SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen[1] Juan Gómez-Luna[1] Izzat El Hajj[2] Yuxin Guo[1] Onur Mutlu[1]
[1]ETH Zürich [2]American University of Beirut

Data movement between memory and processors is a major bottleneck in modern computing systems. The processing-in-memory (PIM) paradigm aims to alleviate this bottleneck by performing computation inside memory chips. Real PIM hardware (e.g., the UPMEM system) is now available and has demonstrated potential in many applications. However, programming such real PIM hardware remains a challenge for many programmers.

This paper presents a new software framework, SimplePIM, to aid programming real PIM systems. The framework processes arrays of arbitrary elements on a PIM device by calling iterator functions from the host and provides primitives for communication among PIM cores and between PIM and the host system. We implement SimplePIM for the UPMEM PIM system and evaluate it on six major applications. Our results show that SimplePIM enables 66.5% to 83.1% reduction in lines of code in PIM programs. The resulting code leads to higher performance (between 10% and 37% speedup) than hand-optimized code in three applications and provides comparable performance in three others. SimplePIM is fully and freely available at `https://github.com/CMU-SAFARI/SimplePIM`.

1. Introduction

Processing-in-memory (PIM) [1–4] is a computing paradigm that places compute units in the memory chip to avoid moving data between the memory and the CPU cores. This paradigm has been shown to offer significantly higher memory bandwidth and lower memory access latency for a wide variety of applications, including graph processing [5–9], machine learning [10–24], database operations [25–30], sparse linear algebra [9, 31], stochastic computing [32], climate modeling [33,34], stencil computations [35], mobile applications [15], and bioinformatics [36–42]. The UPMEM PIM system [43, 44] is the first commercially available PIM hardware with general-purpose cores embedded in the DRAM chip. Prior works show that the UPMEM PIM system can benefit numerous applications [10–12, 26, 29, 31, 40, 45–54]. Several other similar PIM architectures have also been presented and prototyped [13, 25, 55–58].

Programming a real PIM system is a challenging task due to the complexities involved. For example, the UPMEM PIM system requires programmers to distribute data across the DRAM banks, launch PIM kernels on the PIM cores, manage the transfer of data between the DRAM banks and the PIM cores, and orchestrate the execution of multiple PIM threads on each PIM core [59]. This task requires deep knowledge of PIM hardware and system architecture as well as proficiency in low-level APIs, which presents a steep learning curve. Suitable library, programming model, compiler, and tool support is crucial for adopting PIM in real-world systems, as previously discussed in the literature [1, 4].

To facilitate the adoption of PIM in real-world systems, we propose a high-level programming framework called SimplePIM. SimplePIM abstracts the complexities of PIM hardware, supports multiple important applications such as histogram and K-means, and delivers high performance.

The PIM systems targeted by our framework resemble distributed systems, where each PIM core has exclusive access to a memory region. However, a distributed system typically involves individual machines connected via a network and needs to handle node crash failures [60–65] or even Byzantine failures [66–69]. Many distributed systems use voting protocols to elect coordinator nodes for system management [62–65]. However, failure handling should not be a concern for PIM since the PIM cores are not constantly failing and re-joining the system. In PIM systems, the host CPU is responsible for coordinating the PIM cores and handling communication between them. SimplePIM leverages the power of the central host to manage the entire system, including bookkeeping of the framework metadata and merging of intermediate results from PIM cores. Additionally, the host CPU facilitates communication with the outside world, such as network or I/O operations.

To support PIM systems, SimplePIM provides iterators such as `map`, `reduce`, and `zip`. These iterators are commonly found in programming languages (e.g., Python, Haskell) and distributed frameworks (e.g., MapReduce [61], Spark [70]). Their purpose is to separate the application logic from the parallel decomposition of work across cores and threads.

To facilitate communication of data between the host CPU and the PIM cores, SimplePIM also provides `broadcast`, `scatter`, and `gather` collective communication techniques that involve the host CPU as the root node. Communication primitives among the PIM cores, such as `allreduce` and `allgather`, are also available. The communication interface is similar to the message-passing paradigm used in MPI [71]. However, unlike MPI, which is homogenous and fully distributed, in SimplePIM, the host CPU always plays the unique role of the root node in managing the entire system and merging the intermediate results from different PIM cores.

We implement and evaluate SimplePIM on a real PIM system, UPMEM [43], with six different applications: reduction, vector addition, histogram, linear regression, logistic regression, and K-means. These applications have previously been implemented on UPMEM [10–12, 26], providing a baseline for comparing performance, correctness, and code complexity. SimplePIM offers a programmer-friendly interface and

requires 4.4× fewer lines of code, on average, compared to the best existing open-source hand-optimized implementations. In addition, we apply several code optimizations to tailor our SimplePIM implementation to the underlying hardware, making it suitable for UPMEM. Our evaluation results show that SimplePIM performs similarly to hand-optimized implementations in three applications, and outperforms them in the remaining three, despite its general-purpose design. Specifically, for vector addition, logistic regression, and K-means, SimplePIM performs 1.10×, 1.17×, and 1.37× faster than the best prior hand-optimized implementations in weak scaling tests and 1.15×, 1.22×, and 1.43× faster in strong scaling tests.

Our main contributions are:

- We design and introduce SimplePIM, the first high-level programming framework tailored to improve programming productivity in general-purpose PIM architectures. We open-source SimplePIM at `https://github.com/CMU-SAFARI/SimplePIM` to enhance programming accessibility and aid the adoption of PIM systems.
- We implement SimplePIM on the UPMEM PIM architecture and develop six PIM workloads (reduction, vector addition, histogram, linear regression, logistic regression, and K-means) using it. SimplePIM provides a significant reduction in the number of lines of code ranging from 66.5% to 83.1%, i.e., productivity improvements of 2.98× to 5.93× compared to the best prior hand-optimized open-source implementations written using the UPMEM SDK.
- We explore and implement performance optimization techniques both in general and specific to SimplePIM. The SimplePIM implementation of the evaluated workloads performs similarly to hand-optimized implementations in three applications and achieves a speedup of 1.10×-1.43× in the other three applications.

2. Background

Numerous real PIM architectures have been introduced that aim to bring compute units closer to memory [13, 25, 43, 44, 55–58]. For example, AxDIMM [13] places an FPGA near DRAM ranks to accelerate recommendation systems [13] and database operators [25]. FIMDRAM [55] features vector processing units near the banks of High Bandwidth Memory (HBM). FIMDRAM is specifically designed for deep learning applications. SK Hynix AiM [57] is also designed for deep learning applications. AiM places vector processing units near the banks of GDDR6 memory. Alibaba HB-PNM [58] glues together one layer of DRAM and one logic layer with processing elements designed to accelerate recommendation systems.

Our programming framework implementation in this work uses the commercially available UPMEM PIM system [43]. Two major characteristics distinguish UPMEM from other real PIM systems: (1) it integrates general-purpose processing cores in DRAM chips, and (2) it is the only commercially available PIM architecture (as of September 2023).

Previous studies extensively investigate the UPMEM system [10–12, 26, 29, 31, 40, 45–54]. Fig. 1 provides a simplified view of the first-generation UPMEM architecture from a programmer's perspective. The UPMEM hardware is connected to the host system via DRAM DIMM connections. A state-of-the-art UPMEM server contains up to 20 PIM-enabled DIMMs, each with 2 ranks of 8 PIM chips. Each PIM chip has eight 64MB DRAM banks, with a programmable PIM core, a 64KB scratchpad memory, and 24KB instruction memory coupled to each bank. In total, there are up to 2,560 PIM cores. Data transfers between the scratchpad memory and the DRAM bank occur through explicit data transfer commands, with a maximum bandwidth of 800MB/s per bank. The entire PIM system provides a maximum bandwidth of 2TB/s for all PIM cores.

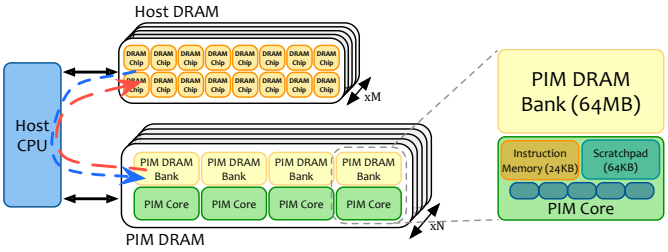

Figure 1: The UPMEM PIM Architecture

The PIM cores operate at 450 MHz and feature an 11-stage pipeline. They can perform one integer addition/subtraction per cycle and 32-bit integer multiplication/division in, at most, 32 cycles when the pipeline is fully utilized. However, floating-point operations may take tens to 2000 cycles to complete, as explained in [26]. The 2,560 independent PIM cores operate in parallel, providing a peak compute throughput exceeding 1 TOPS (Tera operations per second). There is no direct communication mechanism among different PIM cores in hardware. All communication occurs through memory transfers between the host DRAM and the PIM DRAM.

Writing a functionally correct PIM program for a system like the UPMEM PIM architecture is challenging, and optimizing code for performance requires programmers to have a deep understanding of the PIM hardware. To program PIM cores, programmers must manually manage the scratchpad memory to ensure good performance. Data transfers between the scratchpad memory and the corresponding PIM DRAM bank must be 8-byte aligned and not exceed a 2,048-byte limit. For multithreading, the UPMEM SDK [59] provides barrier synchronization, handshakes, and mutexes. At least 11 software threads are required to fully utilize the pipeline [26, 53]. On the host CPU side, programmers allocate/reallocate the number of PIM cores, load the PIM binary, and explicitly launch the PIM program using commands provided in the UPMEM SDK [59]. Communication between the PIM cores and the host CPU is enabled by gather/scatter, broadcast, parallel and serial transfer commands, all of which have different performance implications and data alignment constraints.

Our goal in this work is to design a high-level programming framework for PIM architectures that abstracts these hardware-specific complexities and provides a clean yet powerful interface for ease of use and high program performance.

3. The SimplePIM Programming Framework

A software framework is a universal and reusable software environment that provides a standard abstraction to build and deploy applications. Our software framework, SimplePIM, leverages the massive memory bandwidth and parallelism offered by PIM hardware to operate on arrays of arbitrary size and dimensions. Similar abstractions are widely used in distributed system frameworks (e.g., Spark [70], MapReduce [61]).

SimplePIM offers three key interfaces to support PIM systems like UPMEM. The *management interface* (Section 3.1) stores metadata for the PIM-resident arrays, which can be accessed by the programmer and other parts of SimplePIM as needed. The *communication interface* (Section 3.2) provides abstractions for both Host-PIM and PIM-PIM communication patterns. These patterns are similar to communication patterns in other distributed frameworks such as MPI [71], which makes it easier for SimplePIM to be adopted. The *processing interface* (Section 3.3) leverages PIM's high memory bandwidth and parallelism to execute `map`, `reduce`, and `zip` iterators on PIM arrays. Programmers can combine these iterators to implement many widely-used workloads ranging from simple vector addition to complex machine learning model training. Similar to SimplePIM, systems like Spark [70] provide programmers with some data transformation operations and iterators for processing data.

3.1. Management Interface

The SimplePIM Management Interface provides three main functions: `lookup`, `register` and `free`. These APIs enable tracking of PIM-resident data, data allocation and data deallocation in the form of continuous arrays. This management is centralized and takes place on the host CPU.

The management interface defines and uses two data structures. First, the `array_meta_data_t` struct contains several fields that describe the PIM-resident array. These fields are the ID of the array, its length, data type, and the physical address of its data in the PIM DRAM. Second, the `simple_pim_management_t` struct is responsible for managing all the PIM-resident arrays registered by the programmer. It contains an array of `array_meta_data_t` structs, along with other hardware information such as the number of PIM cores.

Lookup The `lookup` function retrieves the struct of `array_meta_data_t` containing all relevant information of an array from the management unit, based on its unique ID. This function is used by both the communication and processing interfaces to seamlessly access and manipulate the array with a single ID provided by the programmer.

```
array_meta_data_t* simple_pim_array_lookup(const char* id,
    simple_pim_management_t* management);
```

Register The purpose of the `register` function is to register the metadata of an array in the management unit. Typically, the function is called by the processing and communication interfaces when a new output array is created. The programmer provides a unique ID when calling the interfaces, and SimplePIM determines other relevant metadata and registers the output array properly.

```
void simple_pim_array_register(array_meta_data_t* meta_data,
    simple_pim_management_t* management);
```

Free The ID is removed from the management unit, indicating that the array with that ID is no longer available for processing or communication.

```
void simple_pim_array_free(const char* id,
    simple_pim_management_t* management);
```

3.2. Communication Interface

The SimplePIM Communication Interface serves as a comprehensive solution for handling communication between the host CPU and PIM cores, and among PIM cores. This interface effectively manages the complexities of data transfer alignment, address calculation, and different PIM communication commands so that programmers need not worry about them.

To support host-PIM communication, SimplePIM provides three functions. The `broadcast` function sends the same array to all PIM cores. The `scatter` function divides a host array in equal-sized chunks and distributes them across the PIM DRAM banks. The `gather` function reassembles the scattered chunks into a host array.

To support communication among PIM cores, SimplePIM includes `allreduce` and `allgather` functions used in a variety of applications (e.g., machine learning).

Host-to-PIM Communication: SimplePIM Broadcast The `broadcast` function in SimplePIM transfers a host array to all PIM cores in the system, ensuring that all PIM cores have a local copy of the data, as shown in Fig. 2. It then registers the ID of the array with the management interface so that it can be referred to by other functions in the SimplePIM interface. This function can be useful for initializing data or for distributing data that needs to be accessed by all PIM cores.

In the `broadcast` function, the `arr` variable is the source array of the communication on the host side and the `type_size` variable represents the size of a single element for the array.

```
void simple_pim_array_broadcast(char* const id, void* arr,
    uint64_t len, uint32_t type_size, simple_pim_management_t
    * management);
```

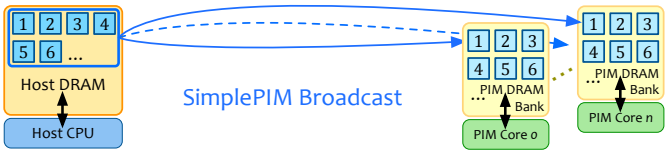

Figure 2: SimplePIM Broadcast function (example with two PIM cores)

Host-to-PIM Communication: SimplePIM Scatter The `scatter` function is designed to take an array located on the host DRAM and divide it into chunks that are distributed to each PIM core's DRAM bank (refer to Fig. 3). This division is performed almost evenly, while taking into account the PIM system's alignment constraints. For example, in the UPMEM

system, data transfers must be aligned to eight bytes. Once this division is complete, the `scatter` function registers the ID of the destination array in the PIM DRAM banks (`id`) with the management interface. As in the `broadcast` function, `arr` and `type_size` are, respectively, the source array and the array element size.

```
void simple_pim_array_scatter(char* const id, void* arr,
    uint64_t len, uint32_t type_size, simple_pim_management_t
    * management);
```

PIM-to-Host Communication: SimplePIM Gather The `gather` function is the counterpart of the scatter function, serving to reassemble a scattered array, as shown in Fig. 3. It works by taking an identifier that corresponds to the scattered array and retrieving the relevant information through the memory management interface. Using this information, the `gather` function retrieves the split portions of the array from each PIM core's DRAM bank, collects them, and reassembles the original array on the host. Finally, the `gather` function returns a pointer to the gathered array.

```
void* simple_pim_array_gather(char* const id,
    simple_pim_management_t* management);
```

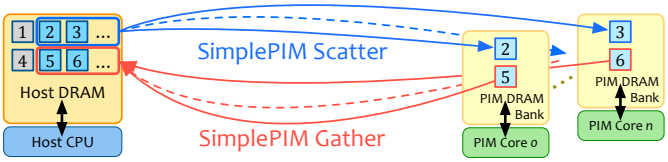

Figure 3: SimplePIM Scatter and Gather function (example with two PIM cores)

PIM-PIM Communication: SimplePIM AllReduce The `allreduce` function accepts arrays equal length across all PIM cores. To execute the `allreduce` operation, the programmer specifies an accumulative function (i.e., the reduction operation) and registers it as a function `handle`. SimplePIM then combines the arrays in place based on the programmer-defined function. Fig. 4 shows an example where the accumulative function is an addition. `allreduce` is often used for algorithm synchronization, for example in machine learning applications. Section 3.3 details how a programmer creates a function `handle`.

```
void simple_pim_array_allreduce(char* const id, handle_t*
    handle, simple_pim_management_t* management);
```

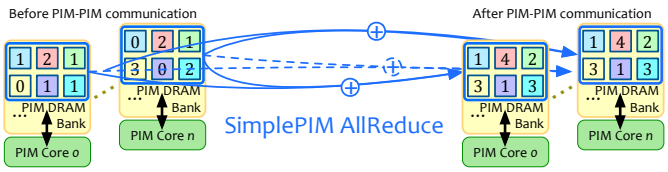

Figure 4: SimplePIM AllReduce function (example with two PIM cores)

PIM-PIM Communication: SimplePIM AllGather The SimplePIM `allgather` function retrieves sections of an array from various PIM cores, combines them, and distributes the complete array to all PIM cores, as shown in Fig. 5. This results in a new array with a unique identifier `new id`.

```
void simple_pim_array_allgather(char* const id, char* new_id,
    simple_pim_management_t* management);
```

Figure 5: SimplePIM AllGather function (example with two PIM cores)

3.3. Processing Interface

The SimplePIM Processing Interface provides iterators that enable the creation of a new array of data after performing operations on an existing array within the PIM system. These iterators can be called from the host and are parallelized automatically by the framework to execute across the PIM cores and threads. Once executed, the resulting output of the iterator is registered via the management interface for future reference.

Creation of a Function Handle In some PIM systems (such as UPMEM), PIM functions are loaded as independent binaries from the main program, compiled using a different compiler that targets the underlying PIM instruction set architecture. This separation ensures that CPU code does not reference PIM functions directly, although the CPU code must be able to pass PIM functions as inputs to SimplePIM's iterators. To facilitate this, the `create_handle` function reads a file containing a PIM function, compiles the function, and provides a handle to the CPU that can be passed as an input to the iterators. The `transformation_type` argument specifies which iterator the handle is for. The programmer-defined functions can execute with a context: the `data` of size `data_size` is broadcast to all PIM cores, and the programmer-defined functions receive this data to aid their executions. For example, for the linear regression workload, the programmer-defined function requires model weight data to compute gradients, and this data is provided as context via the `data` argument.

```
handle_t* simple_pim_create_handle(const char* func_filepath,
    uint32_t transformation_type, void* data, uint32_t
    data_size);
```

Array Map The `simple_pim_array_map` iterator takes a registered input array and a function `handle`, applies the `map_func` function to every element in the data array, and generates a new output array with `dest_id`, as Fig. 6 shows.

```
void simple_pim_array_map(const char* src_id, const char*
    dest_id, uint32_t output_type, handle_t* handle,
    simple_pim_management_t* management);
```

Array Reduction The `simple_pim_array_red` iterator processes each element in an input array `src_id`, calculates an index to an output array `dest_id` and performs a reduction onto the indexed output array element. This operation is similar to the general reduction method proposed in the

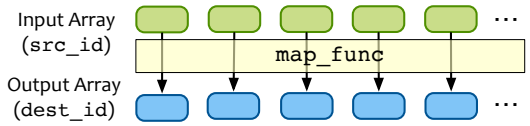

Figure 6: SimplePIM Array Map function

FREERIDE middleware [72]. Later, MATE [73] implemented general reduction and demonstrated its superior performance compared to MapReduce in a multi-core system. SimplePIM's PIM reduction iterator is versatile enough to support various essential applications such as linear regression, K-means, and histogram calculation.

To perform the `PIM array reduction`, the programmer needs to define three functions. The `init_func` initializes all entries in the output array. The `map_to_val_func` function transforms an input element to an output element and determines the corresponding entry in the output array to accumulate the current output element. Finally, the commutative function `acc_func` accumulates the output element that results of the `map_to_val_func` function onto the corresponding entry in the output array. Fig. 7 shows the usage of `map_to_val_func` and `acc_func` functions. In the histogram calculation of an image, for example, the `init_func` initializes all histogram bins to 0. The `map_to_val_func` computes for each image pixel which bin (i.e., an index in the histogram) it belongs to and returns 1. Finally, the `acc_func` is a simple addition that increments the bin count by 1.

```
void simple_pim_array_red(const char* src_id, const char*
    dest_id, uint32_t output_type, uint32_t output_len,
    handle_t* handle, simple_pim_management_t* management)
```

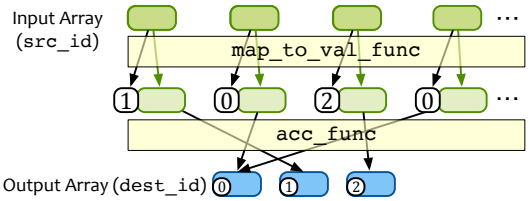

Figure 7: SimplePIM Array Reduction function

Array Zip The `simple_pim_array_zip` iterator takes as input the IDs of two registered arrays that are of the same length. It then generates an output array that combines the elements of the two input arrays, as Fig. 8 shows. By allowing programmers to work with multiple arrays as inputs to the iterators, this function enables greater flexibility in data processing.

```
void simple_pim_array_zip(const char* src1_id, const char*
    src2_id, const char* dest_id, simple_pim_management_t*
    management)
```

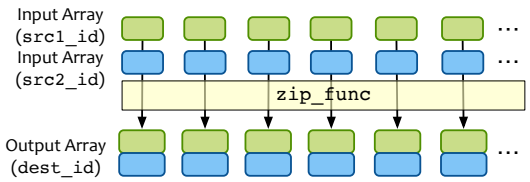

Figure 8: SimplePIM Array Zip function

4. UPMEM Implementation and Optimizations for SimplePIM

PIM cores access memory with higher bandwidth and lower latency (than CPUs or GPUs), making this architecture more compute-bound, and highlighting the importance of code optimization for optimal performance. In this section, we provide detailed information on how we develop and optimize the SimplePIM interface for the UPMEM system.

4.1. Communication Interface

UPMEM provides basic serial commands for transferring data between a PIM core and the host, as well as parallel commands for transferring same-sized continuous data between the host and different PIM cores. The parallel transfer bandwidth increases with the number of PIM cores and can be orders of magnitude higher than the serial transfer bandwidth.

In order to use the parallel transfer commands, memory transfers must be aligned and equal-sized for all PIM cores. SimplePIM automatically pads the communicated arrays and determines the transfer size for the parallel commands based on the array size, data type, and alignment requirements, ensuring that no array element is split across PIM cores and all alignment requirements are met. This provides a clean interface to the programmer, as described in Section 3.2.

SimplePIM leverages parallel data transfers between the UPMEM PIM cores and the host to implement both PIM-PIM and PIM-Host communication primitives. Although previous works such as ABC-DIMM [74] and DIMM-Link [75] demonstrate the potential benefits of hardware mechanisms for improving PIM-to-PIM communication, programmers currently do not have access to a physical direct communication channel among UPMEM PIM cores. In future PIM systems with more efficient communication among PIM cores enabled by hardware, SimplePIM could take advantage of these features to implement PIM-PIM communication more effectively.

4.2. Processing Interface

4.2.1. Array Map. An input array on the host system can be evenly split among the PIM cores using communication primitives. When the `map` iterator is called from the host, all relevant PIM cores are simultaneously invoked to process their respective local arrays. On each of these PIM cores, 12 PIM threads are launched by default, as at least 11 threads are required to fully utilize the pipelined in-order cores of the UPMEM architecture [26, 53]. While we have chosen 12 threads as a convenient even number, the programmers have the flexibility to configure SimplePIM to run any desired number of threads. In SimplePIM, each PIM thread loads its assigned section of the input array from the PIM DRAM bank to the 64KB scratchpad in batches, maximizing the scratchpad-to-DRAM bandwidth. The thread then applies the `map_func` to all elements of the input data and stores the results in the corresponding output address in batches as well.

4.2.2. Array Reduction. To perform array reduction, SimplePIM activates all PIM cores containing segments of the

input and launches 12 PIM threads on each core. These threads operate on the input data in batches, and each PIM core generates its intermediate reduction result in parallel using the programmer-defined `acc_func` PIM function. These intermediate results are then gathered to the host and combined using a host version of `acc_func` with the help of OpenMP [76].

Since an entry in the output array is accessed once per input element, a straightforward optimization is to keep the output array in the 64KB scratchpad memory, if it fits. SimplePIM offers two variants of in-scratchpad `PIM array reduction`: shared and thread-private accumulators.

Shared Accumulator Reduction In this variant, only one reduction output array is present in the scratchpad memory, and one lock is preserved per array entry. One thread initializes this array at the beginning with `init_func`. Then, every thread works on its part of the input array and updates the global output array entries after having acquired the lock associated with each entry.

Thread-Private Accumulator Reduction In this variant, every thread initializes its own local output array with `init_func` and reduces its input segment to that local output array with the `map_to_val` and `acc_func` functions. Once all threads finish the reduction, local outputs are merged in parallel in a ring-reduce manner with `acc_func` and barriers.

These two implementations have a tradeoff between synchronization overhead and scratchpad capacity utilization. The framework automatically chooses an appropriate in-scratchpad reduction variant based on the array sizes and data types. We evaluate the tradeoff between these two variants in Section 5.4.

4.2.3. Array Zip. The `zip` iterator is implemented with a lazy approach to minimize data copying. When the zip iterator is called, the management interface stores the starting addresses, data type sizes, and a common length for the two arrays to be zipped, but does not physically combine them. When a subsequent iterator is passed a lazily zipped array, the management interface extracts the addresses of the two contained arrays and copies the arrays in batches to the scratchpad. Then, the batches are zipped in the scratchpad, and the iterator operation is performed. This approach ensures that data is copied only once in the same loop, reducing both copying and looping overheads. The management interface keeps track of lazily zipped arrays and determines the behavior of the iterators. Currently, the laziness in our implementation is one level deep, which is sufficient for targeting the common case (i.e., multi-input map and reduce functions). If the zip iterator is called with an already lazily zipped array as an argument, the arrays are streamed in batches, combined physically, and stored back to memory in batches. Our experiments (not plotted in the paper) show that lazy zipping improves the performance of vector addition, for example, by more than 2×.

4.3. General Code Optimizations

We implement several code optimization techniques in SimplePIM that are essential for achieving high performance on UPMEM. Some open-sourced UPMEM applications use only some of these optimizations, making them potentially slower than SimplePIM implementations. We list some of the optimizations SimplePIM employs:

1. *Strength Reduction*: The UPMEM chip lacks native support for 32/64-bit integer multiplication, which must be emulated by runtime software and can take tens of cycles to complete. To overcome this challenge, SimplePIM strives to minimize the use of multiplications in the main loop. This is especially important for a general-purpose framework like SimplePIM, where the compiler may not always perform strength reduction automatically. Although the array data type is not known at compile time, the sizes of array elements are often powers of two. SimplePIM takes advantage of this fact by replacing multiplications with shift operations in array offset calculations when the array size is a power of two.
2. *Loop Unrolling*: We have found that loop unrolling can improve the performance of vector addition by up to 20% on UPMEM. We attribute this performance gain to fewer loop counter increments and loop branches. However, loop unrolling can also increase the binary size, which may eventually not fit into the instruction memory of PIM cores. Our SimplePIM implementation for UPMEM uses limited unrolling depth.
3. *Avoiding Boundary Checks*: Many open-sourced UPMEM applications check array boundaries inside the main loop for convenience. In SimplePIM, we evenly pre-partition the work among threads and then process the trailing part of the array separately to avoid boundary checks. For example, we have experienced more than 10% performance degradation due to boundary checks for the vector addition application.
4. *Function Inlining*: SimplePIM inlines programmer-defined functions in the iterator code to avoid the function invocation overhead in the iterator loops. That is, at handle creation time, the functions are not compiled independently, but rather, the iterators that use the functions are also compiled. Compared to compiling iterators and functions separately, we have found that inlining improves the performance of vector addition by more than 2×.
5. Previous studies have shown that the performance of data transfers between the PIM core's scratchpad memory and the corresponding DRAM bank in UPMEM is highly dependent on the data transfer size [26]. We observe that, in previously open-sourced implementations, programmers fix the scratchpad-to-DRAM transfer sizes for convenience, which results in suboptimal performance. An example is a linear regression implementation where the input dimensions are different for different datasets, which would require manual adjustment of transfer sizes. In contrast, SimplePIM automatically and dynamically adapts the scratchpad-to-DRAM transfer size to the input data size and type, achieving better performance while freeing programmers from applying low-level optimizations and allowing them to focus on the application logic.

5. Evaluation

5.1. Benchmarks

We evaluate SimplePIM using six commonly-used PIM-friendly applications: reduction, vector addition, histogram, linear regression, logistic regression, and K-means clustering. As a baseline for comparison, we use hand-tuned implementations from prior works [10–12, 26, 53]. The baselines for reduction, vector addition, and histogram are from the PrIM benchmark suite [26,53,77], while the baselines for the three machine learning algorithms, i.e., linear regression, logistic regression, and K-means, are from [10–12,78]. These prior works develop clean and high-performance open-source codes for benchmarking against CPU and GPU implementations. As such, these codes serve as solid baselines for comparison with our SimplePIM implementations.

Vector Addition We implement `vector addition` in SimplePIM by zipping the input arrays and performing element-wise addition using the `map` iterator. SimplePIM automatically optimizes this operation by performing lazy zipping on the UPMEM device, which results in a high-performance implementation. We evaluate the runtime of `vector addition` for one million 32-bit integer elements per PIM core for weak scaling and 608,000,000 32-bit integer elements for strong scaling, similar to the reference work that provides the baseline implementation [26, 53].

Reduction The `reduction` operation computes the sum of all elements in an input array. In SimplePIM, we implement `reduction` using `PIM array reduction` with an output array of a single element (an accumulator). We select the same number of input elements for weak scaling and strong scaling as for the `vector addition` workload.

Histogram The `histogram` operation is implemented using `PIM array reduction`. We define a programmer-specific `map_to_val` function to compute the corresponding bin for each input element, and a simple addition is used to combine the element counts for each bin. We conducted experiments with 1,572,864 elements per PIM core for weak scaling and 956,301,312 for strong scaling, with the number of histogram bins set to 256 to ensure consistency with the reference baseline implementation [26, 53].

K-Means The UPMEM PIM device currently emulates floating point operations in software, resulting in significantly slower performance than integer operations. To mitigate this issue, our `k-means` benchmark employs input data quantization to integers, following the approach outlined in [10–12]. We conduct experiments with 10 centroids and 10 feature dimensions, using 10,000 elements per PIM core for weak scaling experiments and 6,080,000 elements for strong scaling experiments.

Linear Regression To address the issue of slow floating point operations, we rely on the baseline approach proposed in [10–12], which includes several versions of `linear regression` using various quantization techniques. For our experiments, we use the implementation that employs 32-bit integer operations with bit shifts to prevent integer overflow and underflow. To ensure a fair comparison, we verify that our SimplePIM implementation produces identical results to the baseline approach. We use a feature dimension of ten and generate 10,000 data points per PIM core for weak scaling tests, while for strong scaling tests, we generate a total of 6,080,000 data points, similar to those used in the work that provides the baseline implementation [10–12].

Logistic Regression To enable a fair comparison with the baseline approach [10–12], we apply the same quantization technique used in `linear regression` to `logistic regression`. To minimize computational overhead, we adopt the Taylor series approximation of the sigmoid activation function [79] that the baseline [10–12] uses. However, since the runtime of the approximation depends on the input, we ensure a fair comparison by using the same inputs and initial model weights for both the baseline code [10–12] and the SimplePIM implementation. We also verify that the exact same output is produced. The weak and strong scaling datasets have the same sizes as for `linear regression`.

5.2. Productivity Improvement

Efficiently implementing PIM kernels for these applications requires a significant amount of engineering effort. We measure the programming complexity by counting the lines of effective PIM-related code for each application. This excludes the common code for data loading from a file to the host main memory, host memory allocation, variable definition, and time measurements. We only take into account PIM-related data transfers and PIM kernel execution. Table 1 summarizes the lines of effective code saved by using SimplePIM for the six benchmarks. The amount of coding is reduced by a factor of $2.98\times$ to $5.93\times$ with SimplePIM.

	SimplePIM	Hand-optimized	LoC Reduction
Reduction	14	83	5.93×
Vector Addition	14	82	5.86×
Histogram	21	114	5.43×
Linear Regression	48	157	3.27×
Logistic Regression	59	176	2.98×
K-Means	68	206	3.03×

Table 1: Lines of effective PIM-related code for each benchmark. "LoC Reduction" stands for SimplePIM's reduction in lines of code over hand-optimized baselines.

Productivity improvement is not only achieved by reducing the lines of code, but also by allowing programmers to write plain and easily understandable C code for an uncommon (PIM) architecture. Listing 1 shows the code required to implement a hand-optimized `histogram` operation on the UPMEM PIM architecture. In this code, variable declarations, initialization, as well as code for initializing and computing the histogram are omitted. To program the PIM system, the programmer is responsible for addressing the data structures (based on thread ID or `tasklet_id`) and must be familiar with architecture-specific instructions, such as `mem_reset` (line 5), `mram_read`

(line 14), and `mram_write` (lines 26 & 29). The programmer needs to read the documentation carefully and understand the instructions and their low-level properties in detail. For example, the `mram_read` and `mram_write` instructions, used for DRAM-scratchpad transfers, have an 8-byte alignment requirement and a 2,048-byte transfer limit, and the programmer needs to handle larger transfers manually, as shown in lines 28-30 of Listing 1. Additionally, the code allocates a 2,048-byte buffer for input data transfers (line 7). For more complex applications with variable input element size, such as `linear regression`, the programmer needs to handle data transfers with greater care and effort.

```
1  ... // Initialize global variables and functions for histogram
2  int main_kernel() {
3    ... // Initialize variables and the histogram
4    if (tasklet_id == 0)
5      mem_reset(); // Reset the heap
6    // Allocate buffer in scratchpad memory
7    T *input_buff_A = (T*)mem_alloc(2048);
8
9    for (unsigned int byte_index = base_tasklet; byte_index < input_size;
         byte_index += stride) {
10     // Boundary checking
11     uint32_t l_size_bytes = (byte_index + 2048 >= input_size) ? (
         input_size - byte_index) : 2048;
12
13     // Load scratchpad with a DRAM block
14     mram_read((const __mram_ptr void*)(mram_base_addr_A + byte_index),
         input_buff_A, l_size_bytes);
15
16     // Histogram calculation
17     histogram(hist, bins, input_buff_A, l_size_bytes/sizeof(uint32_t));
18   }
19   ...
20   barrier_wait(&my_barrier); // Barrier to synchronize PIM threads
21   ... // Merging histograms from different tasklets into one histo_dpu
22
23   // Write result from scratchpad to DRAM
24   if (tasklet_id == 0) {
25     if (bins * sizeof(uint32_t) <= 2048)
26       mram_write(histo_dpu, (__mram_ptr void*)mram_base_addr_histo,
         bins * sizeof(uint32_t));
27     else
28       for (unsigned int offset = 0; offset < ((bins * sizeof(uint32_t))
         >> 11); offset++) {
29         mram_write(histo_dpu + (offset << 9), (__mram_ptr void*)(
         mram_base_addr_histo + (offset << 11)), 2048);
30       }
31   }
32   return 0;
33 }
```

Listing 1: Hand-optimized `histogram` code using the UPMEM SDK

Listing 2 illustrates how SimplePIM makes it possible for programmers to implement `histogram` without resorting to any hardware-specific instruction or function calls. The code is straightforward and can be easily understood and implemented by any C programmer. The process involves defining application logic functions (lines 2-14) for the iterator in a separate file (called `histo_filepath` in this example), creating a handle (line 17), and running the iterator with only two additional lines of code on the host side (lines 20 & 23). Similar to Listing 1, the code example in Listing 2 does not show the host-side allocation and data transfers.

In summary, SimplePIM improves programming productivity for PIM systems. It reduces the number of lines of code

```
1  // Programmer-defined functions in the file "histo_filepath"
2  void init_func (uint32_t size, void* ptr) {
3    char* casted_value_ptr = (char*) ptr;
4    for (int i = 0; i < size; i++)
5      casted_value_ptr[i] = 0;
6  }
7  void acc_func (void* dest, void* src) {
8    *(uint32_t*)dest += *(uint32_t*)src;
9  }
10 void map_to_val_func (void* input, void* output, uint32_t* key) {
11   uint32_t d = *((uint32_t*)input);
12   *(uint32_t*)output = 1;
13   *key = d * bins >> 12;
14 }
15
16 // Host side handle creation and iterator call
17 handle_t* handle = simple_pim_create_handle("histo_filepath", REDUCE,
       NULL, 0);
18
19 // Transfer (scatter) data to PIM, register as "t1"
20 simple_pim_array_scatter("t1", src, bins, sizeof(T), management);
21
22 // Run histogram on "t1" and produce "t2"
23 simple_pim_array_red("t1", "t2", sizeof(T), bins, handle, management);
```

Listing 2: SimplePIM `histogram` code

required for an application and abstracts away the underlying architectural complexities, such as managing a scratchpad or software-managed cache, synchronizing PIM threads/cores, determining data transfer sizes and alignments, and allocating/deallocating PIM cores. By doing so, SimplePIM makes PIM programming more programmer-friendly and accessible to a wider range of developers.

5.3. Performance Evaluation

To compare the performance of our SimplePIM code to the baseline code, we conduct experiments on an UPMEM system with 2,432 PIM cores, and measure weak and strong scaling results for each workload on 608, 1,216, and 2,432 PIM cores, as shown in Fig. 9 and Fig. 10, respectively. The number of elements for each workload is similar to the baseline papers. The number of elements for strong scaling tests is set to be equal to the number of elements used for 608 cores in our weak scaling tests.

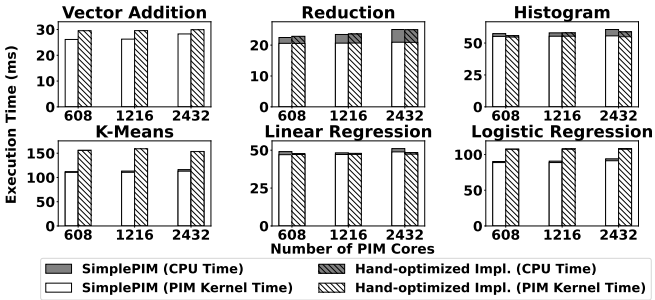

Figure 9: Weak Scaling results for six workloads.

Overall, the experimental results shown in Fig. 9 and Fig. 10 demonstrate that SimplePIM achieves comparable performance to the baseline approach for `reduction`, `histogram`, and `linear regression`, while consistently outperforming the baseline approach for `vector addition`, `logistic regression`, and `k-means` in both weak and strong scaling experiments. We attribute these performance gains to the vari-

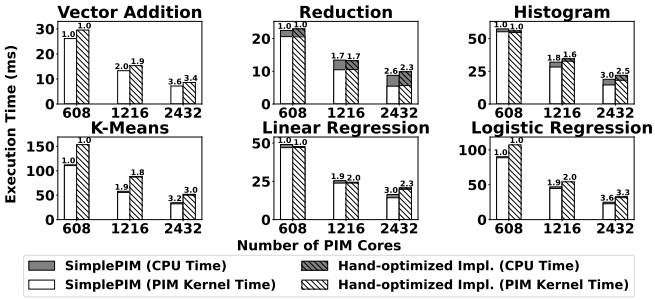

Figure 10: Strong Scaling results for six workloads. The numbers on each bar represent the speedup over 608 PIM cores.

ous optimization techniques implemented in our framework, as discussed in Section 4.

In the weak scaling evaluation results shown in Fig. 9, increasing the number of PIM cores linearly with the input size does not significantly impact the performance of both SimplePIM and hand-optimized code for all six applications. Furthermore, on average, SimplePIM outperforms the hand-optimized implementation in `vector addition`, `logistic regression`, and `k-means` by 1.10×, 1.17×, and 1.37×.

In the strong scaling evaluation results shown in Fig. 10, achieving linear speedup with additional PIM cores is not guaranteed, as communication overheads can become dominant. For the `reduction` workload, which has less kernel execution and comparatively more communication costs, we observe only 1.6× and 2.6× speedup for 2× and 4× more PIM cores. However, for the other five workloads, SimplePIM achieves more than 1.8× speedup with a 2× increase in PIM cores and 3× speedup with a 4× increase in PIM cores. SimplePIM consistently outperforms the hand-optimized implementations of all benchmarks, except for `reduction` with a slight increase in communication cost. SimplePIM outperforms the hand-optimized implementations of `vector addition`, `logistic regression`, and `k-means` by 1.15×, 1.22×, and 1.43× on average across different numbers of PIM cores. These results demonstrate the effectiveness of our framework for programming PIM systems in terms of performance.

We note that, while SimplePIM provides speedup over baseline implementations of some benchmarks, the performance of hand-optimized code can potentially be equal to or even better than that generated by SimplePIM. This requires the programmer to take similar steps as SimplePIM to optimize the code, as Section 4 describes. Achieving such performance requires careful consideration and use of various optimization techniques. SimplePIM frees the programmer from this burden, thereby making them more productive.

5.4. Evaluation of SimplePIM Variants of Array Reduction

SimplePIM provides two variants of `PIM array reduction` (Section 4.2.2). One variant uses shared accumulators (i.e., one shared output array for all threads and one lock per array entry) and the other one thread-private accumulators (i.e., one output array per thread). We compare these two versions of SimplePIM on the `histogram` benchmark with different histogram sizes.

Fig. 11 shows the effect of the two versions on the end-to-end performance of the `histogram` benchmark as we vary the number of bins in the histogram. We make several observations.

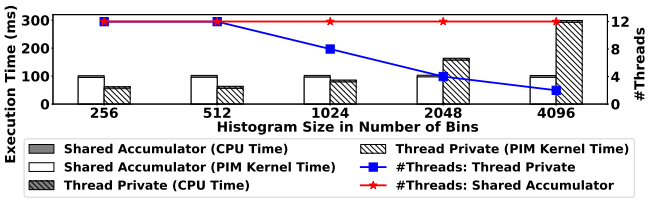

Figure 11: Execution time of SimplePIM's shared accumulator version and thread-private version for the `histogram` benchmark. Red and blue lines represent the numbers of active PIM threads.

First, the thread-private version is faster than the shared accumulator version for histograms of 256, 512, and 1024 bins. Since each thread owns a private output array, there is no need for locks, which avoids the synchronization overhead of the shared memory version. When 12 threads are active (for 256-, and 512-bin histograms), the thread-private version outperforms the shared accumulator version by 1.70×.

Second, the shared accumulator version outperforms the thread-private version for histograms of 2048 and 4096 bins. The cause is a reduction in the number of active PIM threads of the thread-private version (blue line) after 1024 bins. This reduction relates to the occupancy of the scratchpad memory. For t threads with private histograms of n bins (each of d bytes), $t \times n \times d$ bytes of the scratchpad memory are occupied. When the scratchpad size (e.g., 64KB in current UPMEM chips) is not enough for the private histograms (plus buffers for the input array), we should reduce the number of active threads, as we observe for 1024-, 2048-, and 4096-bin histograms. The reduction in the number of active threads causes a linear increase of the execution time, because the pipeline of the PIM cores is not fully busy [26, 53]. As a result, we observe that the execution time of the 2048-bin `histogram` (with 4 threads) is roughly twice as high as that of the 1024-bin `histogram` (with 8 threads). Ditto for the 4096-bin `histogram` (with 2 threads) versus the 2048-bin `histogram`.

6. Discussion

While SimplePIM is currently implemented for the UPMEM PIM architecture [43], it is devised for a broader set of real PIM architectures (e.g., [13, 25, 43, 55–58]), which have one or more common characteristics [12, 52]: (1) there is a host processor with access to standard main memory and PIM-enabled memory, (2) PIM processing elements (PEs) may need to communicate via the host processor, and (3) the number of PIM PEs scales with memory capacity.

With these characteristics in mind, SimplePIM supports host-PIM communication primitives and its management interface runs on the host. SimplePIM's PIM-PIM communication primitives emulate the communication between PIM cores by transparently handling the communication via the host. Given

that PIM-PIM communications play a pivotal role in simplifying and enhancing PIM programming for high performance, we recommend the research and development of more efficient PIM-PIM communication mechanisms in hardware, e.g., as proposed in [74, 75, 80–83].

SimplePIM can accommodate a wide range of computation patterns beyond the current `map`, `reduce`, and `zip` operations. Other parallel patterns, such as prefix sum and filter [26, 53, 77], can be easily incorporated. However, extending support to more complex patterns such as stencil and convolution would necessitate a more fine-grained scatter-gather mechanism to handle halo cells between tiles mapped onto different PIM cores. Similarly, applications with irregular memory access patterns, such as tree data structures [84], are also hard to support due to random access patterns. Future work can extend SimplePIM's capabilities by supporting, implementing, and benchmarking additional communication primitives, iterator functions, and workloads on a variety of PIM systems.

7. Related Work

To our knowledge, SimplePIM is the first high-level programming framework specifically designed for real PIM systems. We first review recent studies of real PIM systems. We then discuss several PIM works that propose programming interfaces and compilers for PIM architectures.

Studies of Real PIM Systems The UPMEM PIM architecture [43] is the first commercially available PIM hardware. Several recent works study this architecture and its suitability to different modern applications. Gómez-Luna et al. [26, 53, 85] present a microbenchmark-based analysis of the UPMEM PIM architecture and a workload suitability study with the PrIM benchmark suite [77], which contains workloads from dense and sparse linear algebra, machine learning, bioinformatics, image processing, graph processing, etc. Nider et al. [48] analyze the UPMEM PIM system for encryption/decryption, compression/decompression, hyper-dimensional computing, and text processing. Other works focus on specific applications or application domains, such as sparse matrix vector multiplication [31, 86], bioinformatics [40, 46, 47, 50, 51, 87], machine learning [10–12, 88], transcendental functions [52], databases [29, 84, 89–91], homomorphic encryption [54], and skyline computation [45].

There are also application studies on other real PIM systems [13, 25, 55–57]. Ke et al. [13] evaluate sparse embedding operators of deep-learning-based recommendation inference [92] on AxDIMM. Lee et al. [25] implement database scan operations on AxDIMM. Ibrahim and Aga [93] implement FFT for commercial PIM architectures, such as Samsung HBM-PIM [55, 56] and SK Hynix AiM [57] (but the evaluation is done with a performance model).

Programming Interfaces for PIM Several works propose programming interfaces for processing-near-memory architectures (i.e., PIM architectures with processing elements near the memory arrays [1]). One approach is to use specialized PIM instructions (e.g., as in [7, 8]) and integrate them into the existing general-purpose sequential execution model. This approach is especially suitable for PIM architectures where communication across PIM processing elements is not possible or easy. When the host processor finds a specialized PIM instruction in the program, it offloads the execution to the PIM processing elements. Another approach is to use remote function calls via message passing between different PIM cores (e.g., as in [5]). This approach is suitable for coarse-grained PIM accelerators with multiple PIM cores that can communicate over an interconnection network.

For processing-using-memory architectures (i.e., PIM architectures that compute by leveraging the analog operational properties of memory components [1]), there are several works that facilitate programming. SIMDRAM [94] provides a framework to generate user-defined operations that are executed via the simultaneous activation of rows inside DRAM subarrays [27, 95]. pLUTo [96] proposes a LUT-based processing-using-DRAM substrate with a compiler that maps complex operations onto LUT queries.

Compilers for PIM Several works propose compilers for simulated PIM architectures. Duality Cache [97] proposes a compiler that accepts existing CUDA programs and maps the computation onto a processing-using-SRAM substrate. Infinity Stream [98] proposes an intermediate representation and a just-in-time compiler for processing-near-memory, processing-using-memory, and host execution. CHOPPER [99] presents a bit-serial compiler for processing-using-DRAM substrates. CINM [100] is a compiler flow for simulated processing-using-memory architectures and processing-near-memory architectures such as UPMEM. It is based on MLIR [101, 102] and it supports linear algebra PIM kernels. SimplePIM is the first programming framework for real PIM architectures that supports a wide variety of PIM kernels.

8. Conclusion

We introduce SimplePIM, which is a new high-level programming framework for real PIM systems. SimplePIM is specifically designed to enable efficient and productive programming. SimplePIM efficiently leverages the host CPU for data management, and incorporates primitives for PIM-host communication (e.g, `scatter`, `gather`, `broadcast`) and primitives for communication among PIM cores (e.g., `allreduce`, `allgather`). SimplePIM's easy-to-use interface provides iterators (e.g., `map`, `reduce`, `zip`) that allow programmers to largely avoid the complexity of the underlying PIM architecture. Our implementation of SimplePIM for the UPMEM system enables programmers to develop PIM programs with $2.98\times$ to $5.93\times$ less code than hand-optimized programs, while providing equal or higher performance. SimplePIM frees programmers from dealing with the complexities and idiosyncracies of the low-level PIM hardware, while also enabling programmer-transparent code optimizations. We believe that SimplePIM is a milestone to ease the programmability and adoption of current and future real PIM systems. We hope that more future work ensues in programming memory-centric systems, and to fa-

cilitate that we open-source our SimplePIM framework at `https://github.com/CMU-SAFARI/SimplePIM`.

Acknowledgments

We acknowledge support from the SAFARI Research Group's industrial partners, especially Google, Huawei, Intel, Microsoft, VMware, and the Semiconductor Research Corporation. This research was partially supported by the ETH Future Computing Laboratory and the European Union's Horizon programme for research and innovation under grant agreement No. 101047160, project BioPIM (Processing-in-memory architectures and programming libraries for bioinformatics algorithms). This research was also partially supported by ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

References

[1] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," *arXiv*, vol. abs/2012.03112, 2020.
[2] O. Mutlu *et al.*, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MicPro*, 2019.
[3] S. Ghose *et al.*, "Processing-in-Memory: A Workload-driven Perspective," *IBM JRD*, 2019.
[4] O. Mutlu *et al.*, "Enabling Practical Processing in and near Memory for Data-Intensive Computing," *DAC*, 2019.
[5] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," *ISCA*, 2015.
[6] S. Angizi and D. Fan, "Graphide: A Graph Processing Accelerator Leveraging In-DRAM-computing," in *GLSVLSI*, 2019.
[7] L. Nai *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[8] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[9] M. Gokhale *et al.*, "Near Memory Data Structure Rearrangement," in *MEMSYS*, 2015.
[10] J. Gómez-Luria *et al.*, "Machine Learning Training on a Real Processing-in-Memory System," in *ISVLSI*, 2022.
[11] J. Gómez-Luna *et al.*, "Evaluating Machine Learning Workloads on Memory-Centric Computing Systems," in *ISPASS*, 2023.
[12] J. Gómez-Luna *et al.*, "An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System," *arXiv preprint arXiv:2207.07886*, 2022.
[13] L. Ke *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2021.
[14] F. Liu *et al.*, "PIM-DH: ReRAM-based Processing-in-Memory Architecture for Deep Hashing Acceleration," *DAC*, 2022.
[15] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[16] A. Boroumand *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," in *PACT*, 2021.
[17] L. Ke *et al.*, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ISCA*, 2020.
[18] N. Park *et al.*, "High-Throughput Near-Memory Processing on CNNs with 3D HBM-Like Memory," *TODAES*, 2021.
[19] B. Kim *et al.*, "MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks," *IEEE TC*, 2020.
[20] H. Sun *et al.*, "An Energy-Efficient Quantized and Regularized Training Framework for Processing-in-Memory Accelerators," in *ASP-DAC*, 2020.
[21] E. Azarkhish *et al.*, "Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes," *IEEE TPDS*, 2017.
[22] J. Park *et al.*, "TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory," in *MICRO*, 2021.
[23] J. H. Lee *et al.*, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *PACT*, 2015.
[24] M. He *et al.*, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," *MICRO*, 2020.
[25] D. Lee *et al.*, "Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM)," *DAMON*, 2022.
[26] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," *arXiv*, vol. abs/2105.03814, 2021.
[27] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," *MICRO*, 2017.
[28] A. Boroumand *et al.*, "Polynesia: Enabling Effective Hybrid Transactional Analytical Databases with Specialized Hardware Software Co-Design," in *ICDE*, 2022.
[29] C. Lim *et al.*, "Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs," *SIGMOD*, 2023.
[30] S. Lloyd and M. Gokhale, "Near Memory Key/Value Lookup Acceleration," in *MEMSYS*, 2017.
[31] C. Giannoula *et al.*, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems," *arXiv*, vol. abs/2204.00900, 2022.
[32] S. Gupta *et al.*, "SCRIMP: A General Stochastic Computing Architecture using ReRAM in-Memory Processing," *DATE*, 2020.
[33] G. Singh *et al.*, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.
[34] G. Singh *et al.*, "Accelerating Weather Prediction using Near-Memory Reconfigurable Fabric," *ACM TRETS*, 2021.
[35] A. Denzler *et al.*, "Casper: Accelerating Stencil Computations Using Near-Cache Processing," *IEEE Access*, 2023.
[36] M. R. de Castro *et al.*, "SparkBLAST: Scalable BLAST Processing using In-memory Operations," *BMC Bioinformatics*, 2017.
[37] F. Zhang *et al.*, "PIM-Quantifier: A Processing-in-Memory Platform for mRNA Quantification," *DAC*, 2021.
[38] S. Angizi *et al.*, "PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly," *DAC*, 2020.
[39] S. Angizi *et al.*, "PIM-Aligner: A Processing-in-MRAM Platform for Biological Sequence Alignment," *DATE*, 2020.
[40] S. Diab *et al.*, "High-throughput Pairwise Alignment with the Wavefront Algorithm using Processing-in-Memory," in *HICOMB*, 2022.
[41] J. S. Kim *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping using Processing-in-memory Technologies," *BMC Genomics*, 2018.
[42] G. Singh *et al.*, "FPGA-based Near-memory Acceleration of Modern Data-intensive Applications," *IEEE Micro*, 2021.
[43] F. Devaux, "The True Processing In Memory Accelerator," *HCS*, 2019.
[44] UPMEM, "Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper)," 2018.
[45] V. Zois *et al.*, "Massively Parallel Skyline Computation for Processing-in-memory Architectures," *PACT*, 2018.
[46] D. Lavenier *et al.*, "DNA Mapping using Processor-in-Memory Architecture," *BIBM*, 2016.
[47] S. Diab *et al.*, "A Framework for High-throughput Sequence Alignment using Real Processing-in-memory Systems," *Bioinformatics*, 2023.
[48] J. Nider *et al.*, "A Case Study of Processing-in-Memory in off-the-Shelf Systems," in *USENIX Annual Technical Conference*, 2021.
[49] J. Nider *et al.*, "Processing in Storage Class Memory," in *HotStorage*, 2020.
[50] D. Lavenier *et al.*, "Variant Calling Parallelization on Processor-in-Memory Architecture," in *BIBM*, 2020.
[51] N. Abecassis *et al.*, "GAPiM: Discovering Genetic Variations on a Real Processing-in-Memory System," *bioRxiv*, 2023.
[52] M. Item *et al.*, "TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems," in *ISPASS*, 2023.
[53] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, 2022.
[54] H. Gupta *et al.*, "Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System," in *IISWC*, 2023.
[55] Y.-C. Kwon *et al.*, "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," *ISSCC*, 2021.
[56] S. Lee *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021.
[57] S. Lee *et al.*, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, 2022.
[58] D. Niu *et al.*, "184QPS/W 64Mb/mm2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.
[59] UPMEM, "UPMEM Software Development Kit (SDK)," https://sdk.upmem.com, 2023.
[60] J. C. Corbett *et al.*, "Spanner: Google's Globally-Distributed Database," *ACM TOCS*, 2012.
[61] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, 2008.
[62] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, 1998.
[63] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC*, 2014.
[64] P. Hunt *et al.*, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC*, 2010.
[65] H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE TC*, 1982.
[66] M. Castro, "Practical Byzantine fault tolerance," in *OSDI*, 1999.
[67] K. R. Driscoll *et al.*, "Byzantine Fault Tolerance, from Theory to Reality," in *SafeComp*, 2003.
[68] D. Dolev and H. R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM J. Comput.*, 1983.
[69] L. Lamport *et al.*, "The Byzantine Generals Problem," *ACM TOPLAS*, 1982.
[70] M. A. Zaharia *et al.*, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *NSDI*, 2012.
[71] M. P. I. Forum, "MPI: A Message-Passing Interface Standard," 1994.
[72] L. Glimcher *et al.*, "Scaling and Parallelizing a Scientific Feature Mining Application using a Cluster Middleware," *IPDPS*, 2004.
[73] W. Jiang *et al.*, "A Map-Reduce System with an Alternate API for Multi-core Environments," *CCGrid*, 2010.
[74] W. Sun *et al.*, "ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast," *ISCA*, 2021.

[75] Z. Zhou *et al.*, "DIMM-Link: Enabling Efficient Inter-DIMM Communication for Near-Memory Processing," *HPCA*, 2023.
[76] L. Dagum and R. Menon, "OpenMP: an Industry Standard API for Shared-memory Programming," 1998.
[77] SAFARI Research Group, "PrIM Benchmark Suite," https://github.com/CMU-SAFARI/prim-benchmarks.
[78] SAFARI Research Group, "PIM Machine Learning Training Benchmarks," https://github.com/CMU-SAFARI/pim-ml.
[79] Z. Qin *et al.*, "A Novel Approximation Methodology and Its Efficient VLSI Implementation for the Sigmoid Function," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.
[80] V. Seshadri *et al.*, "RowClone: Fast and Energy-efficient In-DRAM Bulk Data Copy and Initialization," *MICRO*, 2013.
[81] K. K. Chang *et al.*, "Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM," in *HPCA*, 2016.
[82] Y. Wang *et al.*, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.
[83] S. H. S. Rezaei *et al.*, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *CAL*, 2020.
[84] H. K. Kang *et al.*, "PIM-tree: A Skew-resistant Index for Processing-in-Memory," *Proc. VLDB Endow.*, 2022.
[85] J. Gómez-Luna *et al.*, "Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware," *IGSC*, 2021.
[86] C. Giannoula *et al.*, "Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Architectures," in *SIGMETRICS*, 2022.
[87] L.-C. Chen *et al.*, "UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification," in *DAC*, 2023.
[88] P. Das *et al.*, "Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware," in *SOCC*, 2022.
[89] A. Bernhardt *et al.*, "pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories," in *DaMoN*, 2023.
[90] A. Baumstark *et al.*, "Adaptive Query Compilation with Processing-in-Memory," in *ICDEW*, 2023.
[91] A. Baumstark *et al.*, "Accelerating Large Table Scan using Processing-In-Memory Technology," *BTW*, 2023.
[92] M. Naumov *et al.*, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *arXiv preprint arXiv:1906.00091*, 2019.
[93] M. A. Ibrahim and S. Aga, "Collaborative Acceleration for FFT on Commercial Processing-In-Memory Architectures," *arXiv preprint arXiv:2308.03973*, 2023.
[94] N. Hajinazar *et al.*, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.
[95] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
[96] J. D. Ferreira *et al.*, "pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables," in *MICRO*, 2022.
[97] D. Fujiki *et al.*, "Duality Cache for Data Parallel Acceleration," in *ISCA*, 2019.
[98] Z. Wang *et al.*, "Infinity Stream: Portable and Programmer-friendly In-/Near-memory Fusion," in *ASPLOS*, 2023.
[99] X. Peng *et al.*, "CHOPPER: A Compiler Infrastructure for Programmable Bit-serial SIMD Processing using Memory in DRAM," in *HPCA*, 2023.
[100] A. A. Khan *et al.*, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-memory and Compute Near-memory Paradigms," *arXiv preprint arXiv:2301.07486*, 2022.
[101] C. Lattner *et al.*, "MLIR: A Compiler Infrastructure for the End of Moore's Law," *arXiv preprint arXiv:2002.11054*, 2020.
[102] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *CGO*, 2021.