Received May 29, 2019, accepted June 16, 2019, date of publication June 21, 2019, date of current version July 9, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2924240

Design of Processing-"Inside"-Memory Optimized for DRAM Behaviors

WON JUN LEE[1], CHANG HYUN KIM[1], YOONAH PAIK[1], JONGSUN PARK[1], (Member, IEEE), IL PARK[2], AND SEON WOOK KIM[1], (Senior Member, IEEE)

[1]Department of Electrical and Computer Engineering, Korea University, Seoul 02841, South Korea
[2]SK hynix, Icheon 17336, South Korea

Corresponding author: Seon Wook Kim (seon@korea.ac.kr)

This work was supported in part by the SK hynix, Inc., and in part by the IT R&D Program of Ministry of Trade, Industry and Energy (MOTIE)/Korea Evaluation Institute of Industrial Technology (KEIT) [Development of Processing in Memory Architecture and Parallel Processing for Data Bounding Applications] under Grant 10052653.

ABSTRACT The computing domain of today's computer systems is moving very fast from arithmetic to data processing as data volumes grow exponentially. As a result, processing-in-memory (PIM) studies have been actively conducted to support the data processing in or near memory devices to address the limited bandwidth and high power consumption due to data movement between CPU/GPU and memory. However, most PIM studies so far have been conducted in a way that the processing units are designed *only* as an accelerator on the base die of 3D-stacked DRAM, not involved inside memory while not servicing the standard DRAM requests during the PIM execution. Therefore, in this paper, we show how to design and operate the PIM computing units inside DRAM by effectively coordinating with standard DRAM operations while achieving the full computing performance and minimizing the implementation cost. To make our goals, we extend a standard DRAM state diagram to depict the PIM behaviors in the same way as standard DRAM commands are scheduled and operated on the DRAM devices and exploit several levels of parallelism to overlap memory and computing operations. Also, we present how the entire architecture layers from applications to operating systems, memory controllers, and PIM devices should work together for the effective execution by applying our approaches to our experiment platform. In our HBM2-based experimental platform to include 16-cycle MAC (Multiply-and-Add) units and 8-cycle reducers for a matrix-vector multiplication, we achieved 406% and 35.2% faster performance by the all-bank and the per-bank schedulings, respectively, at $(1024 \times 1024) \times (1024 \times 1)$ 8-bit integer matrix-vector multiplication than the execution of only its operand burst reads assuming the external full DRAM bandwidth. It should be noted that the performance of the PIM on a base die of a 3D-stacked memory cannot be better than that provided by the full bandwidth in any case.

INDEX TERMS Processing-in-memory, DRAM, parallelism, matrix-vector multiplication.

I. INTRODUCTION

The structure of the von Neumann has been followed by most computers today since it was first proposed [1]. However, as a computing unit such as CPU/GPU has more data to handle, it is problematic that the cost of transferring data between memory and the computing unit becomes relatively higher than the cost required only for the data operations. Therefore, for a long time, the computing community has improved the performance by efficiently utilizing caches [2]–[5], embedding larger caches [6], placing accelerators close to CPUs [7]–[9], and so on.

To further observe the performance problem due to the data movement, we measured the performance of untiled and tiled matrix-vector multiplications [10] on 16 multicores of the Dell PowerEdge R720 server [11]. Each core has private 64KB L1 and 256KB L2 caches, and eight cores in one socket share a 20MB LLC. Also, the server has 128GB DRAM, and its maximum bandwidth is 59.7GB/s. For the execution, we disabled hyperthreading [12] and assigned half of the used cores into each socket. The OpenCL's GEMM library [13] was used, and the multiplications were experimented by

The associate editor coordinating the review of this manuscript and approving it for publication was Yue Zhang.

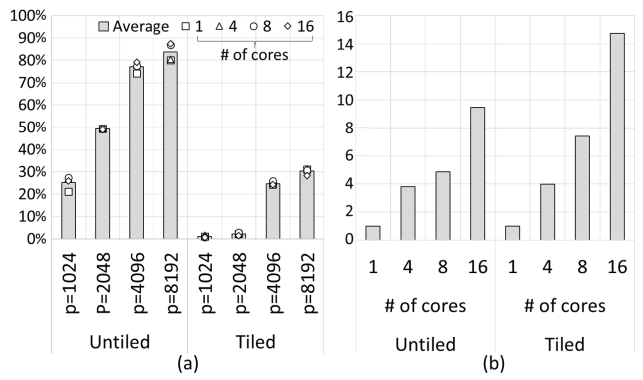


FIGURE 1. (a) Memory bound ratio. (b) Speedup at $p = 8192$. We measured the performance of an 8-bit integer $(p \times p) \times (p \times 1)$ matrix-vector multiplication without and with tiling by increasing p to 1024, 2048, 4096 and 8192.

increasing p to 1024, 2048, 4096 and 8192 at 8-bit integer $(p \times p) \times (p \times 1)$. We initialized all the data and invalidated all the cache lines before starting the multiplication; thus, all the data was available in DRAM at the start. All the performance was measured by using VTune performance analyzer [14].

Figure 1(a) shows that the memory bound ratio increases as the matrix size increases, regardless of the tiled and untiled codes. Also, even if the number of cores is changed, it is shown that the memory bound ratio hardly changes because the matrix-vector multiplications used in this experiment have very regular execution characteristics. In experimental results, the speedup of the tiled code execution was higher than that of the untiled code. However, as shown in Figure 1(b), although the number of cores is increased and the execution of the matrix-vector multiplication exploits complete thread-parallelism, the ideal speedup cannot be achieved because the memory bound does not decrease as shown in Figure 1(a). In conclusion, the tiling technique can help reduce memory performance bottlenecks in applications that use large work memory but cannot eliminate them.

Many technologies have been proposed to alleviate these problems by placing computing and memory units closer, and the most representative of which is called Processing-in-Memory (PIM) technology [15]–[24]. Until the early 2000s, PIM studies such as IRAM [15] had not received much attention from industry, and one of the biggest reasons was that scientific applications, which were formerly major computing domains, were able to utilize cache locality fully. However, in recent years, the computing domain rapidly changes from the computation to data processing such as deep learning [25], [26], in-memory DB [27]–[29], graph computing [30]–[33] and so on. The amount of data to be processed and stored has increased explosively every day, and most of them are very rarely reused. As a result, the bandwidth and power consumption of the data transfer between computing units and memory significantly determine the performance of the entire system [34]–[37], so it has become crucial to minimize the data transfer without compromising computing performance. Besides, along with the development of DRAM process technology, academia and industry are actively conducting PIM research to overcome these problems.

We predicted the performance of PIM in the experimental environment used in Figure 1 for our research motivation. The execution time of PIM could be calculated by multiplying the average memory access latency of the processor (58ns) [38] and the number of memory requests required for data read/write and MAC operations in the matrix-vector multiplication. The PIM execution time was calculated to be very small compared to the CPU execution time in all the cases. PIM can achieve the speedup of 141.1 over the 16-core execution at $p = 8192$. PIM utilizes the internal bandwidth of the DRAM, whereas the CPU performance is bound to the external memory bandwidth. This makes the performance difference.

Most previous PIM studies have focused on designing the application-specific accelerators in the logic die of the 3D-stacked memory, such as HMC [39] or HBM/HBM2 [40], [41], and few studies have implemented the accelerator within DRAM [42]. However, all of them assumed that they did not service the standard DRAM requests during their computation even though handling a standard memory request during the PIM operation is essential for PIM to act as both a memory and an accelerator. Besides, they did not present how the entire architecture layers from applications to operating systems, memory controllers, and PIM devices should work together for achieving significant performance with minimal implementation cost. For example, they did not consider the PIM programming, data mapping from OS page to DRAM banks, a memory controller to schedule both the standard memory and PIM requests, and so on, together for the efficient system development.

We used the following approaches for resolving several critical design issues of PIM development: The address mapping from physical addresses into device addresses in the DRAM memory controller makes the data that consists of one OS page stored and distributed across all the banks in the DRAM. The data layout will not only complicate the PIM design itself but also make the PIM programming very difficult. To facilitate the issues, we develop the PIM framework to match PIM programming and execution concepts with those of parallel programs in multi-core environments. By mapping one bank in the memory to a single core, a memory containing multiple banks functions like a multi-core system. To support this, we make a decision that we design one computing unit per bank. We use the existing bus used to move data between banks and DQ for data exchange or synchronization between banks, and the operation is performed upon memory read/write requests like an explicit message passing cache-coherence. Also, for convenient programming, we use an operator overloading function for the PIM operations, and its interface is provided through the PIM library. As a result, many existing parallel programs can be easily converted into PIM programs.

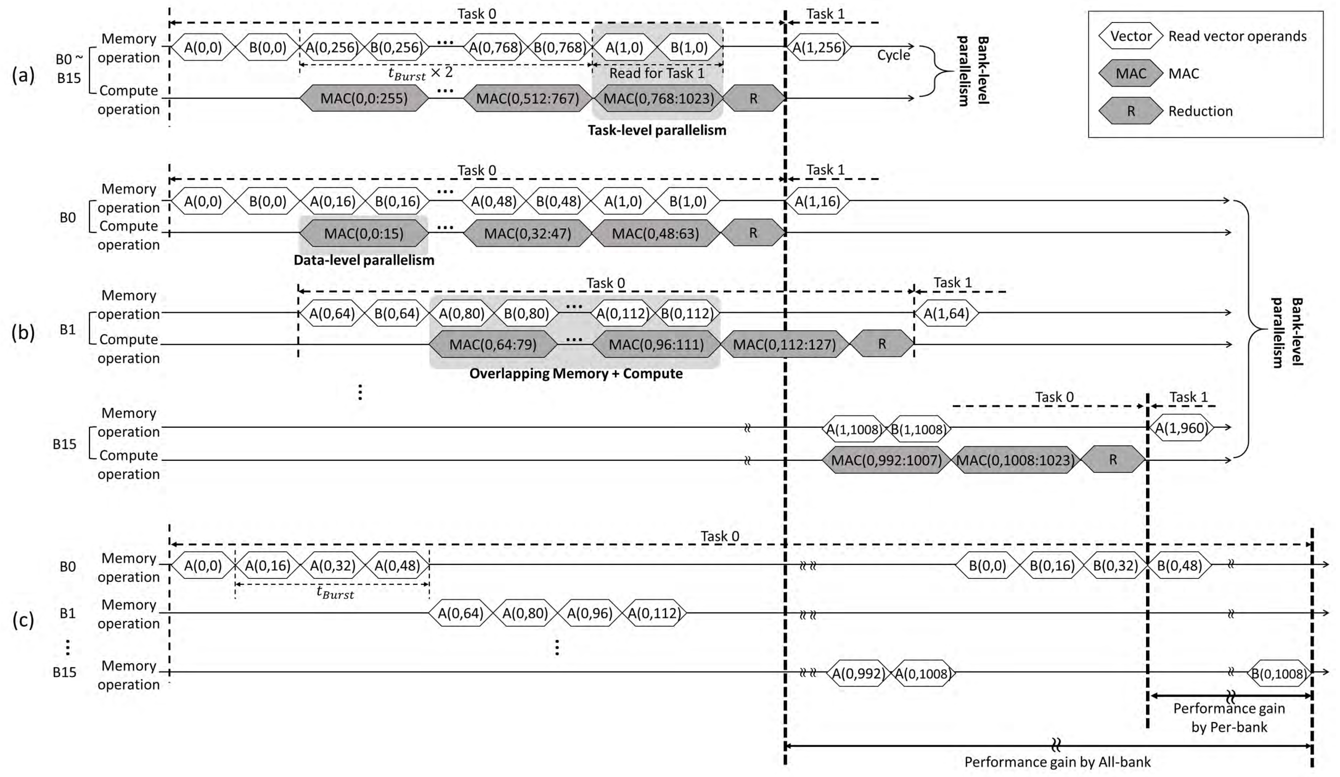


FIGURE 2. **Exploiting various levels of parallelism by different schedulings at (p × 1024) × (1024 × 1) multiplication. (a) All-bank scheduling. (b) Per-bank scheduling. (C) Burst operand reads by standard memory requests from assuming the accelerator usage outside DRAM. We assume 2-cycle MAC and 1-cycle reducer.**

Also, for their seamless co-operation with the standard memory requests and the PIM execution, we add PIM commands to the standard DRAM state diagram for depicting the PIM behaviors in the same way as standard DRAM commands are scheduled and operated on the DRAM devices. The approach allows the PIM execution to be easily adapted and optimized with the standard DRAM behaviors and also makes it possible to service the standard memory requests during the PIM requests.

For maximizing the performance, our PIM architecture exploits several levels of parallelisms by software and hardware as shown in Figure 2 by assuming 2-cycle MAC and 1-cycle reducer: 1) multi-way vector operations by a computing unit per bank (data-level parallelism), 2) independent bank-level execution to use full internal bandwidth in read and write operations (bank-level parallelism), 3) overlapping memory behaviors with computing ones (overlapping memory and compute operations), and 4) exploiting independent PIM operations informed by software (task-level parallelism). To maximize the parallelism exploitation, our memory controller supports all-bank, per-bank, and bank-group command schedulings; thus, the controller would make the PIM computation hide the DRAM behaviors. By using the methods, Figure 2 shows how much performance benefit we can achieve compared to the execution of only its operand burst reads assuming the full DRAM bandwidth. It should be noted that all the read operands to the outside the DRAM are serialized due to the limited memory bandwidth.

To the best of our knowledge, our work is the first study to develop PIM optimized for the DRAM behaviors and describe the entire PIM architecture layers for resolving the critical issues at system-level design. For the verification of our system-level design, we modeled the PIM memory controller and the HBM2-based PIM to include 16-cycle MAC (Multiply-and-Add) units and 8-cycle reducers for a matrix-vector multiplication. Also, we developed the PIM software library for supporting the PIM parallel programming and the OS driver to interact with a host and the PIM device. On our experimental platform, we achieved 406% and 35.2% faster performance by the all-bank and the per-bank schedulings, respectively, at (1024 × 1024) × (1024 × 1) 8-bit integer matrix-vector multiplication than the execution of only its operand burst reads assuming the external full DRAM bandwidth with one channel, i.e., 32GB/s [41]. The performance of the previous studies implementing PIM on a base die of a 3D-stacked memory [19]–[21] cannot be better than that provided by the external full memory bandwidth in any case. The reason is that the previous studies cannot hide the overhead associated with intrinsic memory behaviors such as row activate and precharge, and should wait for completing the operations. Therefore, we did a performance analysis on the baseline of the full bandwidth performance of the memory rather than directly comparing with the previous researches.

This paper consists of the followings: Section II describes PIM design issues and their solution approaches, Section III shows our extension of the DRAM state diagram, Section IV presents our experimental design in detail, Section V shows the performance evaluation, and Section VI discusses the related work. Then, we present the discussion in Section VII and conclude in Section VIII.

II. DESIGN ISSUES AND OUR APPROACHES

In this section, we examine major design issues when supporting computations inside the DRAM.

A. DATA LAYOUT AND EXECUTION MODEL

One of the most challenging issues when implementing PIM, in general, is to minimize the overhead associated with PIM operations while still keeping the existing address mapping and management methods. In other words, memory management for PIM use should not be different from that used in the existing OS.

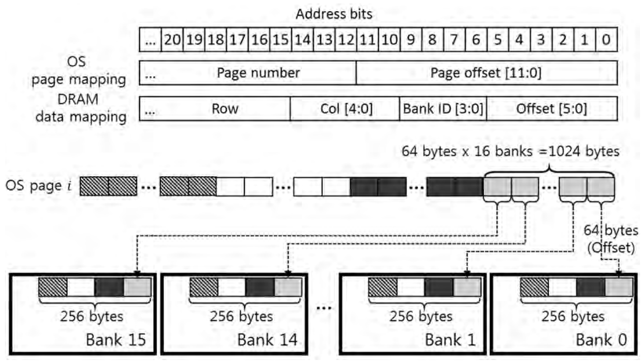


FIGURE 3. Mapping between the physical address of OS page and DRAM internals on our research platform: 4KB OS page and DRAM with 16 banks. One OS page is interleaved with 64 bytes in all 16 banks.

Figure 3 illustrates the relationship between the physical address of the OS page (4KB) and the DRAM device address on our research platform. As shown in the figure, continuous data within one OS page is distributed evenly among 16 banks, with only four interleaved 64 bytes in one bank consisting of one page. *In our PIM design, we preserve this address mapping and convert the maximum internal bandwidth for read/write operations of the DRAM into a computing bandwidth by assigning one computing unit to one bank*; thus, the assignment allows all the banks to perform their computations independently and synchronize between themselves using explicit PIM read/write operations. This model concept provides a great intuition to the design of PIM hardware and software.

The data movement between the banks must be explicitly expressed because our PIM architecture does not support hardware-based data coherence between banks. The overhead of the data movement is the same as the memory copy overhead, i.e., the sum of the latency of data read and write, and their movement between banks. Therefore, the programming and execution concepts of the PIM are the same as the parallel ones to be applied on software-coherent multicores [43].

Also, the occurrence of the DRAM row miss prevents the PIM from computing without idle or stalls because data cannot be retrieved to the computing unit during the next row activation. For example, suppose that the DRAM row miss penalty is 37-cycle (sum of precharge and activation penalties in HBM2 [41]). Whenever the DRAM row miss occurs and the next row becomes active, all the computations are entirely stopped. Therefore, the row miss would result in the significant degradation in performance, so it is essential to ensure that all data is located in one DRAM row as much as possible. In any case, however, when the data size increases, the row miss is unavoidable. In our design, we minimize the associated overhead by overlapping the computation and memory operations, such as activation and precharge.

B. COOPERATING WITH STANDARD MEMORY REQUESTS

If a memory controller receives the standard DRAM memory requests while a PIM operation is in progress, the controller should service the requests as soon as possible to satisfy their performance requirement. However, the previous PIM research did not provide the solutions, and instead, the standard memory requests are assumed to be not received when the PIM operation is in progress [19], [21]–[23], [42]. However, their cooperation should be studied.

For this purpose, *we extend the DRAM state diagram where PIM commands are also expressed with the standard DRAM commands.* It implies that the standard DRAM commands can be processed whenever possible during PIM operations. As a result, we not only generate the PIM requests in the same way as the standard memory requests but also schedule the PIM commands in the same way as the standard DRAM commands. This approach also can minimize implementation and operational overhead for managing the PIM requests and commands.

The PIM instruction generated within the PIM library (discussed in Section IV-E.1) is loaded into a standard write request with 1-bit extension to distinguish itself from a standard memory instruction and passed to a memory controller (MC). The MC decodes the PIM instruction into the PIM command and sends the command to DRAM using the extension of 1 bit to distinguish it from the standard memory commands.

C. PARALLELISM EXPLOITATION

When implementing computing units within the DRAM, the space for the implementation within the DRAM is very limited and thus important to adequately represent the various levels of parallelism in a tight space.

In our research environment, DRAM consists of 16 banks, and one read memory request reads 128-bit data from the bank at one time, and a total of 512-bit in sequence by its burst operation. Thus, since we design an 8-bit integer matrix-vector multiplication, *a computing unit per bank* *supports 16 8-bit integer vector operations at one time, thus exploiting data-level parallelism.* Also, since the data is interleaved across all the banks as shown in Figure 3 and the banks operate independently, we can achieve a total of 16 × 16 data-level parallelisms at one time. In our processing-inside-memory approach, the bank-by-bank parallelism is one of the most fundamental causes of our PIM's superior performance over accelerators to use external interfaces with the limited bandwidth. Also, we can get higher parallelism through software. For example, for a matrix-vector multiplication, since each resulting matrix element is independent, *task-level parallelism can be obtained by making the software express independent/dependent PIM operations.* Similarly, if we calculate using more than 256 operations to obtain one matrix element result, we can get more parallelism in the same way because our model can continue the MAC computations without doing the reduce after every 256 operations. The dependence information about the operations embedded inside the PIM instruction is passed to the memory controller to perform the scheduling. We also *exploit parallelism even in the pipeline of the computing unit by overlapping the DRAM-related operations and the PIM computations on MACs and reducers.* For example, as soon as the DRAM operation completes and their calculations start, the subsequent DRAM operation begins to hide its related overhead.

III. DRAM STATE DIAGRAM EXTENSION

A memory controller converts the standard read/write memory requests generated by a processor into the DRAM commands, which define the DRAM behaviors such as read, write, activation, precharge, and refresh commands. The behaviors are defined as a DRAM state diagram [44], and the state diagram exists per bank in general. To depict PIM behavior in the same way as the standard DRAM commands are scheduled and operated on DRAM devices, we slightly modified the standard DRAM state diagram, and it is shown in Figure 4.

There were two considerations for the extension. PIM always starts to compute by issuing the memory read commands, so we can consider the PIM compute commands as standard read commands with a longer latency. The other consideration is that the standard memory commands use DQ to send and receive data. Instead, the PIM commands send data to the computing unit and receive its results from the unit. The only difference is the source and destination in the commands. Therefore, we can use the same PIM RD and WR command edges as the standard ones without having to define new states.

There are two crucial advantages to expressing the PIM and the standard memory commands in the single state diagram. First, the standard memory commands need to be neither blocked nor handled differently during the PIM execution; thus, at any time during the PIM computation, we can service high priority standard memory requests and naturally satisfy their performance requirement, which was not presented in the previous PIM studies [19], [21]–[23]. Second, the single

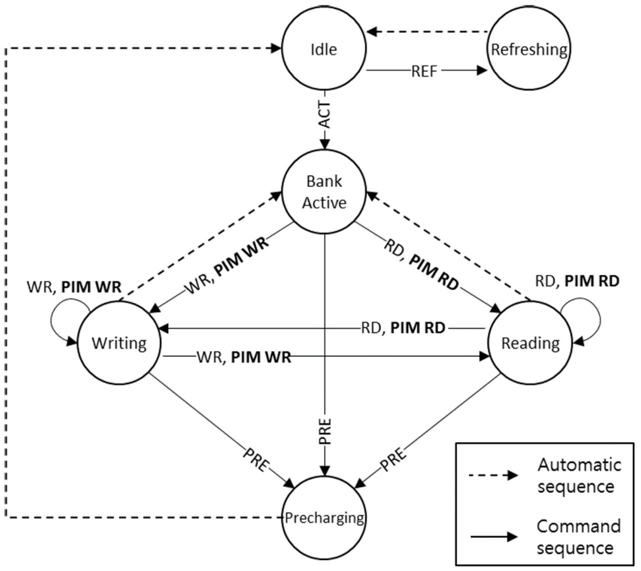


FIGURE 4. **The state diagram for both the standard DRAM and the PIM commands. The bold italic commands are extended for the PIM commands.**

state diagram allows us to design only one queue for both kinds of the commands; thus, we significantly simplify the memory controller design for supporting the PIM operations.

IV. EXPERIMENTAL DESIGN: PIM FOR MATRIX-VECTOR MULTIPLICATION

In this section, we present the detailed implementation of our PIM by using the matrix-vector multiplication which is a core operation for many types of emerging applications such as neural network executions, AR/VR, and so on.

A. OVERALL ARCHITECTURE

Figure 5 shows our PIM overall architecture, which is primarily divided into three components: 1) a software stack to consist of the PIM application, the PIM library, and the PIM device driver, 2) a memory controller, and 3) the PIM device.

The role of the software stack is to map an application's PIM data onto the PIM device, to create PIM instructions using the PIM library and to enable the PIM device driver to offload them into the memory controller. We modified the `mmap` function to perform the PIM data mapping. In order to deliver PIM instructions from a host processor to the PIM device, we can store the PIM instructions in the standard write requests and pass them to the memory controller by adding one-bit representing the PIM request as previously described in Section II-B. However, due to the difficulty of such modification, we used the PIM device driver in our experiment PCIe memory platform to transfer the PIM instructions to the PIM device through the system call interface. If we perform the modification and attach the PIM device directly to the memory bus, the `mmap` and the PIM device driver are removed.

We modeled the memory controller and the PIM device using Verilog HDL in Kintex-7 FPGA attached to PCIe. The memory controller generates the PIM commands from the

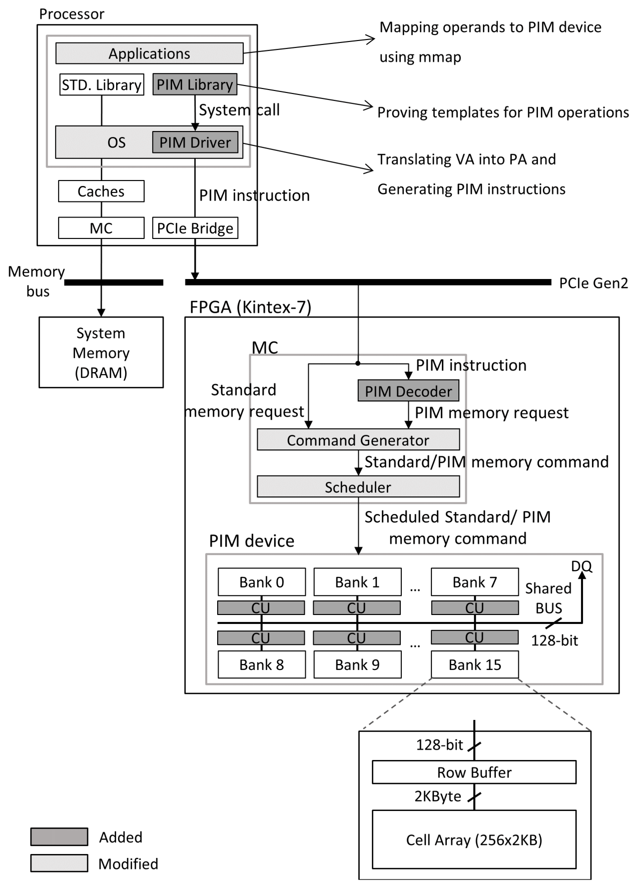


FIGURE 5. **An overall architecture of PIM.**

received PIM instructions, and schedules and sends them to the PIM device with the standard memory commands. Our PIM device consists of 16 banks along with the model of HBM2, and each bank is composed of 256 rows, with each row being 2KB. From each row, one read command reads 128-bit data at one time. Also, all the banks share a 128-bit bus, which is connected with DQ. We designed a computing unit in each bank. The address mapping from MC to the PIM device follows Figure 3.

B. MEMORY DEVICE FOR PIM

HBM2 supports high bandwidth [41], which is approximately 10 times larger than most commonly used DDR devices (DDR4: 25.6GB/s, HBM2: 256GB/s) due to the high number of off-chip pins. The amount of data that can be computed per unit time is proportional to the amount of data that can be read in memory. Also, the banks in one die of HBM2 send data to the DQ pin through a shared bus. In other words, the PIM data coherence between banks can be resolved through the shared bus with the PIM commands without additional overhead in hardware.

We have adopted *only* one die of HBM2 memory as the underlying memory platform for our PIM research, not 3D stacked HBM/HBM2 as our design goal, primarily due to the following considerations: There are many channels of HBM2 to provide high bandwidth and places channel bits directly above the offset in the address mapping. The placement would eventually scatter one OS page through all the channels, i.e., storing one-page data onto all banks on all the dies, which could cause a significant communication overhead. If computation data is assumed to be placed on one channel by changing address mapping, i.e., moving the channel bits to higher bits, the communication overhead for the communication may not occur. However, the channel-level parallelism for the standard memory requests would disappear, and the performance would be degraded. Therefore, many channels are somewhat toxic to the performance of the PIM architecture. We took the issue into account and based on the memory of one die that uses only one channel in the HBM2. One channel configuration also eliminates TSVs for connecting the dies and reduces the burden on developers by implementing computing logic in the corresponding TSV area. Thus, we can store all data in one chip, neither across channels. This consideration can also be taken to a multi-chip (DDR series). However, the synchronization should always be minimized to achieve high performance like parallel execution in multicores.

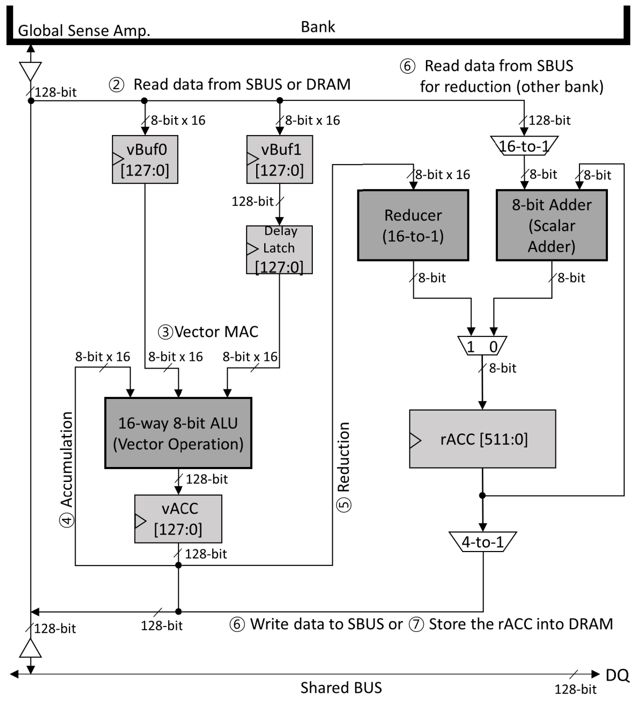


FIGURE 6. **PIM datapath for our matrix-vector multiplication.**

C. PIM DATAPATH AND COMMANDS

Figure 6 shows the PIM datapath design for our 8-bit integer matrix-vector multiplication. In our design, we assign one PIM computing unit to each bank; thus, we have a total of 16 units connected with the already available 128-bit shared bus. Each unit consists of the followings: 1) 128-bit vBUF0 and vBUF1 registers, each of which holds

TABLE 1. PIM commands and their operations.

Type	Opcode [6:4]	Src [3:2]	Dst [1:0]	Operations
Read data	RD (3'b010)	DRAM (2'b00)	vBUF0 (2'b10)	RD vBUF0 $\leftarrow$ DRAM
		DRAM (2'b00)	vBUF1 (2'b11)	RD vBUF1 $\leftarrow$ DRAM
		SBUS (2'b01)	vBUF0 (2'b10)	RD vBUF0 $\leftarrow$ SBUS
		SBUS (2'b01)	vBUF1 (2'b11)	RD vBUF1 $\leftarrow$ SBUS
Write data	WR (3'b001)	vACC (2'b10)	DRAM (2'b00)	WR vACC $\rightarrow$ DRAM
		rACC (2'b11)	DRAM (2'b00)	WR rACC $\rightarrow$ DRAM
		vACC (2'b10)	SBUS (2'b01)	WR vACC $\rightarrow$ SBUS
		rACC (2'b11)	SBUS (2'b01)	WR rACC $\rightarrow$ SBUS
Clear registers	CLR (3b'000)	vBUF0 | vBUF1 | vACC | rACC		Clear the bit-positioned register
Vector operation	MAC (3'b100)	2'bXX	2'bXX	vACC[i] $\leftarrow$ vACC[i] + vBUF0[i] $\times$ vBUF1[i]
Reduce	R_ADD (3'b110)	2'b0X	2'bXX	rACC[i] $\leftarrow$ reduce(+) from vACC
		2'b1X	2'bXX	rACC[i] $\leftarrow$ rACC[i] + SBUS[j]

128-bit data (16 8-bit operands) from DRAM or the shared bus from other banks. 2) One delay latch to hold the first read 128-bit operand since only one memory request is serviced at a time. 3) 16-way vector MACs and each way uses two 8-bit inputs and produces an 8-bit result. 4) 128-bit vector accumulator register (vACC). 5) a 16-to-1 reducer to perform reduction(+) from data in vACC. 6) one 8-bit scalar adder to sum rACC and the reducer result transferred through the shared bus from other banks. Also, 7) a 512-bit rACC register to store the reduced results and to be used for burst writes to DRAM.

The basic PIM operations are performed as follows.

① Initialize the vector accumulator register, vACC.
② Read two 128-bit vector operands one by one from DRAM or the shared bus and storing them in the vBUF0 and vBUF1 registers.
③ Perform vector MAC operations with vACC and store the results in vACC.
④ Repeat Steps 2) and 3) for exploiting higher vector parallelism.
⑤ At the end of the vector operations, perform a reduce operation using the vector values stored in the vACC register, and store the reduced result in the rACC indexed by an offset of the store address.
⑥ If the reduction operation is required with other banks, send the reduce result to other banks or receive the value of rACC from other banks through the shared bus and reduce the values.
⑦ Store the rACC into DRAM by burst writes.

Table 1 illustrates the PIM commands that drive the PIM device. We currently support commands only for the 8-bit integer matrix-vector multiplication. In order to perform various application programs in the PIM, it is sufficient to add more ALU units in the datapath and their associated PIM instructions and commands. We support read and write commands to use DRAM and the shared bus with the registers. Also, we support a clear command to initialize the registers. For the calculation, we develop vector MAC and reduce commands.

One of the most critical issues in implementing PIM is the area overhead of MAC ALUs. McDRAM [42] analyzed the overhead when designing MAC ALUs for BLSAs, column decoders, and I/O drivers in DRAM. Also, [42] showed that the design of 256 8-bit MACs at the column decoder positions incurs the area overhead of about 4.7% of the total DRAM area. Because McDRAM and we used the same MAC configuration, both methods would suffer from similar area overhead. However, the significant difference between McDRAM and our study is that the McDRAM paper envisioned a PIM architecture from the perspective of an accelerator, whereas our work has envisioned a PIM architecture from the perspective of the entire system.

D. MEMORY CONTROLLER

Before designing the PIM memory controller, we modeled and verified the memory controller that generates the standard DRAM commands with Verilog HDL. In the design, we referenced two well-known DRAM simulators, DRAMSim2 [45] and Ramulator [46]. The modeled standard memory controller consists of three parts: 1) a request queue to receive the memory requests, and store and schedule them, 2) a command generator logic to convert the requests into commands, and 3) a command queue per bank to issue the commands.

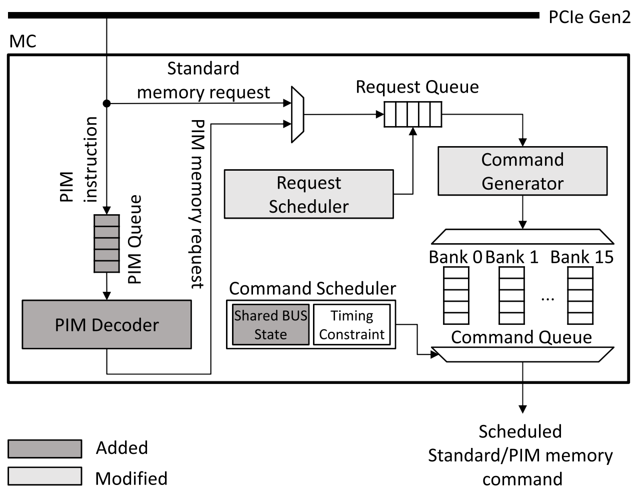


FIGURE 7. A memory controller for PIM.

Figure 7 shows our PIM memory controller design, and the gray-colored ones are either added or modified. The added 8-entry PIM queue and logic for decoding 12 PIM instructions receive the PIM instructions via the PIM device driver, convert them into the PIM memory requests, and store the converted PIM requests in the request queue. A slight addition and modification to the memory controller were required, so the area overhead would be minimal. The greatest strength of our PIM architecture is scheduling the PIM memory requests and commands with the standard DRAM ones together. Therefore, we use only one request queue and only one command queue per bank for both operations with the same queue management scheme. We slightly modified the memory request scheduler to recognize the PIM memory requests and the command generator to translate the PIM memory requests into the PIM memory commands.

In the PIM architecture, however, we need to consider the order of issued commands between the PIM commands and between the PIM commands and the standard DRAM commands. In the schedule between the PIM commands, the memory controller should not change the order of commands if there is a dependence between them. The PIM library generates dependency information inside the PIM instruction. In the schedule between the PIM commands and the standard DRAM commands, we need to take only their priority into account since there is no dependence between them. Our command scheduler uses a round-robin as a priority scheme between banks. Also, we support three kinds of commands for exploiting bank-level parallelism inside the DRAM device: all-bank, per-bank, and bank-group commands. Since there is no limitation in internal bandwidth except for the reduction, we can maximize the bank-level parallelism for the computation inside the memory.

E. PIM SOFTWARE STACK

The software stack for our PIM architecture consists of three parts: the PIM application, the PIM library, and the PIM device driver. In our current research platform, a host processor and the PIM modeled FPGA are connected using PCIe, and we separated the PIM memory region from the system memory region. The only reason for separating the PIM region and the system memory region is that we could not modify the system memory and the memory controller on the real machine for our PIM study. The two memory regions are flat in virtual and physical memory spaces, and the physical address of DRAM modeled as PIM in FPGA is allocated from 0x200000000 to 0x2007FFFFF. The data movement between the two regions must be explicitly expressed by software.

Also, there is no coherence problem between the PCIe memory and caches by assigning the PCIe memory as uncacheable. If the PIM memory is assumed to be cached, the coherence problem can be solved by flushing cached dirty source operands and invalidating the cached destination operands before starting the PIM execution. However, the straightforward method may have a significant influence on the overall system performance [47]; thus, the deep-dived study is needed. In our architecture, a user must access all data with a virtual address, so he cannot know the corresponding physical address. That is, no memory region can be used

```
1   #include <PIM_library.h>
2   #define DST_TYPE uint8_t
3
4   int main()
5   {
6     // Declare matrix-vector operands
7     MatrixPIM <uint8_t> SRC_A(2, 1024);
8     MatrixPIM <uint8_t> SRC_B(1024,1);
9
10    ......
11
12    // Perform the matrix-vector multiplication and store the result into DST_C
13    MatrixPIM <DST_TYPE> DST_C = SRC_A * DST_B;
14    ......
15
16    return 0;
17  }
```

FIGURE 8. An example of the PIM application for the matrix-vector multiplication of (2 × 1024) × (1024 × 1).

only by PIM other than the OS-managed memory regions, so there is no new security issue that arises from using our architecture.

1) APPLICATION AND LIBRARY

An example of the PIM application is shown in Figure 8, where a programmer uses the PIM matrix declaration and operator overloading (Lines 7∼8 and Line 13), similar to a vector of the C++ standard library for declaring the matrix and vector operands, through the PIM templates available in the PIM library.

Figure 9 shows our PIM instruction format and library code. We modified the `mmap` function to assign the PIM data onto the PCIe memory by defining PCIE_FLAG (Lines 12∼15). The data are assumed to be initialized without loss of generality. We also provide convenience to program developers by supporting operator overloading functions (Lines 30∼57). The matrix-vector multiplication using the vector operator "*" requires a typical two-nested loop structure, and there is a difference only in the innermost loop. Because we support 1024 MAC vector operations at one time (16 vector MACs per bank × 16 banks × 4 burst reads), we define the value as VECTOR_SIZE. We can eliminate the `mmap` function if 1) all memory devices include PIM computing units, or 2) OS manages the PIM data only when some of the devices include the PIM units.

Also, the `initial_ordering` in Lines 50∼51 represents the dependence across one-time computation, i.e., a chunk of 1024 vector operands. One element of the result vector may need more than one chunk computations to be calculated. The value 2'b10 means the first chunk execution, and the value 2'b01 does the last chunk computation. If the sizes of the matrix and the vector are large, then we need to compute the chunk calculation several times (Line 43). The value 2'b00 represents the dependence across the chunk computations. In the case of 11, there is no dependence between any PIM instructions. The memory controller schedules the PIM commands by using the information for providing the program correctness. The generated PIM instruction is passed to the PIM device driver via the system call interface on Line 53.

If multiple cores issue two or more PIM executions at the same time, which could lead to data inconsistency. To solve the data consistency, one group of PIM instructions to be

```
struct __PIM_Instruction {
  uint64_t  scr0_addr; // operands
  uint64_t  src1_addr;
  uint64_t  dst_addr;
  uint8_t   opcode;    // opcode
  // dependence: 00: dependence, 01: end, 10: start, 11: independence
  uint8_t   initial_ordering;
};

#define VECTOR_SIZE 1024 // 16 vector ops x 4 bursts x 16 banks
#define MAC        0x04
#define MAP_PCIE  0x40 // for mapping data onto PCIe memory
#define PCIE_FLAG   MAP_PRIVATE | MAP_ANON | MAP_PCIE
#define PCIE_MMAP(SIZE) \
    (PRECISION *)mmap(0,SIZE,PROT_WR|PROT_RD,PCIE_FLAG,-1,0)

template <typename PRECISION>
class MatrixPIM
{
  PRECISION (*__mmap_pcie) = NULL;
  MatrixPIM(unsigned _row, unsigned _col) :row_size(_row), col_size(_col)
  {
    __alloc_size=_row*_col*sizeof(PRECISION);
    if((__mmap_pcie = PCIE_MMAP(__alloc_size)) == NULL) exit(0);
  }
  void operator =(MatrixPIM<dstT> &_ret_obj)
  {
    this->__mmap_pcie = _ret_obj.__mmap_pcie;
  }
  friend MatrixPIM<DST_TYPE> operator *
              (MatrixPIM<PRECISION> &_src0_obj, MatrixPIM<PRECISION> &_src1_obj)
  {
    // create a destination matrix
    MatrixPIM<DST_TYPE> _ret_obj(_src0_obj.row_size, _src1_obj.col_size);

    // initialize PIM instruction
    struct PIM_Instruction __PIM_instr;
    __PIM_instr.opcode = MAC;

    // Generate PIM instructions for matrix-vector multiplication
    int src0_size = _src0_obj.row_size * _src0_obj.col_size;
    for(size_t i=0, dst_pos=0; i<src0_size; i+=_src1_obj.row_size, dst_pos++){
      for(size_t k=0; k<_src1_obj.row_size; k+=VECTOR_SIZE){
        // Virtual addresses of operands
        __PIM_instr.scr0_addr = _src0_obj.__mmap_pcie+i+k;
        __PIM_instr.scr1_addr = _src1_obj.__mmap_pcie+k;
        __PIM_instr.dst_addr = _ret_obj.__mmap_pcie+dst_pos;

        // Initial ordering bit
        __PIM_instr.initial_ordering = (k==0 ? 1:0) << 1 | // first?
          (k==_src1_obj.row_size-VECTOR_SIZE ? 1:0); // last?
        // Send PIM inst to PIM device driver
        SYSCALL(&__PIM_instr);
      }
    }
    return _ret_obj;
  }
  ...
};
```

FIGURE 9. **PIM instruction format and an example of the PIM library for the matrix-vector multiplication.**

processed at a time should be expressed within 64 bytes, which is a general cache block size. If so, it is possible to prevent the commands from being mixed among multiple cores.

2) DRIVER

When delivering PIM instructions from a conventional system to the PIM HW, the PIM instructions must be recognized by a host processor and a memory controller. However, due to the difficulty of such modification, we used the PIM device driver in our experiment PCIe memory platform to transfer the PIM instructions to the PIM device through the system call interface.

Our PIM device driver plays the following two roles: The first is to translate virtual addresses of operands into their physical ones. Since only virtual addresses can be seen in the user space, we make the PIM device driver operating in kernel mode convert the virtual addresses to their physical ones via a page table walk. Second, the PIM device driver sends the PIM instructions to the PIM queue inside our modeled memory controller. We used the writeq (WR_DATA, DST_ADDR) function of the OS to use the general memory write request as it is. Therefore, the PIM instructions can be forwarded to the memory controller by avoiding the modification difficulty. If there is a data dependency between the PIM instructions

```
extern  uint64_t    *PIMQueue;
#define PIMQueue_SRC0   PIMQueue
#define PIMQueue_SRC1   PIMQueue+0x1
#define PIMQueue_DST   PIMQueue+0x2
#define PIMQueue_INSTR PIMQueue+0x3

extern u64 virtual_to_phys(u64 vaddr);

asmlinkage void Send_PIM_Instr
(struct PIM_Instruction *__PIM_instr)
{
  // Copy user space struct to kernel space
  struct PIM_Instruction PIM_instr;
  copy_from_user(&PIM_instr, __PIM_instr)

  // Translate virtual addresses to physical addresses
  PIM_instr.scr0_addr = virt_to_phys(PIM_instr.scr0_addr);
  PIM_instr.scr1_addr = virt_to_phys(PIM_instr.scr1_addr);
  PIM_instr.dst_addr = virt_to_phys(PIM_instr.dst_addr);

  // Send PIM instructions to MC
  // writeq(8-byte DATA, DST address)
  writeq(PIM_instr.scr0_addr, PIMQueue_SRC0);
  writeq(PIM_instr.scr1_addr, PIMQueue_SRC1);
  writeq(PIM_instr.dst_addr, PIMQueue_DST);
  writeq((PIM_instr.opcode<<2 | PIM_instr.initial_ordering), PIMQueue_INSTR);

  // Memory barrier for writeq function
  wmb();
}
```

FIGURE 10. **An example of the PIM device driver.**

sent by the driver and the memory request sent by another application, the request queue scheduler inside the memory controller naturally preserved the dependence at command scheduling.

The implementation of the software we have described so far is a way to operate in our development environment. The PIM device driver is not required if the PIM requests can be sent directly from a memory controller without using the PCIe interface. Also, during the PIM execution, an exception can occur when decoding unsupported PIM instructions and performing the PIM calculations in the arithmetic unit. The PIM instructions executed by the offloading method cannot support precise exceptions. If the exception occurs, PIM notifies an error state to a host process and aborts the execution, in the same way as when an exception occurs in the existing I/O device.

V. PERFORMANCE EVALUATION

In this section, after describing the experimental method, we evaluate the performance of the proposed PIM architecture in terms of the execution cycles, speedup, and overhead in time, and energy consumption through RTL simulation under various execution environment. *We present a performance analysis on the baseline of the external full bandwidth performance of the memory rather than directly comparing with the previous researches since the performance of PIM on a base die of a 3D-stacked memory [19]–[21] cannot be better than that provided by the full bandwidth in any case.*

A. EXPERIMENTAL SETUP

As described in the experimental platform of Section 5, we modeled the memory controller and the PIM device in FPGA for our research. At this time, one die of HBM2 was modeled as the memory device, and our experiment configurations are shown in Table 2. We assumed the execution time ratio of MAC and reducer is 2:1 by the reason that an 8-bit MAC consists of 9 adder stages and a 16 8-bit operands reducer consists of 4 adder stages. We implemented both the PIM memory controller and the PIM memory in Verilog

TABLE 2. Experiment configurations.

Configurations		Parameters
DRAM	# of channels	1
	# of dies	1
	# of banks	16
	# of rows per bank	256
	Row size per bank	2KB
	Data I/O (DQ pin) width	128-bit
	Bandwidth per die	32GB/s
	Burst length	4
	tRCD	16ns
	tRP	16ns
	tRTP	5ns
	tCCD	6ns
	tCWL+tWTR	16ns
ALUs	# of MACs per bank	16
	MAC bit-width	$8 \times 8 = 16$
	Reducer per bank	16-bit adder tree

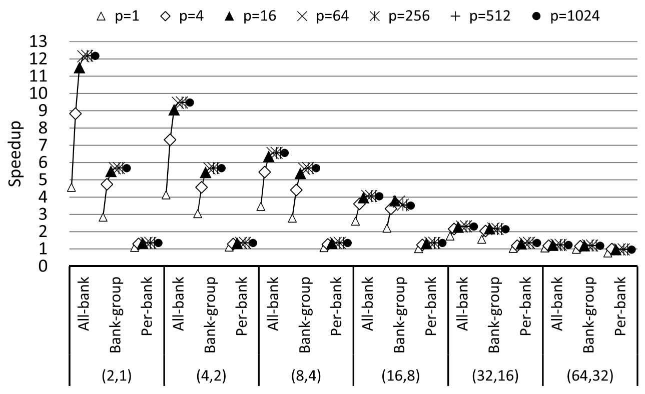


FIGURE 11. Speedup on different compute cycles (*M*, *R*) depending on *p*, where *M* cycles for the MAC operation, *R* cycles for the reduce operation, and ($p \times 1024$) × (1024×1) matrix-vector multiplication.

RTL and obtained accurate execution cycle results through the RTL simulation. The ($p \times 1024$) matrix and (1024×1) vector do not require data movement for the matrix-vector multiplication other than the data movement for the reduction operation when using the data layout described in Section II-A.

We made two assumptions for measuring the performance: 1) All the data required for the matrix-vector multiplication (8-bit×8-bit=16-bit) regardless of the matrix operand sizes were row-aligned in DRAM. After experimenting with the assumption, we analyzed the execution overhead by drawing up a scenario in which a certain probability of row miss occurred during the execution. 2) Also, we assumed that the delays of the modeled memory controller and the instruction transfer from the host processor to the memory controller were ignored.

We have also applied three methods of command scheduling for the PIM execution: 1) all-bank scheduling, 2) per-bank scheduling, and 3) bank-group scheduling. The all-bank scheduling makes all banks operate simultaneously by receiving one command, so it results in the fastest PIM operations. However, the scheduling may suffer from high power consumption. It should be noted that under the scheduling, the standard DRAM operations such as read and write need the change of the existing DRAM structure, but the PIM does not need the change due to no data movement between the PIM device and the memory controller. The per-bank scheduling leverages the current DRAM command interface, which defines the behavior of each bank as a command and commands all banks independently. The reason why each bank receives and processes the standard DRAM commands independently is that each bank performs the different data accesses. However, for the PIM operations, the per-bank scheduling may be considered inefficient because all the banks do the same. However, the high latency of the computing unit would hide the DRAM activity as much as it does. The bank-group scheduling uses the bank group technology added to DDR4 [48] and HBM/HBM2 [40], [41] generations to suggest that a single command operates all the banks in the same bank group, and each group operates independently. We assigned 4 banks to each bank group.

B. SPEEDUP AND EXECUTION TIME

Figure 11 shows the speedup of the various sizes of the matrix-vector multiplications with different latency of the MAC and the reduce operations. The speed was based on the execution time to read all the matrix operands assuming the full DRAM bandwidth (32GB/s) without any calculation divided by the PIM execution time. The experimental results show that we achieved significant speedup by all the schedulings, of which the all-bank scheduling achieved the most speedup over the others. As the latency of the computing unit increases, the speedup decreases, but still very high due to utilizing the full internal bandwidth of DRAM using various levels of parallelism in the PIM execution.

In the ideal design of the computing unit, i.e., (2,1), the ideal speedup of the all-bank scheduling is 16, but the performance measurement result was about 12.2. The performance gap is the same as the overhead caused by inter-bank communication for summation reduction. The speedup of the all-bank scheduling decreases linearly with the latency increment due to that the computing latency gradually dominates the overall execution time. In the case of the bank-group scheduling, we achieved the speedup of 5.7, and in the case of the per-bank scheduling, we achieved 35.4% higher speedup by the burst command to make the bank operations overlapped. In contrast to the all-bank scheduling, the speedup of the per-bank scheduling almost maintains the constant with the computing unit latency increment due to operation overlapped. When increasing the p value, there is no speedup variation since we use both full computing and memory bandwidth by exploiting the task-level parallelism. Therefore, we can conclude that our PIM provides the performance scalability according to the matrix sizes.

Figure 12 shows the ratio of cycle breakdown at the bank execution with $p = 1024$ depending on three scheduling methods in terms of cycles in memory + compute (overlapping), memory-only, compute-only, and idle states. In the

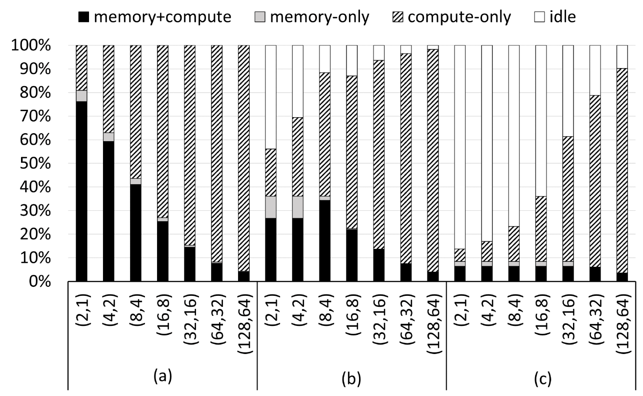


FIGURE 12. Cycle breakdown of the bank execution with $p = 1024$. (a) All-bank. (b) Bank-group. (c) Per-bank.

all-bank scheduling, when the computing latency is (2,1), memory read and computing operations overlap over 70% of the total execution cycle. However, as the computing latency increases, the overlap time decreases and the computing time becomes to dominate the overall execution time. In the per-bank scheduling with the latency (2,1), the idle time occupies more than 80% of the total execution. The reason is that the shared command bus by all the banks prevents the ready commands from being issued, so the overlap execution of computing and memory is minimal. When the computing latency increases, the idle time becomes to the compute-only time. We found that we could not achieve the acceptable performance by the per-bank scheduling due to the limited command bandwidth, but we could reduce the burden of the high-performance ALU design.

The maximum performance of a system using CPU + HBM2 is limited to the external bandwidth, 256GB/s [41]. As discussed in the experiment of Figure 1, it is challenging to expect higher performance gain by CPU + HBM2 than the PIM architecture using the DRAM internal bandwidth. For example, using one die and 16 banks, which are our experimental specification, the internal bandwidth is 512GB/s, which is twice as high as the HBM2 maximum external bandwidth. Therefore, in the case of massively parallel execution in DRAM for memory-bound applications, PIM would achieve significant performance gain.

C. ENERGY CONSUMPTION

Figure 13 shows the log-scaled energy consumption by varying the computing latency and matrix size, p, and the data are normalized by the energy consumption with $p = 1$ execution. We assumed that the read/write operation consumed $168.6mW$, the idle state consumed 52.8mW [49] and the MAC and the Reducer units did 28.8mW and 4mW, respectively [50]. Also, we calculated the scale-point by multiplying the energy consumption of $p = 1$ with p, which represents the energy scalability if the energy consumption of each p approaches to the point. The figure shows that the energy consumption increases linearly with the p value, and all the consumption is very close to the scale-point. Therefore, our system provides excellent scalability in energy consumption.

D. PERFORMANCE VARIANT

We show performance analysis in the following two environments: One is when not all data are aligned to one row, and the other is to perform PIM while servicing other standard memory requests. We found that the per-bank scheduling could tolerate the changes of the execution environment such as row misses and other standard memory requests.

1) OCCURRING ROW MISSES

The study of the execution time by row misses is necessary because there is a limit to DRAM page size and the row miss cannot always be unavoidable. Therefore, the overhead from precharge and activation was measured, assuming that the probability of row misses at every two read accesses were 25%, 50%, and 75%. Figure 14 shows the speedup of the PIM to calculate $(1024 \times 1024) \times (1024 \times 1)$ with the row misses, whose baseline was the execution time by performing operand burst reads at the full memory bandwidth with the corresponding the row miss overhead.

In all the cases, the PIM performance was still higher than one, even at high row miss rate. The performance by the all-bank and the bank-group schedulings with high-performance computing unit was very sensitive to the row misses, i.e., a large number of row misses significantly degraded the performance, since the smaller computing cycle had less scheduling opportunity to hide the overhead. However, in the practical computing unit design, for example, at the (16,8) latency, their performance drops were insignificant even with 75% miss ratio, and more precisely, by 23.3% in the all-bank and 14.5% in the bank-group schedulings.

In the per-bank scheduling, even if the row misses increased, the performance with the row misses was almost the same as that without any row miss, and the speedup variation was only 5.4% at the (16,8) latency with 75% row miss. The row miss overhead was hidden by reducing the idle cycle and increasing the overlap between memory and computing operations. For more detailed performance study, we show the cycle breakdown of the execution at the delay of (8,4), (16,8), and (32,16) in Figure 15. The performance tolerance from the row misses in our all schedulings is one of the significant advantages of our design.

2) EXECUTING WITH THE STANDARD MEMORY REQUESTS

We measured the PIM performance variants while servicing the standard memory requests. The experiment was carried out assuming that one standard memory command is requested per every two PIM memory commands during the PIM execution; thus, one-third of the total memory requests are the standard memory commands. Figure 16(a) shows its speedup, whose baseline was the execution time by performing operand burst reads at full memory bandwidth.

The speedup was degraded in all the cases due to the other memory operations, and more precisely, the performance was dropped by 29.7% in the all-bank, 23.1% in the bank-group, 23.6% in the per-bank schedulings at the (16,8) latency when

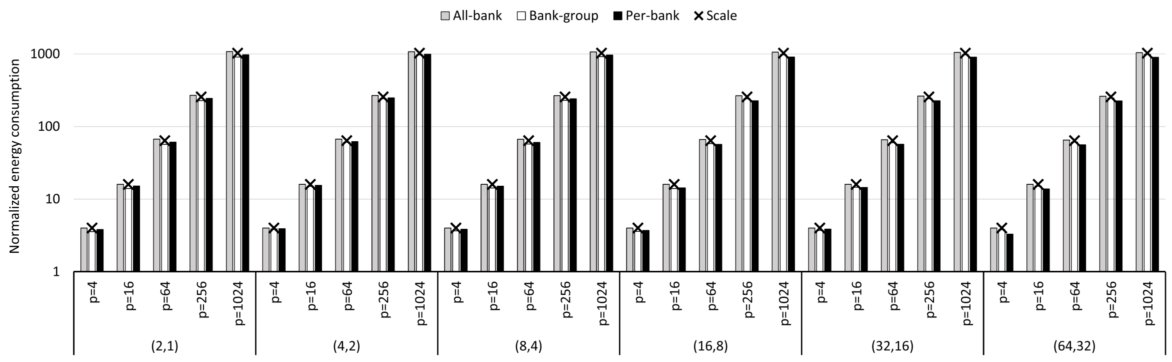


FIGURE 13. Energy consumption by varying the value p. The energy consumption was normalized by the energy consumption with $p = 1$. The “x” symbols represent the scale-points, i.e., the value of multiplying the energy consumption of $p = 1$ execution with p.

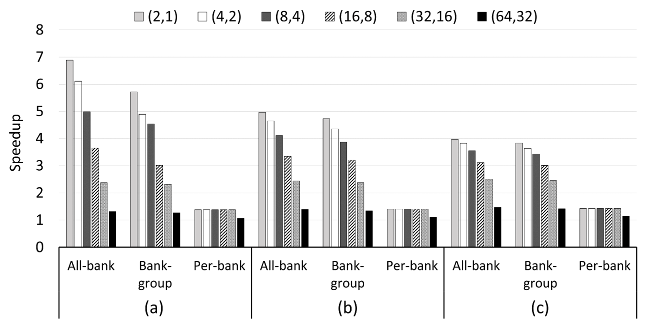


FIGURE 14. Speedup with row misses at $p = 1024$. (a) 25%. (b) 50%. (c) 75%.

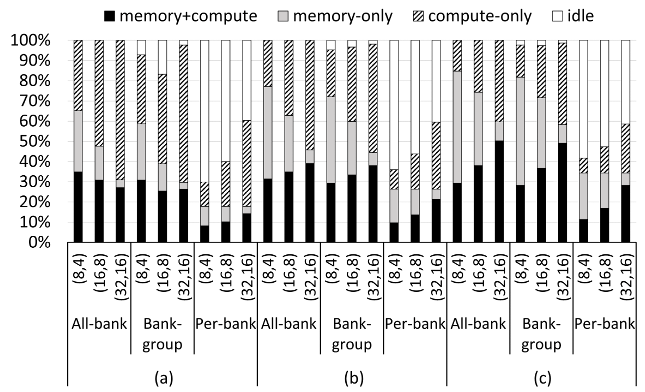
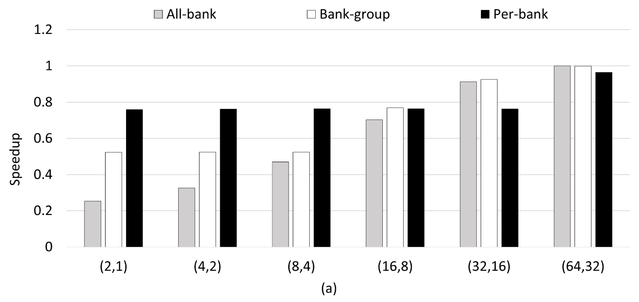


FIGURE 15. Cycle breakdown of Figure 14 at the (8,4), (16,8), and (32,16) latency. (a) 25%. (b) 50%. (c) 75%.



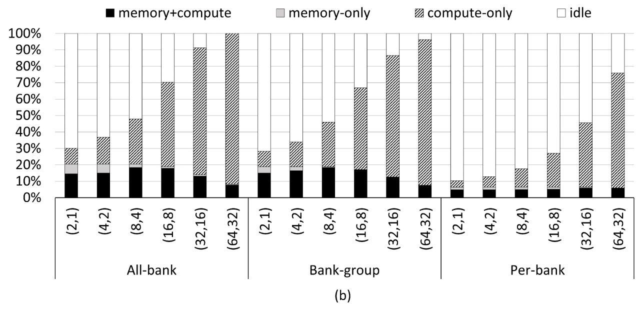


FIGURE 16. Performance when one-third of all the memory requests are the standard memory requests at $p = 1024$. (a) Speedup. (b) Cycle breakdown.

compared with Figure 11. However, the performance degradation was quite surprising when it was estimated that the degradation would be about 33% given that one-third of the total number of memory requests were the standard memory requests. In the cases of the all-bank and the bank-group schedulings, when serving the standard memory requests, all banks must stop; thus, the idle time increased as shown in Figure 16(b). On the other hand, there was no noticeable variant of the cycle breakdown in the per-bank scheduling because a bank could read and compute when other banks are serving other memory requests. The cycle breakdown of the per-bank scheduling is very similar to Figure 12 that did not service any standard memory request. Also, intuitively, the faster computing unit incurred the higher performance drop; 74.6% by the all-bank, 47.6% by the bank-group, and 24.1% by the per-bank schedulings at the (2,1) delay.

VI. RELATED WORK

Recent PIM studies have been extensively researched as Near-Data Processing (NDP) and customized memory devices. The NDP places the hardware logic like general cores and accelerators very near memory devices, in general, on the logic die of the 3D-stacked memory that supports high capacity and high bandwidth. Also, the customized devices use memristors inside the memory to improve the performance of a specific target. In this section, we discuss their research by comparing with our work.

Most PIMs using 3D-stacked memory studies [19]–[23] assigned a task to each general PIM core which is connected with one vault of the 3D-stacked memory in order to perform an offloaded PIM tasks in parallel like multi-core processing. Reference [20] focused on hardware and software interface at the NDP runtime when performing data partitioning, PIM core communication, and task management. Also, this paper proposed memory models between PIM core and a host core. Tesseract [21] focused on the scalability of PIM memory for large-scale graph analysis [32], [33], [51]. It could obtain the scalability through HMC network by proposing a non-blocking message passing algorithm for the communication between intra-cube or inter-cube. GraphPIM [23] supported a software framework, which provided a transparent offloading to a programmer in order to reduce the burden of the program modification. PEI [22] used hardware logic, which profiled data locality according to the cache behavior, to determine the best possible offloaded operations.

PIM research using the existing 3D-stacked memory with general cores incurs several design concerns because a host core and PIM cores exist separately. The first concern is related to address mapping. Since the address mapping between the host core and the PIM cores may be different, overhead from the address translation or re-mapping would occur. Conversely, if the address mapping is the same, the communication overhead between the PIM cores would increase due to the multi-channel interleaving of the 3D-stacked memory. NDP [20] reduced the overhead by moving the channel bits from the existing address mapping. However, this does not guarantee the overall memory performance requirement because standard memory requests cannot utilize the channel-level parallelism. The overhead would become significant in real and multi-workload, but tiny in small memory footprint of applications like in the SPEC 2006 benchmark [52]. The second concern is the high hardware logic overhead for the PIM core, where thermal issues would occur, and the constraints of power dissipation would become greater [53]. NDA [54] proposed the architecture for connecting accelerators with the 3D-stacked DRAM device via TSVs. In this case, the low-power accelerator should be used to resolve the thermal impact, which is an essential problem of the existing 3D-stacked memory. The third concern is ISA. We need to add ISAs to the host processor or extend the existing HMC ISAs for the PIM cores. However, any modification or extension is significant overhead in implementation.

McDRAM [42], very rare but the latest work placed multiply-accumulate (MAC) units in the region of the DRAM column decoder, i.e., inside DRAM, and presented the trade-off in terms of energy consumption, area overhead, and performance. However, McDRAM also neither considered the data layout, serving the standard memory request while executing PIM operations, nor architecture layers.

In the case of the PIM accelerators [54]–[58] for specific targets, both low power and high-performance can be obtained. However, the data movement from the host memory into the accelerator, whose performance is limited by bandwidth. In order to resolve the limitation, several types of research have been proposed [55]–[57], [59]. DaDianNao [56] replaced on-chip memory with a large-capacity eDRAM. TOM [24] changed the on-chip memory of GPU to multiple 3D-stacked memory, but there was an overhead to find the best address mapping through training before the GPU kernel launch to map. [57], [59] suggested the scheduling of CNN operations to increase the reuse of the on-chip memory data.

The customized device PIM is an in-situ analog-based PIM using memristors [60]–[62], where memory cells store not only values but also operations, and most of them used ReRAM [63] as a memory platform. PIM using memristors would simplify the computing the data by the analog operation, so the data need to be handled by ADC/DAC. Therefore, the ADC/DAC area overhead would be significant in memory, and ISAAC [61] proposed new data encoding techniques that can reduce this ADC overhead. Also, there would not be guaranteed that the result of the operation through the ADC always had the same result. In order to reduce the loss of accuracy, PRIME [60] applied the dynamic fixed points [64]. [65] presented a 12×12 memristor and used it as a linear classifier for DNN. However, PIM using a memristor is not feasible at present because the size of the memristor array is sufficiently large to utilize the parallelism [60]–[62].

As a pure software method, the EMU technology [66] is one of the representatives near-data processing techniques in a conventional cluster environment, which migrates and processes threads to nodes where the data is located, rather than moving data to CPU or GPU that needs them.

Our research shows that the PIM can be applied to the real-world computing environment, i.e., would come true shortly by designing a simple computing unit in the most widely used DRAM device and operating it within the range of standard memory behaviors.

VII. DISCUSSION AND FUTURE WORK

When a computing unit is designed to be placed within DRAM, there are so many developmental issues as we discussed before. However, one of the most challenging issues is that the computing unit must be fabricated using the DRAM process. Since the DRAM process different from the logic process, there are several issues. At the implementation, the first process is to make the standard cells using a DRAM process. After the various sizes of the standard cells such as NAND, AND, XOR, NOR, flip-flops, a full-adder, and so on are designed and laid out, the computing unit is Verilog-coded and synthesized with the standard cell library. The designed cells would suffer from considerable delays due to larger Vth of the DRAM process than that of the logic process [67]. With the performance degradation, to support fast read/write access time of memory banks, the computing unit should be deeply pipelined. As the setup time and clock-to-Q delays of flip-flops take a considerable portion of clock cycles in deeply pipelined circuits, reducing those flip-flop delays are one of the main design issues.

When the gate-level netlist is placed and routed after synthesis, large wiring capacitance from a memory bank to the computing unit and the limited number of metal layers are another difficulties encountered when designing the computing unit in DRAM. Long wiring delay path should be pipelined as well with the risks of setup or hold time violations depending on different wire lengths. The wiring congestion incurred by the limited number of metal layers would increase the layout area of computing units after placing and routing (P&R). To satisfy the tight timing closure, the synthesis and P&R tools would employ large standard cells having strong driving strength, which is another reason for significant area overhead in designing computing unit.

VIII. CONCLUSION

In this paper, we proposed the PIM architecture optimized for DRAM behaviors, so we could effectively coordinate the PIM operations with the standard DRAM operations by extending the standard DRAM state diagram for the PIM commands. The extension allowed us to handle the PIM behaviors in the same way as the standard DRAM commands are scheduled and operated on the DRAM devices; thus, PIM can service the standard memory requests during the computation. Also, we proposed the PIM programming and its execution model for quickly applying the conventional parallel execution ones on multi-cores to PIM. We also exploited several levels of parallelism for achieving the full computing performance with minimizing the implementation cost.

By applying our approaches to our HBM2-based experimental platform, we presented how the entire architecture layers from applications to operating systems, memory controllers, and PIM devices worked together for the effective execution. We showed the detailed performance analysis in terms of speedup and energy consumption with varying the computing unit design parameters, row misses, and considering the standard memory requests at the matrix-vector multiplication. By using the 16-cycle MAC and the 8-cycle reducer, we could achieve that the PIM execution was 35.2% and 406% times faster by the per-bank and the all-bank schedulings, respectively, at $(1024 \times 1024) \times (1024 \times 1)$ 8-bit integer matrix-vector multiplication than reading operands at the external full DRAM bandwidth. We are sure that this work provides great insight about real PIM design for energy-efficient data-intensive computing for research and development in our community.

ACKNOWLEDGMENT

(Won Jun Lee and Chang Hyun Kim contributed equally to this work.).

REFERENCES

[1] J. Von Neumann, "First draft of a report on the EDVAC," *IEEE Ann. Hist. Comput.*, vol. 15, no. 4, pp. 27–75, Oct. 1993.

[2] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. 31st Annu. Int. Symp. Comput. Archit. (ISCA)*, p. 212, Jun. 2004.

[3] L. Villa, M. Zhang, and K. Asanovic, "Dynamic zero compression for cache energy reduction," in *Proc. 33rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2000, pp. 214–220.

[4] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 142–149, Feb. 1999.

[5] G. Rivera and C. W. Tseng, "Locality optimizations for multi-level caches," in *Proc. ACM/IEEE Conf. Supercomput.*, p. 2, Nov. 1999.

[6] T. Stocksdale, M.-T. Chang, H. Zheng, and F. Mueller, "Architecting HBM as a high bandwidth, high capacity, self-managed last-level cache," in *Proc. 2nd Joint Int. Workshop Parallel Data Storage Data Intensive Scalable Comput. Syst.*, Nov. 2017, pp. 31–36.

[7] A. Peleg and U. Weiser, "MMX technology extension to the intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, Aug. 1996.

[8] H. Takata, T. Watanabe, T. Nakajima, T. Takagaki, H. Sato, A. Mohri, A. Yamada, T. Kanamoto, Y. Matsuda, S. Iwade, and Y. Horiba, "The D30V/MPEG multimedia processor," *IEEE Micro*, vol. 19, no. 4, pp. 38–47, Jul./Aug. 1999.

[9] M. Tremblay and J. M. O'Connor, "UltraSparc I: A four-issue processor supporting multimedia," *IEEE Micro*, vol. 16, no. 2, pp. 42–50, Apr. 1996.

[10] I. Software. *General Matrix Multiply Sample User's Guide*. Accessed: May 05, 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/4e/e8/intel_ocl_gemm.pdf

[11] Dell. *Dell PowerEdge R720*. Accessed: May 26, 2019. [Online]. Available: https://www.dell.com/downloads/global/products/pedge/dell-poweredge-r720-spec-sheet.pdf

[12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technol. J.*, vol. 6, pp. 4–15, Feb. 2002.

[13] I. Software. *General Matrix Multiply*. Accessed: May 26, 2019. [Online]. Available: https://software.intel.com/en-us/articles/gemm

[14] I. Software. *Intel VTune Amplifier*. Accessed: May 26, 2019. [Online]. Available: https://software.intel.com/en-us/vtune

[15] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): Chips that remember and compute," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 1997, pp. 224–225.

[16] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, Apr. 1995.

[17] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 1999, p. 57.

[18] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *Proc. IEEE Int. Conf. Comput. Design*, Oct. 1999, pp. 192–201.

[19] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3D memory," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, Xi'an, China, Apr. 2017, pp. 751–764.

[20] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Oct. 2015, pp. 113–124. doi: 10.1109/PACT.2015.22.

[21] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ACM/IEEE ISCA*, Jun. 2015, pp. 105–117.

[22] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 336–348.

[23] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 457–468.

[24] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *Proc. 43rd Int. Symp. Comput. Archit.*, Jun. 2016, pp. 204–216.

[25] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," Jul. 2012, *arXiv:1207.0580*. [Online]. Available: https://arxiv.org/abs/1207.0580

[26] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[27] P. Boncz, T. Grust, M. Van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: A fast XQuery processor powered by a relational engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2006, pp. 479–490.

[28] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2013, pp. 1243–1254.

[29] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[30] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, and C.-Y. Lin, "A highly efficient runtime and graph library for large scale graph analytics," in *Proc. Workshop GRAph Data Manage. Exper. Syst.*, Jun. 2014, pp. 1–6.

[31] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Conf. Operating Syst. Design Implement.*, Oct. 2012, pp. 31–46.

[32] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Design Implement.*, Oct. 2012, pp. 17–30.

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[34] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep./Oct. 2011.

[35] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *IEEE Comput.*, vol. 36, no. 12, pp. 39–48, Dec. 2003.

[36] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, Jun. 2010, pp. 175–186.

[37] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[38] Intel. *Thoughts on Intel Xeon E5-2600 v2 Product Family Performance Optimisation–Component Selection Guidelines*. Accessed: May 26, 2019. [Online]. Available: http://infobazy.gda.pl/2014/pliki/prezentacje/d2s2e4-Kaczmarski-Optymalna.pdf

[39] Hybrid Memory Cube Consortium. *Hybrid Memory Cube Specification 2.1*. Accessed: Jun. 25, 2019. [Online]. Avaliable: http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf

[40] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 432–433.

[41] J. H. Cho, J. Kim, W. Y. Lee, D. U. Lee, T. K. Kim, H. B. Park, C. Jeong, M.-J. Park, S. G. Baek, S. Choi, B. K. Yoon, Y. J. Choi, K. Y. Lee, D. Shim, J. Oh, J. Kim, and S.-H. Lee, "A 1.2V 64Gb 341GB/S HBM2 stacked DRAM with spiral point-to-point TSV structure and improved bank group data control," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 208–210.

[42] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "McDRAM: Low latency and energy-efficient matrix computations in DRAM," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2613–2622, Nov. 2018.

[43] I. Park and S. W. Kim, "The distributed virtual shared-memory system based on the InfiniBand architecture," *J. Parallel Distrib. Comput.*, vol. 65, no. 10, pp. 1271–1280, 2005.

[44] SK Hynix. *DDR4 SDRAM Device Operation*. Accessed: Jun. 12, 2018. [Online]. Available: https://www.skhynix.com/static/filedata/fileDownload.do?seq253

[45] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan./Jun. 2011.

[46] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.* vol. 15, no. 1, pp. 45–49, Jan./Jun. 2016.

[47] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 46–50, Jan./Jun. 2017.

[48] K. Koo *et al.*, "A 1.2V 38nm 2.4Gb/s/pin 2Gb DDR4 SDRAM with bank group and $\times 4$ half-page architecture," in *Proc. IEEE ISSCC Dig. Tech. Papers*, Feb. 2012, pp. 40–41.

[49] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved power modeling of DDR SDRAMs," in *Proc. 14th Euromicro Conf. Digit. Syst. Design*, pp. 99–108, Aug./Sep. 2011.

[50] S. Rakesh and K. S. V. Grace, "A survey on the design and performance of various MAC unit architectures," in *Proc. IEEE Int. Conf. Circuits Syst. (ICCS)*, Dec. 2017, pp. 312–315.

[51] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding graph computing in the context of industrial solutions," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2015, pp. 1–12.

[52] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[53] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. Symp. VLSI Technol. (VLSIT)*, Jun. 2012, pp. 87–88.

[54] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 283–295.

[55] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, Mar. 2014, pp. 269–284.

[56] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2014, pp. 609–622.

[57] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, pp. 1–12.

[58] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

[59] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. 43rd Int. Symp. Comput. Archit.*, pp. 367–379, Jun. 2016.

[60] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, pp. 27–39, Jun. 2016.

[61] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 14–26.

[62] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, Mar. 2018, pp. 1–14.

[63] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal—Oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, Jun. 2012.

[64] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," Dec. 2014, *arXiv:1412.7024*. [Online]. Available: https://arxiv.org/abs/1412.7024

[65] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, pp. 61–64, May 2015.

[66] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the Emu system architecture," in *Proc. 6th Workshop Irregular Appl., Archit. Algorithms (IA)*, Nov. 2016, pp. 2–9.
[67] Y.-B. Kim and T. Chen, "Assessing merged DRAM/logic technology," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 1996, pp. 133–136.

WON JUN LEE received the B.S. degree in electrical engineering from Korea University, South Korea, in 2014, where he is currently pursuing the integrated M.S. and Ph.D. degrees with the Compiler and Microarchitecture Lab. His research interests include microarchitecture and memory system designs.

CHANG HYUN KIM received the B.E. degree in electrical engineering from Korea University, Seoul, South Korea, in 2016, where he is currently pursuing the integrated M.S. and Ph.D. degrees with the Compiler and Microarchitecture Lab. His research interests include microarchitecture, memory designs, and SoC design.

YOONAH PAIK received the B.E. degree in electrical engineering from Korea University, Seoul, South Korea, in 2015, where she is currently pursuing the integrated M.S. and Ph.D. degrees with the Compiler and Microarchitecture Lab. Her current research interests include microarchitecture and memory system design.

JONGSUN PARK received the B.S. degree in electronics engineering from Korea University, Seoul, South Korea, in 1998, and the M.S. and Ph.D. degrees in electrical and computer engineering from Purdue University, West Lafayette, IN, USA, in 2000 and 2005, respectively. He joined the Digital Radio Processor System Design Group, Texas Instruments Incorporated, Dallas, TX, USA, in 2002. From 2005 to 2008, he was with the Signal Processing Technology Group, Marvell Semiconductor, Inc., Santa Clara, CA, USA. He joined the Electrical Engineering Faculty, Korea University, in 2008. His research interests include variation-tolerant, low-power, and high-performance very large scale integration architectures and circuit designs for digital signal processing and digital communications. He is a member of the Circuits and Systems for Communications Technical Committee of the IEEE Circuits and Systems Society. He served as a Guest Editor for the IEEE TRANSACTIONS ON MULTI-SCALE COMPUTING SYSTEMS. He is an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS. He has also served in the technical program committees for various IEEE/ACM conferences, including ISCAS, ASP-DAC, HOST, VLSI-SoC, ISOCC, and APCCAS.

IL PARK received the B.S. and M.S. degrees in electronics and electrical engineering from the Pohang University of Science and Technology (POSTECH), and the Ph.D. degree in electrical and computer engineering from Purdue University. He was a Researcher with LG Electronics, Samsung Electronics, and the IBM Thomas J. Watson Research Center. He is currently the Head of design innovation with the DRAM Biz Unit, SK hynix, Inc.

SEON WOOK KIM received the B.S. degree in electronics and computer engineering from Korea University, in 1988, the M.S. degree in electrical engineering from The Ohio State University, in 1990, and the Ph.D. degree in electrical and computer engineering from Purdue University, in 2001. He was a Senior Researcher with the Agency for Defense Development, from 1990 to 1995 and a Staff Software Engineer with Inter/KSL, from 2001 to 2002. He is currently a Professor with the School of Electrical and Computer Engineering, Korea University. His research interests include compiler construction, microarchitecture, system optimization, and SoC design. He is a Senior Member of ACM.