This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2021.3097700, IEEE Micro

Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM

Liu Ke*†, Xuan Zhang†, Jinin So‡, Jong-Geon Lee‡, Shin-Haeng Kang‡, Sukhan Lee‡, Songyi Han‡, YeonGon Cho‡, JIN Hyun Kim‡, Yongsuk Kwon‡, KyungSoo Kim‡, Jin Jung‡, Ilkwon Yun‡, Sung Joo Park‡, Hyunsun Park‡, Joonho Song‡, Jeonghyeon Cho‡, Kyomin Sohn‡, Nam Sung Kim‡, Hsien-Hsin S. Lee*

*Facebook, †Washington University in St. Louis, ‡Samsung

ABSTRACT

Near-memory processing (NMP) is a prospective paradigm enabling memory-centric computing. By moving the compute capability next to the main memory (DRAM modules), it can fundamentally address the CPU-memory bandwidth bottleneck and thus effectively improve the performance of memory-constrained workloads. Using the personalized recommendation system as a driving example, we developed a scalable, practical DIMM-based NMP solution tailor-designed for accelerating the inference serving. Our solution is demonstrated on a versatile FPGA-enabled NMP platform called AxDIMM that allows rapid prototyping and evaluation of NMP's performance potential on real hardware under a realistic system setting using industry-representative recommendation framework. We experimentally validated the performance of a two-ranked AxDIMM prototype which achieves up to 1.89× speedup in latency and 31.6% memory energy saving for embedding operations. For end-to-end recommendation inference serving, AxDIMM improves the throughput up to 1.5× and latency-bounded throughput up to 1.77×, respectively.

1. INTRODUCTION

Personalized recommendation is a fundamental building block of many internet services used by search engines, social networks, online retail, and content streaming [1, 2, 3, 15]. Today's personalized recommendation systems leverage deep learning (DL) to maximize accuracy and deliver the best user experiences [4, 6, 10, 13] These models also consume the majority of the datacenter cycles spent on AI. In 2019, the recommendation models collectively contribute to more than 80% of all AI inference cycles across Facebook's production datacenters [18].

To suggest personalized contents to individual users, generic recommendation models are structured to take advantage of both continuous (dense) and categorical (sparse) features. The latter are captured by large embedding tables with sparse lookup and pooling operations. These embedding operations dominate the run-time of recommendation models and are markedly distinct from other layer types in term of operational intensity, presenting unique challenges: First, while the sparse lookup working set is comparatively small (MBs), the irregular nature of the table indices exhibits poor predictability, rendering typical prefetching and dataflow optimization techniques ineffective. Second, the embedding tables are on the order of tens to hundreds of GBs, overwhelming the affordable capacity of on-chip memory. Finally, the operational intensity of embedding operations is orders of magnitude lower than that of the FC layers. This low intensity dwarfs the full potential of custom hardware such as the specialized datapaths and on-chip memories often found in CNN/RNN accelerators. It suggests that unlike CNNs and RNNs, recommendation models exhibiting low compute-intensity and little to no regularity [12, 18] are incompatible with existing acceleration techniques that exploit regular, reusable dataflow patterns and spatial locality. Given the volume of personalized inferences and their rapid growth rate in datacenters, an effort to improving performance of these predominant models will render substantial impact.

To address these memory-bounded challenges, prior work have proposed near-memory processing as a solution to accelerate the embedding operations for DL-based recommendation [12, 19]. However, much of these work are studied with speculative simulation results, lacking rigorous experimental evidence to support and demonstrate the potential of such technology in real hardware. In this work, we introduce AxDIMM , an FPGA-based prototyping platform that can directly map near-memory processing capability onto its programmable fabric using a DIMM-compatible interface built on top of standard DRAM technology. Similar to prior work, we focus on DDR4-based near-memory processing at the DIMM and rank levels [5, 19] instead of resorting to specialized 2.5D/3D integration processes (e.g. HBM) [9, 11], as the capacity needed for production-scale recommendation models can easily reach hundreds of GB and requires the adoption of commercial DDR DIMM devices at a low cost.

The proposed AxDIMM design exploits rank-level parallelism of the DRAM for a range of sparse embedding inference operators. By performing local lookup and pooling functions inside the specialized FPGA modules near memory, AxDIMM can effectively expose higher intra-DIMM bandwidth and provide signficant performance and energy benefits. Overall, AxDIMM leads to significant embedding access latency speedup (1.71–1.89×) and memory energy saving (31.6%), and improves end-to-end recommendation inference throughput (1.2–1.5×) and reduces the tail-latency (34.5–54.6%). With the reduced tail-latency, AxDIMM sys-

tem improves the latency-bounded throughput up to 1.77×. This work makes the following contributions:

- With its programmable FPGA fabric, AxDIMM provides a versatile platform for fast prototyping of near-memory solutions. It supports both in-order general-purpose processors and specialized accelerator modules. It is lightweight and DDR4 DIMM-compatible.
- We develop the full software stack to carry out end-to-end evaluation of industry-representative recommendation workloads. AxDIMM can seamlessly integrate with production inference serving framework to take into account common datacenter practices and representative production configurations.
- Our experimental results demonstrate that AxDIMM accelerates the execution of a broad class of recommendation models and provides up to 1.89× speedup and 31.6% memory energy savings for embedding operation. Overall, AxDIMM achieves up to 1.5× and 1.77× end-to-end inference throughput and latency-bounded throughput improvement.

2. BACKGROUND

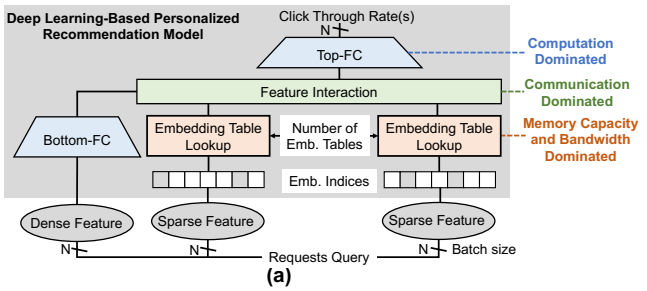
This section describes the general architecture of DL-based recommendation models with prominent sparse embedding features and their performance bottlenecks. As a case study, we conduct a thorough characterization of the Deep Learning Recommendation Model (DLRM) released by Facebook [13]. The characterization illustrates the unique memory requirements and access behavior of production-scale recommendation models and justifies the proposed near-memory accelerator architecture.

2.1 Overview of Recommendation Models

Personalized recommendation is the task of recommending content to users based on their previous interactions and predicted interests. For instance, video ranking (e.g., Netflix, YouTube) recommends a small number of videos, out of potentially millions, to each user. Thus, delivering accurate recommendations in a timely and efficient manner is critical to the business success.

Most modern recommendation models have an extremely large feature set to capture a range of user behavior and preferences. These features are typically separated out into dense and sparse features. While dense features representing continuous inputs in vectors and matrices are processed by typical DNN layers (i.e., FC, CNN, RNN), sparse features for categorical inputs are processed by indexing large embedding tables [13, 18]. A general model architecture of DL-based recommendation systems is captured in Figure 1(a). Similar mixture of dense and sparse features are broadly observable across many alternative recommendation models [4, 6, 14, 15, 20].

Embedding tables are organized as a set of potentially millions of vectors. Their lookup and pooling operations represent sparse features learned during training and generally exhibit Gather-Reduce pattern. Caffe2's family of *SparseLengths* (SLS) operators are widely employed to carry out the embedding operation in production-scale recommendation applications. Our work aims to alleviate the performance



(a)

	Models	# Embs	Emb size	Emb vector	Pooling	FC Layers
DLRM	RMC1	16	1,000,000	64-256 Byte	20-80	6
	RMC2	60	1,000,000	64-256 Byte	20-80	6

(b)

Figure 1: (a)Simplified model architecture reflecting production-scale recommendation models; (b) Parameters of representative recommendation models.

bottleneck attributed to embedding operations and improve system throughput by devising a novel NMP solution to offload the SLS-family embedding operations used in a broad class of recommendation systems.

2.2 DLRM Benchmark Case Study

We use the case study on Facebook's DLRM benchmark [13] to demonstrate the advantages of near-memory processing for at-scale personalized recommendation models. As illustrated in Figure 1(a), dense features in DLRM models are initially processed by the BottomFC operators, while sparse input features are processed through the embedding table lookups. The output of these operators are combined and processed by TopFC producing a prediction of click-through-rate of the user-item pair.

This paper focuses on performance acceleration strategies for two recommendation models representing two canonical classes of the models, RMC1 and RMC2 [18]. These two recommendation model classes consume significant machine learning execution cycles at Facebook's production datacenter, with RMC1 over 29% and RMC2 over 31%. The configuration parameters are shown in Figure 1(b). The distinguishing factor across these configurations is the number of the embedding tables. RMC1 is a comparatively smaller model with fewer embedding tables.

2.3 Performance Characterization

Detailed characterizations of the open-source, production-scale DLRM benchmark have been performed in earlier work [12, 18]. They quantify the potential benefits of near-memory processing in accelerating recommendation inferences and provide the intuition for co-designing the NMP hardware with the algorithmic properties of recommendation models.

A quantitative comparison of the raw compute and memory access requirements is shown in Figure 2(a). Sparse embedding operations, represented by SLS operator, consist of s small sparse lookup into a large embedding table followed by the element-wise summation of the embedding entries. So, circular points in Figure 2(b) show the operational intensity of SLS is orders of magnitude less than FC layers.

Applying the roofline model [16], it has been observed that recommendation models typically lie in the memory

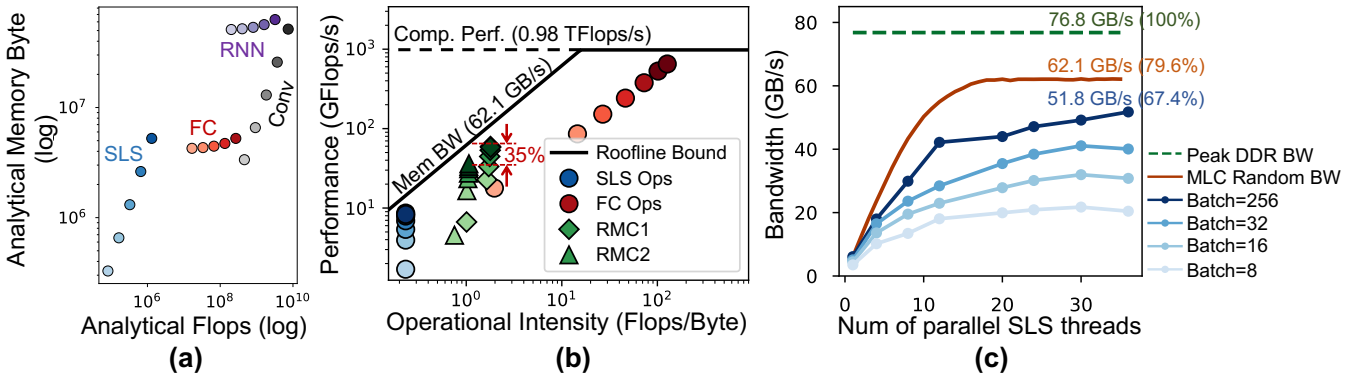


Figure 2: (a) Compute and memory footprint of common deep learning operators, sweeping batch size; (b) Roofline of multi-threaded RMC1, RMC2 sweeping batch size (1-256). Darker color indicates larger batch; (c) Memory bandwidth saturation with increasing number of parallel SLS threads and batch sizes.

bandwidth-constrained region, within 35% of the theoretical roofline performance bound [12]. Figure 2(b) presents the roofline data points for the models, RMC1 and RMC2, as well as their corresponding FC and SLS operators separately. The SLS operator has low compute but higher memory requirements; the FC portion of the model has higher compute needs; and the combined model is in-between. With increasing batch sizes, SLS has low and fixed operational intensity, and FC moves from the region under the memory-bound roofline to the compute-bound region. For the full model, both RMC1 and RMC2 are located in the memory-bound region, as the operational intensity is dominated by embedding operations. It also reveals that, with increasing batch size, the performance of SLS, as well as the entire RMC1 and RMC2, is approaching the theoretical performance bound of the system, and there is tiny room for further improvement without increasing the system memory bandwidth.

It has also been shown experimentally that executing embedding operations can saturate memory bandwidth of real systems at high model- and data-parallelism [12]. Figure 2(c) depicts the memory bandwidth consumption as the number of parallel SLS threads (model-parallelism) is increased for different batch sizes (data-parallelism). The green horizontal line represents the ideal peak DRAM bandwidth (76.8 GB/s, 4-channel, DDR4-2400) and the red curve is an empirical upper bound measured with Intel MLC [7]. The memory bandwidth can easily be saturated by embedding operations especially as both the batch size and the number of threads increase. The memory bandwidth saturates at 51.8GB/s (batch size = 256, number of SLS threads = 30) where 67.4% of the available bandwidth is taken up by SLS. In practice, a higher level of bandwidth saturation beyond this point becomes undesirable as memory latency starts to increase significantly [8].

2.4 Related Work on NMP Architecture

Considering the unique memory characteristics and the sparse, irregular access pattern of personalized recommendation, a number of NMP solutions have recently been proposed [12, 19] to accelerate the predominant embedding operations. Using RecNMP as an example, these DIMM-based NMP solutions often employ NMP processing unit (PU) at DRAM rank-level, as shown in Figure 3. The hardware architecture is minimally modified, locating inside *buffer chip*

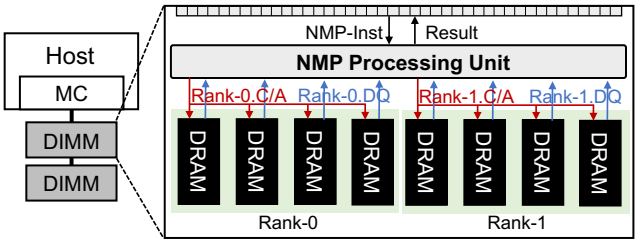


Figure 3: The hardware architecture of prior NMP accelerators [12, 19]

on a DIMM module without requiring any change to the commodity DRAM devices. The NMP PU performs the local embedding lookup an pooling functions at memory-side, producing the general Gather-Reduce execution pattern. The embedding entries are fetched from the concurrently activated ranks, which expands the aggregated memory bandwidth by the number of ranks in the channel (rank-level parallelism). The element-wise summation of the embedding entries is performed inside the NMP PU, and the final pooling result is transferred back to host. The data transfer energy is also reduced, since the embedding entries are pooled inside the NMP PU and only the final results transferred from memory to host.

3. AXDIMM DESIGN

AxDIMM stands for accelerator DIMM and it is a DDR4-compatible FPGA-based platform with standard memory interfaces and compact footprint. With the programmable FPGA fabric inside the buffer chip on the high capacity DIMM, AxDIMM supports both in-order general-purpose processor and specialized accelerator modules. By expanding internal memory bandwidth with multi-rank parallel operations and reducing data movement energy through memory-side data processing, AxDIMM can significantly improve the performance and energy efficiency of the system, making it an ideal prototyping platform for near-memory processing. It can be easily integrated with the host through the standard memory channel interface. In our recommendation applications, the FPGA module of AxDIMM is programmed to instantiate domain-specific accelerators for the embedding operations with gather-scatter pattern to facilitate the near-memory acceleration of personalized recommendation.

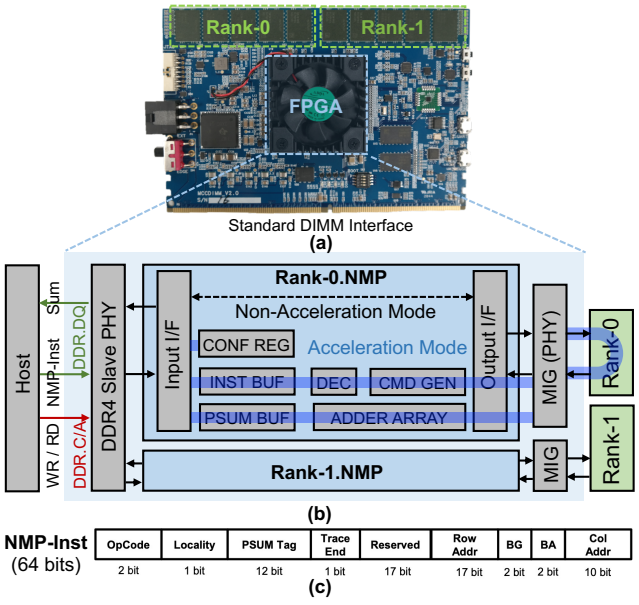


Figure 4: (a) DDR4-compatible FPGA-based AxDIMM platform, (b) hardware architecture of Rank-NMP module, (c) NMP-Instruction format.

3.1 Hardware Architecture

As illustrated in Figure 4(a), the AxDIMM system is an FPGA board with standard DIMM interface that serves as a real-system near-memory processing implementation to exploit rank-level parallelism of its DDR4 memory modules and accelerate embedding lookup and pooling operations. Figure 4(b) present the detailed hardware architecture of the FPGA module instantiated on the AxDIMM board. The internal DRAM ranks are activated in parallel to load embedding entries and the element-wise summation is performed inside the Rank-NMP modules. Two ranks (Rank-0 and Rank-1) are shown in Figure 4(a) and (b). The final pooled results are transferred to the host through the standard memory interface.

To support the interaction with the host, one DDR4 slave PHY is instantiated to receive the DRAM commands and NMP instructions from the host side. With two Rank-NMP modules implemented to interface with the two internal ranks, one AxDIMM can theoretically provide 2× memory bandwidth expansion by exposing the internal memory interface. The memory interface generator (MIG) supports the internal rank accesses between the Rank-NMP module and the commodity DRAM devices.

The execution of Rank-NMP modules is supported in two modes– non-acceleration mode and acceleration mode. In the non-acceleration mode, the host can directly access the end-point DRAM ranks, and AxDIMM functions in the same way as a normal DDR4 DRAM DIMM. All accesses from the host bypass the logic of the Rank-NMP modules. In the acceleration mode, NMP-instructions are issued by the host to perform the embedding lookup and pooling operation inside the Rank-NMP modules and all the regular DRAM accesses from the host are blocked internally. The communication between the host and the Rank-NMP module is performed by normal DDR read/write commands to offload the NMP-instructions

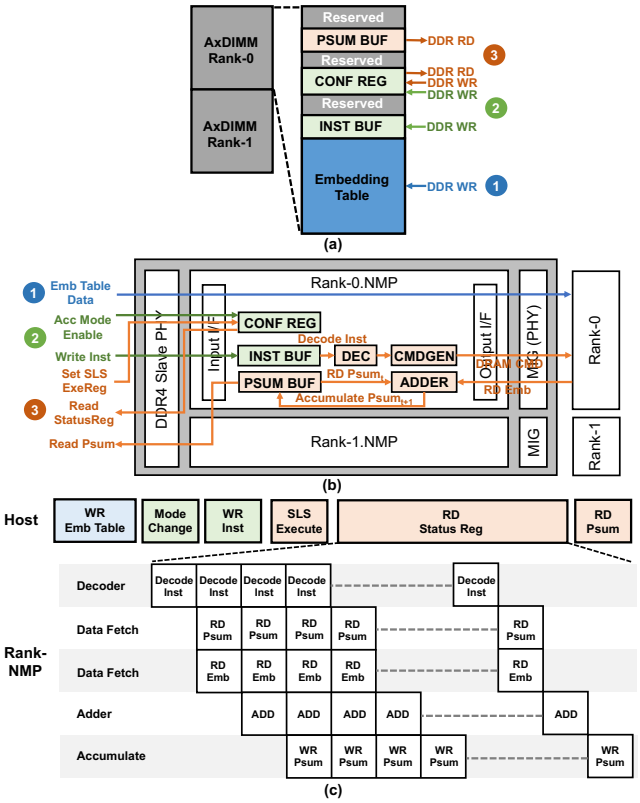


Figure 5: (a) AxDIMM address map, (b) control flow of Rank-NMP, (c) Host-NMP offloading model.

and deliver the final results. As shown in Figure 4(c), each 64-bit NMP-instruction is compatible with the number of the DQ pins of standard memory interface.

The Rank-NMP module is operates following the mechanism of memory-mapped I/O (MMIO). The memory-mapped configuration registers (CONF REG) are set by the host to choose between the two execution mode of a Rank-NMP module. The instruction buffer (INST BUF), a 256KB SRAM, stores NMP-instructions from the host-side; and the partial sum (Psum) buffer (PSUM BUF), also a 256KB SRAM, holds the intermediate Psum values for embedding pooling operations. To load the embedding and Psum vectors from the DRAM devices and Psum buffer, the instruction decoder loads and decodes NMP-instructions from the instruction buffer. The command generator generates the data loading commands to DRAM and Psum buffer to load the embedding and Psum vectors. One DRAM read cycle bursts 64-byte embedding data. The adder array contains 16 floating-point (FP32) adders performing the vector element-wise summation of the loaded embedding entry and Psum vector.

3.2 Execution Flow

The execution flow of embedding operations between the host and the AxDIMM is processed in the following three steps: (1) initializing embedding tables, (2) offloading NMP-instructions, (3) performing NMP operations and loading the results.

First, all the embedding tables are initialized by the host in

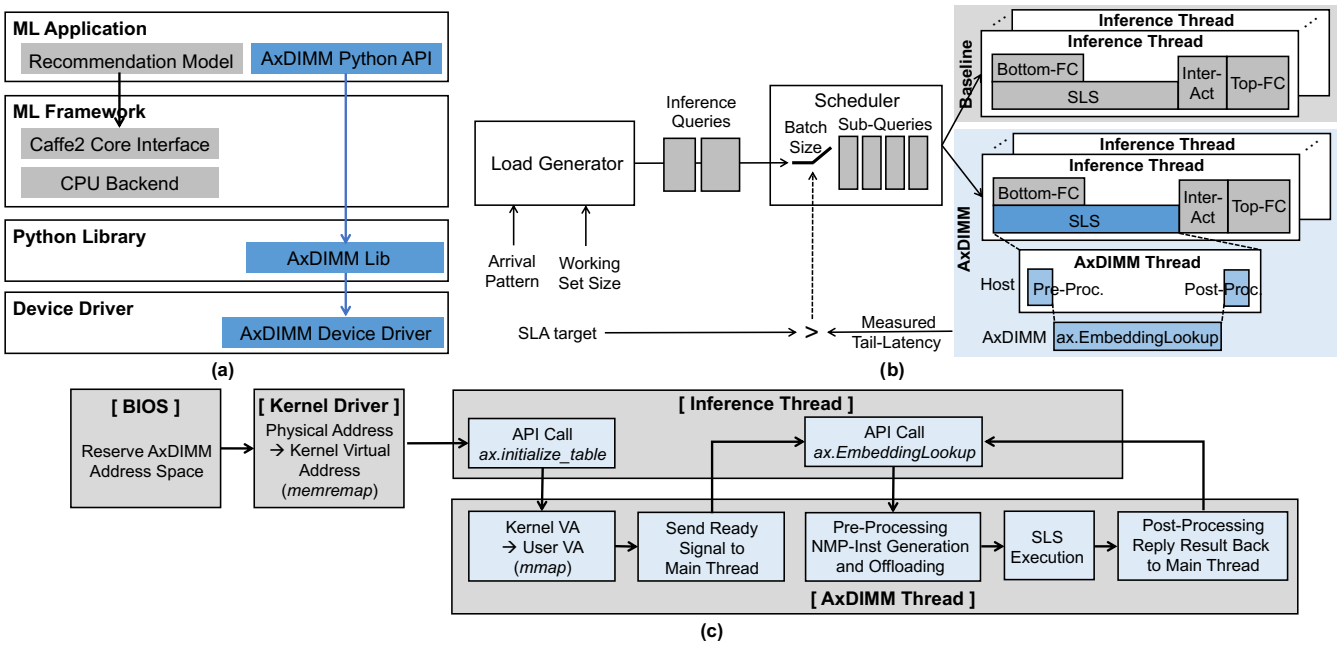


Figure 6: (a) AxDIMM software stack, (b) integrated recommendation inference framework [17] with AxDIMM , (c) SLS offloading flow.

the non-acceleration mode. The embedding tables are written into the DRAM with normal DRAM write commands with no additional control required. Figure 5(a) shows the memory map of AxDIMM . The instruction buffer, the memory-mapped configuration registers, and the Psum buffer are placed at the top of each DRAM rank and exposed to the host, followed by the embedding tables in the memory space.

Next, the Rank-NMP modules are configured to acceleration mode by the host setting the configuration registers. The NMP-instructions for one group of embedding pooling are offloaded from the host by DDR write commands and stored in the instruction buffer.

Then, the Rank-NMP performs the embedding lookup and pooling operation and finally sends the results to the host. The Rank-NMP internal operation is triggered by the host writing the SLS execution register in the configuration registers. The instruction decoder starts loading the NMP-instruction from the instruction buffer and delivers to the command generator. According to the NMP-instruction, the command generator generates the DDR read commands to load the embedding entry from the DRAM rank and load the Psum_t vector from the Psum buffer to its internal FP32 adder array. In one cycle, 64 bytes of embedding and Psum data are loaded to the adder array. The 16 FP32 adders perform element-wise summation of the embedding and Psum vectors, and then store the updated Psum_{t+1} back to the Psum buffer. The host checks the execution status register to track the progress of the embedding operations. When all the operations for one group of embedding pooling are done, the host reads the results from the Psum buffer with a normal DDR read command.

3.3 Software Stack

To provide a seamless programming interface, the AxDIMM software stack integrates the compatible AxDIMM Python library at the user level and the device driver at the kernel level to run recommendation workloads. As shown in Figure 6(a), the AxDIMM Python API is implemented at the user level and supports embedding operations, replacing the existing SLS operators in Caffe2. The AxDIMM library binds the user virtual address to the kernel virtual address for AxDIMM requests, generates the NMP-instructions, and polls the embedding operation output and return results to the recommendation model. The AxDIMM device driver manages the memory allocation and maps the kernel virtual address to the physical address.

The software stack enables a seamlessly integration of AxDIMM with *DeepRecSys* [17], a production inference framework. In Figure 6(b), the load generator models the production query arrival rate and working set size patterns for at-scale recommendation inference. The scheduler optimizes the latency-bounded throughput for at-scale execution exploiting the data- and model-parallelism. To satisfy the strict latency target set by the Service Level Agreement (SLA), large queries are split to multiple sub-queries with smaller batch sizes, and then the sub-queries are processed by parallel inference threads. Same as [17], we adopt the hill-climbing based scheduling policy to achieve the peak latency-bounded throughput by tuning the batch size progressively. On the AxDIMM system, the each main inference thread calls a backend AxDIMM thread. The AxDIMM thread manages the AxDIMM resources and handles the AxDIMM requests from the inference threads by calling AxDIMM APIs.

Figure 6(c) shows the detailed SLS offloading flow during recommendation inference. At beginning, the AxDIMM memory region is reserved by the BIOS and excluded from the kernel usage. The AxDIMM device driver maps the physical addresses (PA) to kernel virtual addresses (VA) by

Table 1: System Parameters and Configurations

	Baseline System	AxDIMM System
Processor	18 cores @ 1.2 GHz	18 cores @ 1.2 GHz
Memory Channel	4 RDIMM channels @DDR4-800 Mbps	2 RDIMM channels @DDR4-800 Mbps, 2 AxDIMM channels @DDR4-800 Mbps,
DIMM Configuration	2 Ranks x8, 16 Gb	
Density	64 GB (SLS) + 64 GB (OS + FC)	
Memory Bandwidth for SLS	2 channels × 800 Mbps	2 channels × 2 rank × 800 Mbps

memremap. During the initialization stage, the AxDIMM thread calls AxDIMM API, *ax.initialize_table*, writing embedding tables to a specified DRAM rank according to AxDIMM address mapping and the AxDIMM library maps the kernel VA to user VA by *mmap*. When the embedding initialization finished, the main inference thread calls AxDIMM API *ax.EmbeddingLookup* to perform the embedding operation. During the pre-processing stage, the NMP-instructions are generated and offloaded to AxDIMM hardware. Then the AxDIMM starts the embedding operation based on the offloaded NMP-Instruction. Finally, during the post-processing stage, the results are polled from the AxDIMM to host, and the main inference thread performs the execution of the following FC layers.

4. EXPERIMENTAL METHODOLOGY

To set up the experimental evaluation in real hardware, we run production-scale recommendation models on the server-class Broadwell CPU. This allows us to measure the impact of accelerating embedding operations on AxDIMM for end-to-end inference serving. Figure 7(a) and (b) show the baseline system and the AxDIMM system used in our evaluation. Both systems contain four memory channels. In the baseline system, all the four memory channels are normal registered DIMM (RDIMM), while in the AxDIMM system, two channels are normal RDIMM and two channels are AxDIMM .

On the AxDIMM system, the memory space of AxDIMM under memory channel CH1 and CH3 is reserved for embedding operations and separated from the main memory space. The rest of the normal processes reside in the normal RDIMM under CH0 and CH2. Similar to the AxDIMM system, CH0 and CH2 of normal RDIMM in the baseline system are also allocated for the normal processes. CH1 and CH3, operated also as normal RDIMM, are exclusively reserved for storing embeddings.

Due to the internal timing constraints of the DDR IO buffer in Xilinx XCZU19EG FPGA on AxDIMM , the overall system speed was purposely slowed down to keep up with the FPGA IO speed. If the DDR IO buffer is implemented with ASIC as will be done in a real product, AxDIMM will operate at the normal speed. As depicted in Table 1, we fixed the IO speed of DDR4 interface of both the baseline and the AxDIMM to 800 Mbps (1/3 of a normal DDR4 memory channel). In commensurate with the memory channel speed degradation, the CPU frequencies for both the baseline and the AxDIMM are also lowered down from 3.2 GHz to 1.2 GHz, the minimum CPU frequency supported by Intel Xeon.

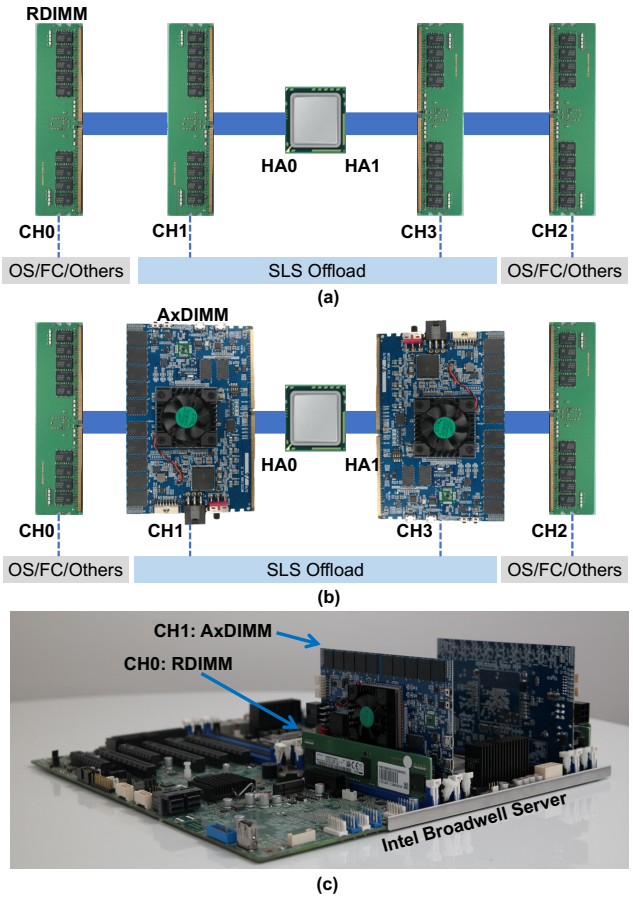


Figure 7: (a) Baseline system configuration; (b) AxDIMM system configuration; (c) Real-system configuration AxDIMM integrated with Intel Broadwell.

5. EVALUATION

This section presents a quantitative evaluation of AxDIMM . We first evaluate the latency improvement and energy saving of the offloaded SLS operators on AxDIMM system. Then, an end-to-end evaluation of throughput, tail-latency and latency-bounded throughput for the entire recommendation model is present.

5.1 SLS Operator Speedup

Performance Gain. Since AxDIMM exploits the rank-level parallelism, the memory bandwidth will be scaled linearly with the number of ranks under the memory channel. We validated the speedup of embedding operations on AxDIMM from the expanded memory bandwidth. In Figure 8(a), we sweep the size of memory footprint (0.08–1.25 MB) of the SLS operator by batch size from 16 to 256. The AxDIMM system achieves 1.71–1.89x latency speedup by utilizing the doubled internal bandwidth of 2 ranks on AxDIMM .

Energy Gain. Compared with the baseline system, the AxDIMM system also demonstrates a significant memory energy saving by reducing the data movement between the host and AxDIMM performing the embedding pooling in memory-side. As shown in Figure 8(b), AxDIMM system achieves

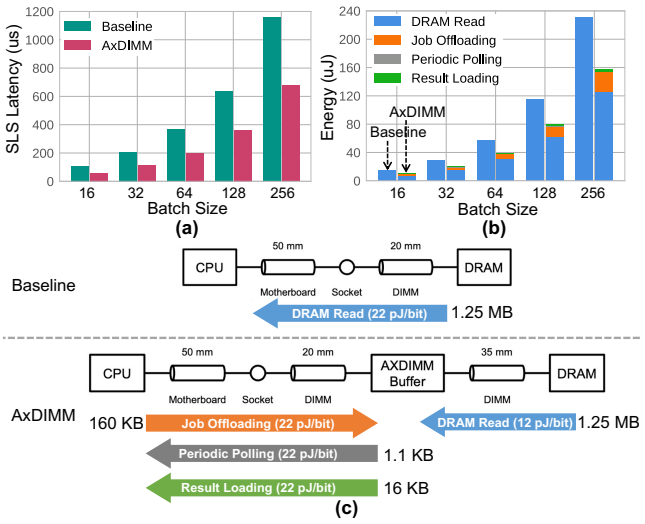


Figure 8: (a) Latency and (b) energy cost of launching SLS operators on baseline and AxDIMM systems, (c) Energy model of baseline and AxDIMM systems

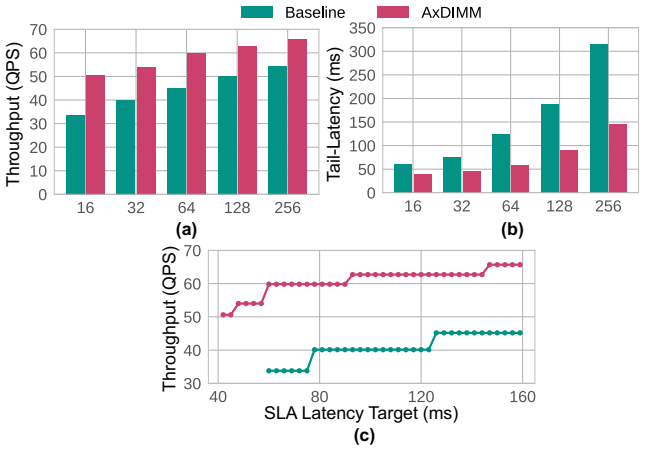


Figure 9: (a) Throughput, (b) tail-latency and (c) latency-bounded throughput of RMC2 on baseline system and AxDIMM system.

31.6% energy saving over the baseline system. In the baseline system, illustrated in Figure 8(c), every loaded embedding entry must be transferred from the DRAM, via the long circuit trace on the motherboard, back to the CPU. Worse yet, such swarm of short-lived data will evict other useful data from the cache hierarchy, destructing on-chip data locality. On the contrary, in the AxDIMM system, embedding accesses and their corresponding pooling operations are to be performed directly inside the NMP modules at the memory-side. Therefore, the host only needs to send the NMP-instructions, check the register status, and read the result from the DRAM when these operations are finished.

5.2 End-to-end Model Speedup

To evaluate the improvement of end-to-end recommendation inference, we measured the model-level throughput and tail-latency with the integrated recommendation inference serving framework [17] for RMC2 on baseline system and AxDIMM system. With the accelerated embedding operation on AxDIMM , in Figure 9(a), the AxDIMM system achieves 1.2–1.5× inference throughput improvement measured in QPS (queries per second), and reduces the tail-latency (p95) by 34.5–54.6% varying inference batch size from 16 to 256.

As a latency-critical service, the recommendation systems must satisfy strict SLA latency targets, therefore recommendation systems are optimized to achieve the high latency-bounded throughput. The reduced inference latency by AxDIMM system allows the inference scheduler tunes the batch size to a higher level while still satisfying the latency target. In Figure 9(c), we observed that the latency-bounded throughput of RMC2 is improved by 1.39–1.77× on AxDIMM system.

6. CONCLUSION

In this work, we present a versatile FPGA-based near-memory processing prototyping platform called AxDIMM and demonstrate its application in accelerating embedding processing of personalized recommendation systems. Our measurement results of a two-ranked AxDIMM achieve up to 1.89× latency speedup and 31.6% energy saving for embedding operation. For the end-to-end recommendation inference, the inference serving throughput is improved up to 1.5× and the tail-latency reduced up to 54.6%. With the reduced tail-latency, AxDIMM system improves the latency-bounded throughput up to 1.77×.

REFERENCES

[1] Amazon Personalize, https://aws.amazon.com/personalize/.

[2] Fortune, https://fortune.com/2019/04/30/artificial-intelligence-walmart-stores/.

[3] Google Cloud Platform, https://cloud.google.com/solutions/recommendations-using-machine-learning-on-compute-engine.

[4] Guorui Zhou et al., "Deep Interest Network for Click-Through Rate Prediction," in *KDD*, 2018, pp. 1059–1068.

[5] Hadi Asghari-Moghaddam et al., "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *MICRO*, 2016.

[6] Heng-Tze Cheng et al., "Wide & Deep Learning for Recommender Systems," in *RecSys*, 2016, pp. 7–10.

[7] Intel Memory Latency Checker (MLC), https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[8] Joseph Izraelevitz et al., "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," in *arXiv preprint arXiv:1903.05714*, 2018.

[9] Junwhan Ahn et al., "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015, pp. 336–348.

[10] Kim Hazelwood et al., "Applied machine learning at Facebook: a datacenter infrastructure perspective," in *HPCA*, 2018, pp. 620–629.

[11] Lifeng Nai et al., "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.

[12] Liu Ke et al., "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ISCA*, 2020, pp. 790–803.

[13] Maxim Naumov et al., "Deep Learning Recommendation Model for Personalization and Recommendation Systems," in *arXiv preprint arXiv:1906.00091*, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[14] Miguel Campo et al., "Competitive Analysis System for Theatrical

Movie Releases Based on Movie Trailer Deep Video Representation," in *Arxiv*, 2018. [Online]. Available: https://arxiv.org/abs/1807.04465

[15] Paul Covington et al., "Deep Neural Networks for YouTube Recommendations," in *RecSys*, 2016, pp. 191–198.

[16] Samuel Williams et al., "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," in *Communications of the ACM*, 2009.

[17] Udit Gupta et al., "DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference," in *ISCA*, 2020.

[18] Udit Gupta et al., "The Architectural Implications of Facebook's DNN-based Personalized Recommendation," in *HPCA*, 2020.

[19] Youngeun Kwon et al., "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *MICRO*, 2019, pp. 740–753.

[20] Zhe Zhao et al., "Recommending What Video to Watch Next: A Multitask Ranking System," in *RecSys*, 2019, pp. 43–51.