AUT Journal of Electrical Engineering
10.22060/EEJ.2026.26237.6085

Efficient Dense Matrix Multiplication on UPMEM Processing-in-Memory Architecture: Optimization and Performance Analysis

Elham Cheshmikhani[1, *]

[1]Assistant Professor, Department of Computer Science and Engineering, Shahid Beheshti University, 1983969411 Tehran, Iran
*Corresponding author: Elham Cheshmikhani, email: e_cheshmikhani@sbu.ac.ir

Abstract:

Matrix multiplication is a fundamental operation in image processing and many data-intensive applications, and its growing computational and memory demands increasingly challenge conventional CPU- and GPU-based systems. Processing-near/in-Memory (PnM/PiM) architectures address this challenge by reducing data movement between memory and computation units. UPMEM is a commercially available DRAM-based PIM platform that integrates lightweight processors within memory chips; however, efficiently mapping compute-intensive kernels such as dense matrix multiplication onto this architecture remains non-trivial due to limited arithmetic support, restricted on-chip memory, and low per-core frequency. In this work, we investigate the feasibility and performance of dense matrix multiplication on UPMEM. Starting from naive implementation, we apply loop reordering and tiling optimizations tailored to the UPMEM execution model. We analyze the impact of key architectural parameters, including the number of DPUs and tasklets, and compare UPMEM-based implementations against optimized CPU baselines under identical input sizes and precision constraints. Experimental results show that exploiting tasklet-level parallelism and DPU scalability yields significant performance improvements, with the tiling-based UPMEM implementation achieving up to 2.76x speedup over CPU tiling and outperforming loop-reordered CPU implementations for large matrices using 32-bit arithmetic.

Keywords:

Processing in Memory (PiM), Processing near Memory (PnM), UPMEM, Memory wall, Von Neumann bottleneck, Data-intensive computing, Memory-centric architecture.

1. Introduction

Classical computer architectures based on the von Neumann model (Figure 1.a) suffer from a fundamental inefficiency, the physical separation between the processing unit (CPU) and main memory. This separation gives rise to the well-known *memory wall*, a situation where memory access latency and limited memory bandwidth significantly degrade performance, especially for workloads that frequently move large volumes of data between memory and processor [1, 2]. In these data-intensive applications, the overhead of data movement becomes the dominant performance bottleneck. One canonical example is matrix multiplication, a core building block in numerous image processing tasks such as filtering, transformation, and compression. In such algorithms, input data must be repeatedly fetched from memory and results written back, leading to substantial latency, energy consumption, and throughput limitations when executed on traditional CPU-based systems.

To address these limitations, the paradigms of Processing-near-Memory (PnM) and Processing-in-Memory (PiM), as shown in Figure 1.b and 1.c, respectively, have emerged as promising alternatives, aiming to bring computation closer to where the data resides (memory banks in gray color). In PnM/PiM architectures, Processing Elements (PEs) are integrated either near or within memory chips, substantially reducing, or even eliminating, the need for frequent data transfers between separate memory and CPU. As a result, PnM/PiM can dramatically decrease memory latency and energy overhead, while enabling high parallelism for data-intensive workloads [3, 4].

Over the past decades, a variety of PnM/PiM designs have been proposed, ranging from DRAM-based to non-volatile memory (NVM) PiM, as well as hybrid approaches [5-7]. Recent reviews show that PnM/PiM architectures hold particular promise for workloads in domains such as machine learning, graph processing,

10.22060/EEJ.2026.26237.6085

databases, and scientific computing, i.e., tasks that are fundamentally data-bound rather than compute-bound [8, 9].

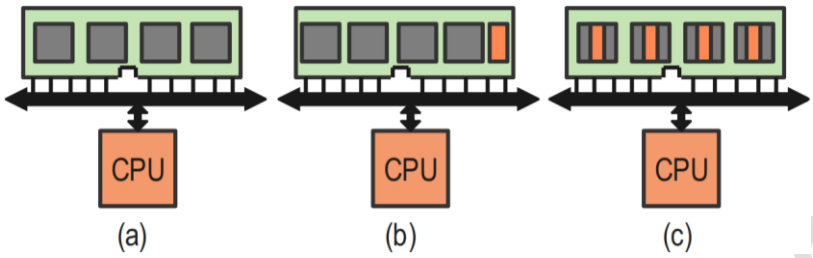

Figure 1. Processing architectures, a) von Neumann model, b) near-memory computing, and c) in-memory computing.

Among these, matrix operations, including matrix multiplication and convolution, stand out as natural targets for acceleration via PnM/PiM, because they involve large data sets and can benefit from massive parallelism and reduced data movement. For example, in/near-memory matrix/vector multiplication and convolution have been demonstrated using Memristive systems, achieving substantial speedups over conventional architectures [5, 10].

One of the most compelling commercial realizations of PnM is the UPMEM architecture, which integrates many small-scale processing units (called DPUs) directly with commodity DRAM chips [11, 12]. Each DPU co-locates a small in-order core, scratchpad or local memory, and part of the DRAM banks, enabling code to execute directly where data resides [9, 13]. In UPMEM-based systems, the aggregated internal memory bandwidth and parallelism scale with the number of DPUs, making them particularly promising for memory-bound workloads such as matrix multiplication [14, 15]. However, programming for PnM/PiM is not simply a matter of recompiling conventional CPU code. Several challenges impede efficient use of PnM/PiM architectures. For instance, many PnM/PiM systems favor fixed-point or limited-precision arithmetic, requiring quantization or adaptation of algorithms; data layout and memory mapping must respect the bank and bank-parallelism constraints of PnM/PiM memory; also, hybrid systems that combine

traditional DRAM and PnM/PiM memory must manage data movement between DRAM and PnM/PiM address spaces, which can reintroduce overheads [5, 11]. Furthermore, PnM/PiM-enabled systems must deal with issues of cache coherence, consistency, and efficient scheduling to avoid negating the benefits of in-memory computing [6, 8].

In this work, we explore the feasibility and performance of matrix multiplication within the UPMEM PnM framework, with a view toward image-processing applications. By carefully adapting the matrix multiplication algorithm, including data layout, chunking, mapping strategies, and precision considerations, we implement and optimize the operation for UPMEM. We then compare its performance against conventional CPU-based implementations, with a focus on latency.

Through this study, we aim to address the ambiguities, such as the extent to which PnM-based architectures like UPMEM can overcome the memory wall in data-intensive matrix operations. It can determine the main bottlenecks or limitations in practice (e.g., data mapping, precision, memory management). On the other hand, it can determine the way that future PnM/PiM systems or software toolchains can be designed to make such implementations more efficient and portable. Our experimental evaluation and analysis provide insight into these concerns, offering both practical data and design guidelines for leveraging PnM/PiM in image-processing and other matrix-heavy workloads.

2. Background and Preliminaries

Modern workloads continue to grow in scale and data intensity, exposing fundamental limitations of compute-centric systems in terms of memory bandwidth, latency, and energy efficiency. Addressing these challenges requires rethinking the balance between computation and data movement and exploring architectures that bring processing closer to where data resides. The concepts introduced here provide architectural context for PnM/PiM systems such as UPMEM, clarifying their advantages in overcoming conventional bottlenecks.

10.22060/EEJ.2026.26237.6085

This section presents the UPMEM PnM platform in depth, including its system organization, execution model, and the internal structure of its Data Processing Units (DPUs). This analysis forms the basis for the optimization strategies and experimental evaluations in later sections.

The concept of *Processing-near/in-Memory* dates back to the 1960s, when researchers identified fundamental limits of von Neumann architectures [16]. As workloads grew, the continuous data transfer between processor and memory became a major bottleneck, later termed the *memory wall*, where improvements in CPU speed alone yielded limited system gains [4, 8, 16].

In traditional systems, computation and storage are physically separated, requiring frequent data movement that incurs latency and energy overhead. This is especially problematic in data-intensive applications such as machine learning, big-data analytics, and scientific computing, where memory access dominates execution time [6]. Matrix multiplication is a representative example, requiring repeated access to large input matrices and intermediate results.

PIM addresses these limitations by integrating processing capability directly into memory. Lightweight compute units are embedded within memory chips, enabling local execution of operations such as arithmetic and logic. This reduces data movement, thereby lowering latency and improving energy efficiency [4].

A key feature of PIM is massive parallelism, where many simple processing units operate concurrently on different memory regions, making it highly suitable for data-parallel workloads [8]. Another key principle is data locality, since computation is performed where data is stored, minimizing transfer overhead. This also significantly improves energy efficiency, as data movement dominates power consumption in conventional systems [6].

Several practical implementations have realized the PIM concept, most notably UPMEM and the Hybrid Memory Cube (HMC) [7]. HMC uses 3D-stacked memory with logic layers for near-memory processing, while UPMEM is the first commercial DRAM-based PIM platform integrating processing units directly inside memory chips, and is the focus of this work [15].

2-1- UPMEM Architecture

UPMEM is a commercial PIM platform integrating lightweight processors inside standard DDR4 DRAM chips. Unlike GPUs or FPGAs, computation is embedded directly in memory, enabling high internal bandwidth and massive parallelism for memory-bound workloads [15, 17]. As shown in Figure 2, a UPMEM system consists of 1) a host CPU (x86, ARM64, or RISC-V), 2) conventional main memory, and 3) UPMEM DIMM modules. Each DIMM contains DRAM chips enhanced with embedded Data Processing Units (DPUs), operating in parallel as a coprocessor [15].

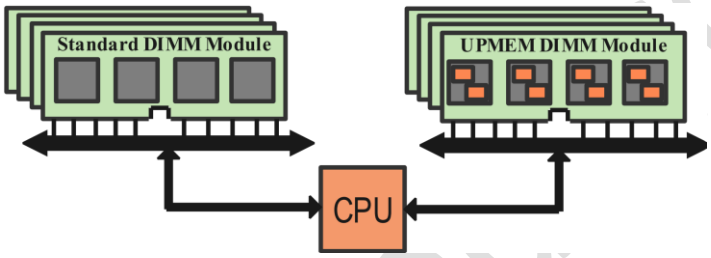

Figure 2. System Architecture using UPMEM.

The execution model follows a host-to-DPU offloading scheme: the host selects tasks, transfers data and code to DPUs, each DPU executes independently on its local memory, and results are returned to main memory. Each DRAM chip is divided into 64-MB regions, each paired with a dedicated DPU, making compute capacity scale with memory size. This tight coupling enables efficient execution of memory-intensive workloads such as matrix operations, graph analytics, and image processing.

UPMEM follows a heterogeneous model where the host handles control and scheduling, while DPUs execute compute-intensive kernels. This design allows UPMEM to act as a scalable accelerator without replacing CPUs [15]. UPMEM DIMMs are deployed as standard memory modules in conventional servers, making them easily integrable. To software, they appear as memory devices, accessed via runtime libraries. Applications include database acceleration, graph analytics, machine learning inference, genomics, and sparse linear algebra, consistently showing speedup and energy gains over CPU-only systems [9, 15].

10.22060/EEJ.2026.26237.6085

2-2 DPU Architecture

Each DRAM chip integrates multiple DPUs (Figure 2), which form the computational core of the system [15]. Figure 3 shows the internal DPU structure. DPUs follow a RISC-like pipelined architecture optimized for energy-efficient fixed-point computation. Each DPU uses a three-level memory hierarchy:

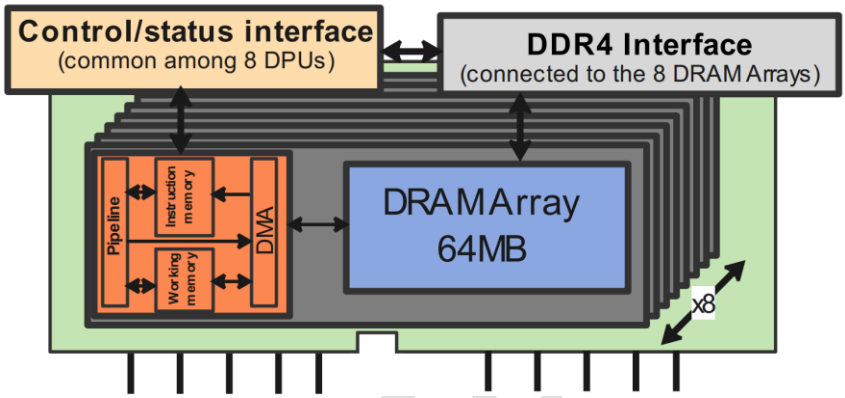

Figure 3. DPU Architecture.

- MRAM: 64MB DRAM region for data storage.
- IRAM: on-chip SRAM for instructions.
- WRAM: low-latency SRAM scratchpad for intermediate data.

Data transfers between MRAM and internal memories are handled by DMA, supporting two modes:

- Execution Mode: instruction execution.
- Copy Mode: data transfer between MRAM and host memory.

Execution and data transfer cannot occur simultaneously, requiring explicit scheduling by the program [15]. DPUs lack an MMU, meaning there is no address translation layer as in conventional systems. This avoids abstraction overhead but limits compatibility with traditional memory management.

IRAM and WRAM provide significantly lower latency than MRAM. While DRAM uses 25nm technology, DPUs are implemented in 65nm technology [15]. Architecturally, DPUs combine Harvard-style internal memory access (IRAM/WRAM separation) with von Neumann-like external MRAM access via DMA, forming a hybrid Harvard–von Neumann design. Unlike other PIM systems, UPMEM does not support direct inter-DPU communication; all communication is mediated by the host CPU.

Each DPU includes an 11-stage pipeline and supports 24 hardware threads (tasklets) for latency hiding. The deep pipeline necessitates fine-grained multithreading to maintain utilization. DPUs also lack FPUs, requiring fixed-point computation or software emulation, making quantization essential for workloads such as matrix multiplication [15]. Table 1 summarizes key DPU characteristics. Overall, the design prioritizes energy efficiency and memory-bound throughput over floating-point performance [15].

Table 1. UPMEM DPU Architectural Specifications.

Parameter	Value
Number of DPUs per chip	8
Number of DPUs per DIMM	128
Total DPUs in 20 DIMMs	2560
MRAM per DPU	64 MB
WRAM per DPU	64 KB
IRAM per DPU	24 KB
DPU area	3.75 mm^2
Power consumption per DPU	120 mW
Operating frequency	350 MHz
Number of hardware threads (tasklets)	24
Registers per tasklet	3

3- Related Work

PIM has attracted significant research attention as a promising solution to overcome the memory wall and the inefficiencies of traditional von Neumann architecture. Early studies established the theoretical foundations of near/in-memory computing, emphasizing the fundamental relationship between data

movement, energy consumption, and system performance [16]. Recent studies categorize PIM architectures into DRAM-based, non-volatile memory (NVM)-based, and near-memory processing systems, and analyze their applicability across workloads such as machine learning, databases, and scientific computing [4, 8].

Several comprehensive reviews have focused on the system-level challenges of PIM adoption, including programmability, data placement, coherence, and scheduling. Shafiee *et al.* studied architectural and runtime challenges of near-data processing and emphasized the importance of memory-centric optimization for achieving real-world performance gains [18]. Mutlu *et al.* presented a modern PIM primer, analyzing how recent commercial PIM platforms fit into contemporary memory hierarchies and data-intensive computing stacks [14]. These works demonstrate that although PIM offers strong theoretical advantages, architectural constraints and programming complexity remain major obstacles to widespread adoption.

Among commercial PIM solutions, UPMEM is currently one of the most mature and widely deployed DRAM-based PIM platforms. Di Sanzo provided a detailed performance and energy-efficiency review of workloads executed on UPMEM systems, demonstrating that memory-bound workloads benefit the most from near/in-memory execution [19]. Nider *et al.* presented a real-world case study of UPMEM deployed in off-the-shelf systems, highlighting both the significant performance gains and the limitations imposed by host-DPU data movement and restricted programming models [15]. In addition, recent work by Lee *et al.* proposed a specialized memory management unit (PIM-MMU) to reduce data-transfer overheads in commercial PIM systems, showing that memory virtualization and address translation represent critical bottlenecks for large-scale deployments [20].

Matrix operations represent one of the most intensively studied classes of workloads for PIM systems. Leitersdorf *et al.* proposed MatPIM, a memristive in-memory architecture for accelerating dense matrix operations and convolution, achieving significant speedup and energy efficiency compared to GPU-based execution [10]. Giannoula *et al.* introduced SparseP, a framework for sparse matrix-vector multiplication on real DRAM-based PIM systems, demonstrating substantial reductions in memory traffic and improved

scalability [21]. These studies confirm that matrix multiplication and related linear algebra kernels are particularly well-suited to in-memory execution due to their data-parallel and memory-bound characteristics.

Beyond scientific computing, PIM has also been explored in the context of deep neural networks and image-processing workloads. Kaur *et al.* reviewed various PIM architectures for neural network acceleration and emphasized that convolution, matrix multiplication, and activation functions are natural candidates for in-memory acceleration [9, 20]. These findings directly align with image-processing pipelines, where matrix multiplication forms the computational backbone of filtering, transformation, and feature-extraction operations.

It should be noted that UPMEM is a relatively recent commercial innovation in the field of PIM, and as a result, research projects developed specifically for this platform are still emerging and rapidly expanding. To date, UPMEM-based studies have explored a wide range of application domains, including genomics and bioinformatics, database systems, machine learning, and deep learning workloads. In this paper, we focus on a problem that has not yet been comprehensively studied on UPMEM hardware, namely dense matrix multiplication. This operation constitutes a fundamental computational kernel in many image processing and scientific computing applications. Existing UPMEM-based matrix-related studies have primarily focused on vector addition, matrix–vector multiplication, and sparse matrix operations, leaving dense matrix–matrix multiplication largely unexplored.

4- Problem Definition and Motivation

Dense matrix multiplication is a fundamental operation in computer science, mathematics, engineering, and especially in image processing and data-intensive applications. Due to its high computational complexity and large data volume, optimizing and accelerating matrix multiplication has long been a central challenge in the design of modern processing systems. Traditional CPU implementations, while flexible and general-

purpose, are limited by the number of processing cores and memory bandwidth. Although GPUs offer massive parallelism and high performance for such kernels, they incur high energy consumption and significant hardware cost [22, 23].

With the emergence of PIM architectures, such as UPMEM, new opportunities arise for optimizing memory-bound operations like matrix multiplication. By collocating computation with memory, PIM reduces data movement and memory access latency, leading to potential improvements in both throughput and energy efficiency [4, 19].

PIM platforms such as UPMEM offer a promising alternative to traditional von Neumann systems by significantly reducing data movement between memory and compute units. However, several architectural and implementation-specific constraints limit their effectiveness on dense computational kernels. The lack of native support for high-precision arithmetic necessitates compiler-driven subroutines with high cycle count, preventing DPUs from achieving high throughput in compute-bound workloads [4, 12].

Scalability is another non-trivial challenge. Although UPMEM can support hundreds of DPUs running in parallel, obtaining linear or near-linear speedup for dense matrix multiplication requires careful load balancing, memory partitioning, and minimization of host–DPU communication overhead, especially given the DPUs' relatively low clock frequency (e.g., 350–500 MHz) and limited per-core arithmetic Throughput [12, 19].

Altogether, the UPMEM PIM architecture provides substantial benefits for data-intensive workloads, but its applicability to compute-heavy kernels such as dense matrix multiplication is limited by several inherent architectural constraints. UPMEM DPUs are 32-bit processors without hardware support for 32-bit fixed-point multiplication, 32-bit division, or any floating-point arithmetic. This limitation originates from cost-reduction strategies and the restricted number of metal layers available during DPU fabrication. As a consequence, multiplication and division are emulated using iterative shift-and-add instructions (e.g.,

10.22060/EEJ.2026.26237.6085

Algorithm 1 Measuring Cycle Count of a Multiplication on UPMEM DPU

1: **Inputs:** None
2: **Outputs:** Number of execution cycles for multiplication
3: Initialize `A` as maximum 16-bit integer
4: Initialize `B` as maximum 16-bit integer
5: Declare `C` as 32-bit integer
6: Configure performance counter to count cycles
7: Perform multiplication:

$$C \leftarrow A \times B$$

8: Read the performance counter value:

$$nb\_cycles \leftarrow \texttt{perfcounter\_get()}$$

9: Print: "number of cycles:" nb_cycles

mul_{step} and div_{step}), which may take many cycles depending on operand values. For higher-precision operations, the UPMEM compiler inserts software subroutines (e.g., *mulsi3*, *addsf3*, *muldf3*, *divdf3*), significantly increasing execution latency (see [18, 19]).

To concretely examine the impact of the mentioned architectural constraints, we focus on the fundamental operation of dense matrix multiplication, which is representative of many compute- and memory-intensive workloads. We consider the multiplication of two dense matrices,

$$C = A \times B, \tag{1}$$

where $\mathbf{A} \epsilon \mathbb{R}^{M \times K}$, $\mathbf{B} \epsilon \mathbb{R}^{K \times N}$, $\mathbf{C} \epsilon \mathbb{R}^{M \times N}$. This operation serves as a useful microbenchmark for evaluating the efficiency of UPMEM DPUs, as it combines high arithmetic intensity with frequent memory accesses and repeated multiply–accumulate operations. Due to the absence of floating-point units and native high-precision multipliers in the DPU pipeline, all computations are performed using fixed-point integer arithmetic, making the cost of multiplication a dominant factor in overall execution time.

By implementing the code depicted in Algorithm 1 on UPMEM, the results are extracted and shown in Table 2. As demonstrated, while small-width integer arithmetic (8-bit, 16-bit) usually completes in a few

cycles, 32-bit multiplication requires markedly more cycles, and floating-point operations often take hundreds or thousands of cycles.

These overheads make high-precision or floating-point arithmetic impractical on current UPMEM DPUs, motivating the use of 32-bit integer matrices in this study (consistent with earlier discussion about PIM limitations [4, 12]).

In summary, dense matrix multiplication on UPMEM must contend with limited arithmetic capabilities, manual memory management, communication bottlenecks, and scalability constraints. This work addresses this challenge by implementing and evaluating a 32-bit integer matrix multiplication kernel on UPMEM, analyzing its performance and scalability trade-offs under real hardware constraints. By implementing and optimizing dense matrix multiplication on UPMEM using 32-bit integer arithmetic, we aim to assess whether the architectural strengths of PIM can compensate for the limited arithmetic capabilities of DPUs,

Table 2. Cycle count of arithmetic operations on UPMEM DPU for different operand types

Operands	Add (cycles)	Sub (cycles)	Mul (cycles)	Div (cycles)
Int8	< 16	< 16	< 16	16
Int16	< 16	< 16	16	16
Int32	< 16	< 16	48	16
float	64	64	192	1040
double	80	64	640	2144

and to provide a practical performance characterization of a core matrix kernel.

5- Proposed Method

UPMEM integrates a large number of lightweight Data Processing Units (DPUs) directly within DRAM chips, enabling highly parallel execution of data-centric workloads [12]. In this paper, we propose an optimized dense matrix multiplication framework specifically tailored to the architectural characteristics and constraints of the UPMEM PIM platform. The proposed approach aims to mitigate the high cost of

arithmetic operations and memory access inherent to DPU execution by carefully restructuring data layout, computation order, and parallel workload distribution.

Starting from a straightforward matrix multiplication implementation, we incrementally refine the design to improve data locality, reduce redundant MRAM accesses, and better exploit the massive parallelism provided by DPUs and their hardware tasklets. The resulting methodology emphasizes memory-efficient tiling, balanced workload partitioning across DPUs, and explicit management of MRAM-WRAM data transfers, thereby aligning the computation with the strengths of the UPMEM architecture while minimizing its inherent limitations.

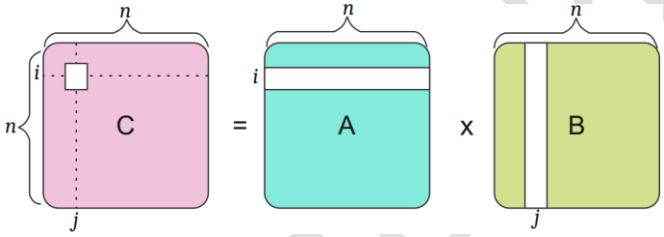

Figure 4. Naive implementation of matrix multiplication.

5-1- Baseline Dense Matrix Multiplication

The simplest implementation of dense matrix multiplication, referred to as the *naive implementation* and illustrated in Figure 4, directly follows the mathematical definition:

$$C[i][j]=\sum_{k=0}^{k-1} A[i][k]\times B[k][j] \quad (2)$$

A naive triple-nested loop formulation is commonly used as an initial approach in conventional CPU implementations:

for *i* **in range** (0, rows_A):

    for *j* **in range** (0, cols_B):

        for *k* **in range** (0, cols_A):

10.22060/EEJ.2026.26237.6085

$$C[i][j] += A[i][k] \times B[k][j]$$

On the UPMEM platform, this baseline implementation is adapted to exploit the available parallelism across DPUs. Matrix **A** is partitioned row-wise among DPUs, while matrix **B** is either replicated or streamed depending on its size and available MRAM capacity. Each DPU computes a disjoint subset of rows of matrix **C**, thereby eliminating inter-DPU communication and enabling fully independent execution.

Due to the three-level memory hierarchy of DPUs, MRAM stores the assigned blocks of matrices **A** and **B** as well as the resulting block of matrix **C**. WRAM is used as a low-latency scratchpad to cache input sub-blocks and intermediate accumulation values, while the DPU kernel instructions reside in IRAM. Because MRAM access latency is significantly higher than WRAM access latency, explicit DMA transfers are used to stage data into WRAM before computation.

Each DPU supports up to 24 hardware tasklets, which are leveraged to parallelize the inner dimension of the matrix multiplication. In the baseline design, tasklets compute partial sums over disjoint segments of the shared dot-product dimension *K*. Partial results are accumulated in WRAM using synchronization primitives to ensure correctness. This approach enables effective latency hiding for MRAM accesses while improving arithmetic utilization of the DPU pipeline.

5-2- Data Partitioning and Load Distribution

Efficient mapping of the computation $C = A \times B$ onto a distributed set of DPUs is a key challenge in dense matrix multiplication. The proposed method partitions the input matrices into sub-blocks that fit within the 64MB MRAM available per DPU, accounting for two input sub-matrices, one output sub-matrix, and auxiliary buffers. Given the absence of a hardware MMU in UPMEM and the need for explicit data transfers, we employ a block decomposition strategy inspired by classical 2D partitioning techniques [24]. Each DPU is responsible for computing one output tile. To maximize memory utilization and parallelism, the host determines an appropriate tiling scheme based on matrix dimensions and the number of available

DPUs. Tiles are padded as needed to satisfy alignment constraints imposed by the UPMEM SDK, ensuring that data transfers occur in multiples of eight bytes for efficient DMA operation.

5-3- Host and DPU Program Structure

The implementation is divided into a host program and a DPU program.

5-3-1- Host Program

The host constructs an array of tile descriptors containing metadata such as matrix offsets, dimensions, and memory addresses. These descriptors are transferred to each DPU's MRAM using the UPMEM runtime. The host launches DPU kernels asynchronously and retrieves partial results upon completion, overlapping data transfers with computation whenever possible to hide communication latency. This design follows prior work on minimizing host--PIM traffic in hybrid systems [20].

5-3-2- DPU Program

Each DPU loads its assigned sub-blocks of matrices *A* and *B* from MRAM into WRAM buffers. Given the limited WRAM capacity and lack of dynamic memory allocation, a streaming buffer mechanism is employed to iteratively load and process sub-segments. Inside the DPU, a parallel inner multiplication loop is executed using multiple tasklets, each computing a portion of the output tile. Partial results are accumulated locally before being written back to MRAM. This approach is conceptually similar to blocked matrix multiplication on GPUs [25], but is adapted to the scratchpad-based memory model and pipeline structure of UPMEM DPUs.

5-4- Memory and Execution Optimizations

Due to the absence of caches and floating-point units on DPUs, several optimizations are employed:

- Blocking and tiling: Matrices are decomposed into blocks that fit within WRAM to maximize data reuse and minimize MRAM accesses.

- Chunked streaming: Input sub-blocks are streamed into WRAM buffers to reduce redundant MRAM traffic.
- Tasklet parallelism: Multiple tasklets per DPU process, independent sub-computations concurrently balance workload and reduce idle cycles.
- Padding alignment: Data transfers are aligned to 8-byte boundaries to satisfy SDK constraints and improve bandwidth utilization.

These optimizations follow principles used in high-performance linear algebra libraries such as BLIS and cuBLAS [26, 27].

5-5- Addressing UPMEM Constraints

UPMEM DPUs lack hardware support for 32-bit multiplication, division, and floating-point arithmetic. To accommodate this limitation, all computations are restricted to 32-bit integer arithmetic, following UPMEM programming guidelines and performance profiling results [12, 19]. Tile sizes are selected to ensure that the combined memory footprint of input tiles, output tiles, and auxiliary buffers fits within the MRAM capacity of each DPU.

5-6- Scalability and Parallel Execution

To utilize all available DPUs, the host employs a scheduling strategy that distributes tiles in a round-robin or block-cyclic manner, mitigating load imbalance and stragglers. Data transfers and computation are overlapped using double buffering, allowing one buffer to be processed while the next is being filled. This approach is inspired by scalable parallel matrix multiplication techniques used in distributed systems [28].

The overall execution proceeds as follows:

- The host initializes matrices **A** and **B** in main memory.

- Data is transferred from host DRAM to UPMEM MRAM.
- DPU kernels are launched in execution mode.
- Tasklets perform parallel fixed-point matrix multiplication.
- Output submatrices of **C** are written back to MRAM.
- The host retrieves and assembles the final output matrix.

In summary, this paper implements dense matrix multiplication on the UPMEM PIM platform using three progressively optimized versions of the algorithm. Beginning with a naive baseline implementation, we introduce loop-level and memory-level optimizations that improve data locality, reduce MRAM access overhead, and enhance tasklet-level parallelism. These implementations serve both as a performance study and as a foundation for understanding how classical numerical kernels can be effectively adapted to PIM architecture.

6- Optimizing Matrix Multiplication

Over the years, many methods have been proposed to implement matrix multiplication, as one of the most fundamental and widely used operations, efficiently on different hardware architectures, ranging from simple CPU-based implementations to optimized cache-aware algorithms and parallel divide-and-conquer strategies.

The simplest implementation of matrix multiplication is the naive implementation that consists of a triple-nested loop. While conceptually simple, this method suffers from several performance issues. It incurs a large number of memory accesses; each iteration reads from *A* and *B* and writes to *C*. Likewise, it has poor cache utilization for large matrices because of repeated data reloading, leading to many cache misses [29]. Despite its poor performance, the naive algorithm is often used as a baseline for performance comparison.

The initial implementation of the naive matrix multiplication algorithm on the UPMEM architecture played a crucial role in understanding the challenges associated with matrix-based computations in this PIM

framework. This experimental stage allowed us to examine architectural limitations, memory behavior, and performance bottlenecks, ultimately guiding the development of an optimized Tiling-based approach.

The key drawbacks identified during this phase are summarized below:

- Identifying performance bottlenecks: Running the naive multiplication algorithm revealed that frequent accesses to MRAM significantly increase latency and reduce overall DPU performance. This observation highlighted the need for improved data locality and allowed us to design a Tiling strategy that manages data more efficiently in WRAM.
- Understanding data-access patterns: Analysis of the naive approach showed that storing matrix *A* in row-major order and matrix *B* in column-major order reduces WRAM cache misses and improves read efficiency during Tiling. These insights directly informed the memory-layout optimizations applied later.
- Recognizing the impact of DPU- and tasklet-level parallelism: The initial implementation demonstrated how tasklets operate independently and how their workload distribution affects performance. This helped us determine how each tile should be split among tasklets in the optimized Tiling method to maximize parallelism.

This first implementation served as a foundational step, providing a comprehensive understanding of the optimization opportunities within the UPMEM platform. These observations directly enabled the design of our Tiling-based method, reducing computational overhead and improving memory management to fully exploit the computational capabilities of DPUs during matrix multiplication.

6-1- Tiling/Cache Blocking

Cache blocking (or tiling) further enhances performance by dividing matrices into smaller sub-blocks (tiles) that fit entirely in cache. Within each tile, the data is reused multiple times before eviction, dramatically reducing main memory accesses.

10.22060/EEJ.2026.26237.6085

The tiled algorithm proceeds as follows:

for i_{block} **in range** (0, rows_A, block_size):

for j_{block} **in range** (0, cols_B, block_size):

for k_{block} **in range** (0, cols_A, block_size):

for *i* **in range** (i_block, i_block + block_size):

for *j* **in range** (j_block, j_block + block_size):

for *k* **in range** (k_block, k_block + block_size):

$$C[i][j] += A[i][k] \times B[k][j]$$

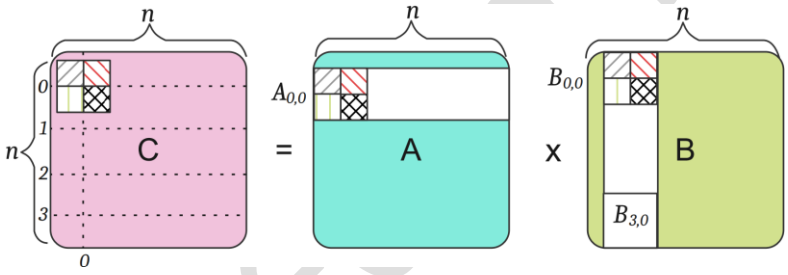

Figure 5. Tiling implementation of matrix multiplication.

The primary steps include dividing matrices into tiles based on cache size, as shown in Figure 5. Then, multiplication on tiles that fit in the cache should be performed.

Finally, intermediate results need to be accumulated to form the final result.

Tiling can significantly improve performance for large matrices by maximizing cache reuse and minimizing costly main memory access. However, it requires knowledge of cache sizes and can be less robust on multi-tasking systems due to context switches flushing cache lines [30].

6-2- Loop Reordering Optimization

One of the major performance challenges in matrix multiplication is the effective use of the CPU cache. CPUs store multi-dimensional arrays in row-major order (i.e., elements of each row are contiguous in memory). In the naive implementation, the inner loop index traverses rows of matrix *B*, which does not exploit spatial locality in contiguous memory.

A simple yet effective optimization consists of reordering the loop structure of the conventional matrix multiplication algorithm. As illustrated in Figure 6, this transformation changes the computation order such that each element of a row of matrix **A** is multiplied with the corresponding row of matrix **B**, contributing incrementally to the elements of the output matrix **C**. By hoisting the access to *A[i][k]* outside the innermost loop, the value is loaded once and reused across all column computations, thereby improving data locality and reducing redundant memory accesses.

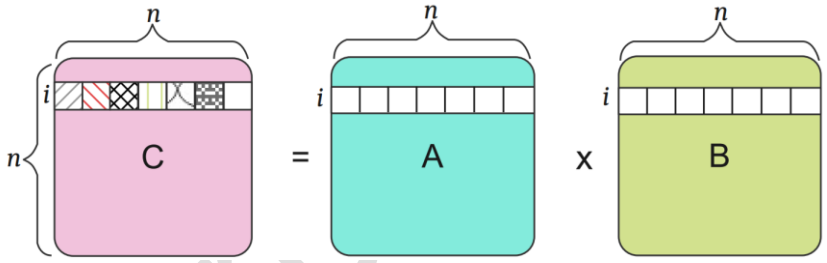

Figure 6. Loop reordering implementation of matrix multiplication.

for *i* **in range** (0, rows_A):

for *k* **in range** (0, cols_A):

$$temp = A[i][k]$$

for *j* **in range** (0, cols_B):

$$C[i][j] += temp \times B[k][j]$$

This loop reordering improves cache behavior by accessing memory in a more sequential manner, thereby reducing cache misses and improving runtime performance. Cache-aware loop ordering has been proposed

in many optimization studies [31]. However, loop reordering does not improve performance on UPMEM because it lacks a cache component and its local memory is managed by software.

7- Evaluation

This section evaluates the performance of the proposed dense matrix multiplication implementations on the UPMEM PIM platform and compares them against optimized CPU-based implementations. The evaluated methods include the naive iterative algorithm, the loop-reordered optimization, and the tiled (cache-blocked) approach. These implementations, introduced in the previous sections, are assessed to highlight the impact of memory access patterns, data locality, and architectural constraints on execution efficiency. The evaluation focuses on understanding how UPMEM's architectural parameters, specifically the number of DPUs and the number of tasklets per DPU, affect execution time and scalability. We further investigate the impact of arithmetic precision on performance, given the lack of native support for high-precision operations in UPMEM DPUs.

Implementing matrix multiplication on the UPMEM platform follows a heterogeneous programming model that consists of two tightly-coupled components: a host program and a DPU program. The host-side program is responsible for preparing input data, partitioning matrices across available DPUs, and transferring the required data to each DPU through dedicated memory buffers. Once data transfer is complete, the host launches the DPUs for execution and, upon completion, retrieves the partial results from each DPU and consolidates them in the main memory for further use.

The DPU-side program implements the core computation kernel. Based on the metadata and configuration information provided by the host, each DPU processes its assigned portion of the input matrices independently and stores the resulting partial output in its local memory. This separation of responsibilities allows large-scale parallel execution but also introduces several architectural and implementation challenges that directly influence performance.

10.22060/EEJ.2026.26237.6085

Several implementation-specific challenges arose when mapping dense matrix multiplication onto the UPMEM architecture. First, the problem had to be adapted to the distributed nature of DPUs. This was addressed by partitioning the input matrices according to the number of available DPUs and assigning each DPU a distinct submatrix for computation. Second, data transfer constraints imposed by the UPMEM compiler required that all data sent to DPUs adhere to strict alignment rules. To satisfy these constraints, input matrices were padded such that their dimensions became divisible by eight. Another challenge involved defining a consistent data layout for each DPU after partitioning and padding. This was resolved through a preprocessing stage on the host, where all necessary metadata describing matrix dimensions, offsets, and memory boundaries were computed in advance. These metadata were labeled and transferred to DPU memory using UPMEM-provided communication primitives, ensuring that each DPU had a complete and correct view of its assigned workload.

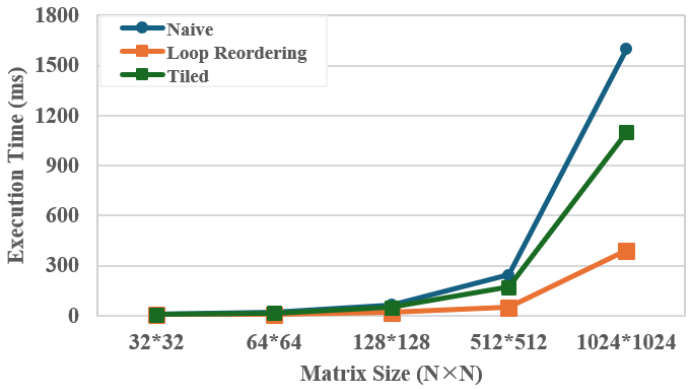

Figure 7. Performance comparison of different matrix multiplication implementations on CPU.

Efficient parallel data transfer presented an additional challenge. To avoid serial communication overhead, fixed-size buffers were allocated for each DPU based on the maximum number of rows that a DPU could process, multiplied by the padded column size. This design ensured that all DPUs could receive their data concurrently while operating strictly within their predefined memory regions. Finally, the limited on-chip memory capacity of each DPU (64MB) imposed a strict upper bound on the problem size. Each DPU must store two input submatrices, one output submatrix, auxiliary metadata, and additional working memory.

Consequently, the available memory was divided into four parts: two for the input matrices, one for the output matrix, and one for metadata and temporary storage. Under these constraints, each DPU can process a maximum matrix size of approximately 2048×2048 elements when using 32-bit integer arithmetic. Larger matrices are supported by increasing the number of DPUs and distributing the workload accordingly.

7-1- Implementation on CPU

7-1-1- Naive Matrix Multiplication

As shown in Figure 7, the naive implementation is the slowest among all three methods, especially for larger matrix sizes. It demonstrates that the execution time increases from approximately 20ms for a 256×256 matrix to 1700ms for a 1024×1024 matrix.

This poor performance primarily stems from excessive memory accesses. Each element in matrix $\boldsymbol{C}$ requires three memory reads and one write. The other reason is a misaligned memory layout. Matrix $\boldsymbol{A}$ is stored in row-major order, while matrix $\boldsymbol{B}$ is stored in column-major order, causing a high number of cache misses.

7-1-2- Loop-Reordered Matrix Multiplication

The loop-reordered version achieves the best performance among the tested implementations. For instance, on a 1024×1024 matrix, it performs about 4.42 times faster than the naive version. The main reasons for this improvement include changing the loop order from (m, n, k) to (m, k, n), which yields a more cache-friendly memory access pattern. Although in this case, data locality is improved, causing a reduction in the frequency of cache misses and preventing unnecessary memory reloads.

7-1-3- Tiled Matrix Multiplication

The tiled implementation performs better than the naive approach but is still slower than the loop-reordered version. As depicted in Figure 7, for a 1024×1024 matrix, it is 2.5 times slower than the loop-reordered method. Its limitations arise from being a cache-agnostic approach, meaning that the tile size does not adapt

to system-specific cache characteristics. On the other hand, the presence of operating system context switches, which may evict loaded tiles from cache, reduces efficiency.

In summary, the best method is loop-reordered matrix multiplication, while the worst method is naive iterative matrix multiplication. The tiled method improves over the naive version but shows weaker performance on general-purpose systems compared to loop reordering due to cache unpredictability.

7-2- Implementation on DPU

To ensure a fair comparison, all implementations operate on identical input sizes and use fixed-point integer arithmetic unless otherwise stated. Three matrix multiplication variants are considered. On UPMEM, we evaluate both the naive and tiled approaches, while on the CPU, all three methods are analyzed as described in the previous subsections.

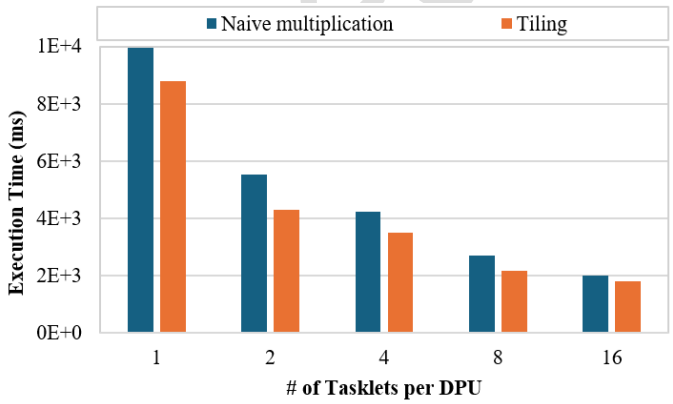

Figure 8. Performance scaling of naive matrix multiplication and optimized tiling method with respect to the number of DPUs on UPMEM

7-2-1- Impact of Tasklet-Level Parallelism

We first examine the effect of tasklet-level parallelism on matrix multiplication performance. In this experiment, a single DPU is used while the number of tasklets is varied from a small number up to the maximum supported value of 24. The input matrices are fixed to a size of *512×512*.

10.22060/EEJ.2026.26237.6085

As shown in Figure 8, the results demonstrate a substantial reduction in execution time as the number of tasklets increases for both the naive and tiled implementations. This improvement is expected, as tasklets operate independently and allow the workload to be decomposed into parallel sub-computations along the dot-product dimension. In our DPU kernels, tasklets process disjoint portions of the input data and compute partial sums concurrently, which are later accumulated in WRAM. As a result, increased tasklet parallelism directly translates into higher arithmetic utilization and more effective hiding of MRAM access latency. These observations confirm that tasklets are a critical source of intra-DPU parallelism and must be fully exploited to achieve acceptable performance on UPMEM.

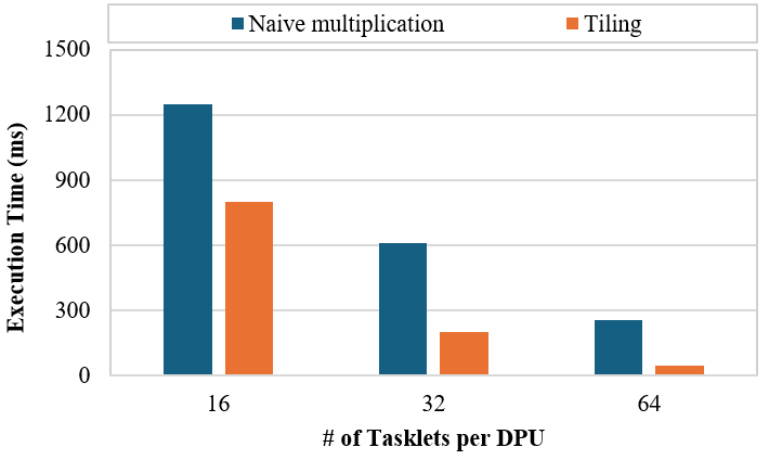

Figure 9. Effect of increasing the DPU count on the execution time of naive matrix multiplication and optimized tiling method on UPMEM.

7-2-2- Impact of the Number of DPUs

After establishing the benefits of tasklet parallelism, we fix the number of tasklets per DPU to the maximum value of 24 and study the effect of increasing the number of DPUs. Due to limitations of the UPMEM simulator, the maximum number of available DPUs in our experiments is 64. Nevertheless, this configuration is sufficient to evaluate scalability trends for the considered matrix sizes. As the number of DPUs increases, execution time decreases significantly for both matrix multiplication implementations, as demonstrated in Figure 9. This behavior highlights the effectiveness of data partitioning across DPUs, where each DPU processes an independent tile of the output matrix. Similar to tasklet-level parallelism,

10.22060/EEJ.2026.26237.6085

increasing the number of DPUs improves throughput by distributing both computation and memory accesses across multiple processing units. The observed scalability confirms that dense matrix multiplication on UPMEM benefits from coarse-grained parallelism, provided that the workload is evenly balanced and communication between the host and DPUs is minimized.

Table 3. Execution time comparison of matrix multiplication algorithms on UPMEM (32-bit operations) and CPU.

Matrix Size	Naive UPMEM	Tiled UPMEM	Naive CPU	Tiled CPU
128×128	1.8	0.3	4.1	1.5
256×256	15	5	20	14
512×512	208	39	245	120
1024×1024	1420	398	1700	1100

7-2-3- CPU versus UPMEM Performance Comparison

With the effects of tasklets and DPUs characterized, we compare the performance of matrix multiplication on UPMEM against CPU-based implementations. Table 3 summarizes the execution times for different matrix sizes using 32-bit integer arithmetic. The naive matrix multiplication on UPMEM already outperforms the naive CPU implementation. For example, for a 1024×1024 matrix, the UPMEM naive implementation achieves approximately a 1.19x speedup compared to the CPU counterpart. However, this implementation does not fully exploit the architectural capabilities of UPMEM and remains slower than optimized CPU algorithms such as loop reordering. In contrast, the tiled matrix multiplication on UPMEM achieves significantly higher performance. Compared to the naïve CPU implementation, it provides a speedup of approximately 4.27x for large matrices. Moreover, it outperforms the tiled CPU implementation

by about 2.76x. These results demonstrate that, when properly optimized, UPMEM can be competitive with and even superior to conventional CPUs for dense integer-based matrix multiplication.

7-2-4- Effect of 64-bit Arithmetic

Finally, we investigate the impact of using 64-bit integer arithmetic on performance. As discussed earlier, UPMEM DPUs lack hardware support for 64-bit operations, which must be emulated through software routines. To quantify this overhead, we repeat the experiments using int64 data types. The results in Table 4 show a dramatic performance degradation for both naive and tiled implementations. For a 1024×1024 matrix, the naive implementation becomes 43.4% slower compared to its 32-bit counterpart, while this

Table 4. Execution time comparison of matrix multiplication algorithms on UPMEM using 64-bit integer operations

Matrix Size	Naive Matrix Multiplication (UPMEM)	Tiling-Based Matrix Multiplication (UPMEM)
128×128	10 ms	6 ms
256×256	175 ms	11 ms
512×512	270 ms	70 ms
1024×1024	2510 ms	610 ms

value for tiled implementation is 34.7%. This degradation is consistent with the high cycle count of software-emulated arithmetic operations on DPUs. These findings clearly indicate that UPMEM is not well-suited for workloads requiring high-precision or wide integer arithmetic. In such cases, execution on the host CPU yields significantly better performance.

7-3- Discussion

Overall, the evaluation demonstrates that UPMEM can deliver substantial performance benefits for dense matrix multiplication when computations are carefully optimized to match its architectural constraints. Tasklet-level and DPU-level parallelism play a decisive role in achieving high throughput, while tiling and

data-layout optimizations are essential for reducing MRAM access overhead. However, the results also highlight important limitations.

The absence of hardware support for high-precision arithmetic significantly restricts UPMEM's applicability to workloads involving 64-bit or floating-point operations. Consequently, UPMEM-based PIM architectures are best suited for data-intensive, low-precision computations, whereas compute-heavy and high-precision workloads remain more efficiently handled by conventional CPUs.

8- Conclusion

This paper investigated dense matrix multiplication on the UPMEM PIM platform through three implementations: naive, loop-reordered, and tiled. By adapting classical optimization techniques to the architectural constraints of UPMEM DPUs, we demonstrated the importance of data locality, explicit memory management, and fine-grained parallelism for achieving high performance on PIM systems. Experimental results show that both tasklet-level parallelism and DPU scalability significantly reduce execution time. For 32-bit integer arithmetic, the tiled UPMEM implementation outperformed CPU-based naïve and tiled methods and achieved competitive performance with optimized loop-reordered CPU implementations, despite the DPUs' low clock frequency and lightweight cores. These results confirm that UPMEM can effectively accelerate dense matrix multiplication when computation is carefully aligned with its memory-centric design. However, the lack of native support for high-precision arithmetic leads to severe performance degradation for 64-bit operations, limiting the suitability of UPMEM for compute-bound or precision-sensitive workloads. Overall, this study provides a practical performance characterization of dense matrix multiplication on UPMEM and offers insights that can guide future PIM-oriented algorithm design, as well as motivate further research into compiler optimizations, mixed-precision methods, and hybrid CPU–PIM execution models.

References

[1] J. Gómez-Luna, I.E. Hajj, I. Fernandez, C. Giannoula, G.F. Oliveira, O. Mutlu, Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture, arXiv preprint arXiv:2105.03814, (2022).
[2] E. Cheshmikhani, B. Peccerillo, A. Mondelli, S. Bartolini, A General Framework for Accelerator Management Based on ISA Extension, IEEE Access, 10 (2022) 120702-120713.
[3] K. Asifuzzaman, N.R. Miniskar, A.R. Young, F. Liu, J.S. Vetter, A survey on processing-in-memory techniques: Advances and challenges, Memories-Materials, Devices, Circuits and Systems, 4 (2023) 100022.
[4] M.V. Pathak, Processing-In-Memory Techniques: Survey, Advances, and Challenges, International Journal for Research in Applied Science and Engineering Technology, 12(5) (2024) 4956-4970.
[5] M. Aghaei, S. Ebrahimi, M.S. Arafati, E. Cheshmikhani, D. Rahmati, S. Gorgin, J. Kim, Modeling and Simulation Frameworks for Processing-in-Memory Architectures, arXiv preprint arXiv:2512. 00096, (2025).
[6] K. Khan, S. Pasricha, R.G. Kim, A survey of resource management for processing-in-memory and near-memory processing architectures, Journal of Low Power Electronics and Applications, 10(4) (2020) 30.
[7] N. Zarif, Offloading Embedding Lookups to Processing-In-Memory for Deep Learning Recommender Models, Masters Thesis, The University of British Columbia, 2023.
[8] G.F. Oliveira, J. Gómez-Luna, S. Ghose, O. Mutlu, Methodologies, Workloads, and Tools for Processing-in-Memory: Enabling the Adoption of Data-Centric Architectures, in: IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2022, pp. 261-266.
[9] R. Kaur, A. Asad, F. Mohammadi, A Comprehensive Review of Processing-in-Memory Architectures for Deep Neural Networks, Computers (MDPI), 13(7) (2024) 174.
[10] O. Leitersdorf, R. Ronen, S. Kvatinsky, MatPIM: Accelerating matrix operations with memristive stateful logic, in: 2022 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2022, pp. 215-219.
[11] P. Das, Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing-in-Memory Hardware, Masters Thesis, Rochester Institute of Technology, 2022.
[12] I. Cutress, Hot Chips 31 Analysis: In Memory Processing by Upmem, Anandtech, Aug, 18 (2019).
[13] Y. Falevoz, J. Legriel, Energy efficiency impact of processing in memory: A comprehensive review of workloads on the upmem architecture, in: European conference on parallel processing, Springer, 2023, pp. 155-166.
[14] O. Mutlu, S. Ghose, J. Gómez-Luna, R. Ausavarungnirun, A modern primer on processing in memory, in: Emerging computing: from devices to systems: looking beyond Moore and Von Neumann, Springer, 2022, pp. 171-243.
[15] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, Others, A case study of Processing-in-Memory in off-the-Shelf systems, in: 2021 USENIX Annual Technical Conference (USENIX ATC 21), 2021, pp. 117-130.
[16] W.A. Wulf, S.A. McKee, Hitting the memory wall: Implications of the obvious, ACM SIGARCH computer architecture news, 23(1) (1995) 20-24.
[17] C. Lim, Others, Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs, Proceedings of the ACM on Management of Data, 1(2) (2023) 1-27.
[18] A. Shafiee, Others, Enabling Efficient Processing-in-Memory with Near-Data Processing, IEEE Computer Architecture Letters, 21(2) (2022).

[19] P.D. Sanzo, Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture, in: Euro-Par 2023 Parallel Processing Workshops, Springer, 2024, pp. 155-166.
[20] D. Lee, B. Hyun, T. Kim, M. Rhu, Pim-mmu: A memory management unit for accelerating data transfers in commercial pim systems, in: 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2024, pp. 627-642.
[21] C. Giannoula, I. Fernandez, J. Gómez-Luna, N. Koziris, G. Goumas, O. Mutlu, SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory systems, arXiv preprint arXiv:2201.05072, (2022).
[22] J. Sanders, E. Kandrot, CUDA by example: an introduction to general-purpose GPU programming, Addison-Wesley Professional, 2010.
[23] D.B. Kirk, W.-M.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Elsevier, 2016.
[24] J.J. Dongarra, B.A. Wiberg, Others, A Survey of Numerical Linear Algebra Methods on Parallel Architectures, Springer, 1988.
[25] V. Volkov, Understanding latency hiding on GPUs, University of California, Berkeley, 2016.
[26] F.G. van Zee, J.J. Dongarra, A Model for Performance Prediction and Design of BLIS, High Performance Extreme Computing Conference (HPEC), (2015).
[27] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable Parallel Programming with CUDA, in: ACM SIGGRAPH, 2008.
[28] K. Bergman, P. Boyle, Others, Exascale Computing Study: Technology Challenges in Achieving Exascale Systems, 2008.
[29] D.R. Grayson, D.R. Steiger, Matrix Multiplication Techniques and Optimization, in: Computing Surveys, ACM, 1994.
[30] R.C. Whaley, J.J. Dongarra, Automatically Tuned Linear Algebra Software, Supercomputing '98 Proceedings, (2001).
[31] M.S. Lam, E.E. Rothberg, M.E. Wolf, The Cache Performance and Optimizations of Blocked Algorithms, ACM SIGARCH Computer Architecture News, 19(2) (1991) 63-74.