs, the **gather-reduction pattern** is selected and studied widely in acceleration designs nowadays [1], [2], [4], [6]–[8]. Compared with the inner-product operation, which is a natural fit for regular-structured ReRAM crossbars, the gather-reduction pattern exhibits an irregular and sparse computational characteristic, posing greater challenges to the acceleration design on ReRAM crossbars.

In this paper, we present a PIM-based DLRMs accelerator with a heterogeneous memory hierarchy. We exploit ReRAM crossbars for the embedding operations on high-access-frequency embedding items (*fast path*). The embedding items of low access frequency are fetched from the dual-inline

978-3-9819263-7-8/DATE23/© 2023 EDAA

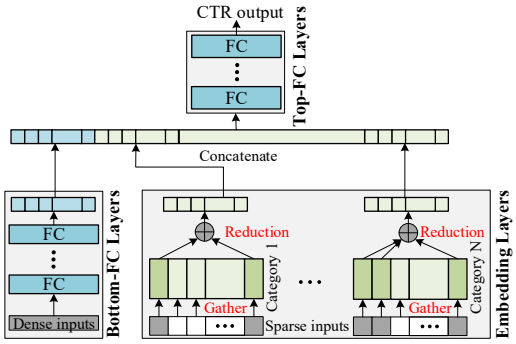

Fig. 1: Architecture of DLRMs.

memory modules (DIMM) for computation (*slow path*). These two combined fast and slow data paths work in a pipeline style that can hide the access latency in DIMM. We then explicitly propose runtime and offline strategies to exploit the irregular sparsity and improve the embedding operations on ReRAM crossbars. Our contributions could be summarized as follows:

- We present a PIM-based DLRMs architecture with a heterogeneous memory hierarchy, exploiting the locality in embedding layers, namely PIMPR. This architecture reduces the bandwidth requirement. Meanwhile, the large degree of parallelism and high energy efficiency characteristics of ReRAM crossbars are leveraged.
- We optimize the throughput by a three-stage runtime strategy to exploit the large sparsity in embedding layers, and a K-partition heuristic-based offline mapping strategy for load balance. A latency-matching pipeline is further proposed to improve the inference throughput.
- Experiments on various industry datasets show that the PIMPR brings 6.72× speedup and 36.2× energy efficiency improvement over NDP recommendation system design TensorDIMM [6].

II. BACKGROUND

A. Personalized Recommendation Models

Figure 1 shows a simplified architecture of the deep learning personalized recommendation model (DLRM) [3]. The model mainly comprises FC layers, embedding layers, and concatenation layers. DLRM receives a set of user characteristics and the user's past interactions as input vectors. The user characteristics are dense inputs containing personal information (e.g., age, gender, etc.). While the user's past interactions are sparse inputs, each non-zero indicates a preferred item in the embedding table. Dense and sparse inputs are processed by the bottom FC layers and embedding layers, respectively. The embedding layer contains several embedding tables, each containing all the item feature vectors in a category. During inference, the *gather operation* collects the item feature vectors looked up by sparse input. Next, the *reduction operation* merges the item vectors by element-wise summation. Compared with the FC layers with a highly regular computational pattern, the embedding layer shows simple element-wise operations, but an irregular access pattern [6].

The industry datasets for DLRMs exhibit two characteristics: First, the number of items in an embedding table can reach several million that occupy tens GBs of the memory space [4], [5], which exceeds the capacity of normal on-chip memory [4]. Besides, transferring this large amount of data puts much pressure on bandwidth. Second, although each embedding table consists of a large number of item vectors, only tens of vectors interact with a user [5]. This character induces an extremely sparse pattern of embedding operation.

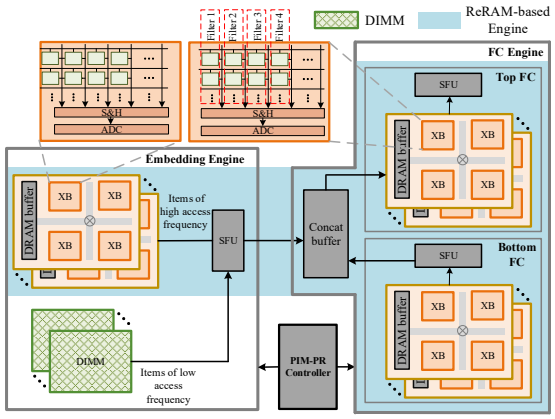

Fig. 2: PIMPR architecture overview.

B. Architecture for Recommendation Models

Near-Data-processing (NDP) architectures have been proposed to accelerate the embedding layers of DLRM. TensorDIMM [6] and RecNMP [7] propose to use specialized DIMM based NDP designs. Multiple DIMMs are required to improve the overall throughput of these two designs due to the low bandwidth of commodity DIMMs. SPACE [4] employs a high bandwidth 3D-stacked DRAM (HBM) and places the near memory processing elements on the logic die. The memory bandwidth, however, is still the main limiting factor for high throughput. By doubling the HBM modules, SPACE can achieve a 1.8× performance improvement.

ReRAM-based accelerator featured by high density, fast read access, and low leakage power [9] has become a promising solution for deep learning networks. One ReRAM-based accelerator for special personalized recommendation models featured by using inner-product operation as the feature interaction method has been proposed and named REREC [10]. However, it still suffers from losing general support for widely used gather-reduction pattern in embedding operations and the tremendous embedding tables in industrial scenarios.

III. DESIGN DETAILS IN PIMPR

A. Overall Architecture of PIMPR

Fig. 2 shows the overall architecture of PIMPR. PIMPR has a heterogeneous memory hierarchy, with the ReRAM-based engine serving as the computing cache and DIMMs assisting in expanding storage capacity. A central controller is responsible for loading the inputs and generating instructions to control the whole computation process. From a functional point of view, PIMPR can be further split into an FC engine and an embedding engine. The FC engine consists of the bottom FC part and the top FC part. Each part contains crossbars and Special Functions Units (SFUs). SFUs include Shift-and-Add units (S&A) and scalar Arithmetic and Logic Units (sALU) to further process

Fig. 3: Access frequency distribution of each item in different categories of datasets.

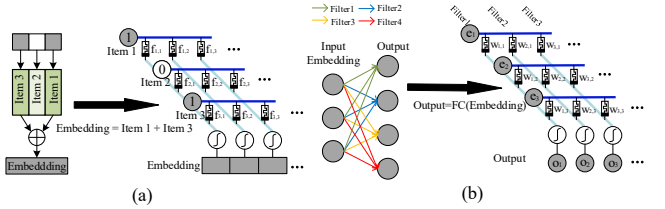

Fig. 4: Implementing embedding layer on ReRAM crossbars.

the partial results from the crossbars. Each FC layer's filter is mapped to a column of the crossbar cells and vector-multiplied with input features via each bitline. In the embedding engine, besides the ReRAM crossbars and SFUs, we include DIMM chips featured by large storage capacity assisting the ReRAM crossbars in storing the whole embedding tables.

1) Item access locality: As shown in Figure 3, we show the distributions of access frequency of each item in various industry datasets, which indicates a strong locality: A large proportion of accesses for an embedding table come from a small portion of items (the statistical analysis shows that 90% of the total access frequency comes from the top 6.2% of the items with high access frequency in these four datasets, averagely). Note that the items with high access frequency also exhibit temporal locality, i.e., these "hot" items are changed and updated at a very slow pace.

2) Heterogeneous memory hierarchy: The item access locality motivates us to cache the item feature vectors with the highest access frequency (denoted as H_vectors) on ReRAM crossbars and store the rest of the item feature vectors (denoted as L_vectors) in DIMMs. By principle, we cache as many H_vectors as possible in ReRAM. Thus the proportion of H_vectors depends on the capacity of the ReRAM module in PIMPR. During the inference of the embedding layers, the gather and reduction operations of these H_vectors carry out locally on crossbars (detailed in Sec III-A3). Afterward, PIMPR loads the partial sums from buffers in ReRAMs and the L_vectors from DIMMs, then sums them up through SFUs to derive the final embedding results.

3) Implementing DLRMs on crossbar: For embedding layers, Take a simple embedding operation with three item feature vectors stored in the embedding table shown in Figure 4(a) as an example. We write each dimension of an item feature vector on a row of the crossbars (the dimensions of the item feature vectors are hundreds of thousands in production recommendation models [4], and we store a vector to the same row of several crossbars). Then we gather the chosen items by inputting "1" in corresponding wordlines. Thus, we can achieve the reduction result by quantizing the analog signals in each bitline. As for FC layers, we use the "weight stationary" computational design as in [13] shown in Figure 4(b).

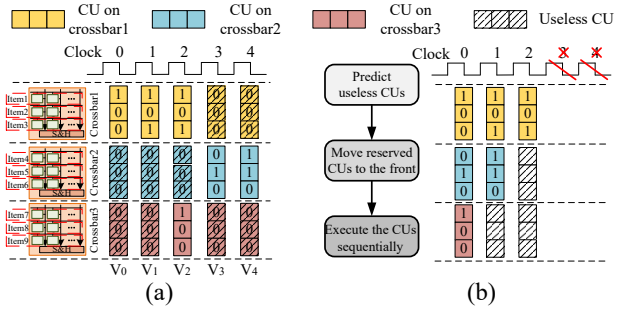

Fig. 5: Comparison between (a) the base design and (b) the design using runtime strategy.

4) Advantages of our architecture: First, the demand for the bandwidth of DIMMs is dramatically reduced. Second, the dominating embedding operations are executed through the high-parallel energy-efficient ReRAM crossbars. This **fast data path** contributes the most computation. Third, compared with the typical main-memory & cache & PEs architecture [7] wherein the operands must be dynamically loaded from the main memory to the cache, PIMPR directly loads the data from DIMMs to SFUs. This **slow data path** can avoid writing the ReRAM crossbars, which incurs large latency, and power consumption issues.

5) Analysis of the low-efficiency problem induced by sparsity in embedding operations: The base execution process (denoted as **Base design**) of embedding operation on PIMPR is shown in Figure 5(a). In this figure, each row of the crossbars stores one item feature vector. V_0 - V_4 are **user interaction vectors**. Each "1" in the vector means the user interacts with the corresponding item. During inference, the three crossbars can execute one input vector's gather and reduction operation in one cycle. Thus, it takes five cycles to complete the embedding operation for all five users. However, we observed that for many cycles there was no item interaction with the user on some bars. Here, we denote the computation on a crossbar in one cycle as a computation unit (**CU**). And the CUs with no item interacting with the user are denoted as **useless CUs**. These useless CUs occupy an extremely high proportion in real-world situations. Take the Amazon CDs & Vinyl dataset containing 1944316 item feature vectors [5] as an example. We store the item feature vectors of the top 5.6% access frequency on 128×128 ReRAM crossbars. In this case, the useless CUs account for 94.7%. This high proportion of useless CUs results in extremely low efficiency in the base design.

6) Runtime strategy overview: We propose the runtime strategy to improve the computing efficiency as shown in Figure 5(b). Specifically, the runtime strategy is composed of three phases: 1) predict the useless CUs (**Prediction Phase**). 2) move the reserved CUs to the front in the timeline (**Skipping Phase**). 3) execute the reserved CUs sequentially on each crossbar (**Execution Phase**). Using this strategy, only three cycles are needed to finish the total embedding operations. Besides, as we can predict useless CUs, the crossbars and the peripheral circuit (e.g., DACs) can be turned off in corresponding cycles to reduce energy consumption.

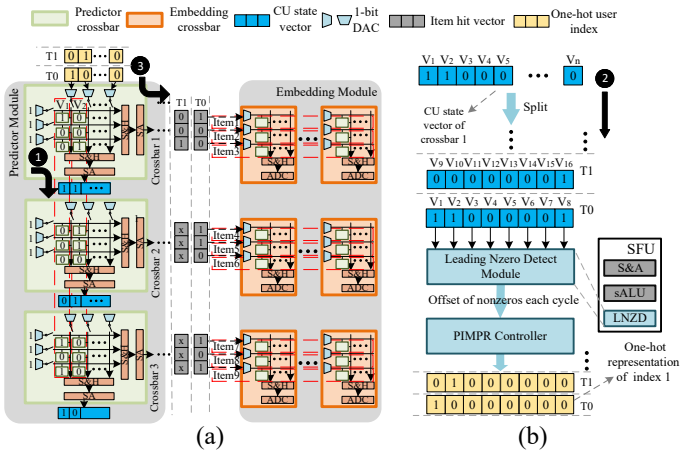

Fig. 6: Modules and dataflows in the three phases: (a) the two-mode predictor crossbars and the dataflows in prediction phase and execution phase. (b) the optimized LNZD module and the dataflow in the skipping phase.

B. Module design and dataflow for the runtime strategy

We design the dedicated hardware architecture to efficiently implement the above strategy at runtime. The modules and dataflow of the architecture are described as follows.

The **Embedding Module** consists of an array of ReRAM crossbars, where each row of embedding modules can store an item feature vector. The **Predictor Module** contains a group of crossbars arranged in a vertical manner. A batch of user interaction vectors is placed in sequence across all the columns. Each predictor crossbar links to one row of the embedding crossbars as shown in Figure 6(a). The predictor crossbar has two function modes, i.e., the *predicting mode* for the prediction phase and the *reading mode* for the execution phase. Unlike conventional crossbars only containing a DAC on each row, the predictor crossbars deploy switchable DACs on each row and each column. Besides, an optimized **Leading Nonzero Detection Module (LNZD)** is added in SFUs for the skipping phase, as shown in Figure 6(b).

1) Prediction phase: In the prediction phase, we turn on the switches of the DACs on each row to implement *predicting mode* on predictor crossbars. The data flow is shown as 182 in Figure 6(a): PIMPR inputs "1" to DACs on all rows of predictor crossbars; and consequently, each predictor crossbar can summarize the values on each column. Suppose the resulting sum of a column is ≥ 1, indicating that at least one item interacts with the user. In this case, the corresponding sense amplifier (SA) on the bottom of the predictor crossbar then outputs a "1" bit to reserve the corresponding CU in the embedding crossbar. Otherwise, the predictor crossbar yields a "0" bit on this column as a signal to skip the useless CU. The outputs of all columns of a predictor crossbar are combined into a *CU state vector* for further use in the skipping phase.

2) Skipping phase: The data flow of the skipping phase is shown as 183 in Figure 6(b). The CU state vectors from predictor crossbars are fed to LNZDs, which operate at a much higher frequency and output each nonzero's offset in the CU state vector cycle by cycle. The central controller converts these offsets into corresponding one-hot format (denoted as *one-hot user index*) to represent the global index of the users interacting with at least one item in the batch. The *one-hot user index* is later used in the execution phase to search the interactive items on the corresponding embedding crossbars.

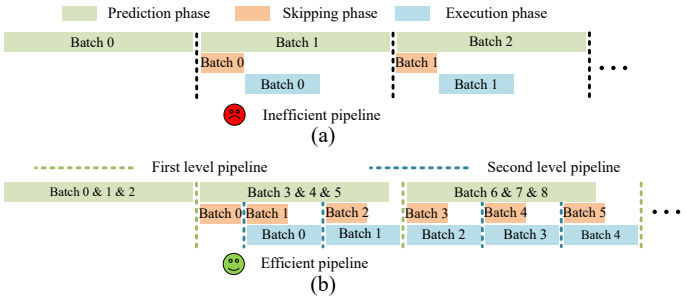

Fig. 7: Pipeline optimization: (a) batch-by-batch pipeline. (b) two-level latency-matching pipeline.

The input size of each LNZD should match the largest batch size (e.g., 128), which causes a great burden on hardware resources. Here, we split the long CU state vector into short vectors to simplify the LNZD's logic cost, as shown in Figure 6(b). Note that the digital circuit of LNZD operates at a frequency (e.g., 1.5 GHz) which is hundreds of times the operating frequency of the ReRAM crossbars (e.g., 10 MHz [13]). The production throughput of *one-hot user index* from the LNZD is therefore high enough for the consumer (embedding crossbars).

3) Execution phase: In the execution phase, the predictor crossbar works in reading mode by turning on the switches of the DACs on each column as shown in 184. For the one-hot user index that goes through the DACs on columns of predictor crossbars, only one column of a predictor crossbar is "read out" by the right SA. The output *Item-hit vector* from each predictor crossbar drives each row of the embedding crossbar to select the feature vectors of the interactive items for gather and reduction operations.

With our runtime strategy, crossbars may compute partial results for different users in an iteration. We buffer these partial results instead of merging these partial results immediately. Then, the controller records the original user interaction vectors and performs an index-selection process to merge the corresponding partial results to get the final embedding results for each user.

C. Latency-matching Pipeline

It incurs long latency to write the user interaction vectors onto the ReRAM-based crossbars, not only because the write latency is much longer (e.g., (r) $29.3ns$/(w) $50.9ns$ [14]), but also the crossbar only supports writing one row per cycle. Therefore, the latency of the prediction phase is much longer than the skipping and execution phases in a batch (e.g., $2.75\times$ and $4.1\times$ longer on Google Maps [4] with a batch size of 128, respectively). A straightforward pipeline design, as shown in Figure 7(a), inevitably causes considerable idle time. We propose a latency-matching pipeline design to improve efficiency as shown in Figure 7(b). We triple the resources of predictor crossbars and write the three batches of data into three rows of predictor crossbars in parallel. Meanwhile, the prediction and execution phases of the three batches are arranged in a finer-grained manner where each phase is executed batch

Algorithm 1: Heuristic K-partition for Offline Strategy

Input: access frequency of n items in descending order $\mathbf{F} = \{F_1, \ldots, F_n\}$, crossbar number k
Output: item assignments of k crossbars $\mathbf{A} = \{A_1, \ldots, A_k\}$

Create $\mathbf{A} = \{A_1, \ldots, A_k\}$ and generate min-heap $\mathbf{A}$ based on $\mathbf{A_i}.val$;
/* $A_i.val$ is the access frequency of crossbar, initialized to 0; A_i.item is the list of assigned items, initialized to empty */
for $j \leftarrow 1$ *to* n **do**
    $root \leftarrow$ pop the root element of $\mathbf{A}$;
    $root.val$ += F_j;
    append j to $root.item$;
    push $root$ into $\mathbf{A}$;
end

Tab. I. Parameters of the PIMPR Architecture

Component	Configuration	Area ($mm^2 \times 10^{-3}$)	Power (mW)
Crossbar	Embedding Crossbar: 128×128 2-bit/cell, number: 9504×8	1900.8	22809
	Predictor Crossbar: 128×128 1-bit/cell, number: 96×8	17.4	190.8
	For FC layer: 128×128 2-bit/cell, number: 8192	204.8	2457.8
DAC	For Embedding Crossbar: 1-bit, number: 76000×8	102.75	2371.2
	For FC layer: 2-bit number: 5120	1.7	30.72
ADC	6-bit,1.2GSps number: 3700×8+128	17395	19024
SA	1.2GSps, number:1200×8	1094.3	950.4
S&H	number:1216000×8+5120	299.21	3.95
Buffer	512KB×9	1843.2	2520
SFU	-	8854.4	976
Controler	-	1842.9	67.3
DIMM	2.4GHz, 4 channel 8 GB per channel	-	8500
Total		**33.56 mm^2**	**59.64 w**

by batch. Experiments show an average 2.69× improvement of throughput on the workloads shown in Table II using our latency-matching pipeline.

Besides the pipeline of these three phases, we also exploit a system-level pipeline for computing on ReRAM chips and loading low-frequency items from DIMMs. The latency of loading these interactive L_vectors in a pipeline stage is 0.29× of the latency of computing on ReRAM chips averagely in our evaluation. This number indicates that the bandwidth of DIMMs is no longer a bottleneck in our design.

D. Offline Mapping Strategy for Workload Balancing

Although the runtime strategy can improve the computing efficiency of embedding crossbars, there still be many idle cycles on amounts of crossbars for the unbalanced workloads as shown in Figure 5(b). Here, we propose an offline mapping strategy to balance the workload and revive those vacant crossbars. The key idea is to reorder the items stored on the embedding crossbars to make each embedding crossbar has the same probability of being activated. The problem of minimizing the difference among the access frequencies of crossbars can be relaxed as a linear K-partition problem solved by a greedy heuristic as shown in Algorithm 1.

IV. EXPERIMENT

A. Experiment Setup and Benchmark

Table I shows the area & power parameters of the components in PIMPR (consisting of eight ReRAM-based chips, one DIMM for the embedding engine, and one ReRAM-based chip for the FC engine). We use CACTI 7 [15] at 32nm to model buffers. We use the same ReRAM model as in [13] to obtain the power consumption parameters of the ReRAM crossbars. The read/write latency and read/write energy cost are 29.31ns/50.88ns, 1.08pJ/3.91nJ, respectively [14]. For ADC, DAC, and SA, the model from [16] is used. According to the statistical data, up to 21 rows in an embedding crossbar are activated. Thus 6-bit ADC is sufficient for the accumulation results on each bitline. We implement the overall digital circuitry (including the central controller, multiplexers, decoders of the input vector controller, SFU component, etc.) in Verilog RTL. We then synthesize the RTL using 32nm [17] technology with an operating frequency of 1.5GHz. For DIMM, we use Ramulator [18] to get a cycle-accurate behavioral model. We modified NVSim [19] with these models to estimate time, area, and energy consumption. The overall area and power consumption of PIMPR are 33.56mm^2 and 59.64W, respectively.

We use Facebook's deep learning recommendation model (DLRM) [20] with eight embedding tables and six FC layers in our experiments. The batch size is 128. We select six datasets in real-world recommendation systems. (1) Amazon - CDs & Vinyl (cds) [5]. (2) Amazon - Kindle Store (kds) [5]. (3) Amazon - TV & Movie (tvm) [5]. (4) Google Maps (map) [4]. (5) Anime (ani) [4]. (6) Steam Game (stm) [4]. In experiments, we combine four datasets as a big workload for evaluating the production-level recommendation systems. The workloads are listed in Table II.

Tab. II. Workload Configurations

Workload	Dataset Configurations	Workload	Dataset Configurations
CDS	cds-cds-cds-cds	KDS	kds-kds-kds-kds
TVM	tvm-tvm-tvm-tvm	MAP	map-map-map-map
ANI	ani-ani-ani-ani	STM	stm-stm-stm-stm
MIX1	cds-kds-tvm-map	MIX2	kds-tvm-map-stm

B. Overall Performance

We compare PIMPR with state-of-the-art NDP recommendation accelerators TensorDIMM [6], RecNMP [7], SPACE [4], TRiM [8]. We do not include the comparisons with REREC [10] for the ReRAM crossbar capability is limited and the mainstream embedding operations using the gather-reduction pattern are not supported in REREC. Figure 8 shows the relative performance and energy efficiency of PIMPR and the other NDP systems normalized to TensorDIMM×2 when executing the total DLRM. Each suffix means the number of memory chips used in the specific NDP design. Averagely, PIMPR achieves 6.72× and 1.79× speedup, 36.2×, and 5.1× energy efficiency improvement compared with TensorDIMMx2 and TRiM.

In these NDP systems, the parallelism of the processing units is limited by the huge cost of the digital circuit implementation and constrained memory bandwidth. By comparison, each ReRAM crossbar in ReRAM crossbar arrays can perform the reduction of multiple items in one cycle, and we can execute calculations on multiple crossbars simultaneously, which provides a huge potential for computational parallelism. Besides,

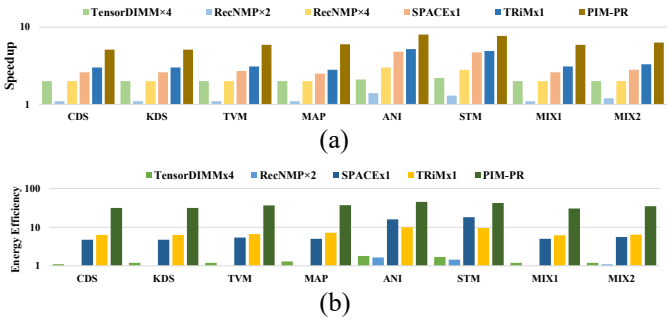

Fig. 8: (a) Speedup and (b) energy efficiency comparisons when executing the total DLRM.

Tab. III. Effect of Our Strategies

	Workloads	CDS	KDS	TVM	MAP
Base design	latency(ms)	1379.2	1720.4	2555.84	2126.4
	Energy consume(J)	62.8	89.6	127.12	103.44
Base+RTS	latency(ms)	503.36	627.2	1213.6	771.2
	Energy consume(J)	20.4	25.44	47.36	31.84
Base+RTS +OMS (PIMPRS)	latency(ms)	**184.8**	**471.5**	**665.04**	**286.4**
	Energy consume(J)	**8.16**	**20.8**	**29.75**	**13.28**

PIMPR only needs to load the low-access items from DIMMs, which greatly reduces bandwidth pressure (In our experiments, the latency of loading items from DIMMs is only 0.29× of the latency of computing on ReRAM chips averagely. It indicates that our design can solve the "memory wall" problem of DLRM.) Moreover, the two proposed strategies further contribute to better performance, for they greatly improve the efficiency of the embedding operations on crossbars.

C. Analysis of Our Proposed Strategies

We also provide ablation studies to demonstrate the effects of our runtime strategy (RTS) and offline mapping strategy (OMS). We evaluate the latency/energy consumption with/without the two strategies on CDS, KDS, TVM, MAP. The results are shown in Table III. We observe that (RTS) alone brings 2.74× latency improvement and 3.10× energy reduction averagely on the four workloads. With both RTS and OMS, PIMPRS further achieves 5.78× performance improvement and 6.05× energy reduction on average.

Here, we calculate the average of the following values in all batches to further explain the effect of the two strategies: **1)** the maximum value of the reserved CUs on each crossbar, which determines the inference latency of a batch according to the "Cask Effect". **2)** the mean value of the reserved CUs on each crossbar. The higher the value, the longer the average working hours of each crossbar in a batch, causing more energy consumption. **3)** the variance of the reserved CUs on each crossbar, which indicates the degree of the load balance among the crossbars. The statistical results are shown in Figure 9. We can figure that the averaged maximum value using *RTS* or *RTS+OMS* is smaller (0.36× or 0.17× averagely) than that in base design, which contributes to the corresponding improvement (2.74× or 5.78× averagely) on throughput. The averaged variance using *RTS+OMS* is much smaller than that using *RTS*, which indicates a more balanced workload distribution using *OMS*. Besides, the averaged mean value using *RTS* or *RTS+OMS* is smaller (0.173× or 0.135× averagely) than that in base design, thus the average energy consumption using *RTS+OMS* is also the least.

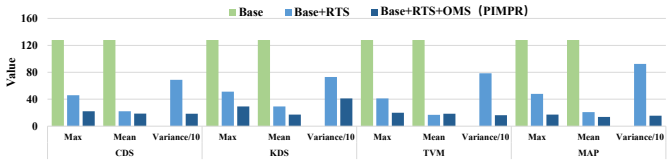

Fig. 9: The maximum & mean number and the variance of the reserved CUs using different strategies on the four datasets, which determines the total latency, energy consumption, and ReRAM load-balancing degree in inference, separately.

V. CONCLUSION

The tremendous data movement becomes the bottleneck for a high-performance DLRM accelerator. Facing this challenge, we present the first PIM-based DLRM accelerator with a heterogeneous memory hierarchy. Moreover, sparsity widely exists in DLRMs, we propose runtime & offline strategies and latency-matching pipeline to exploit sparsity on the heterogeneous architecture. Experimental results show that PIMPR performs better than the previous NDP-based DLRM accelerators on various datasets.

REFERENCES

[1] P. Covington *et al.*, "Deep neural networks for youtube recommendations," in *RecSys*, 2016.
[2] Facebook. [Online]. Available: http://www.facebook.com.
[3] U. Gupta *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *HPCA*, 2020.
[4] H. Kal *et al.*, "Space: locality-aware processing in heterogeneous memory for personalized recommendations," in *ISCA*, 2021.
[5] J. Ni *et al.*, "Justifying recommendations using distantly-labeled reviews and fine-grained aspects," in *EMNLP-IJCNLP*, 2019.
[6] Y. Kwon *et al.*, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *MICRO*, 2019.
[7] L. Ke *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *ISCA*, 2020.
[8] J. Park *et al.*, "Trim: Enhancing processor-memory interfaces with scalable tensor reduction in memory," in *MICRO*, 2021.
[9] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *HPCA*, 2015.
[10] Y. Wang *et al.*, "Rerec: In-reram acceleration with access-aware mapping for personalized recommendation," in *ICCAD*, 2021.
[11] Y. Qu *et al.*, "Product-based neural networks for user response prediction," in *ICDM*, 2016.
[12] H. Guo *et al.*, "Deepfm: a factorizationmachine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.
[13] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ACM SIGARCH Computer Architecture News*, 2016.
[14] D. Niu *et al.*, "Design of cross-point metal-oxide reram emphasizing reliability and cost," in *ICCAD*, 2013.
[15] R. Balasubramonian *et al.*, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *TACO*, 2017.
[16] D. Fujiki *et al.*, "In-memory data parallel processor," in *ACM SIGPLAN Notices*, 2018.
[17] Synopsys. [Online]. Available: https://www.synopsys.com/community/university-program/teaching-resources.html.
[18] Y. Kim *et al.*, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, 2015.
[19] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, 2012.
[20] M. Naumov *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv:1906.00091*, 2019.