PIMSys: A Virtual Prototype for Processing in Memory

Derek Christ
derek.christ@iese.fraunhofer.de
Fraunhofer IESE
Kaiserslautern, Germany

Lukas Steiner
lukas.steiner@rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

Matthias Jung
m.jung@uni-wuerzburg.de
JMU Würzburg
Würzburg, Germany

Norbert Wehn
norbert.wehn@rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

ABSTRACT

Data-driven applications are increasingly central to our information technology society, propelled by AI techniques reshaping various sectors of our economy. Despite their transformative potential, these applications demand immense data processing, leading to significant energy consumption primarily in communication and data storage rather than computation. The concept of processing-in-memory (PIM) offers a solution by processing data within memory, reducing energy overheads associated with data transfer. PIM has been an enduring idea, with recent advancements in DRAM test chips integrating PIM functionality, indicating potential market adoption.

This paper introduces a virtual prototype of Samsung's PIM-HBM architecture, leveraging open-source tools like gem5 and DRAMSys, along with a custom Rust software library facilitating easy utilization of PIM functionality. Key contributions include the first gem5 based full-system simulation of PIM-HBM, experimental validation of the virtual platform with benchmarks, and the development of a Rust library enabling PIM functionality at the software level. Our benchmarks evaluated speedup for PIM in the range of 6.0 × to 17.5 × compared to a respective non-PIM system for different memory-bound workloads.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**; • **Computer systems organization** → *Parallel architectures*; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

DRAM, PIM, Virtual Platforms

1 INTRODUCTION

Data-driven applications are increasingly becoming the focal point of our information technology society, with AI techniques fundamentally altering various sectors of our society and economy. A common characteristic of these applications is the vast amount of data they require to be captured, stored, and processed. Consequently, many of these applications, e. g., large language models (LLMs) or other artificial intelligence workloads are bound by the memory performance. Furthermore, a significant portion of energy is consumed by communication and data storage rather than computation. As demonstrated by Jouppi et al. [8], in a 7nm process, a 32-bit floating-point multiplication requires 1.31 pJ, whereas a 64-bit DRAM memory access demands 1300 pJ. This energy is expended in transferring data from memory through the network on chip, arbiters, and various levels of caches. Hence, it would be considerably more energy-efficient to process data where it resides, particularly within the memory itself. This approach works very well with data-flow oriented applications. In other words, rather than transmitting data to computational units, the computational instructions should be sent to the memory housing the data.

This concept, known as processing-in-memory (PIM), has been around for many years. For instance, Stone already proposed it in the 1970s [26]. Since then, similar to the field of artificial intelligence, this idea has experienced "summer" and "winter" periods in research over the past decades. However, recently, different companies have developed DRAM test chips with integrated PIM functionality, showing promising potential for entry into the market.

For instance, UPMEM introduced the first publicly available real-world general-purpose PIM architecture [4]. UPMEM integrates standard DDR4 DIMM-based DRAM with a series of PIM-enabled UPMEM DIMMs containing multiple PIM chips. Each PIM chip houses eight DRAM processing units (DPUs), each with dedicated access to a 64 MiB memory bank, a 24 KiB instruction memory, and a 64 KiB scratchpad memory. These DPUs function as multithreaded 32-bit reduced instruction set computer (RISC) cores, featuring a complete set of general-purpose registers and a 14-stage pipeline [4]. Even prior to UPMEM, Micron introduced its automata processor [28]. It features a nondeterministic finite automaton (NFA) inside the DRAM to accelerate certain algorithms. In 2020, SK Hynix, a leading DRAM manufacturer, unveiled its PIM technology, named Newton, utilizing High Bandwidth Memory (HBM) [5]. Unlike UPMEM, Newton integrates small MAC units and buffers into the bank area of the DRAM to mitigate the area and power overhead of a fully programmable processor core. Following SK Hynix's lead, Samsung, another major DRAM manufacturer, announced its own PIM DRAM implementation named Function-In-Memory DRAM (PIM-HBM or FIMDRAM) one year later [11].

With these new architectures on the horizon, it becomes crucial for system-level designers to assess whether these promising developments can enhance their applications. Furthermore, these emerging hardware architectures necessitate new software paradigms. It remains unclear whether libraries, compilers, or operating systems will effectively manage these new devices at the software level. Therefore, it is imperative to establish comprehensive virtual platforms for these devices, enabling real applications to be tested within a realistic architectural and software platform context.

This paper introduces a virtual prototype of Samsung's PIM-HBM, developed using open-source tools such as gem5 [13] and the DRAM simulator DRAMSys [24]. Additionally, the virtual prototype is accompanied by a custom Rust software library, simplifying the utilization of PIM functionality at the software level.

In summary, this paper makes the following contributions:

- We propose, to the best of our knowledge, for the first time full-system simulation of PIM-HBM with a virtual platform consisting of gem5 and DRAMSys.
- We provide an experimental verification of the virtual prototype with benchmarks.
- We propose a modern Rust library to provide the PIM functionality up to the software level.

Using this novel full-system simulation framework, it is possible to evaluate the effectiveness of PIM-HBM for real-world applications in a detailed and realistic manner and to examine the implications of integrating this PIM solution into these applications.

The paper is structured as follows. Section 2 shows the related work in the area of PIM simulation. Section 3 gives a brief background on the relative PIM architectures, whereas Section 4 explains the proposed PIM virtual platform. The Sections 5 and 6 show experimental simulation setup and the results, which are compared with already published results from PIM vendors. The paper is finally concluded in Section 7.

2 RELATED WORK

Several virtual prototypes of PIM architectures have been object to research in the past. The authors of NAPEL [23] used Ramulator-PIM, which is based on the processor simulator ZSim [18] and the DRAM simulator Ramulator [9], to build a high-level performance and energy estimation framework. Yu et al. [31] introduced MultiPIM, a PIM simulator, which is also based on Ramulator and ZSim, capable of simulating parallel PIM cores, distributed over a memory network. However, these publications evaluate the PIM systems only from a high level of abstraction. With PIMSim [30], the authors provide a configurable PIM simulation framework that enables a full-system simulation of user-specified PIM logic cores. The authors of DP-Sim [32] present a full-stack infrastructure for PIM, based on a front-end that generates PIM instructions by instrumenting a host application and executing them in a PIM-enabled memory model. In a similar way, Sim^{2}PIM [3, 19] uses instrumentation to simulate only the PIM side of a host application. The MPU-Sim [29] simulator focuses on general-purpose near-bank processing units based on 3D DRAM technology, while neglecting the data transfers between the host CPU and the PIM devices. These instrumentation approaches are less accurate when it comes to integration with the host processor because they primarily focus on simulating the PIM units. Recently, the authors of [1] presented a novel Active Compute Memory (ACM) architecture that allows for key-value sorting within the Dynamic Random Access Memory (DRAM). To investigate the performance and energy improvements, they implemented a virtual prototype based on ZSim and DRAMSim3 [12]. A slightly different approach is taken by PiMulator [14], which does not simulate but emulates PIM implementations such as RowClone [20] or Ambit [21] by implementing a soft-model in an FPGA.

In addition to PIM architectures from research, there are also virtual prototypes of industry architectures. Very recently, the authors of [6] introduced uPIMulator, a cycle-accurate simulator that models UPMEM's real-world general-purpose PIM architecture. In addition to its automata processor, Micron introduced another PIM architecture called In-Memory Intelligence [2]. The new architecture places bit-serial computing elements at the sense amplifier level of the memory array. Evaluations of In-Memory Intelligence are based on a custom Micron discrete event simulator that implements the hardware models. Similarly, to analyze the potential performance and power impact of Newton, SK Hynix developed a virtual prototype based on the DRAMSim2 [16] cycle-accurate memory simulator, which models a High Bandwidth Memory 2 (HBM2) memory and the extended Newton DRAM protocol. However, DRAMSim2 is more than 10 years old and several orders of magnitude slower than DRAMSys [25]. The simulated system is compared with two different non-PIM systems: an ideal non-PIM host with infinite compute bandwidth and a GPU model of a high-end Titan-V graphics card using a cycle-accurate GPU simulator. SK Hynix finds that Newton achieves a $54\times$ speedup over the Titan-V GPU model and a speedup of $10\times$ for the ideal non-PIM case, setting a lower bound on the acceleration for every possible non-PIM architecture. With PIMSimulator [22], Samsung provides a virtual prototype of PIM-HBM, also based on DRAMSim2. PIM-Simulator offers two simulation modes: it can either accept pre-recorded memory traces or generate very simplified memory traffic using a minimal host processor model that essentially executes only the PIM-related program regions. However, both approaches do not accurately model a complete system consisting of a host processor running a real compiled binary and a memory system that integrates PIM-HBM. As a result, only limited conclusions can be drawn about the performance improvements of PIM-HBM and the necessary modifications to the application code to support the new architecture. In Samsung's findings, the simulated PIM-HBM system provides a speedup in the range of $2.1\times$ to $2.6\times$ depending on the simulated workload with an average speedup of $2.5\times$ compared to the system with standard HBM2 memory. Based on both the Newton and PIM-HBM architectures, PipePIM [7] pipelines the operation of the bank-level processing units, achieving speedups of $2.16\times$ and $1.74\times$, respectively, over the base PIM architectures. The simulation environment is based on Ramulator, but few details are given about how detailed the host is simulated.

Looking beyond the simulation frameworks presented, this work aims to provide a virtual prototype of an existing PIM architecture to enable functionally correct full-system simulations: from the integration of the PIM software stack into the application, over the detailed simulation of a processor running the real compiled binary, to the simulation of a model of PIM-HBM, while obeying the complex DRAM-related timing dependencies.

3 BACKGROUND DRAM-PIM

Many types of deep neural networks (DNNs) used for language and speech processing, such as recurrent neural networks (RNNs), multilayer perceptrons (MLPs) and some layers of convolutional neural networks (CNNs), are severely limited by the memory bandwidth that the DRAM can provide, making them *memory-bound* [5].

PIM is a good fit for accelerating memory-bound workloads with low operational intensity. In contrast, compute-bound workloads tend to have high data reuse and can make excessive use of the on-chip cache and therefore do not need to utilize the full memory bandwidth.

A large number of modern DNN layers can be expressed as a matrix-vector multiplication. The layer inputs can be represented as a vector and the model weights can be viewed as a matrix, where the number of columns is equal to the size of the input vector and the number of rows is equal to the size of the output vector. Pairwise multiplication of the input vector and a row of the matrix are used to calculate an entry of the output vector. Such an operation, defined in the widely used Basic Linear Algebra Subprograms (BLAS) library [15], is also known as a GEMV routine. Because one matrix element is only used exactly once in the calculation of the output vector, there is no data reuse in the matrix. Further, as the weight matrices tend to be too large to fit into the on-chip cache, such a GEMV operation is deeply memory-bound [5]. Consequently, such an operation is a good fit for PIM.

Many different PIM architectures have been proposed by researchers in the past, and more recently real implementations have been introduced by hardware vendors. These proposals differ largely in the location of the processing operation, ranging from analog distribution of capacitor charges at the DRAM subarray level to additional processing units at the global I/O level. Each of these approaches comes with different advantages and disadvantages. The closer the processing is located to the DRAM subarray, the higher the energy efficiency and achievable processing bandwidth, as a higher level of parallelism can be achieved. This is because the processing bandwidth is not limited by the narrow data bus, but by the respective hierarchical level of the processing units. On the other hand, the integration of the PIM units inside the memory array becomes more difficult as area and power constraints limit the integration [27].

One real PIM implementation of the DRAM manufacturer Samsung, called Function-In-Memory DRAM (PIM-HBM or FIMDRAM), was presented in 2021 [10, 11]. PIM-HBM is based on the HBM2 memory standard and it integrates 16-wide single-instruction multiple-data (SIMD) engines directly into the memory banks, exploiting bank-level parallelism, while preserving the highly optimized memory subarray [10]. A special feature of PIM-HBM is that it does not require any modifications to components of modern processors, such as the memory controller, i.e., it is agnostic to existing HBM2 platforms. Consequently, for the operation of the processing units (PUs), mode switching is required for PIM-HBM, which makes it less useful for interleaved PIM and non-PIM traffic and small batch sizes.

At the heart of PIM-HBM lie the PUs, where one of which is shared by two banks of the same pseudo channel (pCH). The architecture of such a PU is illustrated in Figure 1. A PU contains two sets of SIMD floating-point units (FPUs), one for addition and one for multiplication, where each set contains 16 16-bit wide FPUs each. Besides the FPUs, a PU contains a command register file (CRF), a general register file (GRF) and a scalar register file (SRF) [11]. The 16-wide SIMD units correspond to the 256-bit prefetch architecture of HBM2, where 16 16-bit floating-point operands are passed directly from the secondary sense amplifiers (SSAs) to the FPUs as

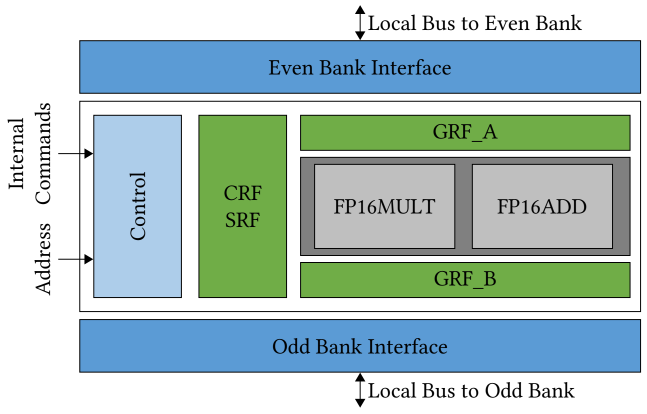


Figure 1: The architecture of a PU, according to [11].

the result of a single memory access. As all PIM units operate in parallel, with 16 banks per pCH, a singular memory access loads a total of 256 bit · 8 PUs = 2048 bit into the FPUs. As a result, the theoretical internal bandwidth of PIM-HBM is 8 × higher than the external bus bandwidth to the host processor.

PIM-HBM defines three operating modes: In the default **Single-Bank (SB) mode**, the PIM-HBM has identical behavior to normal HBM2 memory. To switch to another mode, a specific sequence of activate (ACT) and precharge (PRE) commands must be sent by the memory controller to specific row addresses. The **All-Bank (AB) mode** is an extension to the SB mode where the PIM execution units allow for concurrent access to half of the DRAM banks at the same time. This provides 8 × more bandwidth than the standard operation mode, which can be used for the initialization of memory regions across all banks. With another predefined DRAM access sequence, the memory switches to the **All-Bank-PIM (AB-PIM) mode**. In this mode, a single memory access initiates the concurrent execution of the next instruction across all processing units. In addition, the I/O circuits of the DRAM for the data bus are completely disabled in this mode, reducing the power required during PIM operation. Both in AB mode and in AB-PIM mode, the total HBM2 bandwidth per pCH of 16 GB/s is 8 × higher with 128 GB/s or in total 2 TB/s for 16 pCHs.

Due to the focus on DNN applications in PIM-HBM, the native data type for the FPUs are 16-bit floating-point (FP16) numbers, which is motivated by the significantly lower area and power requirements for FPUs compared to 32-bit floating-point numbers. The SIMD FPUs of the processing units are implemented as both an 16-wide FP16 multiplier unit and an 16-wide FP16 adder unit, providing support for these basic algorithmic operations.

The CRF acts as an instruction buffer, holding the 32 32-bit instructions to be executed by the processor when performing a memory access. A program that is stored in the CRF is called a *microkernel*. Each GRF consists of 16 registers, each with the HBM2 prefetch size of 256 bits, where each entry can hold the data of a full memory burst. The GRF of a processing unit is divided into two halves (GRF-A and GRF-B), with eight register entries allocated to each of the two banks. Finally, in the SRFs, a 16-bit scalar value is replicated 16 × as it is fed into the 16-wide SIMD FPU as a constant summand or factor for an addition or multiplication. It is also divided into two halves (SRF-A and SRF-M) for addition and multiplication with eight entries each. The PIM-HBM instruction set provides a total of 9 32-bit RISC instructions, each of which falls into one of three groups: control flow instructions (NOP, JUMP, EXIT), arithmetic instructions (ADD, MUL, MAC, MAD) and data movement instructions (MOV, FILL).

Since the execution of an instruction in the microkernel is initiated by a memory access, the host processor must execute load (LD) or store (ST) instructions in a sequence that perfectly matches the loaded PIM microkernel. When an instruction executes directly on data that is provided by a memory bank, the addresses of these memory accesses specify the exact row and column where the data should be loaded from or stored to. This means that the order of the respective memory accesses for such instructions is important and must not be reordered by the processor or memory controller, as it must match the corresponding instruction in the microkernel. One solution to this problem would be to introduce memory barriers between each LD and ST instruction of the processor, to prevent any reordering, however this comes at a significant performance cost and results in memory bandwidth being underutilized. To solve this overhead, Samsung has introduced the address aligned mode (AAM) mode for arithmetic instructions. In the AAM mode, the register indices of an instruction are ignored and decoded from the column and row address of the memory access itself. Using this approach, the register indices and bank addresses remain synchronized, even if the memory controller reorders the access order.

4 PIM VIRTUAL PLATFORM

To build a virtual prototype of PIM-HBM, an accurate model for HBM2 is needed, in which the additional PIM-PUs can be integrated. For this, the cycle-accurate DRAM simulator DRAMSys [25] is used and its HBM2 model is extended to include the previously described PUs into the pCHs of the PIM-activated channels. The PIM-HBM model itself does not need to model any timing behavior: its submodel is essentially untimed, since it is already synchronized with the operation of the DRAM model of DRAMSys. Consequently, the model focuses on implementing the functional behavior of PIM-HBM, while implicitly being accurate with respect to DRAM timing constraints. To achieve a full-system simulation, detailed processor and cache models are required in addition to the PIM-enabled memory system. For this, the gem5 simulator is used, which generates memory requests by executing the instructions of a compiled workload binary.

While PIM-HBM operates in the default SB mode, it behaves exactly like a normal HBM2 memory. Only when the host initiates a mode switch of one of the PIM-enabled pCHs, the processing units become active. When entering AB mode, the DRAM model ignores the specific bank address of incoming write (WR) commands and internally performs the write operation for either all even or all odd banks of the pCH, depending on the parity of the original bank index. After the transition to the AB mode, the DRAM can further transition to the AB-PIM mode, which allows the execution of instructions in the processing units. The AB-PIM mode is similar to the AB mode in that it also ignores the concrete bank address except for its parity, while additionally passing the column and row address and, in the case of a read, also the respective fetched bank data to the processing units. Only then, the PU model executes the instructions of the microkernel that operate on the read input data. In the case of a write access, the output of the processing unit is written directly into the corresponding bank, ignoring the actual data of the transaction object coming from the host processor. This is equivalent to the real PIM-HBM implementation, where the global I/O bus of the memory is not actually driven, and all data movement is done internally in the banks.

The model's internal state of a processing unit consists of the GRF register files GRF-A and GRF-B, the SRF register files SRF-A and SRF-M, the program counter, and a jump counter that keeps track of the current iteration of a JUMP instruction. Depending on a RD or WR command received from the DRAM model, the control flow is dispatched into one of two functions that execute an instruction in the CRF and increment the program counter of the corresponding PIM unit. Both functions calculate the register indices from the memory address that are used by the AAM execution mode, and dispatch using a branch table to the handler of the current instruction. In case of the data movement instructions MOV and FILL, the model executes a move operation that loads the value of one register or the bank data and assigns it to the destination register. The arithmetic instructions fetch the operand data from their respective sources and perform the operation, and write back the result by modifying the internal state of the PU. Note that while the MAC instruction can iteratively add to the same destination register, it can not reduce the 16-wide FP16 vector itself. Instead, it is the responsibility of the host processor to reduce these 16 floating point numbers to a single FP16 number that represents an entry in the output vector.

With this implementation of a PIM-HBM model, it is now possible to write a user program that controls the execution of the PIM-PUs directly in the HBM2 model. However, correctly placing the input data in the DRAM and arbitrating its execution is a non-trivial task. Therefore, a software library based on the Rust programming language [17] is provided. Due to its strict aliasing rules, Rust allows for a safe execution of the microkernels, as it can guarantee that the PIM data is not accessed by the program during operation of the PUs. The library contains the logic to safely switch between SB, AB and AB-PIM modes by writing to a designated memory region. Additionally, it offers data structures to facilitate the assembly and transfer of microkernels to the PIM units. In order to place input operands in a specific memory layout required for PIM, data structures are also provided to facilitate this. After mode switching and programming of the microkernel, the library implements functionality to execute a user-defined microkernel by issuing the necessary memory requests through the execution of LD and ST instructions.

The use of AAM requires a special memory layout so that the register indices are correctly calculated from the column and row addresses of a memory access. The mapping of an exemplary weight matrix used for a GEMV operation is illustrated in Figure 2. The actual memory layout in the linear address space required to achieve this mapping depends on the address mapping of the memory controller. To use all eight GRF-A registers of a PU, each matrix row must be placed in its own bank. Because most memory controllers implement bank interleaving, where adjacent memory accesses

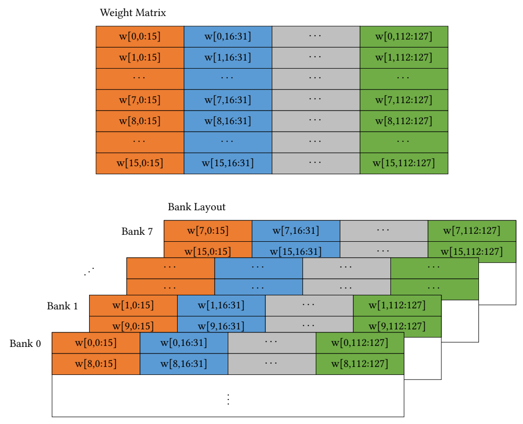


Figure 2: Mapping of the weight matrix onto the memory banks.

```
#[repr(C, align(32768))]
struct Matrix<const R: usize, const C: usize>(
    [[F16x16; R]; C / 16],
);
```

Listing 1: Pseudo code for the definition of a PIM-enabled FP16 matrix.

cycle between banks, the matrix must adhere to a column-major layout. In a column-major matrix layout, the entries of a column are stored sequentially before switching to the next column, according to the `MATRIX[R][C]` C-like array notation. However, the concrete element type of such an array is not a single FP16 element, but a vector of 16 FP16s packed together, since this corresponds to a 32 B memory access. This results in 16 FP16 matrix row elements being stored sequentially before switching to the next 16 FP16 elements in the next row of the same 16 columns, ensuring that a SIMD processing unit always contains the data of only its associated matrix row.

To guarantee the correct placement of the first matrix element at the boundary of the first bank of the pCH, an alignment for the matrix data structure of 512 B would need to be explicitly enforced. However, when using the AAM execution mode, this is not sufficient. As already mentioned in Section 3, the GRF-A and GRF-B indices are calculated from the column and row address of the triggering memory access. With an alignment of 512 B, no assumptions can be made about the initial value of the GRF-A and GRF-B indices, while for the execution of a complete GEMV kernel, both indices should start with zero. Therefore, to accommodate the additional six address bits corresponding to the indices, the weight matrix must be aligned to a stricter requirement of $2^6 \cdot 512\,\mathrm{B} = 32\,768\,\mathrm{B}$. The simplified pseudo code for defining a matrix with R rows and C columns is given in Listing 1. It is important to note that while the matrix itself follows a column-major layout, 16 FP16 elements are packed together.

Following operand initialization, the host processor proceeds to execute the PIM microkernel. It begins by transitioning to the AB-PIM mode and subsequently issues the necessary memory RD and WR requests through the execution of LD and ST instructions. When executing control instructions or data movement instructions that operate only on the register files, the RD and WR requests must be located in a dummy region of memory where no actual data is stored, but which must be reserved for that purpose. Further, when data is read from or written to the memory banks, these memory requests are issued with the correct address for the data. As half the banks in a pCH operate at the same time, from the viewpoint of the host processor, the data accesses occur very sparsely. In the case of the input vector, where one 16-wide SIMD vector of FP16 elements is repeated as often as there are banks in a pCH, a burst access must occur every $32\,\mathrm{B} \cdot \text{number of banks per pCH} = 512\,\mathrm{B}$ over the entire interleaved input vector for a maximum of $8\times$. To then perform the repeated MAC operation with the weight matrix as bank data, a similar logic must be applied. Since each row of the matrix resides in its own memory bank, with an interleaving of the size of a 16-wide SIMD vector of FP16 elements, also one memory access must be issued every 512 B. As the input address of the weight matrix grows, the GRF-A and GRF-B indices are incremented in such a way that the GRF-A registers are read repeatedly to multiply the weights by the input vector, while the GRF-B registers are incremented in the outer loop to hold the results of additional matrix rows.

Besides generating memory requests, an important task of the software library is to maintain the data coherence of the program. The compiler may introduce invariants with respect to the value of the output vector, since it does not observe that the value of the vector has changed without the host explicitly writing to it. As a result, the compiler may make optimizations that are not obvious to the programmer, such as reordering memory accesses, that cause the program to execute incorrectly. To avoid this, not only between non-AAM instructions in the microkernel, but also after initializing the input operands and before reading the output vector, memory barriers must be introduced to ensure that all memory accesses and PIM operations are completed.

When performing a gem5 simulation, there are three options to choose from: syscall emulation mode, full-system Linux mode, and full-system bare-metal mode. Due to the added system complexity of simulating a complete operating system, the bare-metal option was chosen over the full-system Linux mode. A self-written kernel provides full control for implementing a minimal example using PIM-HBM, but some setup is required, such as initializing page tables for memory management.

5 SIMULATIONS

Our simulations are based on the gem5 simulator and the DRAMSys memory simulator. The comparison between non-PIM and PIM architectures considers a hypothetical ARM host processor with infinite compute capacity. In this ideal approach, memory bandwidth is the only limiting constraint, so only memory-bound effects are considered. This approach provides a lower bound on the possible speedups PIM can achieve: As the memory bound can only become less significant, real systems will see higher speedups due to the additional compute overhead. The configuration of HBM2 DRAM is summarized in Table 1.

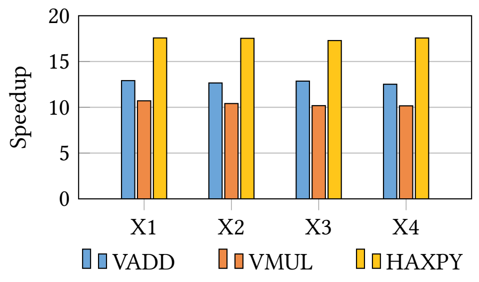
Our benchmarks are divided into two classes: vector benchmarks, which perform level 1 BLAS-like operations, and matrix-vector

Parameter	Description	Value
Number of Bank Groups	Bank Groups per pCH	4
Number of Banks	Banks per pCH	16
Number of pCHs	pCHs per Channel	2
Number of Channels	Total Number of Channels	1
Number of Columns	Columns per Memory Array	128
Number of Rows	Rows per Memory Array	65536
Width	Width of the Data Bus	64

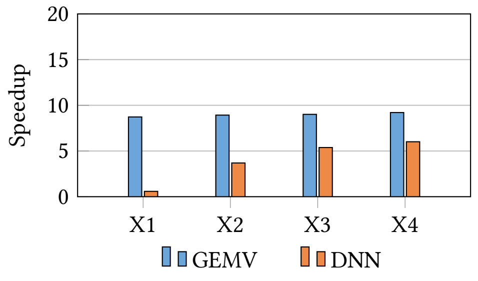
Table 1: The configuration of HBM2.

Level	Vector	GEMV	DNN
X1	2M	(1024×4096)	(256×256)
X2	4M	(2048×4096)	(512×512)
X3	8M	(4096×8192)	(1024×1024)
X4	16M	(8192×8192)	(2048×2048)

Table 2: Operand dimensions.



(a) Vector Benchmarks



(b) Matrix-Vector Benchmarks

Figure 3: Speedup of PIM compared to non-PIM.

benchmarks, which perform level 2 BLAS operations. Both classes of benchmarks are typically memory-bound, since little or no data is reused during the operation. For the first class of benchmarks, two FP16 vectors are added (VADD), multiplied (VMUL), or combined in a half precision $a \cdot x + y$ (HAXPY) fashion. The second class of benchmarks performs a GEMV matrix-vector multiplication or models a simple fully connected neural network with multiple layers and applying the activation function rectified linear unit (ReLU) in between. The ReLU operation is executed in PIM-HBM during a MOV instruction, by setting a specific instruction flag. Between the network layers, control is switched back to the host, since it must first reduce the partial sums computed by PIM-HBM to produce the input vector of the next layer. Each benchmark is executed with a set of different operand dimensions, called levels, which are listed in Table 2. The column for the vector benchmark describes the dimension of both operand vectors, while the columns for the GEMV and DNN benchmarks describe the matrix dimensions.

The benchmarks' focus lies on the achievable performance gain of PIM. In each run simulation, the relative performance (speedup) of PIM compared to non-PIM is analyzed.

6 RESULTS

The results in Figure 3 show significant speedups for all vector benchmarks in all simulated operand dimensions, with the following average values: 12.7 × for VADD, 10.4 × for VMUL and 17.5 × for HAXPY. On the other hand, the achieved speedup for the matrix-vector simulations varies with the simulated operand dimensions. The GEMV benchmark achieved a speedup in the range 8.7 × to 9.2 × with an average value of 9.0 ×, while the fully connected neural network layers experience a higher variance: With a range of 0.6 × to 6.0 ×, the DNN benchmark experiences both a slowdown and an acceleration of the inference time. Therefore, there is a break-even point between dimensions X1 and X2 where PIM can be expected to become viable.

In addition to its own virtual prototype, Samsung used a real hardware accelerator platform for its analysis, based on a unmodified high-end processor with 60 compute units and using real manufactured PIM-HBM memory packages. Similar to the simulation setup of this paper, Samsung has used different input dimensions for its microbenchmarks for both its GEMV and its vector ADD workloads. These are consistent with the previous dimension levels.

The performed ADD microbenchmark of Samsung shows an average speedup of around 1.6 × for the real system and 2.6 × for the virtual prototype. Compared to this paper, where the speedup is approximately 12.7 ×, this result is almost an order of magnitude lower. Samsung explains the low speedup by the fact the processor has to introduce memory barrier instructions between every 8 ADD instructions, resulting in a severe performance degradation. However, this memory barrier was also implemented in our VADD kernel. One possible explanation for the deviation could be architectural differences between the simulated ARM-based system and Samsung's GPU-based system. The simulated platform can speculatively execute instructions, which may result in better utilization of memory bandwidth. In addition, the vector benchmarks require more memory barriers relative to the number of arithmetic instructions, as their microkernels do not contain any loops. So the effects of architectural differences caused by these memory barriers would affect the vector benchmarks more than the matrix benchmarks.

The GEMV microbenchmark on the other hand shows a more matching result with an average speedup value of 8.3 × for Samsung's real system and 2.6 × for their virtual prototype, while this

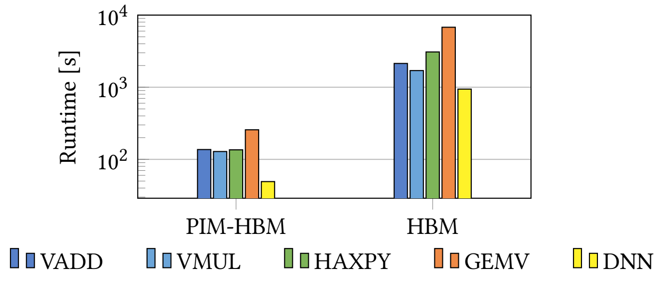


Figure 4: Runtimes of the simulation workloads on the host system.

paper achieved an average speedup of 9.0 ×, which is well within the reach of the real hardware implementation.

Figure 4 shows the simulation runtimes of the various workloads on the host system. With PIM enabled, the runtime drops by about an order of magnitude for some workloads, indicating the reduced simulation effort on gem5's complex processor model, as only new memory requests are issued by the model during operation of PIM. Therefore, exploring the effectiveness of different PIM-enabled workloads may be less time-consuming than traditional workloads due to the reduced simulation complexity.

7 CONCLUSION

In this paper, we presented a virtual prototype of Samsung's PIM-HBM architecture for simulation and evaluation of real-world applications. Leveraging the open-source tools gem5 and DRAMSys, the PIM-HBM implementation integrates seamlessly into sophisticated simulation frameworks that enable the realistic exploration of a wide-range of workloads using full-system simulation. In addition to the hardware perspective, the analysis includes considerations from a software point of view and identifies the necessary modifications to the data layout in applications in order to efficiently make use of PIM-HBM. Using this simulation framework, we conducted an analysis of the potential feasibility and effectiveness of PIM across a range of microbenchmarks. The simulations demonstrated a reduction in execution time by 9.2 × for matrix-vector operations, and for simplified neural network tasks, a reduction by up to a factor of 6.0 ×. These findings are largely consistent with the results reported by Samsung, with the exception of a deviation observed in the vector microbenchmarks. Furthermore, an examination of the wallclock time for simulations comparing non-PIM and PIM approaches showed that the decreased complexity of simulations can lead to a reduction by up to an order of magnitude. In this work, the first system-level virtual prototype of Samsung's PIM-HBM is presented, enabling the rapid exploration and feasibility analysis of various workloads in a realistic and detailed manner. Looking ahead, future work should focus on providing estimations on the energy efficiency of the PIM architecture and on expanding the software framework to a Linux implementation, enabling further research on real-world AI applications.

FUNDING

This work was partly funded by the German Federal Ministry of Education and Research (BMBF) under grant 16ME0934K (DI-DERAMSys).

REFERENCES

[1] Pouya Esmaili-Dokht et al. 2024. $O(n)$ Key–Value Sort With Active Compute Memory. *IEEE Trans. Comput.* 73, 5 (May 2024), 1341–1356. https://doi.org/10.1109/TC.2024.3371773
[2] Tim Finkbeiner et al. 2017. In-Memory Intelligence. *IEEE Micro* 37, 4 (2017), 30–38. https://doi.org/10.1109/MM.2017.3211117
[3] Bruno E. Forlin et al. 2022. Sim 2 PIM: A Complete Simulation Framework for Processing-in-Memory. *Journal of Systems Architecture* 128 (July 2022), 102528. https://doi.org/10.1016/j.sysarc.2022.102528
[4] Juan Gómez-Luna et al. 2021. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. *CoRR* abs/2105.03814 (2021). arXiv:2105.03814 https://arxiv.org/abs/2105.03814
[5] Mingxuan He et al. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Athens, Greece, 372–385. https://doi.org/10.1109/MICRO50266.2020.00040
[6] Bongjoon Hyun et al. 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. arXiv:2308.00846 [cs]
[7] Taeyang Jeong et al. 2024. PipePIM: Maximizing Computing Unit Utilization in ML-Oriented Digital PIM by Pipelining and Dual Buffering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. https://doi.org/10.1109/TCAD.2024.3410842
[8] Norman P. Jouppi et al. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA52012.2021.00010
[9] Yoongu Kim et al. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456
[10] Young-Cheon Kwon et al. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. IEEE, San Francisco, CA, USA, 350–352. https://doi.org/10.1109/ISSCC42613.2021.9365862
[11] Sukhan Lee et al. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 43–56. https://doi.org/10.1109/ISCA52012.2021.00013
[12] Shang Li et al. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109. https://doi.org/10.1109/LCA.2020.2973991
[13] Jason Lowe-Power et al. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR]
[14] Sergiu Mosanu et al. 2022. PiMulator: A Fast and Flexible Processing-in-Memory Emulation Platform. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Antwerp, Belgium, 1473–1478. https://doi.org/10.23919/DATE54114.2022.9774614
[15] Netlib. 1979. BLAS (Basic Linear Algebra Subprograms). https://www.netlib.org/blas/.
[16] P Rosenfeld et al. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan. 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4
[17] Rust Foundation. 2015. The Rust Programming Language. https://www.rust-lang.org/.
[18] Daniel Sanchez et al. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *ACM SIGARCH Computer Architecture News* 41, 3 (June 2013), 475–486. https://doi.org/10.1145/2508148.2485963
[19] Paulo C. Santos et al. 2021. Sim2PIM: A Fast Method for Simulating Host Independent & PIM Agnostic Designs. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Grenoble, France, 226–231. https://doi.org/10.23919/DATE51398.2021.9474104
[20] Vivek Seshadri et al. 2013. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Davis California, 185–197. https://doi.org/10.1145/2540708.2540725
[21] Vivek Seshadri et al. 2020. In-DRAM Bulk Bitwise Execution Engine. arXiv:1905.09822 [cs]
[22] Shin-haeng Kang et al. 2023. PIMSimulator. https://github.com/SAITPublic/PIMSimulator.

[23] Gagandeep Singh et al. 2019. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, Las Vegas NV USA, 1–6. https://doi.org/10.1145/3316781.3317867

[24] Lukas Steiner et al. 2020. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*. Springer.

[25] Lukas Steiner et al. 2022. DRAMSys4.0: An Open-Source Simulation Framework for In-depth DRAM Analyses. *International Journal of Parallel Programming* 50, 2 (April 2022), 217–242. https://doi.org/10.1007/s10766-022-00727-4

[26] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* C-19, 1 (1970), 73–78. https://doi.org/10.1109/TC.1970.5008902

[27] Chirag Sudarshan et al. 2022. A Critical Assessment of DRAM-PIM Architectures - Trends , Challenges and Solutions. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Marc Reichenbach, and Matthias Jung (Eds.). Vol. 13511. Springer International Publishing, Cham, 362–379. https://doi.org/10.1007/978-3-031-15074-6_23

[28] Ke Wang et al. 2016. An Overview of Micron's Automata Processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, Pittsburgh Pennsylvania, 1–3. https://doi.org/10.1145/2968456.2976763

[29] Xinfeng Xie et al. 2022. MPU-Sim: A Simulator for In-DRAM Near-Bank Processing Architectures. *IEEE Computer Architecture Letters* 21, 1 (Jan. 2022), 1–4. https://doi.org/10.1109/LCA.2021.3135557

[30] Sheng Xu et al. 2019. PIMSim: A Flexible and Detailed Processing-in-Memory Simulator. *IEEE Computer Architecture Letters* 18, 1 (Jan. 2019), 6–9. https://doi.org/10.1109/LCA.2018.2885752

[31] Chao Yu et al. 2021. MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator. *IEEE Computer Architecture Letters* 20, 1 (Jan. 2021), 54–57. https://doi.org/10.1109/LCA.2021.3061905

[32] Minxuan Zhou et al. 2021. DP-Sim: A Full-stack Simulation Infrastructure for Digital Processing In-Memory Architectures. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. ACM, Tokyo Japan, 639–644. https://doi.org/10.1145/3394885.3431525