Scalability Limitations of Processing-in-Memory using Real System Evaluations

GILBERT JONATAN, KAIST, Republic of Korea
HAEYOON CHO, KAIST, Republic of Korea
HYOJUN SON, KAIST, Republic of Korea
XIANGYU WU, KAIST, Republic of Korea
NEAL LIVESAY, Northeastern University, USA
EVELIO MORA, Universidad Católica de Murcia, Spain
KAUSTUBH SHIVDIKAR, Northeastern University, USA
JOSÉ L. ABELLÁN, Universidad de Murcia, Spain
AJAY JOSHI, Boston University, USA
DAVID KAELI, Northeastern University, USA
JOHN KIM, KAIST, Republic of Korea

Processing-in-memory (PIM), where the compute is moved closer to the memory or the data, has been widely explored to accelerate emerging workloads. Recently, different PIM-based systems have been announced by memory vendors to minimize data movement and improve performance as well as energy efficiency. One critical component of PIM is the large amount of compute parallelism provided across many PIM "nodes" or the compute units near the memory. In this work, we provide an extensive evaluation and analysis of real PIM systems based on UPMEM PIM. We show that while there are benefits of PIM, there are also scalability challenges and limitations as the number of PIM nodes increases. In particular, we show how collective communications that are commonly found in many kernels/workloads can be problematic for PIM systems. To evaluate the impact of collective communication in PIM architectures, we provide an in-depth analysis of two workloads on the UPMEM PIM system that utilize representative common collective communication patterns – AllReduce and All-to-All communication. Specifically, we evaluate 1) embedding tables that are commonly used in recommendation systems that require AllReduce and 2) the Number Theoretic Transform (NTT) kernel which is a critical component of Fully Homomorphic Encryption (FHE) that requires All-to-All communication. We analyze the performance benefits of these workloads and show how they can be efficiently mapped to the PIM architecture through alternative data partitioning. However, since each PIM compute unit can only access its local memory, when communication is necessary between PIM nodes (or remote data is needed), communication between the compute units must be done through the host CPU, thereby severely hampering application performance. To increase the scalability (or applicability) of PIM to future workloads, we make the case for how future PIM architectures need efficient communication or interconnection networks between the PIM nodes that require both hardware and software support.

CCS Concepts: • **Computer systems organization** → **Parallel architectures**.

Authors' addresses: Gilbert Jonatan, gilbertjonatan@kaist.ac.kr, KAIST, Republic of Korea; Haeyoon Cho, haeyoon.cho@kaist.ac.kr, KAIST, Republic of Korea; Hyojun Son, processor@kaist.ac.kr, KAIST, Republic of Korea; Xiangyu Wu, wuxiangyu@kaist.ac.kr, KAIST, Republic of Korea; Neal Livesay, n.livesay@northeastern.edu, Northeastern University, USA; Evelio Mora, eamora@ucam.edu, Universidad Católica de Murcia, Spain; Kaustubh Shivdikar, shivdikar.k@northeastern.edu, Northeastern University, USA; José L. Abellán, jlabellan@um.es, Universidad de Murcia, Spain; Ajay Joshi, joshi@bu.edu, Boston University, USA; David Kaeli, kaeli@ece.neu.edu, Northeastern University, USA; John Kim, jjk12@kaist.edu, KAIST, Republic of Korea.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2024 Copyright held by the owner/author(s).

ACM 2476-1249/2024/3-ART5
https://doi.org/10.1145/3639046

Additional Key Words and Phrases: processing-in-memory, interconnection networks, collective communication

ACM Reference Format:
Gilbert Jonatan, Haeyoon Cho, Hyojun Son, Xiangyu Wu, Neal Livesay, Evelio Mora, Kaustubh Shivdikar, José L. Abellán, Ajay Joshi, David Kaeli, and John Kim. 2024. Scalability Limitations of Processing-in-Memory using Real System Evaluations. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 1, Article 5 (March 2024), 28 pages. https://doi.org/10.1145/3639046

1 INTRODUCTION

Emerging workloads including deep neural networks require a significant amount of computation and general-purpose architectures do not provide the necessary compute. As a result, domain-specific architectures are commonly used to provide a significant increase in the amount of compute [19, 39, 48]. With the increasing amount of compute throughput, modern computer system performance is often bottlenecked by the memory system and, in particular, by the movement of data to/from the main memory system [17, 38, 73]. This bottleneck is becoming more problematic due to increasingly data-intensive workloads that require high memory capacity and memory bandwidth [34, 45, 58]. As today's domain-specific accelerators and hardware platforms provide significantly improved compute throughput [13, 21, 47, 70], the gap between compute throughput and memory bandwidth continues to increase.

Processing-in-memory (PIM) or near-data processing (NDP) have been proposed as potential solutions to reduce the memory bandwidth gap and accelerate overall performance [28, 59]. PIM accelerates applications by moving the computations to where the data is stored (i.e., main memory). PIM is not new as the concept has been proposed since the 70s [74]; however, it has gained renewed interest recently with emerging memory-intensive workloads such as machine-learning workloads [21, 37, 76]. In addition to growing interest from the research community on PIM and NDP, recent announcements by multiple memory vendors include different types of PIM memory modules, including the Samsung HBM function-in-memory (FIM) [53], SK Hynix GDDR-based PIM [49], and the UPMEM processing-in-DRAM engine [22]. The type of computation provided across the different PIM devices varies in terms of compute flexibility/programmability and efficiency, as summarized in Table 1. For example, SK Hynix GDDR-AiM [49] have fixed functional units that specifically target GEMV (GEneral Matrix-Vector multiplication), while the Samsung HBM FIM [55] provides some (limited) programmable compute logic near the memory banks for different machine-learning operations. In comparison, UPMEM [22] provides a general-purpose compute core near the memory banks to provide the highest flexibility as compared to other PIM implementations. However, this comes at the cost of reduced efficiency, similar to CPU vs ASIC or domain-specific accelerators comparison. The AxDIMM [42] also provides general-purpose compute near-memory but compute is located within the buffer chip of DDR4 and does not provide the amount of parallelism compared to alternative PIM architectures. All PIM systems pursue a similar goal of trying to minimize changes to the standard memory interface (e.g., DDRx protocol) while providing the ability to enable computation near the memory.

The different PIM architectures have been evaluated across various workloads, including machine learning applications [79], matrix-vector multiplications [64], and sparse-matrix vector multiplications [29]. Prior studies have demonstrated various degrees of benefits from PIM – e.g., 3.5× improvement on speech recognition on HBM2-PIM [46] and 6.7× improvement on decoder block processing of GPT-3 on GDDR6-AiM [49], compared to non-PIM systems. In comparison, ***this work provides an in-depth analysis of PIM architecture scalability challenges on a real PIM architecture***. In particular, many workloads or kernels require collective communication where parallel compute units need to exchange data. In this work, we focus on two representative

collective communication patterns (AllReduce and All-to-All) that are commonly found in many workloads [77]. We analyze two different types of kernels from emerging workloads that can be potentially accelerated through PIM while requiring collective communication – 1) embedding tables used in recommendation systems [61]; and 2) Number Theoretic Transform (NTT) computations used in Fully Homomorphic Encryption [11]. The two kernels present different challenges for the PIM as embedding table kernel is memory-intensive and requires high memory bandwidth as well as high memory capacity. However, it requires a relatively small amount of compute (e.g., reduction) near memory while global reduction is often necessary. In comparison, NTT is more compute-intensive from modular multiplications but requires All-to-All communication to shuffle data around. These kernels have been analyzed [56, 58, 66, 69, 80] and PIM–based architectures for these kernels have also been proposed [32, 33, 41, 50, 62, 65, 67]; however, to the best of our knowledge, performance and scalability of these kernels have not been analyzed on real PIM architectures.

In this work, we first implement these kernels on the UPMEM PIM system [22] and provide an extensive evaluation and analysis of their performance. In particular, high performance (and utilization) can be obtained on a single PIM node (module) or UPMEM DPU (or DRAM processing unit). We then quantify the performance improvement and scalability as the problem (kernel) size increases when multiple PIM nodes are used to increase the amount of parallelism. While there are performance benefits from PIM when parallelism is exploited across a large number of PIM nodes, we find that communication between PIM nodes is needed, similar to other parallel, distributed systems. As a result, ***performance scalability of PIM is limited when the communication of data between the PIM modules becomes a bottleneck***. In this work, we define **PIM locality** as the locality of data (or memory) for PIM compute logic. For workloads with high PIM locality, PIM scalability is not a problem; however, when PIM locality is reduced (i.e., data is needed from non-local memory), then performance scalability can be limited. However, all modern PIM systems only support communication through the host (or the CPU) and do not provide direct communication between the PIM nodes. We evaluate the performance overhead during the movement of data between the PIM module and the host, as well as the computation overhead from managing intermediate data within the host processor – which fundamentally limits the amount of performance improvement that can be obtained from PIM architecture.

To enable scalable PIM architecture, we discuss how ***future PIM systems should provide hardware (and software) support for processing-in-memory (PIM) interconnect*** between the PIM nodes (or banks). PIM interconnect can enable direct communication between the PIM compute nodes – avoiding the high cost of having to communicate through the host (or the CPU) and enabling higher scalability. Unlike traditional interconnects, the challenges and constraints of DRAM require that both the hardware (i.e., interconnect architecture, switches, etc.) and the software (i.e., communication and synchronization primitives) need to be modified to enable such PIM interconnect. The main contributions of this work include the following.

- We present the design and implementation of two kernels from emerging workloads on the UPMEM PIM architecture that utilizes collective communication and the potential performance benefits from PIM.
- We demonstrate the performance scalability limitations as the number of PIM nodes scale and inter-PIM communication needs to be done through the host (or CPU) communication.
- To enable a high-performance and scalable PIM architecture in the future, we discuss how an interconnection network between the PIM nodes that supports direct communication between them is necessary for future PIM systems to avoid communicating through the host.

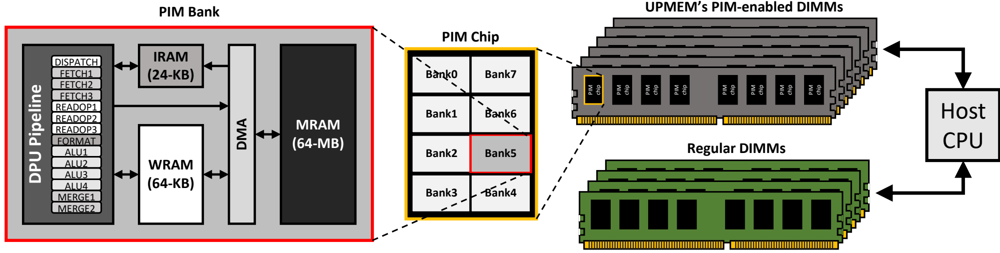


Fig. 1. High-level block diagram of UPMEM Processing-in-Memory system. This work refers to PIM "node" as the unit of near-data processing or a single DPU (or a bank) that has its own compute and local memory.

	Memory type	Level of parallelism	Supported compute	Performance	Flexibility
GDDR6-AiM [49]	GDDR6	Bank	GEMV	+ +	- -
HBM-PIM [55]	HBM2	Bank	GEMV,ReLU,Vector ADD/Mult	+ +	-
AxDIMM [42]	DDR4	Rank	General compute (CPU+FPGA)	-	+
UPMEM [22]	DDR4	Bank	General compute (CPU)	+	+ +

Table 1. Qualitative comparison of different PIM systems.

2 BACKGROUND

In this section, we provide an overview of the UPMEM PIM system used in our evaluation. We also provide the background of two kernels that we explore: embedding table found in deep learning recommendation models (DLRM) as well as Number Theoretic Transform (NTT), commonly used in Fully Homomorphic Encryption.

2.1 UPMEM architecture

While other PIM architectures are available such as Samsung HBM FIM [55] or SK Hynix GDDR PIM [49], the UPMEM architecture was used in this work because it provides the most flexibility in terms of near-data compute that could be exploited for near-data processing. A UPMEM-based PIM system is made up of standard DDR4 2,400-DIMM modules with 8 or 16 UPMEM chips per DIMM. Figure 1 depicts a high-level block diagram for an UPMEM chip. Inside each chip, there are 8 DPUs (DRAM processing units) and 8 64-MB memory banks. As 20 is the maximum number of UPMEM DIMM modules in today's configurations, users can benefit from up to 2,560 (20×128) DPUs and a total memory capacity of 160 GB [71]. All DPUs in the UPMEM modules operate together as a parallel coprocessor to a host CPU. The DPU is a multithreaded 32-bit processor that supports up to 24 hardware threads, called *tasklets*. Each tasklet is equipped with 24 32-bit (or 12 64-bit) general purpose registers, 4 fixed common registers (i.e., common to all tasklets), and 4 fixed thread index registers. The DPU consists of 14 pipeline stages and thus, multiple tasklets are required to fully utilize the DPU pipeline [35]. Apart from a DPU and a 64-MB main memory bank (MRAM), each of the eight PIM chip slices also contains a 24-KB instruction memory (IRAM), and a 64-KB scratchpad memory (a software-managed cache called WRAM). The host CPU can read/write data from/to MRAM through the DDR4 interface with UPMEM APIs [72]. Only data located in WRAM can be used in an operation, hence a DMA module is responsible for moving the data from MRAM to WRAM and from WRAM to MRAM by utilizing DMA read and write instructions, respectively, from the DPU. A DPU can only access local data stored in WRAM. Hence any inter-DPU communication is a very costly operation that must occur through the host CPU.

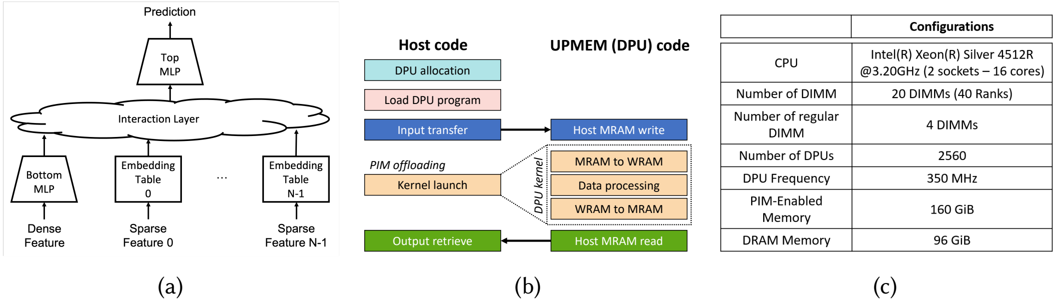
Kernel/Workload	Description	Applications	Type of collective communication	When collective communication is used?
Fast Fourier Transform (FFT)	Signal conversion from time/space domain to frequency domain	Signal visualization, Spectrum analysis	All-to-all	Transpose intermediate matrix
Number Theoretic Transform (NTT)	Specialized Discrete Fourier Transform (DFT) algorithm for finite integer field	Homomorphic encryption	All-to-all	Transpose intermediate matrix
Embedding table lookup	Lookup of the corresponding embedding values for a list of IDs	Deep Learning Recommendation model (DLRM), NLP	AllReduce	Global reduction of output across row partitions
Molecular Dynamics	Predict behavior of atoms in molecular system	Drug discovery, Protein structure prediction	AllReduce	Collect global properties (e.g. total energy, stress, temperature)
PageRank	Graph mining application	Web search	AllReduce	Update vertex score vector
BFS	Graph traversal	Shortest path, Minimum spanning tree	AllGather	Compute union of local next frontier and copy global next fontiers
Multi-Layer Perceptron (MLP)	Feedforward neural network consisting of fully connected neurons	Classification, Regression	AllGather	Broadcast partial output vectors
Distributed Machine Learning	Training or inference of large neural-network model using multiple machines	Large-scale neural networks	AllReduce, All-to-all	Reduce gradients across nodes in data parallelism (AllReduce), data exchange in hybrid parallelism (All-to-all)

Table 2. Example of collective communication across various workloads.

2.2 Applications

In this work, we explore applications that require collective communications, which are commonly used in distributed parallel processing. A summary of different examples of collective communication across different workloads is summarized in Table 2. In particular, we focus on two representative collective communication patterns – AllReduce and All-to-All. AllReduce is a communication pattern that reduces partial outputs from different nodes to produce fully accumulated outputs and then copies the same reduced outputs across all nodes. AllReduce communication is used in many different application fields including graph mining such as PageRank [81, 82], scientific applications like molecular dynamics [30], embedding table lookup, and machine learning [68]. For the All-to-all communication pattern, a unique subset of data is exchanged between all possible pairs of nodes. All-to-all communication is used in scientific applications like Fast Fourier Transform (FFT) [10], NTT computation in Homomorphic Encryption, DLRM training with hybrid parallelism [58], and Homomorphic Encryption [47]. In the following sections, we describe two kernels that we focus on – embedding tables that are accessed in recommendation systems as well as the NTT kernel that is commonly used within homomorphic encryption.

2.2.1 Embedding Tables. Deep learning-based recommendation model (DLRM) is an important class of model for recommendation systems [14, 61]. A high-level block diagram of DLRM is shown in Figure 2a and consists of the bottom MLP, embedding layer, interaction layer, and the top MLP. Bottom MLP handles dense input while the embedding layer handles sparse input. The results of the bottom MLP and the embedding layer are combined by the interaction layer and can consist of concatenation, summing, averaging, etc. The output of the interaction layer is fed to the Top MLP before the final result (e.g., click probability) is generated. While the MLP layers are compute-intensive, the embedding layer is very memory-intensive since it consists of embedding table lookups with sparse inputs as indices. The indices are inputs to recommendation system models and represent a unique identifier or index to an embedding table entry. Another important parameter in the recommendation model and the embedding tables is the *pooling factor*.



	Configurations
CPU	Intel(R) Xeon(R) Silver 4512R @3.20GHz (2 sockets – 16 cores)
Number of DIMM	20 DIMMs (40 Ranks)
Number of regular DIMM	4 DIMMs
Number of DPUs	2560
DPU Frequency	350 MHz
PIM-Enabled Memory	160 GiB
DRAM Memory	96 GiB

Fig. 2. (a) High-level overview of deep-learning recommendation model (DLRM) that consists of N embedding tables and MLP computation, (b) UPMEM basic program structure, and (c) system configuration of the UPMEM server used in the evaluation.

The pooling factor determines the number of entries read from a single embedding table that are combined (or reduced) to generate a single output value. The pooling factor is a hyperparameter determined by the recommendation system model designer and can vary – from a value of 1 (e.g., YouTube [83]), 10s (e.g., Meta/Facebook [34, 58]), and up to 100s (e.g., Alibaba [84]). The embedding table requires not only large memory capacity as the embedding table size for the recommendation system can reach several TBs [51, 58] but also high memory bandwidth due to the large number of memory accesses and some compute (e.g., reduction of the entries read from the single table). In this work, we explore how the embedding table can leverage PIM architecture – in particular, unlike prior work on DLRM PIM [41, 42, 50, 65], we identify the challenges when scaling out embedding tables across multiple PIM nodes.

2.2.2 FHE and NTT. Fully Homomorphic Encryption (FHE) is an emerging technology that enables computation on encrypted data. The security of modern FHE schemes—including BGV [9], BFV [24], TFHE [12], and CKKS [11]—is based on the hardness of the Ring Learning with Errors (RLWE) problem [57]. A key computational bottleneck in such schemes is *polynomial multiplication*, where the coefficients are elements of a finite field. Negacyclic convolution is commonly implemented using the Number Theoretic Transform (NTT) as demonstrated by prior works (e.g., GPU [40], ASIC [47], and PIM [62]). An NTT is simply the FFT specialized to a finite field. The product $\mathbf{a} * \mathbf{b}$ of two polynomials $\mathbf{a}$ and $\mathbf{b}$ is related to the NTT by the following [15]:

$$\mathbf{a} * \mathbf{b} = \mathbf{\Psi}^{-1} \odot \mathrm{iNTT}(\mathrm{NTT}(\mathbf{\Psi} \odot \mathbf{a}) \odot \mathrm{NTT}(\mathbf{\Psi} \odot \mathbf{b})) \tag{1}$$

where $\odot$ denotes Hadamard product, NTT and iNTT denote the NTT and its inverse respectively, and $\mathbf{\Psi}$ is a vector consisting of powers of a primitive $2N$th root of unity.

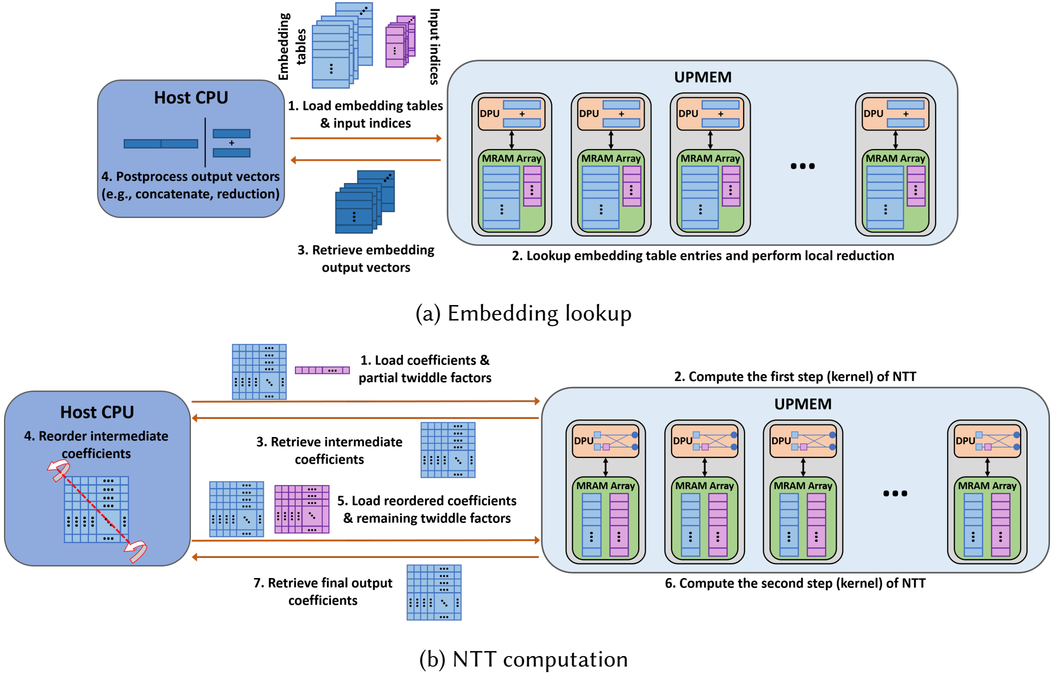
Stridden memory access in NTT leads to poor memory localization and data reuse, which makes NTT difficult to parallelize. To counter this drawback, hierarchical NTT [8] divides the NTT workload into smaller, localized, and manageable pieces that are easier to compute in parallel. An M-dimensional NTT breaks N-point NTT down to M computational steps (i.e., smaller NTT chunks, ideally $N^{1-\frac{1}{M}} \times \sqrt[M]{N}$-point NTT per step) with a synchronization required between each step – e.g., a 65536-point 2D NTT has 2 computational steps, each step containing 256×256-point NTTs. Synchronization consists of an All-to-all data exchange between multiple parallel processes (e.g., DPUs). For the current UPMEM system, this translates to a costly inter-DPU communication. In this work, 2D NTT was chosen for the small number of synchronization requirements (i.e., only one synchronization).

2.3 Related work

Procesing-in-Memory (PIM) and UPMEM: There have been many prior works on processing-in-memory (PIM) or near-data processing (NDP), both within academia as well as industry [3, 5, 7, 25–27, 60]. Recently, memory vendors have proposed different PIM architectures including Samsung HBM-based PIM [46, 55] and GDDR-based PIM from SK hynix [49]. UPMEM proposed PIM by adding a general purpose core near each memory bank [72] and we exploit UPMEM architecture in this work because of the flexibility provided. Prior work [29, 54, 63] have also explored accelerating various workloads, including sparse matrix-vector multiplication, DNA sequencing, and AES encryption on the UPMEM architecture while providing an in-depth analysis of the UPMEM system. In addition, a benchmark suite for UPMEM architecture has also been released [35]. This work also explores the performance benefits of UPMEM architecture; however, unlike prior work, we focus on potential limitations of the UPMEM (and other PIM) architectures when communication is necessary between the PIM nodes.
PIM for Recommendation Systems: RecNMP [41], TensorDIMM [50], and TRiM [65] proposed adding a computing unit in the DIMM buffer and performing the embedding operation at DIMM-side for DLRM. RecNMP [41] proposed to connect PIM to the host memory controller with customized NMP instructions, while TensorDIMM [50] proposed to compose the memory pool called TensorNode with several DIMMs that are interconnected to the host with high bandwidth network channels such as NVLink. TRiM [65] exploits bank-level parallelism as well as rank-level parallelism. Tensor Casting [51] proposes the solution for *training* DLRM with near-data processing in the DIMM. AxDIMM [42] also accelerates recommendation systems through near-data processing with compute logic placed within the buffer chip of a DIMM. However, prior work do not evaluate or analyze the scalability challenges of embedding tables on a real system when distributed across multiple "nodes."
PIM for FHE: Different PIM architecture to accelerate FHE have been proposed [32, 33, 62, 67] CiM-HE [67] proposes a PIM-based accelerator for the BFV scheme homomorphic operations on SRAM while other work (Crypto-PIM [67], FHE-PIM [33], and MemFHE [32]) proposes PIM for RRAM memory technology and accelerates NTT. Unlike prior work, this is one of the first works to explore an important kernel within FHE (i.e., NTT) on a real PIM system and understand its scalability limitations.
Interconnection Network: There has been a significant amount of work done on different types of interconnection networks, including network-on-chip [18, 20] and large-scale networks. Memory-centric network [43] explores interconnect between memory modules as well as intra-module interconnect that utilizes a crossbar; however, the interconnect is not necessarily appropriate for PIM-to-PIM communication. Memory channel network [5] proposed a practical near-memory processing system by assuming each near-memory processor runs an OS and communicates through the existing network stack of the OS. However, host communication is still required when communication between near-memory processors is required. Recent work [75, 85] have proposed to interconnect the DIMMs (or ranks); however, they do not necessarily enable communication between the PIM *nodes*. To the best of our knowledge, this is one of the first works to discuss the need for interconnection networks between PIM nodes to enable scalable PIM computation.

3 EVALUATION METHODOLOGY

In this study, we choose the UPMEM architecture [71] due to its flexibility in terms of computing near memory (i.e., general purpose cores are available near memory); however, our observations on the scalability (or lack thereof) are not limited to UPMEM but can be generalized to other PIM architecture. For example, Samsung's HBM-PIM [55] provides somewhat programmable compute



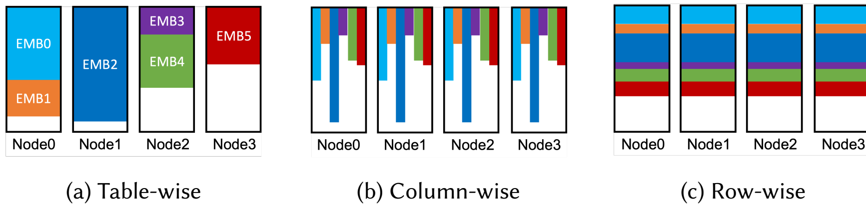
(a) Embedding lookup

(b) NTT computation

Fig. 3. Block diagram illustrating offloading of embedding lookup and NTT computation.

logic within each bank, and SK Hynix's AiM [49] provides fixed compute logic within each bank; however, accessing data from a different bank is a challenge for these PIM architectures as well. The system configuration shown for the UPMEM system used in our evaluation is summarized in Figure 2c. To evaluate the performance, the code for the different kernels was implemented with UPMEM API [72] that is based on C. UPMEM programming shares many similarities with GPU (CUDA, OpenCL) programming. The source code is partitioned into host code that is executed on the CPU, and kernel code that is offloaded to the PIM architecture or the DPUs. Instead of threads (or warps), UPMEM provides units of *tasklets*, which are specified during compilation, with a maximum of 24 tasklets supported for each DPU [22]. The basic control flow for the UPMEM host application is: a) DPU allocation, b) load DPU program, c) input transfer, d) kernel launch, and e) retrieve the output, as summarized in Figure 2b. In comparing the performance of PIM with CPU-only (i.e., no PIM), the same host or the CPU was used in the evaluation.

This work explores the performance benefits of PIM, especially within the context of *PIM scalability*. We evaluate the potential benefits (and limitations) of PIM on two important kernels, 1) embedding tables in recommendation systems and 2) NTT within Fully Homomorphic Encryption. A high-level overview of the execution of the two kernels is shown in Figure 3. For embedding tables, the table and the indices are first loaded from the host and then, executed on the PIM (or the DPU) by memory access and reduction computation. Afterward, the output is sent back to the host for any potential postprocessing (based on how the data is partitioned) (Figure 3a). For the NTT kernel which is partitioned into two stages, communication to/from the PIM nodes occurs across two steps, and in between the two steps, shuffling of the data occurs in the host (Figure 3b). In our implementation, NTT effectively results in the need for executing two kernels because of the data shuffling that is required between the two kernels (or the two stages of NTT). While not



(a) Table-wise (b) Column-wise (c) Row-wise

Fig. 4. Alternative embedding table partitioning approaches for a 4-node system with 6 different embedding tables. The different colors represent the different embedding tables.

shown in detail, UPMEM programming requires data to be copied to "scratchpad" memory (i.e. WRAM) [72] (Sec 2.1)– thus, data from the main memory (i.e. MRAM) is first copied over to the WRAM in our implementation. Additional details of the kernel implementation and how data are partitioned across multiple PIM nodes are provided in Sec 4 and Sec 5.

4 EMBEDDING TABLE KERNEL

4.1 UPMEM Implementation

The embedding table kernel for UPMEM was written and compared against a CPU implementation to validate functional correctness. The offloaded compute to PIM includes the embedding table lookup as well as a reduction of the embedding table entries that are accessed. To accelerate performance on UPMEM, we used the WRAM (or the scratchpad memory within the DPU) and 24 *tasklets* (or threads) to keep the pipeline busy. Unless otherwise stated, a batch size of 512 is assumed and each tasklet is responsible for executing a fraction of the total batch. Analysis is first performed on a single DPU and then, evaluated across multiple DPUs.

4.2 Embedding Table Scalability through Partitioning

The size of the embedding tables can be very large as the size of an individual table can be hundreds of MBs to up to a few TBs [1, 58], and multiple tables are commonly used. As a result, embedding tables need to scale across multiple nodes, especially if a given table cannot fit into a single node, and need to be partitioned across multiple nodes. Partitioning of tables is more problematic for PIM architectures since the amount of memory per PIM node is often limited (e.g., 64 MBs for UPMEM architecture). There are three approaches for partitioning the embedding tables [58] – table-wise, column-wise, and row-wise (Figure 4). Table-wise partitioning distributes one or more tables to each node – the simplest form of partitioning for embedding tables, but can lead to imbalanced memory accesses across nodes. However, the number of tables per node is limited by the memory capacity; thus, for PIM nodes where the memory per node is limited, the memory capacity is often too small to support even one table, thus an alternative partitioning strategy is needed.

Column-wise partitioning divides each table vertically (Figure 4b). For an N node system, each table is partitioned into N columns and each node contains a single column for a given table – thus, the number of memory accesses to each node is identical and memory accesses are balanced. One trade-off is that since the same entry or row needs to be accessed across all nodes, the network (or communication) traffic is increased, as the same table indices need to be sent to every other node. However, the bigger limitation is the scalability of column-wise partitioning since the scalability is limited by the table width or the dimension of the table. Each dimension of an embedding table entry contains data (e.g., 4B floating point data) that need to be combined with other embedding table entries – thus, partitioning the table beyond the unit of a dimension is infeasible and an embedding table with d dimensions can be partitioned across, at most, d nodes. Given that the

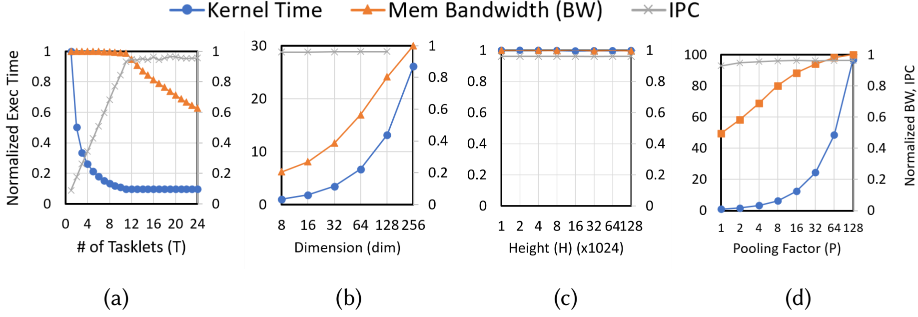


Fig. 5. Single DPU performance (kernel execution time) and analysis (memory bandwidth and DPU compute IPC), with a single embedding table as the following parameters, are varied – (a) number of tasklets (T), (b) embedding table dimension (dim), (c) embedding table height (H), and (d) pooling factor (P). Unless otherwise stated, the default parameters used are $dim = 64, H = 1k, T = 24, P = 16$.

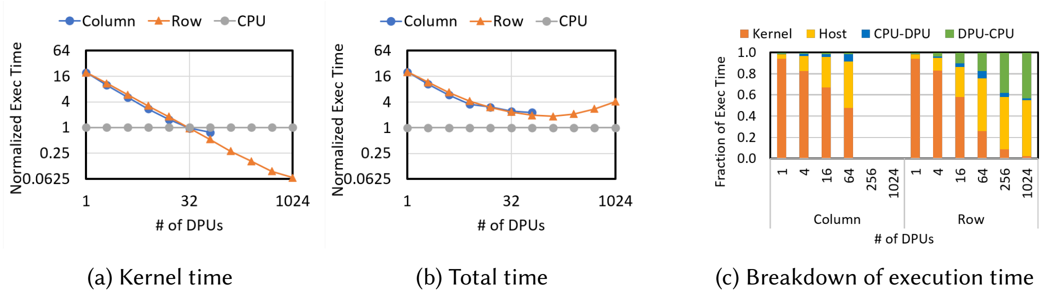
common dimension size is approximately 4-384 [58], the scalability of column-wise is also limited. An alternative approach is dividing the table horizontally (Figure 4c), where each node contains unique entries (or rows) of a given embedding table. This reduces the need for duplicating the embedding table indices across multiple nodes. If the different entries of the tables are placed in different nodes, then more data needs to be transferred from the memory nodes, since only a partial reduction is done at each node and a global reduction is required. Because of scaling limitations, we evaluate *hybrid* approaches that leverage both row-wise and column-wise partitioning in the following sections.

4.3 Embedding Table Evaluations

Embedding tables are first evaluated using synthetic embedding tables on a single DPU and then, we evaluate the scalability of embedding tables across multiple DPUs with different table partitioning approaches described earlier in Section 4.2. We conclude with evaluations based on production-scale embedding tables [58].

4.3.1 Single DPU Evaluations. Analysis of embedding tables on a single DPU is shown in Figure 5 as various parameters of the embedding table are varied, including the table height (H), table width or dimension size (dim), pooling factor (P), and the number of tasklets (T). The figure plots the performance (kernel execution time - thus, lower is better) and results are normalized to the leftmost data point. The analysis provides for the compute and memory – using the memory bandwidth and the DPU IPC (instructions per cycle) metrics. Since the DPU is an in-order, single-issue processor, the maximum IPC that can be obtained is a value of 1. Current UPMEM implementation does not provide performance counters to measure internal MRAM memory bandwidth; thus, bandwidth is estimated based on execution time and the amount of data transferred from the MRAM.

The embedding table implementation requires 11 tasklets to fully utilize the DPU pipeline, suggesting that embedding table implementation in UPMEM is compute-bound instead of memory-bound. When the number of tasklets increases beyond 11, the memory bandwidth utilization actually drops as the performance is bottlenecked by the compute and not necessarily the memory bandwidth. As the dimension size increases, more data needs to be fetched from the memory and thus, proportionally increases the bandwidth as well as the execution time. As expected, the height of the table (or the capacity) does not impact performance since the table is assumed to fit within a single DPU. The pooling factor has an interesting impact on the result as for small pooling factors



(a) Kernel time (b) Total time (c) Breakdown of execution time

Fig. 6. Normalized performance (execution time) measuring (a) kernel (DPU) execution time, (b) total execution time using UPMEM PIM, and (c) execution time analysis as the number of DPUs is varied for column-wise and row-wise partitioning.

($P \leq 8$), the impact on performance is relatively small but beyond 16, the pooling factor has a significant impact on performance. As both dimension and pooling factor increase, more internal bandwidth becomes utilized as well.

OBSERVATION #1: ***For memory-bound workloads such as embedding table kernels, as parallelism is exploited through higher batch size, the compute resource near memory can be fully utilized and the workload can become compute-bound because of the limited compute resource near memory.***

4.3.2 Multi-DPU Evaluations. Performance as the number of DPUs increased is shown in Figure 6, comparing the performance of row-wise partitioning (`Row`) and column-wise partitioning (`Column`) with CPU baseline. The performance comparison metric is execution time – including the execution time of the kernel or the time spent executing on the PIM (or the DPU) as well as the total execution time, which includes the overhead of communication to/from the DPU as well as any inter-DPU communication. In this evaluation, we use a single embedding table with $dim = 64$ (i.e., 64 columns), $H = 128k$, and $P = 32$. The column-wise partition scalability is limited to 64 DPUs since there are only 64 “columns” in the tables and a column cannot be further partitioned. In comparison, row-wise partitioning can scale beyond 64 DPUs. Results show how kernel time continues to improve as the number of DPUs increases and with more than 64 DPUs, the performance of UPMEM can exceed that of the CPU when only the kernel time is considered (Figure 6a). The performance of column-wise and row-rise are relatively similar, aside from the fact that row-wise provides scalability beyond 64 DPUs. However, in terms of the total execution time (Figure 6b), the overhead from PIM results in the CPU always outperforming UPMEM PIM as the CPU exceeds UPMEM by 2.27× with 64 DPUs. Beyond 64 DPUs, the total execution time actually *increases* with row-wise because of the PIM overhead.

Analysis of the total execution time for column-wise and row-wise partitioning is shown in Figure 6c that includes kernel execution time (`Kernel`), time spent on the host CPU (`Host`), sending data from CPU to DPU (`CPU-DPU`) and from DPU to CPU (`DPU-CPU`). In column-wise partitioning, `Host` consists mostly of data movement (or re-organization) to properly concatenate the embedding vector, while row-wise partitioning includes the global reduction that needs to be done by the host CPU. As the number of DPUs increases, the fraction of time spent doing compute near memory (or `Kernel`) becomes smaller – e.g., with 1024 DPUs and row-wise partitioning, `Kernel` represents only 2.3% of the total execution time. However, the overhead, including `Host` and `DPU-CPU`, dominates. The `DPU-CPU` includes transferring the embedding output vectors back to the host and increases

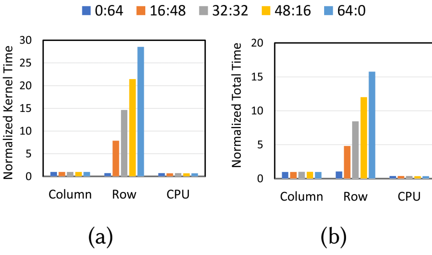


Fig. 7. Impact of load-imbalance on the (a) kernel and the (b) total execution time. With a pooling factor of 64, $x : y$ notation is used where x is the number of indices that are biased while y is the number of indices that are randomly assigned.

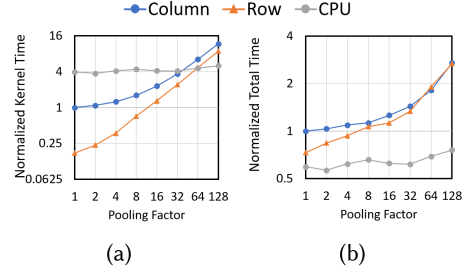


Fig. 8. Normalized (a) kernel and (b) total execution time as the pooling factor is varied, with 64 DPUs and a single embedding table (dim=64, H=128K, T=24).

when only partial reductions are done within each PIM node. The CPU–DPU overhead is relatively negligible since it only involves transmitting the indices to the UPMEM PIM. To provide a fair comparison, we assume the embedding tables are already loaded into the main memory (for CPU baseline) and also loaded in the MRAM for the UPMEM.

OBSERVATION #2: ***Even if the kernel execution time on PIM scales linearly, the communication overhead between the host CPU and PIM nodes (i.e., DPUs) can significantly minimize the potential benefit of PIM as the number of PIM nodes increases.***

The impact of potential load-imbalance is demonstrated in Figure 7. Prior work [23, 52] have shown that embedding table accesses can be skewed and result in "hot" accesses. Thus, we create a synthetic access pattern for embedding tables with an imbalance. Assuming a pooling factor of 64 (i.e., 64 memory accesses), ($x : y$) notation is used to represent the amount of imbalance – x refers to the number of indices of the pooling factor that are "biased" or accesses within the same DPU while y represents the number of indices that are randomly distributed. For example, 0:64 means all inputs are generated randomly while 64:0 means all inputs are generated by bias and access a single DPU. Results show that column-wise partitioning has a negligible impact on performance, regardless of the amount of imbalance while the impact is noticeable for row-wise partitioning. In column-wise partitioning, the same indices are sent to all PIM nodes and thus, load-balanced access occurs across all nodes. However, for row-wise partitioning, the indices are only sent to nodes that have the embedding table entries being accessed – thus, leads to load-imbalanced memory accesses across nodes and cannot fully exploit the available memory bandwidth.

Given the skewed access for embedding tables in real recommendation systems (often following a power-law distribution), it results in hot-data and this can be exploited by reordering the indices [78] to exploit the locality of the CPU memory hierarchy by placing frequently accessed entries near each other. However, a reordered embedding table can be problematic for PIM architectures where the embedding tables are distributed across multiple "nodes." For example, if the row-based (e.g., row-wise or hybrid partitioning) approach is used, the "hot" portion of the table entries will only be distributed across a small number of PIM nodes – thus, creating an imbalance in the number of accesses and not maximize the total amount of memory bandwidth (and compute) across all PIM nodes.

OBSERVATION #3: ***Spatial locality of the embedding table is not necessarily beneficial since it can create imbalance across the different PIM nodes.***

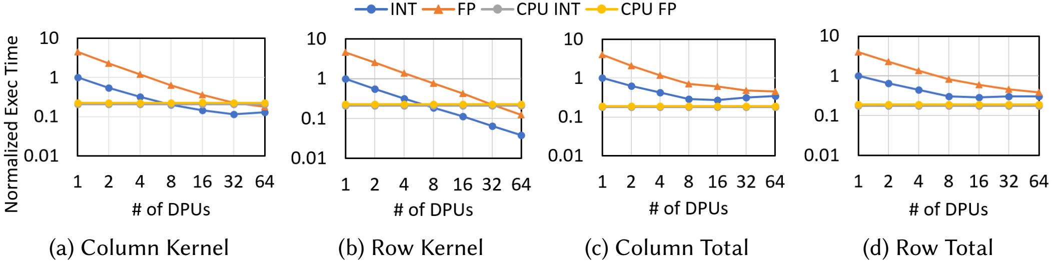
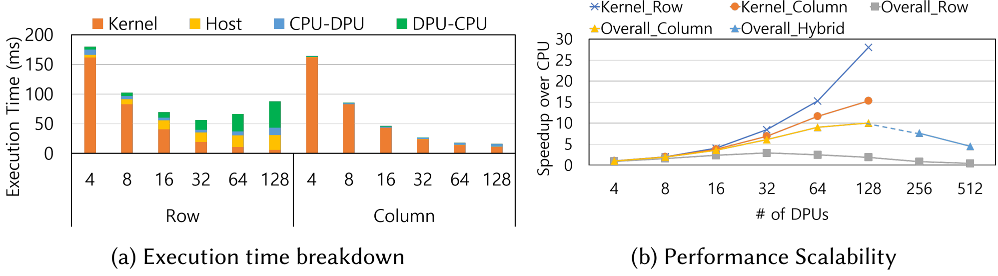


Fig. 9. Comparison between integer (INT) and floating point (FP) implementation of the embedding table (dim= 64, H=128K, T=24, P=32, and 64 DPUs).

The impact of the pooling factor is shown in Figure 8 and as the pooling factor increases, the amount of memory access proportionally increases and thus, results in higher execution time. [1] For small pooling factors, row-wise provides significant improvement in kernel execution time compared to column-wise. One benefit of column-wise is that compute (or memory access) across all of the nodes is load-balanced as each PIM node whereas row-wise can result in imbalance since the amount of accesses to each PIM node can vary (e.g., if a table is partitioned row-wise across two nodes, all accesses to the table can occur on one node while the other node might not have any accesses). However, one benefit of row-wise is memory access granularity, compared to column-wise. For example, if the DRAM (or MRAM) access granularity is 64B, sequential read that accesses the 64B results in more efficient memory bandwidth usage. With row-wise partitioning, each "row" of the embedding table maps consecutively within the DRAM. However, for column-wise where only one element (or one dimension) is mapped to each DPU, only 4B corresponding to one element of the embedding vector is needed resulting in poor memory bandwidth utilization. As the pooling factor increases, the performance gap decreases since imbalance can occur with row-wise, and the overhead (as well as global reduction at the host) results in an increase in the total execution time (Figure 8b).

Embedding tables for recommendation models use floating point data representation [61]. Unfortunately, current UPMEM does not have native support for floating point operations and thus, floating point operations are emulated by the DPU [72]. However, given that some PIM architectures have support for floating point operations [49, 55], we analyze the impact if the compute operations are accelerated. In addition, some prior work [6, 31] have shown a minimal loss in accuracy when using integer-based quantization for DLRM inference acceleration. Thus, we explore the impact of using integer-based embedding tables and its impact on overall performance. Performance comparison is shown in Figure 9 using an embedding table that consists of 32-bit integers (INT) and tables that consist of 32-bit floating point numbers (FP). We also compare against the same implementation using CPU-only baseline implementation. The use of INT results in a significant reduction in kernel time, compared to FP for UPMEM. However, it is interesting to note that the scalability gets *worse* with INT – e.g., the kernel execution time stops improving at around 16 or 32 DPUs while the total execution time flattens out at around 8 DPUs. Although the compute time decreases with INT operations, the overhead, including the global reduction at the host or the DPU-to-CPU data movement does not change – thus, the overhead becomes a larger fraction and further limits scalability. Note that the impact of INT and FP has minimal impact on CPU evaluation since the memory bandwidth is the dominant factor in determining the overall performance.

[1]The execution time for the CPU does not change significantly because of the large capacity of the last-level cache and the locality that it provides.



(a) Execution time breakdown (b) Performance Scalability

Fig. 10. Execution time breakdown and scalability comparison between different partitioning schemes. For scalability, row-column hybrid partitioning is used instead of column partitioning.

OBSERVATION #4: ***The memory bandwidth utilization overhead from small-granularity memory accesses can degrade overall performance. This can be more problematic as the compute throughput of PIM increases.***

Results in Figure 10a provide a detailed performance breakdown when the number of DPUs is scaled from 4 to 128. An embedding table of 128 MB is used such that the table can fit within 4 DPUs. We evaluate the strong scaling out to 128 DPUs. The plot shows performance in terms of execution time (thus, lower is better). In general, row-wise partitioning has better scalability in terms of *compute* speedup – e.g., row-wise achieves nearly a 30× improvement in compute as the number of DPUs is increased by 32×. In comparison, the compute scalability for column-wise partitioning is limited to less than 16× as column-wise partitioning uses a smaller memory access granularity when moving data between MRAM and WRAM which results in inefficiency.

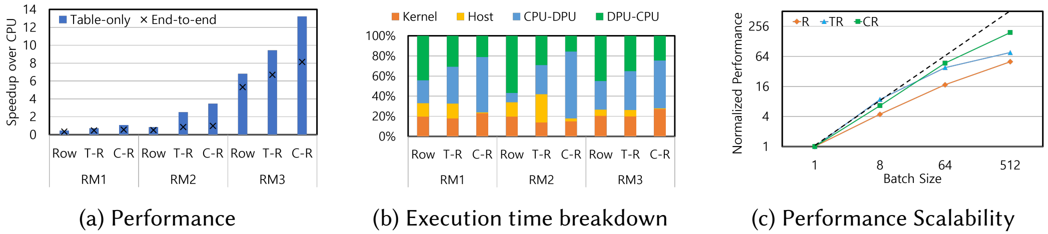
However, the *overall* speedup is lower for row-wise partitioning, as the amount of time for `DPU-CPU` transfers, i.e., the communication overhead to transfer partially reduced data back to the host, and `Host` time, i.e., the reduction computation in the CPU, increases as the number of DPUs increases. With row-wise partitioning, partial reduction is done across different DPUs and the partially reduced output data needs to be transferred between the DPUs and the host. As the number of DPUs increases, the amount of data transferred back to the host will also increase proportionally – thus, explaining the slowdown as the number of DPUs reaches 64 and 128 with row-wise partitioning. In comparison, column-wise partitioning continues to show overall performance improvements, while kernel time decreased when employing more DPUs, the total size of data needed to transfer remained unchanged when the number of DPUs changed.

Another fundamental limitation of column partitioning is that a column (even with just a single dimension) can exceed the memory capacity of a single PIM bank for large tables. As a result, a *hierarchical* partitioning needs to be used – e.g., a group of nodes or DPUs are used for the first-level partition scheme and then, among the groups, a different type of partitioning scheme is used. Overall scalability is shown in Figure 10b. Column-wise partitioning cannot scale beyond 128 DPUs due to the embedding table's dimension size. To address this limitation, hybrid (hierarchical) partitioning is employed, where a given column is horizontally partitioned (i.e., row-wise) to provide more scalability. However, both row-wise and hybrid partitioning suffer from communication and host computation overhead, resulting in a similar performance degradation.

OBSERVATION #5: ***Hybrid partitioning is necessary to scale embedding table kernel across a large number of PIM nodes. While hybrid partitioning performs better than naive partitioning (e.g., row partitioning), the scalability is still limited because of inter-PIM communication.***

	RM1	RM2	RM3
Embedding Dimension (Width)	32	32	32
Hash Size (Height)	16M	16M	160M
Avg. Pooling Factor	80	80	20
# of Tables	10	40	5
Total Table Size	20GB	80GB	100GB

Table 3. Different embedding table configurations based on production-scale [34] used in our evaluation.



(a) Performance (b) Execution time breakdown (c) Performance Scalability

Fig. 11. (a) Performance comparison of different partitioning schemes on large-scale embedding tables and end-to-end DLRM models, (b) execution time breakdown, and (c) performance scalability comparison between different partitioning schemes for RM2. `C-R`: Column-Row, `T-R`: Table-Row hybrid partitioning.

4.3.3 DLRM-based Embedding Table Evaluation: In addition to synthetic embedding tables, we evaluate the performance of a large-scale embedding table configuration that is based on production-size recommendation models [34]. We scaled it to fit within the UPMEM server memory capacity. The configuration of the three different types of recommendation models is summarized in Table 3. Figure 11a shows a performance comparison of UPMEM with a CPU for embedding table lookup-only and end-to-end DLRM for different partitioning schemes. Performance is normalized to the execution time of the CPU baseline without any PIM. For production model-based evaluations, we used a batch size of 16 to model inference where smaller batch sizes are used. For embedding table-only evaluation, row-wise partitioning results in a significant performance degradation (up to 54% for the RM1 model) because of the significant overhead from data movement from the DPU back to the CPU host, as well as host computation (`Host`) (Figure 11b). Adopting hierarchical (hybrid) approaches can provide performance benefits – up to 13.2× for RM3. However, even for hybrid partitioning, row-wise partitioning is still necessary to scale out embedding table recommendations across a large number of nodes. Note that host computation (as well as DPU to CPU communication) represents a significant fraction of the execution time. For end-to-end DLRM evaluation, we assume that only the embedding table kernels are offloaded and the other operations of DLRM (e.g., MLP) are done on the host CPU in the UPMEM implementation. Speedup over CPU is shown in RM3 for different partitioning schemes. `C-R` hybrid partitioning can provide more than 8× improvement for RM3, as even row partitioning outperforms baseline CPU. Figure 11c shows performance scalability across different partitioning schemes for RM2 as the batch size is increased. While there is performance improvement or speedup as the batch size is increased, row-based partitioning (`R`) results in lower performance benefits as the batch size is scaled. However, hybrid partitioning is able to achieve higher performance and approach linear speedup. To scale embedding tables across a large number of nodes with PIM capability, row-wise partitioning is necessary but this

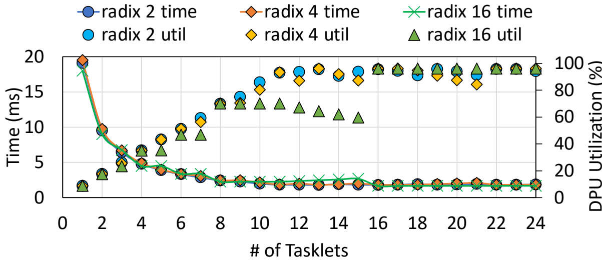


Fig. 12. Single DPU NTT ($N = 2^8$) runtime and utilization with various tasklet counts across different radix configurations.

Butterfly radix	**2**	**4**	**16**
Butterfly/NTT stage	128	64	16
Butterfly size	1	4	32
# of sync	7	3	1

Table 4. Different radix butterfly comparison for 256-point NTT.

leads to the need to perform a *global* reduction between the PIM modules. However, such *inter-PIM communication or global communication can only be done through the host in modern PIM architectures and severely limits scalability.*

5 NTT KERNEL

5.1 UPMEM Implementation

We implemented the Number Theoretic Transform (NTT) for the UPMEM architecture and verified its functional correctness by comparing it with CPU implementation in [4]. We implemented the NTT kernel based on an iterative FFT algorithm and merged Cooley–Tukey NTT optimization from [69]. We started with a single DPU implementation of the NTT with $N = 2^8$ where N is the number of coefficients. We changed the number of tasklets for each butterfly radix, sweeping the value from 1 to 24, to evaluate NTT performance and the DPU pipeline utilization. Using hierarchical NTT [8], we scaled the NTT to $N = 2^{16}$. Our initial implementation is based on 2D NTT and based on our results, we extrapolated the results to 3D and 4D NTT. However, because of the communication overhead, 2D NTT results are the most optimal performance on UPMEM and we use 2D NTT for the rest of our evaluation. We evaluated the 2D NTT kernel through a strong scaling experiment, where we varied the number of DPUs while maintaining a fixed problem size. Additionally, we experimented with variations in the $\log Q$ value which corresponds to the number of limbs (i.e., the number of NTTs). Finally, we evaluated the homomorphic multiplication of the CKKS scheme on the UPMEM system.

5.2 NTT Evaluations

5.2.1 Single DPU Evaluation. Figure 12 shows NTT evaluation on a single DPU with various numbers of tasklets and radix configurations. The x-axis shows the number of hardware tasklets and the right side y-axis shows the DPU pipeline utilization with 100% representing that the DPU is executing 1 instruction per cycle. The left-side y-axis shows the kernel performance (execution time) and thus, lower is better. We evaluate the implementation of radix 2, 4, and 16 and use 256-point NTT since it is the smallest (larger than 16) NTT that can be broken down into radix 2, 4, and 16. The number of butterfly operations per stage, the size of each butterfly operation, and the necessary thread synchronization vary depending on the radix. The highest radix (i.e. radix-16) has the fewest stages (with the largest butterfly operations size per stage) and results in the smallest number of synchronizations. The impact on the NTT implementation from the different radix sizes is shown in Table 4.

M-point NTT using radix-k implementation results in $\log_k(M)$ stages, with each stage consisting of M/k butterfly operations. During each NTT stage, butterfly operations will be assigned to each

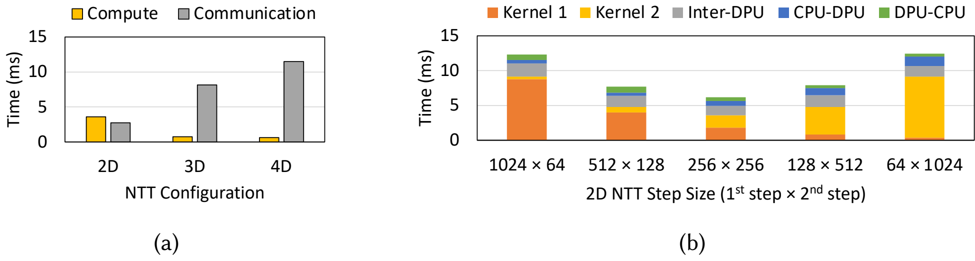


Fig. 13. Analysis of (a) hierarchical NTT and (b) different step size implementation for 2D NTT.

tasklet in a round-robin fashion, and then synchronize using a barrier API provided by UPMEM, with other tasklets, before the next NTT stage. Figure 12 shows that the pipeline starts to saturate around 11 tasklets for radix-2 and radix-4. However, for radix-16, the utilization actually drops significantly before reaching 16 tasklets. The reduction in utilization is caused by workload fragmentation with the amount of fragmentation equaling ($M/k \mod T$) where T is the number of tasklets. Thus, while fragmentation also occurs for radix-2 and 4, the amount of fragmentation is much lower and the impact on utilization is also very small. In general, more tasklets result in higher performance and higher radix, when "balanced" without any fragmentation across the tasklets, provides the highest performance) For example, in the 16 tasklets implementation, radix-16 performs better than radix-2 by approximately 8% from fewer synchronizations and better temporal locality.

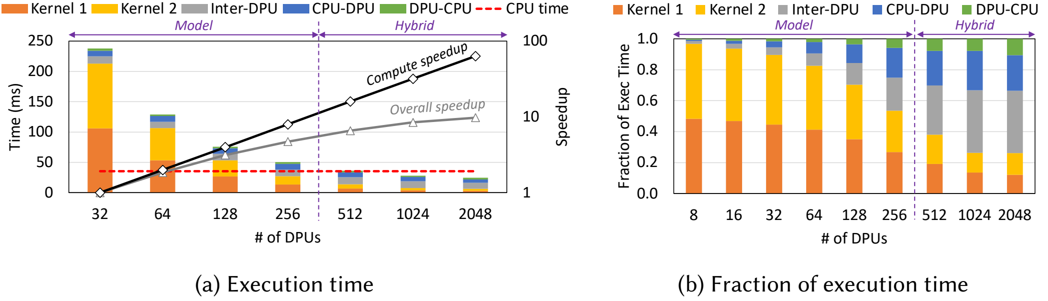
OBSERVATION #6: ***High-radix NTT results in the highest performance if the butterfly operations are load-balanced across the tasklets and maximize compute utilization.***

5.2.2 Hierarchical NTT exploration. Hierarchical NTT is often used to partition a large NTT into smaller-sized NTTs. The total number of computations does not change but hierarchical NTT partitions NTT into multiple "steps" or dimensions – with smaller NTT size calculations performed within each step or dimension. We refer to the local NTT size as the *step size.* [2] Increasing the number of steps or dimensions partitions the NTT into smaller step size and allows parallelism to be exploited. However, there is a trade-off as synchronization is needed after each step. In the UPMEM architecture, the overhead of synchronization within a DPU is relatively small but if synchronization is needed between the DPUs, the overhead is more significant since synchronization is effectively done through the host CPU.

The two main components of NTT are the computation (or kernel computation on UPMEM DPU) and the communication or synchronization between the DPUs. To understand the impact of these two components, the compute and the communication are measured on UPMEM to analyze the trade-off between the two components. As the NTT dimension size increases, the NTT compute is measured as well as the communication of the data to/from the host for the synchronization. As shown in Figure 13a, as the number of dimensions increases, the computation time is reduced as the additional amount of parallelism with higher dimensions can be exploited with a larger number of DPUs (i.e., using 256 DPUs for 2D but leveraging 2048 DPUs for 4D). However, the communication time increases as more synchronization is needed after each step – thus, overall execution time increases.

Analysis of different step sizes for 2D NTT is shown in Figure 13b. `CPU-DPU` and `DPU-CPU` represent communication to/from the DPU while `Kernel1` and `Kernel2` represent the two steps of the 2D NTT. The `Inter-DPU` is the communication time between the DPU that occurs between

[2] For a hierarchical NTT, the step size for each dimension can differ.



(a) Execution time

(b) Fraction of execution time

Fig. 14. Strong scaling experiment results of 8 NTTs ($N = 2^{16}$) with model- and hybrid-level parallelism.

Kernel1 and Kernel2. The different step sizes for the two kernels result in an unbalanced compute time between the kernels – thus, a balanced step size results in optimal performance and minimizes overall execution time.

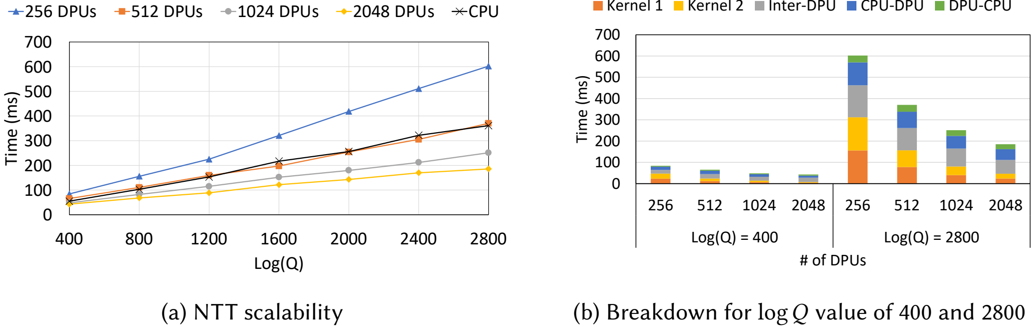
OBSERVATION #7: ***High-dimensional, hierarchical NTT reduces the local NTT size and increases the parallelism but increases DPU-level synchronization. Our analysis demonstrates a "balanced" 2D NTT provides a balance between computation and communication.***

5.2.3 Multi-DPU Evaluations. We implemented 2D NTT – i.e., a two-computational step NTT with each step composed of 256× 256-point NTTs using 16 tasklets for each DPU. Each tasklet performs a single radix-16 NTT per stage to increase data reuse and minimize thread synchronization (see Section 5.2.1). We carried out a strong scaling experiment from 32 to 2048 DPUs. Given the limit of 256 DPUs for partitioning a 2D 2^{16}-point NTT, we evaluated eight independent NTTs in parallel, to scale up to 2048 DPUs. For DPUs ranging from 32 to 256, we used model-level parallelism, distributing each 256-point NTT among the DPUs. To scale beyond 256 DPUs, we combined model- and input-level parallelism (shown as *Hybrid* in Figure 14). This involved partitioning the input NTTs and distributing them across multiple sets of 256 DPUs. For instance, to execute 8 NTTs with 512 DPUs, the first 4 NTTs are allocated to a set of 256 DPUs, while the remaining 4 NTTs are assigned to another set of 256 DPUs, thus in total 512 DPUs are allocated. Consequently, each DPU computes 4× 256-point NTT in every kernel.

The MRAM stores the input and twiddle factors (i.e., sets of integer constants to facilitate the transformation from one domain to another). Data transfer to the WRAM occurs prior to NTT execution, and afterward, the results are returned to the MRAM. In each DPU, we further break down the 256-point NTT through an additional level of 2D NTT, resulting in two computational steps of 16× 16-point NTTs. Each tasklet computes a 16-point NTT (i.e., radix-16 butterfly operation) for each step. Thus, the NTT is effectively a hierarchical 4D NTT calculation. With this NTT configuration, only one DPU synchronization is required between the first and second steps (i.e., kernel). Following the DPU synchronization, an All-to-all communication is conducted to shuffle data among the DPUs before the second kernel.

The scalability of the NTT kernel as the number of DPUs increases is shown in Figure 14a. CPU-DPU and DPU-CPU represent communication to and from the DPU, while Kernel1 and Kernel2 represent the two compute steps for the 2D NTT. The Inter-DPU represents the time for communication between the DPU that occurs between Kernel1 and Kernel2. [3] As the number of DPUs

[3]Even though the communication occurs through the host in the UPMEM PIM architecture, it is categorized differently to isolate the impact of inter-DPU communication on overall performance.



(a) NTT scalability

(b) Breakdown for $\log Q$ value of 400 and 2800

Fig. 15. NTT evaluation with varying $\log Q$ (i.e., number of limbs) and number of DPUs.

increases, the compute time (i.e., computation time from UPMEM's performance measurement cycle counter) scales almost linearly – approximately 63.5× speedup as the number of DPUs increases from 32 to 2048. However, the overall speedup is limited to approximately 9.7×. For a smaller number of DPUs, `Inter-DPU` is negligible but represents approximately 40.3% of the total execution time for 2048 DPUs (Figure 14b). Thus, as the compute time linearly decreases, communication becomes a bigger bottleneck in overall performance.

OBSERVATION #8: ***For strong scaling, DPU kernel execution time scales linearly with the number of DPUs for NTT but inter-DPU overhead does not scale linearly and limits overall scalability.***

5.2.4 NTT Scalability: The coefficient modulus size or $\log Q$ is an important FHE parameter, which has a direct effect on the multiplicative depth (i.e., how many multiplications can be done before bootstrapping is needed). The value of $\log Q$ impacts the number of limbs and in this experiment each limb corresponds to an NTT. In Figure 15 we vary the $\log Q$ size across four UPMEM implementations (256, 512, 1024, and 2048 DPUs) and CPU implementation. Figure 15a shows the scalability for NTT and all implementations scale linearly. The 2048 DPUs configuration results in the best performance across all values of $\log Q$. Figure 15b shows the execution time breakdown for two values of $\log Q$. While inter-DPU time continues to dominate the total execution time for the 2048 DPUs, the kernel runtime scales linearly with the number of DPUs. Inter-DPU time for the 2048 DPUs is faster than the 256 DPUs due to higher total memory bandwidth.

The inter-DPU communication for NTT, which consists of copying intermediate data from the DPU back to the host, re-arranging the data, and then, re-distributing the data or moving the data back to the DPU, is effectively a "software" implementation of All-to-All communication in modern PIM architecture. As the results show, it results in performance overhead and limits scalability – i.e., it represents 34.9% of the total execution time for 2048 DPUs with $\log Q = 2800$.

OBSERVATION #9: *With hybrid parallelism, the maximum number of DPUs provides the highest performance because of the additional parallelism (both memory bandwidth and compute); however, scalability can still be improved if the inter-DPU communication overhead is reduced.*

We also evaluated NTT within an end-to-end homomorphic multiplication of the CKKS scheme from [36] on the UPMEM system, and results are shown in Figure 16. The following parameters were used: $N = 65536$, $L = 23$, 8 extension limbs, 54-bit q, and $dnum = 3$ [2]. The performance of each primitive operation on the homomorphic multiplication of the CKKS scheme was measured

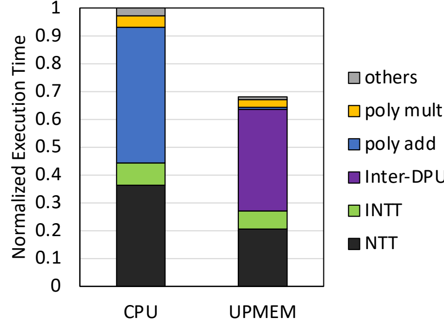
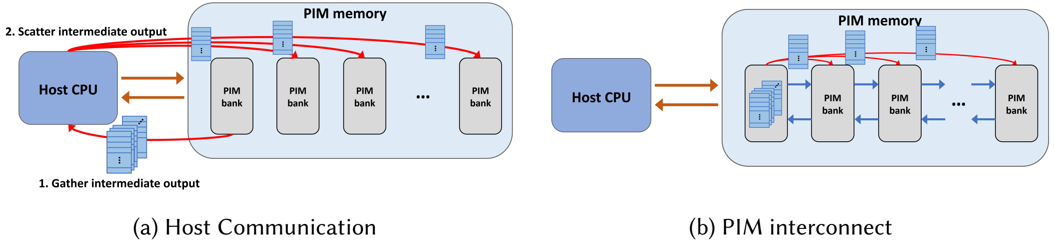


Fig. 16. End-to-end CKKS homomorphic multiplication evaluation on the UPMEM system.



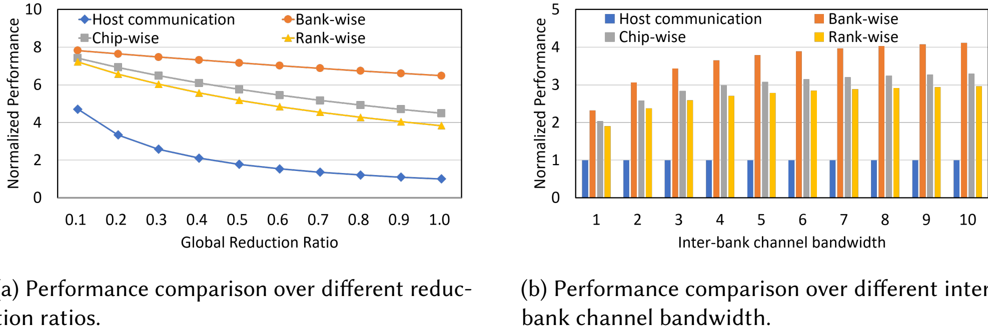
(a) Host Communication (b) PIM interconnect

Fig. 17. A high-level block diagram of (a) conventional PIM system where communication occurs through the host and (b) PIM system with PIM interconnect that enables PIM-to-PIM communication.

separately, based on its sequence of occurrence Compute-wise, UPMEM provides speedup compared to CPU across all primitive functions. The significant amount of speedup on the polynomial addition (`poly add`) primitives is primarily due to the high compute throughput provided by the PIM core. However, the absence of an integer multiplication unit in UPMEM [72], results in limited compute speedup for any kernel involving modular multiplication (e.g., `poly mult` and NTT). In addition, the significant `Inter-DPU` portion in Figure 16 is caused by synchronization in NTT and its inverse.

6 DISCUSSION: CASE FOR PIM INTERCONNECT

A key aspect of PIM systems to enable scalable performance with a large number of PIM nodes is inter-PIM communication. In this section, we argue that ***future PIM architectures need hardware/software support for PIM-to-PIM communication (or PIM interconnect) to enable scalability.*** The need for communication between nodes is not new in a distributed (parallel) system [16] and interconnection networks [18] are commonly found in many systems to enable such communication. Similar communication is necessary in a PIM system but current PIM systems communicate "indirectly" through the host CPU. To the best of our knowledge, this is one of the first work to demonstrate the limitations of a scalable PIM system and argue how an interconnection network is necessary even in a PIM-based system. A high-level block diagram of the potential benefit of PIM interconnect is shown in Figure 17 with an example of All-to-all operation. In today's PIM systems, any communication between the PIM banks/nodes needs to occur through the host (Figure 17a). This creates a bottleneck in the host-to-memory interface and more importantly, the communication (and any additional processing at the host) is serialized through the host. In comparison, a PIM interconnect enables low latency communication directly between



(a) Performance comparison over different reduction ratios.

(b) Performance comparison over different inter-bank channel bandwidth.

Fig. 18. Modeling of the potential performance benefit of PIM interconnect compared to PIM communication through the host.

the PIM banks (Figure 17b) and more importantly, enables *parallel* communication. For simplicity, data communication from only the left PIM node is shown in the figure. We outline some of the hardware/software challenges for PIM interconnect in this section.

Hardware Challenges: Unlike traditional interconnection networks [18] such as those used in large-scale supercomputers or network-on-chip, the constraints of PIM and DRAM are very different, and traditional interconnect architectures cannot be leveraged. Especially, because of the limited amount of logic and the limited bandwidth available, a conventional hardware-based interconnect router microarchitecture is not necessarily appropriate. To enable practical PIM interconnect, the following principles need to be exploited.

- **Hierarchical Network:** The hierarchical packaging constraints (bank, chip, rank) will require a hierarchical interconnect organization to match the packaging constraints. However, given the limited pin bandwidth available in a DRAM system, existing bandwidth, including PIM-host bandwidth, will likely need to be shared to enable efficient PIM interconnect.
- **Low Cost:** Buffered router-based interconnect is likely not feasible because of the high cost of input buffers. Thus, an efficient, minimally buffered router approach [44] is necessary to enable PIM interconnect.

Software Challenges: In addition to hardware support, including channels and any “router” logic, software support is necessary to enable practical PIM-to-PIM communication. In particular, we highlight the following main challenges.

- **Minimal impact on DDR interface:** Any PIM-to-PIM communication needs to minimize its impact on the DDR interface, similar to how modern PIM systems [22, 49, 55] have been designed. Hence, remote accesses will need to be viewed like local Load/Store operations but enable PIM interconnect access through appropriate PIM instructions.
- **Deterministic behavior:** Introducing non-determinism can be problematic for the host (CPU) and PIM interface (as well as the DDR interface). Therefore, the software (and the hardware) needs to guarantee deterministic behavior to ensure minimal impact on the programming and usage of the PIM-to-PIM communication.

To understand the potential benefit of PIM interconnect, we model PIM interconnect to analyze potential performance benefits using the embedding table kernel (with hybrid partitioning policy (`C-R`) and the results are shown in Figure 18. The key components that are modeled are the host-PIM bandwidth as well as the PIM-to-PIM bandwidth that is enabled with the PIM interconnect. For host-based communication, the model assumes global reduction is performed by reading partial

embedding outputs. For the analysis, we ignore the cost of the global reduction within the host and estimate the performance based on the amount of data transferred and the host-PIM bandwidth. For the PIM interconnect, we assume a point-to-point interconnect is available between the PIM banks to create a hierarchical network, across the bank, chip, and rank hierarchy. We vary two components to determine the potential benefit of PIM interconnect – (a) global reduction ratio or the amount of computation that would need to be done by the host and (b) PIM interconnect (or PIM-to-PIM channel) bandwidth.

Figure 18a shows the performance improvement of PIM interconnect over host communication as the *global* reduction ratio is varied. We analyze the benefit of PIM interconnect across three different numbers of PIM nodes — (1) Bank-wise where 8 PIM banks are within the same DRAM chip, (2) Chip-wise where 64 PIM banks are within the same rank, and (3) Rank-wise where 256 PIM banks across 4 DRAM ranks, need to communicate with each other. Performance (or inverse of time) is shown and thus, higher is better and the results are normalized to the host communication with a global reduction ratio of 1. Not shown but when the ratio is 0, no inter-PIM reduction is necessary as all reduction is done locally and thus, the performance between the two architectures is identical. However, when the value is 1, all reduction is done globally and maximizes the difference between the two architectures. When there is a small amount of global reduction, the performance benefit is only 66% but when most of the reduction is global, the potential benefits from PIM interconnect can be approximately 6.5×.

For this analysis, we assumed the inter-bank channel bandwidth of the PIM interconnect is 2.8 GB/s (64b link width at 350MHz). Figure 18b shows how the performance of PIM interconnect changes over different inter-bank channel bandwidths of PIM interconnect. For this analysis, we assumed a 50% reduction ratio and varied inter-channel bandwidth from 0.35GB/s to 3.5GB/s (x-axis is normalized to 0.35GB/s) and analyzed performance improvement of embedding table lookup with hybrid partitioning that requires different scope of AllReduce. Even with 0.35GB/s of bandwidth, PIM interconnect shows significant performance improvement over host communication. This analysis demonstrates how PIM interconnect can improve communication and overall performance, even when assuming limited bandwidth for PIM interconnect.

OBSERVATION #10: ***PIM interconnect that provides direct communication between PIM bank (or nodes) is critical to enable scalable performance. PIM interconnect will not only require hardware support but also software support to ensure the DRAM interface can support PIM-to-PIM communication.***

7 CONCLUSION

Processing-in-memory (PIM) architecture enables the acceleration of emerging workloads by minimizing data movement as computation is moved near the data. In this work, we addressed the potential PIM scalability challenges as the number of PIM nodes increases and communication is necessary between the PIM nodes. In particular, we observed that performance scalability in PIM can be achieved when load-balancing occurs across multiple PIM nodes; however, we also identified on real UPMEM PIM hardware that workload scalability can be limited when communication (e.g., collective communication) is needed and occurs through the host (or the CPU). Based on the analysis, we present the case for PIM interconnect or an interconnection network between the PIM nodes and argue how it can benefit next-generation PIM architectures to provide scalable performance.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and the shepherd for their insightful comments that improved the quality of the paper. This research was supported in part by Korea Institute of Science and Technology Information(KISTI)(No. K-23-L02-C06), NRF-2023R1A2C200422911, IITP No.RS-2023-00228255, Experiential AI and the NSF IUCRC Center for Hardware and Embedded Systems Security and Trust (CHEST), NSF CNS 2312275, NSF CNS 2312276, and in part by grant RYC2021-031966-I funded by MCIN/AEI/10.13039/501100011033 and the "European Union NextGenerationEU/PRTR."

REFERENCES

[1] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. 2021. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 802–814. https://doi.org/10.1109/HPCA51647.2021.00072

[2] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An fpga-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 882–895. https://doi.org/10.1109/HPCA56546.2023.10070953

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. https://doi.org/10.1145/2749469.2750386

[4] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 53–63. https://doi.org/10.1145/3560827.3563379

[5] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. 2018. Application-transparent near-memory processing architecture with memory channel network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 802–814. https://doi.org/10.1109/MICRO.2018.00070

[6] Michael J. Anderson, Benny Chen, Stephen Chen, Summer Deng, Jordan Fix, Michael Gschwind, Aravind Kalaiah, Changkyu Kim, Jaewon Lee, Jason Liang, Haixin Liu, Yinghai Lu, Jack Montgomery, Arun Moorthy, Nadathur Satish, Sam Naghshineh, Avinash Nayak, Jongsoo Park, Chris Petersen, Martin Schatz, Narayanan Sundaram, Bangsheng Tang, Peter Tang, Amy Yang, Jiecao Yu, Hector Yuen, Ying Zhang, Aravind Anbudurai, Vandana Balan, Harsha Bojja, Joe Boyd, Matthew Breitbach, Claudio Caldato, Anna Calvo, Garret Catron, Sneh Chandwani, Panos Christeas, Brad Cottel, Brian Coutinho, Arun Dalli, Abhishek Dhanotia, Oniel Duncan, Roman Dzhabarov, Simon Elmir, Chunli Fu, Wenyin Fu, Michael Fulthorp, Adi Gangidi, Nick Gibson, Sean Gordon, Beatriz Padilla Hernandez, Daniel Ho, Yu-Cheng Huang, Olof Johansson, Shishir Juluri, and et al. 2021. First-generation inference accelerator deployment at facebook. *CoRR* abs/2107.04140 (2021). arXiv:2107.04140 https://arxiv.org/abs/2107.04140

[7] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783753

[8] D. H. Bailey. 1989. FFTs in external or hierarchical memory. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. 234–242. https://doi.org/10.1145/76263.76288

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[10] Sébastien Cayrols, Jiali Li, George Bosilca, Stanimire Tomov, Alan Ayala, and Jack Dongarra. 2022. Lossy all-to-all exchange for accelerating parallel 3-d ffts on hybrid architectures with gpus. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 152–160. https://doi.org/10.1109/CLUSTER51413.2022.00029

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[13] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[14] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. Association for Computing Machinery, New York, NY, USA, 191–198. https://doi.org/10.1145/2959100.2959190

[15] Richard Crandall and Barry Fagin. 1994. Discrete weighted transforms and large-integer arithmetic. *Math. Comput.* 62 (1994), 305–324. https://doi.org/10.2307/2153411

[16] David Culler, Jaswinder Pal Singh, and Anoop Gupta. 1998. *Parallel computer architecture: A hardware/software approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[17] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.

[18] William Dally and Brian Towles. 2004. Principles and practices of interconnection network. (01 2004).

[19] William J. Dally, Stephen W. Keckler, and David B. Kirk. 2021. Evolution of the graphics processing unit (gpu). *IEEE Micro* 41, 6 (2021), 42–51. https://doi.org/10.1109/MM.2021.3113475

[20] W. J. Dally and B. Towles. 2001. Route packets, not wires: On-chip interconnection networks *(DAC '01)*. ACM, New York, NY, USA, 684–689. https://doi.org/10.1145/378239.379048

[21] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (jun 2020), 48–57. https://doi.org/10.1145/3361682

[22] Fabrice Devaux. 2019. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*. IEEE, 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[23] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2018. Bandana: Using non-volatile memory for storing deep learning models. arXiv:1811.05922 [cs.LG]

[24] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[25] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 283–295. https://doi.org/10.1109/HPCA.2015.7056040

[26] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 113–124. https://doi.org/10.1109/PACT.2015.22

[27] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 751–764. https://doi.org/10.1145/3037697.3037702

[28] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), 3:1–3:19. https://doi.org/10.1147/JRD.2019.2934048

[29] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 21 (feb 2022), 49 pages. https://doi.org/10.1145/3508041

[30] Steven Gottlieb, W. Liu, D. Toussaint, R. L. Renken, and R. L. Sugar. 1987. Hybrid-molecular-dynamics algorithms for the numerical simulation of quantum chromodynamics. *Phys. Rev. D* 35 (Apr 1987), 2531–2542. Issue 8. https://doi.org/10.1103/PhysRevD.35.2531

[31] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. 2019. Post-training 4-bit quantization on embedding tables. *CoRR* abs/1911.02079 (2019). arXiv:1911.02079 http://arxiv.org/abs/1911.02079

[32] Saransh Gupta, Rosario Cammarota, and Tajana Rosing. 2022. MemFHE: End-to-end computing with fully homomorphic encryption in memory. *arXiv preprint arXiv:2204.12557* (2022).

[33] Saransh Gupta and Tajana Šimunić Rosing. 2021. Accelerating fully homomorphic encryption with processing in memory. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1335–1338.

[34] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[35] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a new paradigm: experimental analysis and characterization of a real processing-in-memory system. *IEEE Access* 10 (2022), 52565–52608. https://doi.org/10.1109/ACCESS.2022.3174101

[36] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 364–390.

[37] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. https://doi.org/10.1145/3282307

[38] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[40] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[41] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating personalized recommendation with near-memory processing. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 790–803. https://doi.org/10.1109/ISCA45697.2020.00070

[42] 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, and Hsien-Hsin S. Lee. 2022. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro* 42, 1 (2022), 116–127. https://doi.org/10.1109/MM.2021.3097700

[43] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 145–155. https://doi.org/10.1109/PACT.2013.6618812

[44] John Kim. 2009. Low-cost router microarchitecture for on-chip networks. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 255–266. https://doi.org/10.1145/1669112.1669145

[45] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. 2022. Aquabolt-xl hbm2-pim, lpddr5-pim with in-memory processing, and axdimm with acceleration buffer. *IEEE Micro* 42, 3 (2022), 20–30. https://doi.org/10.1109/MM.2022.3164651

[46] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. 2021. Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. https://doi.org/10.1109/HCS52781.2021.9567191

[47] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. Bts: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 711–725.

[48] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963

[49] Yongkee Kwon, Guhyun Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, Jeongbin Kim, Changhyun Kim, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyeongdeok Lee, Seungyeong Park, Wonjun Lee, Seongju Lee, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, John Kim, Euicheol Lim, and Junhyun Chun. 2023. Memory-Centric Computing with SK Hynix's Domain-Specific Memory. In *2023 IEEE Hot Chips 35 Symposium (HCS)*. 1–26. https://doi.org/10.1109/HCS59251.2023.10254717

[50] 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 the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 740–753. https://doi.org/10.1145/3352460.3358284

[51] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2021. Tensor casting: Co-designing algorithm-architecture for personalized recommendation training. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 235–248. https://doi.org/10.1109/HPCA51647.2021.00029

[52] Youngeun Kwon and Minsoo Rhu. 2022. Training personalized recommendation systems from (gpu) Scratch: Look forward not backwards. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 860–873. https://doi.org/10.1145/3470496.3527386

[53] Young-Cheon Kwon, Jaehoon Lee, Suk Han fhand Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 64. 350–352. https://doi.org/10.1109/ISSCC42613.2021.9365862

[54] Dominique Lavenier, Jean-Francois Roy, and David Furodet. 2016. DNA mapping using processor-in-memory architecture. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 1429–1435.

[55] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware architecture and software stack for pim based on commercial dram technology : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. https://doi.org/10.1109/ISCA52012.2021.00013

[56] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. 2021. MERCI: Efficient embedding reduction on commodity hardware via sub-query memoization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3445814.3446717

[57] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23.

[58] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 993–1011. https://doi.org/10.1145/3470496.3533727

[59] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems* 67 (2019), 28–41.

[60] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[61] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep learning recommendation model for personalization and recommendation systems. https://doi.org/10.48550/ARXIV.1906.00091

[62] Hamid Nejatollahi, Saransh Gupta, Mohsen Imani, Tajana Simunic Rosing, Rosario Cammarota, and Nikil Dutt. 2020. Cryptopim: In-memory acceleration for lattice-based cryptographic hardware. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[63] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, et al. 2021. A case study of processing-in-memory in off-the-shelf systems.. In

USENIX Annual Technical Conference. 117–130.
[64] NVidia. 2012. cuBLAS library. https://docs.nvidia.com/cuda/cublas/
[65] 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 *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 268–281. https://doi.org/10.1145/3466752.3480080
[66] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *Progress in Cryptology – LATINCRYPT 2015*, Kristin Lauter and Francisco Rodríguez-Henríquez (Eds.). Springer International Publishing, Cham, 346–365.
[67] Dayane Reis, Jonathan Takeshita, Taeho Jung, Michael Niemier, and Xiaobo Sharon Hu. 2020. Computing-in-memory for performance and energy-efficient homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 11 (2020), 2300–2313.
[68] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alexander Sergeev, and Michael Matheson. 2022. Accelerating collective communication in data parallel training across deep learning frameworks. (4 2022). https://www.osti.gov/biblio/1862153
[69] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact ring-lwe cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 371–391.
[70] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annu. IEEE/ACM Int. Symp. on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070
[71] UPMEM SAS. 2021. The pim reference platform. Retrieved October 31, 2022 from https://www.upmem.com/technology/
[72] UPMEM SAS. 2021. UPMEM documentation. Retrieved October 31, 2022 from https://sdk.upmem.com/2021.3.0/
[73] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. 2022. Täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 42–58. https://doi.org/10.1145/3470496.3527379
[74] Harold S. Stone. 1970. A logic-in-memory computer. *IEEE Trans. Comput.* C-19, 1 (1970), 73–78. https://doi.org/10.1109/TC.1970.5008902
[75] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. ABC-DIMM: Alleviating the bottleneck of communication in dimm-based near-memory processing with inter-dimm broadcast. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 237–250. https://doi.org/10.1109/ISCA52012.2021.00027
[76] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740
[77] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.* 19, 1 (feb 2005), 49–66. https://doi.org/10.1177/1094342005051521
[78] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. 2022. EL-rec: Efficient large-scale recommendation model training via tensor-train embedding table. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41404.2022.00075
[79] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. https://doi.org/10.48550/ARXIV.1609.08144
[80] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 49–72.
[81] Huasha Zhao and John Canny. 2013. Sparse allreduce: Efficient scalable communication for power-law data. arXiv:1312.3020 [cs.DC]
[82] Huasha Zhao and John Canny. 2014. Kylix: A sparse allreduce for commodity clusters. In *2014 43rd International Conference on Parallel Processing*. 273–282. https://doi.org/10.1109/ICPP.2014.36
[83] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 43–51.

[84] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1059–1068.

[85] Z. Zhou, C. Li, F. Yang, and G. Sun. 2023. DIMM-Link: Enabling efficient inter-dimm communication for near-memory processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 302–316. https://doi.org/10.1109/HPCA56546.2023.10071005

Received February 2023; revised January 2024; accepted January 2024