IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 28 December 2022, accepted 14 January 2023, date of publication 23 January 2023, date of current version 27 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3238812

RESEARCH ARTICLE

PISA-DMA: Processing-in-Memory Instruction Set Architecture Using DMA

WON JUN LEE, CHANG HYUN KIM, YOONAH PAIK, AND SEON WOOK KIM, (Senior Member, IEEE)
Department of Electrical Engineering, Korea University, Seoul 02841, South Korea

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

This work was supported in part by SK Hynix Inc., and in part by the Institute for Information & Communications Technology Promotion (IITP) Grant funded by the Korea Government (MSIP) (Memory-Centric Architecture Using the Reconfigurable PIM Devices) under Grant 2022-0-00441.

ABSTRACT Processing-in-memory (PIM) has attracted attention to overcome the memory bandwidth limitation, especially for computing memory-intensive DNN applications. Most PIM approaches use the CPU's memory requests to deliver instructions and operands to the PIM engines, making a core busy and incurring unnecessary data transfer, thus, resulting in significant offloading overhead. DMA can resolve the issue by transferring a high volume of successive data without intervening CPU and polluting the memory hierarchy, thus perfectly fitting the PIM concept. However, the small computing resources of DRAM-based PIM devices allow us to transfer only small amounts of data at one DMA transaction and require a large number of descriptors, thus still incurring significant offloading overhead. This paper introduces PIM Instruction Set Architecture (ISA) using a DMA descriptor called PISA-DMA to express a PIM opcode and operand in a single descriptor. Our ISA makes PIM programming intuitive by thinking of committing one PIM instruction as completing one DMA transaction and representing a sequence of PIM instructions using the DMA descriptor list. Also, PISA-DMA minimizes the offloading overhead while guaranteeing compatibility with commercial platforms. Our PISA-DMA eliminates the opcode offloading overhead and achieves 1.25x, 1.31x, and 1.29x speedup over the baseline PIM at the sequence length of 128 with the BERT, RoBERTa, and GPT-2 models, respectively, in ONNX runtime with real machines. Also, we study how our proposed PISA affects performance in compiler optimization and show that the operator fusion of matrix-matrix multiplication and element-wise addition achieved 1.04x speedup, a similar performance gain using conventional ISAs.

INDEX TERMS Processing-in-DRAM, direct memory access, instruction set architecture, PIM offloading.

I. INTRODUCTION

Most modern computers are based on a stored-program concept, i.e., the von Neumann architecture [1], where instructions and data are stored in a separate memory and handled the same. Therefore, when processing low-locality data-intensive applications, such as recently emerging DNN (Deep Neural Network) [2], e-commerce [3], [4], [5], [6] graph applications [7], [8], and so on, the memory performance determines the overall system performance.

Processing-in-Memory (PIM) architectures have been actively studied by placing computing units close to [9], [10], [11], [12], and [13] or inside memory [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] to overcome the memory bandwidth limitation. PIM can maximize internal memory bandwidth for the computation using bank-level parallelism [14], [15], [17], [18], [22], [23], [24], [25], [26], thus providing high computation performance. For example, the decoupled PIM [26] achieved a speedup of 75.8x and

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

1.2x over CPU and GPU at the Level-3 BLAS, respectively. Samsung FIM [23] achieved a speedup of 11.2x and 3.5x over CPU for memory-bound neural network kernels and applications [27], [29], [32], respectively.

Despite the PIM's performance advantages, in-DRAM PIM has yet to be commercially available because of the following two factors. First, most designs have pursued ''accelerator the first approach'' instead of ''memory the first,'' affecting all architecture layers' design, such as cores [9], [11], [12] and memory controllers [14], [15], [17], [18], [24], thus incompatible with our current computing platforms. For example, the latest PIM studies from Samsung [23] and UPMEM [30] separated the PIM memory area from the non-PIM memory to avoid incompatibility with the JEDEC memory standard [31] for supporting all-bank execution. Our recent work and baseline for this research, the decoupled PIM [26] satisfies the standard memory interface. However, its performance is lower than the all-bank PIMs due to its per-bank execution.

Second and more importantly, the PIM instruction set architecture (ISA) is unavailable, which fully supports compatibility with commercial computing platforms and perfectly fits the PIM concept and the PIM-target application characteristic. Most PIMs developed their own PIM instructions [11], [12], [22], [23], [30] and offloaded them to the PIM engine by modifying the bus interface and using the CPU load/store instructions. It increases the hardware cost and raises the system performance issues, such as making the core very busy and incurring high latency to access uncacheable PIM areas. Recent PIMs use Direct Memory Access (DMA) as the offloading mechanism [22], [25], [26], [30], [32] to resolve the performance issue by transferring opcodes and large-size operands without CPU intervention.

However, the DMA-based offloading method for in-DRAM PIM has only been used to reduce the offloading overhead. From a performance point of view, due to the PIM's design characteristics, the approach still introduces significant offloading overhead. There is little space available in commercial DRAMs where we can implement the PIM engines, i.e., only a 64-byte register for one source operand and a 16-byte latch for the other are allowed. The tiny resource limits one DMA descriptor's data transfer size (in general, 64-bytes$\times$number of banks), thus requiring a significantly large number of DMA descriptors for fetching PIM-targeted large-size operands. One PIM execution stream, in general, alternately fetches two source operands, differently performing in the datapath, thus consisting of one opcode descriptor, two source operand descriptors, and one destination operand descriptor per OS page. Therefore, a smaller DMA data transfer size requires more operand descriptors and, consequently, more opcode descriptors, thus incurring significant operand and opcode offloading overhead, which would substantially degrade the performance. An even worse problem is that the opcode size is only 64-byte, which does not fit well with the characteristics of DMA transfers. Our experiment shows that the opcode descriptors occupy about 25% of the total.

In this paper, we propose the first Processing-in-Memory Instruction Set Architecture (PIM ISA) to guarantee full compatibility with current commercial computing platforms, which is critical to the success of the PIM products in the market, and make the following contributions.

- Contribution 1: We introduce PIM ISA using a DMA descriptor called PISA-DMA and use the DMA engine as the PIM ISA offloading engine.

Most fields of the DMA descriptor are already well-defined to fit the PIM concept and the PIM-target application characteristic, i.e., not involving CPU execution and repeatedly executing the large-size data with the same operation. We specify the PIM operands in the source and destination fields of the DMA descriptor and the PIM opcode in the unused bit fields of the DMA descriptor. As a result, we can depict the PIM operand and opcode in one DMA descriptor and use the DMA engine as the PIM offload engine, thus neither adding any hardware components nor modifying computing platforms. Our approach is very cost-effective in implementation.

- Contribution 2: Our PISA-DMA makes PIM programming intuitive.

We can think of committing one PIM instruction as completing one DMA transaction and representing a sequence of PIM instructions using the DMA descriptor list. Therefore, the PIM programming is the same as the DMA programming. Also, since the DMA transactions are serviced one by one, we can think of PIM ISAs as being processed in order as well. When processing a transaction, i.e., each PISA-DMA, we can execute memory requests in parallel across banks by exploiting the bank-level parallelism and in-order inside a bank. The separate opcode and operand offloading in the previous works [22], [32] asked a user to carefully program for synchronizing the PIM memory requests between the opcode and the operand execution, thus incurring high programming complexity and related execution overhead.

- Contribution 3: Our PISA-DMA minimizes the offloading overhead.

We cannot reduce the operand fetching for the PIM execution, i.e., difficult to reduce the operand descriptors. However, if we express the opcode and operands in a single descriptor together, we can eliminate the opcode descriptors. The elimination allowed us to reduce the total number of descriptors by 25.8%, 26.1%, and 24.9%, thus achieving significant speedups of 1.25x, 1.31x, and 1.29x compared to the baseline PIM [26] in BERT [33], RoBERTa [34], and GPT-2 [35] models, respectively, with a sequence length of 128.

- Contribution 4: We can apply traditional compiler optimization techniques to our PIM code generation.

We fused the matrix-matrix multiplication with the following element-wise addition operators in BERT and found that PISA reduced 2.9% descriptors and achieved 1.04x speedup

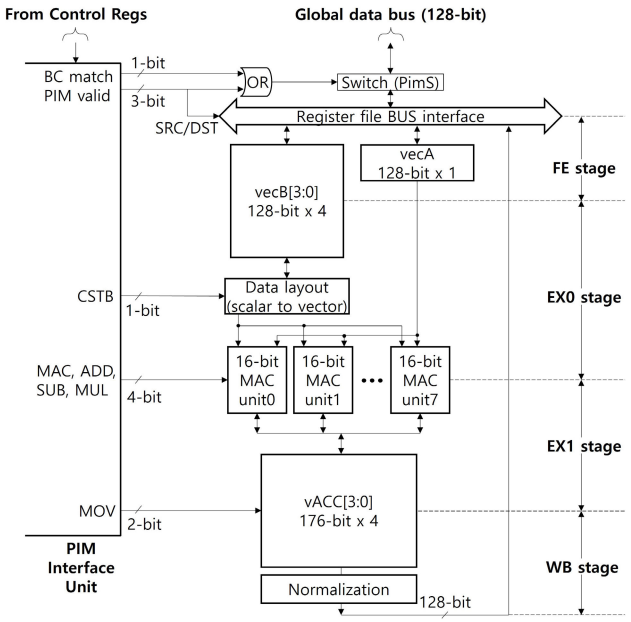

FIGURE 1. The decoupled PIM datapath [26].

compared with the unfused PISA execution, similar to the CPU parallel execution. Therefore, we decided that we would apply various compiler optimization techniques to our PIM code generation without concerns.

This paper consists of the followings: Section II describes decoupled PIM, the baseline architecture of this paper, and Section III proposes PISA-DMA instructions using DMA descriptors and describes the PISA-DMA execution flow. Section IV shows the performance evaluation, and Section V discusses the related works. Then, we present the conclusion in Section VI.

II. BACKGROUND: DECOUPLED PIM

We propose the PISA-DMA based on the decoupled PIM [26], so we review its datapath, interface unit, and execution flow of matrix-matrix multiplication.

A. DATAPATH

Fig. 1 shows the decoupled PIM datapath with the 128-bit DRAM bank data bit-width [25], [26]. We could not afford abundant computing resources due to the space limitation in DRAM for the PIM development. Nevertheless, to compute with the 4-cycle burst standard read/write requests, each bank embeds the datapath including 8 bfloat MACs for the 8-way vector computations, one 128-bit general vector register (`vecA`), one 128-bit×4 general vector register (`vecB[3:0]`), and one 176-bit×4 accumulator register (`vACC[3:0]`). `vecB[3:0]` stores the whole burst, and `vecA` stores only 1-cycle burst from the whole burst. The datapath consists of 4 pipeline stages: FE stage for fetching operands from a bank, EX0/EX1 stages for the MAC computations (multiplication in the first stage and addition in the second stage), and WB stage that writes `vACC` to the bank.

The control unit configures the datapath before one DMA transaction fetches and stores the PIM operands by decoding the information in control registers in the decoupled PIM Interface Unit, provided by the PIM opcode offloading. The pre-configured datapath allows us to significantly lower power consumption by avoiding instruction decoding at every computation.

B. INTERFACE UNIT

Fig. 2 shows the decoupled PIM Interface Unit, which is inside the PIM DRAM and shared by all banks, complying with the JEDEC standard memory interface, receiving `Command`, `Address`, and `Data` signals from standard memory requests as a conventional DRAM device. A programmer (PIM library) offloads the opcode, i.e., stores either the PIM source or destination operand address, its size, and the datapath configuration information of the PIM engine in uncached memory-mapped Control Regs (`REG A/B/C/D`) before the PIM execution. After initializing all the registers, we ask the CPU to initiate the DMA transfer, and the DMA engine issues memory requests to the PIM device. After completing the transfer, the DMA engine interrupts the CPU to notify the completion of the PIM execution.

During the PIM execution, the PIM Request Identification Unit (RIU) distinguishes the PIM memory requests by matching the PIM operand information in the operand registers (`REG A/B/C`) from the incoming memory requests and provides the data to the engine if matched. Each bit of the configuration register (`REG D`) corresponds to one control signal of the PIM datapath.

The decoupled PIM performs two execution phases of memory and computation by controlling each bank's `PimS` switch in Fig. 2 by the PIM memory requests. All memory requests use the `DataS` switch to connect to the global bus, i.e., to place data on the bus for read requests and acquire data from the bus for write requests. At the memory phase, we read the bank-private operands from a bank and turn on the corresponding bank's `DataS` switch to place them on the global bus. At the same time, RIU recognizes the bank-private operands and turns on the bank's `PimS` switch to store them in its PIM engine's registers. After performing the memory phase bank-by-bank, at the computation phase, we read the bank-shared operands from a bank and turn on the corresponding bank's `DataS` switch to place them on the global bus. At the same time, RIU recognizes the bank-shared operands and generates the `BC match` signal to notify the broadcast to all banks' engines. The signal turns on all banks' `PimS` switches so that all banks' engines receive the broadcast data from the global data bus and perform the computation [26]. The broadcast makes all banks perform the computation simultaneously, thus reaching the computing throughput of the all-bank PIM without any data conflict on the global bus.

C. MATRIX-MATRIX MULTIPLICATION ALGORITHM

Fig. 3 shows the decoupled PIM's matrix-matrix multiplication using an optimal tiling size of $(32 \times 32) \times (32 \times 16)$ [26].

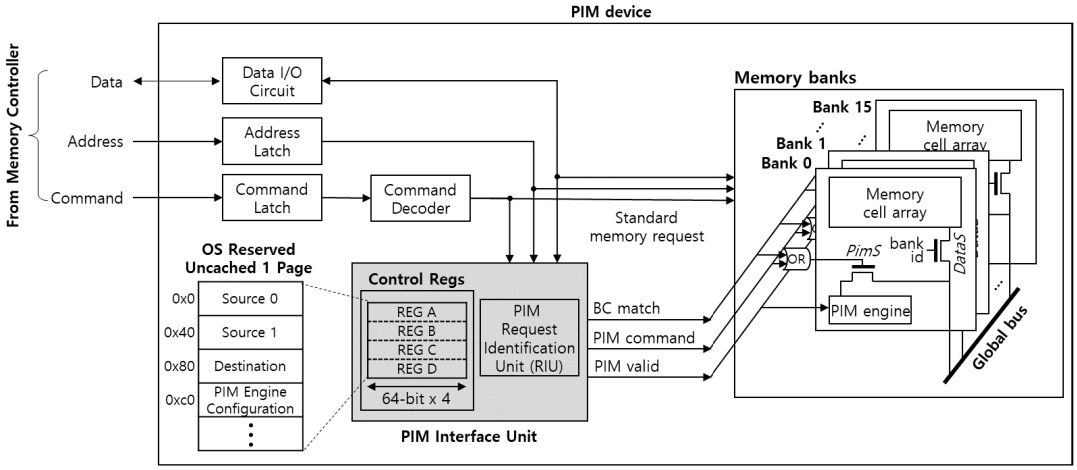

FIGURE 2. The decoupled PIM [26] device interface. The graded component was modified for our work.

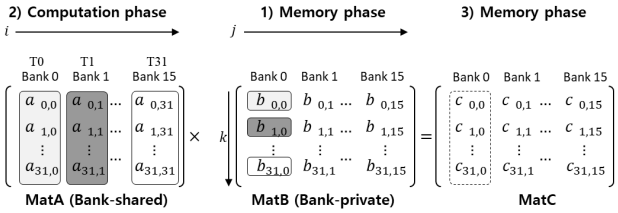

FIGURE 3. Matrix-matrix multiplication algorithm of the decoupled PIM [26].

Each bank j ($j = 0, 1, \ldots, 15$ where j is a bank number) multiplies pairs of ($a_{0:31,0}$, $b_{0,j}$), ($a_{0:31,1}$, $b_{1,j}$), $\cdots$, ($a_{0:31,31}$, $b_{31,j}$) and accumulates the multiplication results one-by-one to calculate $c_{0:31,j}$. The execution performs the following phases in order:

1) *Memory phase for fetching bank-private operands*: Each bank reads 64-byte (a burst size) columns of the bank-private operand *MatB* from its memory cell array and stores them to `vecB`.
2) *Computation phase for all-bank execution*: We reuse the *MatB* elements in `vecB` and broadcast a column of *MatA* one by one for the all-bank execution. We repeat 32 times during Ti where $i = 0, 1, \ldots, 31$ for broadcasting $a_{0:31,i}$. At $T0$, the first element of `vecB` is multiplied with $a_{0:31,0}$ to generate the partial sum of $c_{0:31,j}$. The DRAM burst broadcasts 64-byte $a_{0:31,i}$ and stores its 16 bytes per cycle to vecA, i.e., $a_{0:7,i}$, $a_{8:15,i}$, $a_{16:23,i}$, and $a_{24:31,i}$ in order. At every cycle, the storing triggers MAC operations; 8 ALUs multiply `vecA` with $b_{0,j}$ and produce a partial sum of $c_{0:31,j}$ through 4 cycles; $c_{0:7,j}$, $c_{8:15,j}$, $c_{16:23,j}$, and $c_{24:31,j}$. The element $b_{0,j}$ is reused for the multiplication with $a_{0:31,0}$, i.e., 32 times. After repeating the operations from $T0$ to $T31$, $c_{0:31,j}$ is available in `vACC` of each bank. We return to the previous memory phase whenever requiring a new bank-private operand *MatB* if the $k > 32$.
3) *Memory phase for storing the results into memory*: Each bank stores a 64-byte `vACC` in the memory cell array.

We transposed all the matrices considering the DRAM's read/write granularity of 64 bytes and followed the conventional address mapping, locating 9 to 6 address bits as bank id [25]. In other words, the continuous data distributes in all 16 banks with interleaved 64 bytes.

III. PISA-DMA: PIM INSTRUCTION SET ARCHITECTURE USING DIRECT MEMORY ACCESS

In this section, we describe how to design and execute the PISA-DMA, which represents both the PIM opcode and operand in one descriptor without modifying any architecture layers.

A. PIM EXECUTION BEHAVIOR VS. DMA

We should carefully handle the PIM opcode and operands for the correct execution, requiring the following methods: 1) The control registers in the PIM Interface Unit should be uncached memory-mapped, as discussed in Section II-B. 2) The most up-to-date source operands should be in DRAM before the PIM execution. 3) Similarly, the valid destination operands should be in DRAM after the PIM execution.

To support the second and third methods, we should flush cached PIM operands into DRAM before the PIM execution and invalidate them. The cache flush and invalidation incur significant overhead, so most PIM studies declared the PIM operands as uncached attributes [22], [23], [24], [25], [26]. However, the CPU access to the uncached data is too slow because of their strictly ordered memory operations. The slowness also affects the first method in terms of performance.

Most PIM-target application performs the computation using bulk and continuous memory requests. Therefore, the memory requests generated by the CPU incur significant overhead due to their large amount and slow uncached

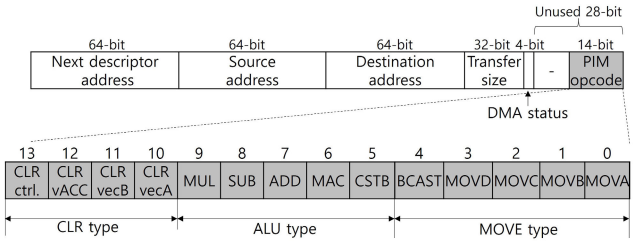

FIGURE 4. The PISA-DMA instruction format using the DMA descriptor.

attribute, which degrades the overall performance. A DMA engine is designed to transfer bulk and successive data without CPU intervention from one memory to another, i.e., suitable for uncached data. Therefore, when considering the PIM opcode and operand management and the PIM application characteristics, we can conclude that the DMA engine is well suited for the PIM operation.

B. PISA-DMA INSTRUCTIONS

1) FORMAT

Fig. 4 represents a PISA-DMA format using a DMA descriptor [36]. Each PISA-DMA consists of an opcode and one source and destination operands (their start addresses and size) like most ISAs, i.e., x86, ARM, RISC-V, etc. The descriptor already contains the fields for specifying the operands (64-bit source and destination addresses, 32-bit transfer size), so we use 14 bits of the unused descriptor field for storing the opcode. Also, the descriptor contains the next descriptor address, allowing us to represent a sequence of PISA-DMA instructions. We can represent both the PIM opcode and operands information together in one DMA descriptor without modifying a DMA descriptor format, thus not requiring modification of a DMA engine. When the DMA engine fetches a descriptor, the PISA-DMA Interface Unit recognizes a PISA-DMA instruction and configures the PIM datapath using the PIM opcode specified in the descriptor. Then, the DMA engine fetches the operands.

In the data transfer operation, the DMA engine issues both read requests using the source address and write requests to store the read data using the destination address with the transfer size. However, PIM is different, i.e., it needs only one operand. The DMA engine issues only read requests to fetch the PIM operand from memory and provide the read data to the datapath for the computation; therefore, we do not need to specify the destination operand, which the DMA descriptor requires. For this purpose, we reserve eight pages (32KB), the maximum size of the consecutive memory accesses by the PIM math library. Similarly, we assign the source operand as the reserved space for storing the PIM operands in memory.

2) OPCODES

Table 1 shows the PISA-DMA opcodes to configure the PIM datapath. We divide them into MOVE, ALU, and CLR types.

The MOVE type specifies operand fetch and store between a memory bank and registers. MOVA/MOVB configures the movement path of incoming data from a bank to the `vecA`/`vecB` register by the PIM read request, respectively. MOVC configures the movement path for storing data in the `vACC` register to a bank by the PIM write request. MOVD is an opcode to configure the data movement from the `vecB` register to the `vACC` register for element-wise ADD/SUB operations, requiring two source operands from `vACC` and `vecA`. BCAST is an opcode to broadcast the data from one bank to all banks.

The ALU type specifies arithmetic operations, such as MAC/ADD/SUB/MUL. CSTB is an opcode for constant value broadcasting, and k represents a counter value. k is auto-incremented by 1 for every PIM read request, and a 16-bit constant value is selected from the 512-bit `vecB` register. The selected 16-bit is broadcast to each input of the entire 8-way 16-bit MAC unit. When k is 31, it is initialized because all data of the 512-bit `vecB` register have been used. All the computation outputs are stored only in the `vACC` register.

The CLR type is an opcode for clearing the broadcasting counter, control registers, and general registers.

C. INTERFACE UNIT

In the decoupled PIM, we store the information of two source operands (src0 and src1) and one destination operand (dest) *at one time, i.e., using one write memory command*, representing the execution of "src0 `op` src1 → dest." Then, we read src0, read src1, and store dest in order for the execution using memory requests. In PISA-DMA, we represent one operand and one opcode in one descriptor. Therefore, we reshape the execution of the decoupled PIM into "read src0; `op` src1 → acc; store acc into dest," needing only one operand at one time.

Therefore, we can reduce two source registers (`REG A/B`) and one destination register (`REG C`) in RIU [25] to one (`Rop`). However, we need one more register (`Rdesc`) to store the address information about the PISA-DMA descriptors. As a result, we could reduce 64-bit×4 registers to 64-bit×3 registers and 3 address matching logics to 2 address logics. The configuration register of the decoupled PIM (`REG D`) was renamed to (`Rconf`) for PISA-DMA.

D. EXECUTION

The PISA-DMA execution steps are as follows, and Fig. 5 shows them:

① *Offloading the PIM instructions*: CPU stores the PISA-DMA descriptors (i.e., instructions) in DRAM and its address in `Rdesc` of the PIM device's PISA-DMA IU.

② *Enabling the DMA engine*: CPU stores the descriptor address in the DMA control register and activates the DMA engine.

③ *Identifying and decoding the PISA-DMA descriptors*: When the DMA engine fetches the descriptors from DRAM, the PISA-DMA interface unit in DRAM identifies the PISA-DMA descriptor by comparing all the incoming read request addresses with `Rdesc`. When

TABLE 1. PISA-DMA opcodes.

Types	Opcode	Mnemonic	Operation (i=0∼3 for a burst length)
MOVE	0x0001	MOVA vecA, MemSrc	vecA ← MEM[i]
	0x0002	MOVB vecB, MemSrc	vecB[i] ← MEM[i]
	0x0004	MOVC MemDst, vACC	MEM[i] ← vACC[i]
	0x0008	MOVD vACC, vecB	vACC[i] ← vecB[i]
	0x0010	BCAST $\text{vecA}_{all\ banks}$, MemSrc	$\text{vecA}_{all\ banks}$ ← MEM[i]
ALU	0x0020	CSTB MAC_{srcB}, vecB	MAC_{srcB} ← vecB[i][16*(k+1)-1:16*k] (k=0∼31)
	0x0040	MAC vACC, vecA, vecB	vACC[i] ← vecA×vecB[i] + vACC[i]
	0x0080	ADD vACC, vecA	vACC[i] ← vecA + vACC[i]
	0x0100	SUB vACC, vecA	vACC[i] ← vecA – vACC[i]
	0x0200	MUL vACC, vecA, vecB	vACC[i] ← vecA×vecB[i]
CLR	0x0400	CLR vecA, 0	Clear vecA
	0x0800	CLR vecB, 0	Clear vecB
	0x1000	CLR vACC, 0	Clear vACC
	0x2000	CLR CTRL_REGS, 0	Clear CTRL_REGS

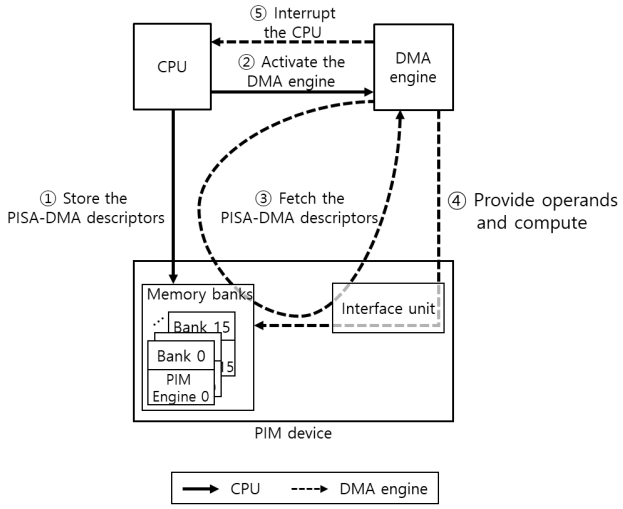

FIGURE 5. The execution flow of the PISA-DMA instructions.

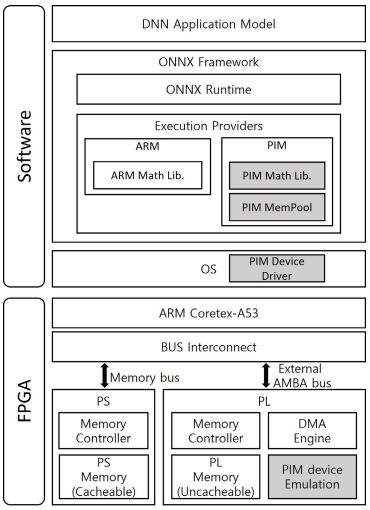

FIGURE 6. The overall architecture of the experimental platform. The graded components were added and modified for our work from the decoupled PIM [26].

identified, the interface unit decodes the descriptor and stores the decoded information in the interface unit, i.e., the PIM opcode in `Rconf` to configure the PIM engine datapath and the operand in `Rop`.

④ *Executing the PIM computation*: Then, the DMA engine transfers memory requests and, thus, fetches the PIM operands. PISA-DMA IU recognizes the incoming memory requests as the PIM memory requests by matching with the `Rop` register and performs the PIM computation by the configured PIM datapath.

⑤ *Completing and continuing more PISA-DMA descriptors*: After completing the fetched PISA-DMA descriptor, the DMA engine fetches the subsequent descriptors in order, if available. After completing all the descriptors, the DMA engine interrupts the CPU, and we finish the execution.

E. REPRESENTING A SEQUENCE OF PISA-DMA INSTRUCTIONS

A DMA engine generally supports a descriptor list operation to reduce the overhead of handling multiple descriptors. The descriptor list can be one or multiple descriptors; each descriptor in the list means a single DMA transaction.

CPU stores the descriptor list, i.e., a sequence of descriptors, before a DMA transfer. Then, when the CPU initiates a DMA transfer, a DMA engine fetches the descriptor from memory one by one and generates a DMA transaction. When completing a DMA transfer corresponding to one descriptor, a DMA engine fetches the next descriptor using the next descriptor address, as shown in Fig. 4. When completing the DMA transfer of all descriptors in the descriptor list, a DMA engine sends a completion interrupt to the CPU only once.

PISA-DMA represents both an opcode and an operand in one DMA descriptor; therefore, we can represent the sequence of PISAs as the descriptor list. One PISA-DMA instruction commit is equivalent to completing one DMA transaction; the commit of all PISA instructions stored in the descriptor list is the same as completing the code sequence. It allows a programmer to express the PIM codes intuitively and reduces the PISA-DMA execution overhead.

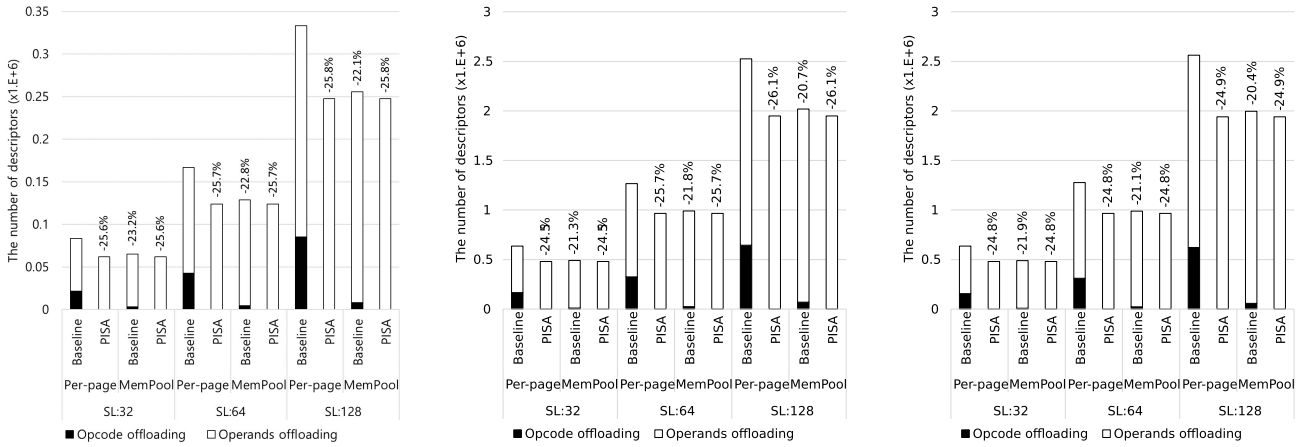

FIGURE 7. The number of the DMA descriptors for the PIM execution when varying the sequence length from 32 to 128 in (a) BERT, (b) RoBERTa, and (c) GPT-2. The percentage numbers represent the PISA's descriptor reduction from the baseline+per-page.

TABLE 2. Experiment configurations.

DNN Model	BERT [33], RoBERTa [34], and GPT-2 [35] Sequence length: 32, 64, and 128
Math Library	CPU: MLAS library [37] PIM: Decoupled PIM library [26]
Framework	ONNX Runtime version 1.6.0 [38] ONNX version 1.9.0 [38]
FPGA Board	HTG-Z920 (Virtex UltraScale)
Host Processor	ARM Cortex-A53 (4 cores) 1.5GHz Private L1 I/D cache 32KB Shared L2 cache 1MB
Memory	PS: 2GB DDR4 DRAM PL: 4GB DDR4 DRAM
DMA Engine	Xilinx CDMA [36]

IV. PERFORMANCE EVALUATION

A. EXPERIMENTAL ENVIRONMENT

Fig. 6 represents our experimental platform to use the PISA-DMA, extended from the decoupled PIM [26] on HTG-Z920 (Xilinx Virtex UltraScale board), and Table 2 describes the experiment configuration. Our baseline architecture targets one channel of HBM [25], [26], where all banks are connected to one shared bus inside the chip.

We measured the performance of three DNN application models, BERT, RoBERTa, and GPT-2, using the decoupled PIM (baseline), the CPU serial execution (CPU_S), and the CPU parallel execution using OpenMP (CPU_P).

We did not modify any hardware components except the PIM device, including the Xilinx DDR4 memory controller and Xilinx CDMA IP, as shown in Fig. 6. We only modified the interface unit from the decoupled PIM device [26] in Section II-B and the PIM library to represent the PISA-DMAs. We developed the PIM in the Programmable Logic (PL) area, i.e., we used the PL memory as the PIM memory. Since the operating frequencies of the PS and PL memory controllers were different, we scaled them for a fair performance comparison with the CPU execution. The decoupled PIM and PISA-DMA PIM used Xilinx CDMA (Central Direct Memory Access) IP [36] as their offloading engine, supporting coalesce of up to 255 descriptors' completion interrupt. We allocated the PISA-DMA descriptors into the PL memory.

Also, we developed the PIM MemPool to provide large contiguous physical pages by utilizing a huge page [39] mechanism at an application level without modifying the OS (+MemPool) in order to see the strength and weakness of the PISA-DMA with and without huge contiguous physical pages. The support of the huge contiguous physical pages can allow us to reduce operand descriptors.

B. PIM DESCRIPTORS: OFFLOADING OVERHEAD

The PIM execution consists of three factors, as mentioned in Section III-D: 1) the offloading to generate and store the DMA descriptors in DRAM, 2) the computation by read/write operands using DMA, and 3) the notification to CPU after completing the PIM execution. Since we used the same PIM math algorithm and PIM architecture for all the PIM's performance studies, only 1) and 3) determine their performance difference. More precisely, the number of the DMA descriptors for the PIM offloading determines the performance, and Fig. 7 shows the numbers when varying the sequence length of 32 to 128. The higher the sequence length (SL), the larger the operand size and the more descriptors. The sequence length represents p in the matrix-matrix multiplication of $(p \times q) \times (q \times r)$.

Our PISA completely eliminated the opcode descriptors by combining them with the operand descriptors; thus, it reduced the total descriptors by 25.8%, 26.1%, and 24.9% compared to baseline+per-page, i.e., without MemPool, in BERT, RoBERTa, and GPT-2, respectively. The MemPool library provides contiguous physical pages, thus significantly reducing the opcode descriptors to specify the operand's address range. However, the library still needs the opcode descriptors,

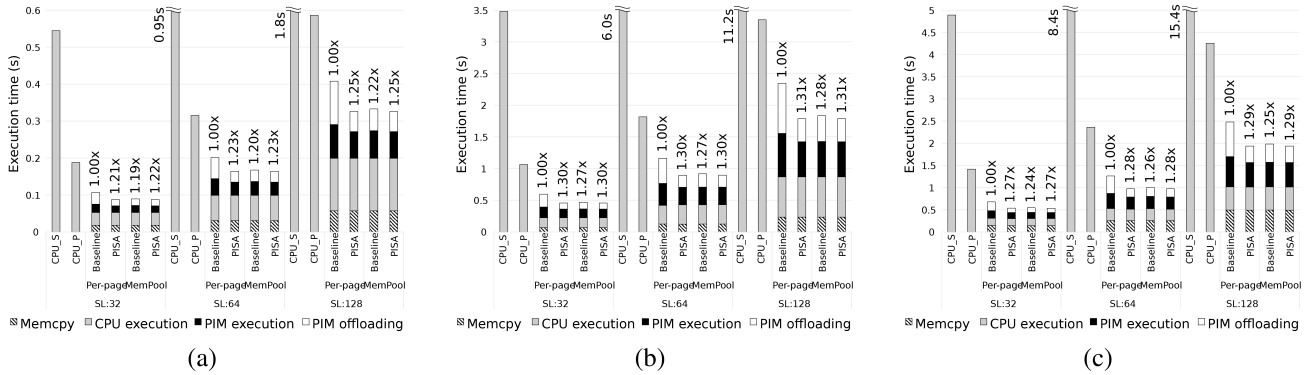

FIGURE 8. The speedup and the execution time breakdown when varying the sequence length from 32 to 128 in (a) BERT (b) RoBERTa, and (c) GPT-2. The numbers represent the speedup with respect to PIM's baseline+per-page.

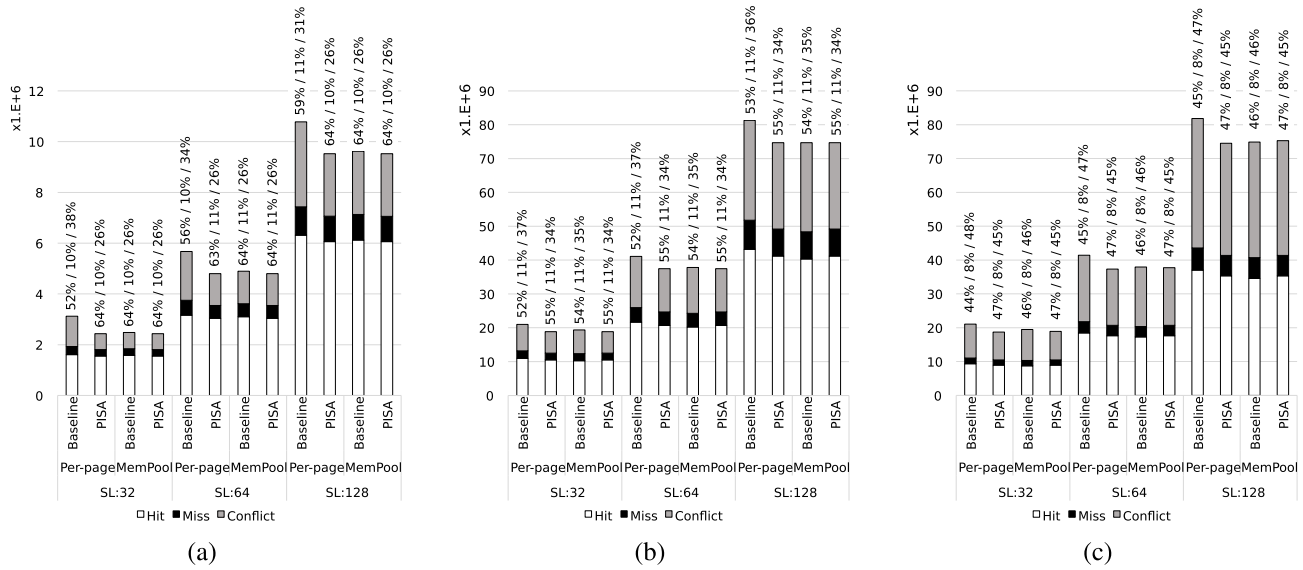

FIGURE 9. The breakdown of row buffer hit/miss/conflict ratio in the PIM executions when varying the sequence length from 32 to 128 in (a) BERT, (b) RoBERTa, and (c) GPT-2. The percentage numbers represent the row buffer hit, miss, and conflict ratio.

and PISA further removes them by 4.8%, 6.8%, and 5.7% compared to baseline+MemPool. As a result, we conclude that we do not need the MemPool function, which shows the strength of our PISA-DMA.

C. SPEEDUP AND EXECUTION TIME

Fig. 8 shows the speedup of total execution time with the PIM's execution time breakdown. When varying the sequence length from 32 to 128, CPU_P achieved the speedups of 2.90∼3.04x, 3.27∼3.37x, and 3.45∼3.63x in BERT, RoBERTa, and GPT-2, respectively, with respect to CPU_S. The speedup was saturated at about 3.5x due to its 4-core execution.

The number of descriptors in Fig. 7 directly affected the PIM performance, and PISA achieved the highest in all the experiments. Memcpy in the PIM execution represents data copy between the CPU and PIM for providing data coherence. The CPU execution in PIM represents that the CPU performs operations not supported by the PIM device. Therefore, they are the same in all cases of the baseline and PISA.

PISA achieved a significant speedup; 6.22x and 5.48x, 7.63x and 6.31x, and 9.15x and 7.98x at the sequence lengths of 32 and 128 in BERT, RoBERTa, GPT-2, respectively, comparing with CPU_S. PISA consistently achieved higher speedup than the baseline PIM with per-page and MemPool approaches by remarkably reducing the offloading time: PISA+per-page achieved the speedups of 1.21∼1.25x, 1.30∼1.31x, and 1.27∼1.29x compared to the baseline+per-page in the three models, respectively. Also, we found that MemPool did not contribute any performance improvement with PISA since the per-page operand descriptor transfers sufficiently large data, at least 4KB at one time. The large size can diminish the timing gap between multiple per-page operand descriptors by providing many memory requests to a memory controller.

D. DRAM BEHAVIOR

Fig. 9 shows the breakdown of the row buffer hit/miss/conflict of the baseline decoupled PIM and our PISA-DMA. The row buffer behaviors are related to 1) the CPU's storing

descriptors, 2) the DMA engine's fetching descriptors, 3) its offloading opcodes, and 4) its fetching and storing operands. We implemented the performance counter inside the Xilinx memory controller for profiling the DRAM behaviors.

The row miss or conflict occurs whenever the different sequence of 1) to 4) occurs. In the case of PISA eliminating the offloading opcode, the row buffer conflict occurs when starting to read a different operand or consuming all data of the opened row buffer. However, in the decoupled PIM, if re-configuration is needed, the row buffer conflict is encountered because the DMA engine reads the opcode descriptor from different DRAM rows and writes the opcode to control registers in the interface unit to configure the PIM engine. Therefore, the baseline PIM incurs higher conflict row misses than PISA. As a result, the row buffer hit ratio of PISA-DMA was about 8.2%, 2.7%, and 2.7% higher, and the row buffer conflict ratio was about 8.4%, 2.8%, and 3.0% lower than of baseline+per-page in BERT, RoBERTa, and GPT-2, respectively. MemPool reduces the number of engine re-configurations in the decoupled PIM, so DRAM behavior was almost similar to PISA.

E. OPERATOR FUSION

Traditionally, a compiler's code optimization on CPU improves performance by executing fewer instructions, and thus, we study how our proposed PISA affects performance in a compiler's code optimization.

We applied the operator fusion to a pair of the matrix-matrix multiplication and its following element-wise addition popularly used for improving performance [40] by removing storing the multiplication results and reloading them for the addition, i.e., spills. We measured the performance variant by fusing 24 matrix multiplications with element-wise additions among 32 matrix multiplication operators in BERT with a sequence length of 128.

Fig. 10(a) shows the number of descriptors in each execution without and with the operator fusion. Without PISA, in the baseline+per-page and baseline+MemPool executions, the fusion totally decreased the descriptors by 0.8% and 2.0% in each execution, reducing the operand descriptors by 3% and 3% but increasing the opcode descriptors by 5% and 25%, respectively. The fusion increases the opcode descriptors due to their interleaved execution (alternatively executing multiplication and addition), thus increasing the PIM device re-configuration. On the other hand, PISA does not require the opcode offloading, thus further reducing the PISA descriptors by 2.9% in both the per-page and MemPool executions.

Fig. 10(b) shows the speedup of the fused execution with respect to the unfused one with their execution time breakdown. The CPU_P improved the performance by 3% compared with the unfused CPU_P execution. Also, the baseline+fusion improved the performance by 3% and 2% in the per-page and MemPool executions, respectively, compared with the unfused baseline execution. On the contrary, the PISA took more performance advantage from the fusion by

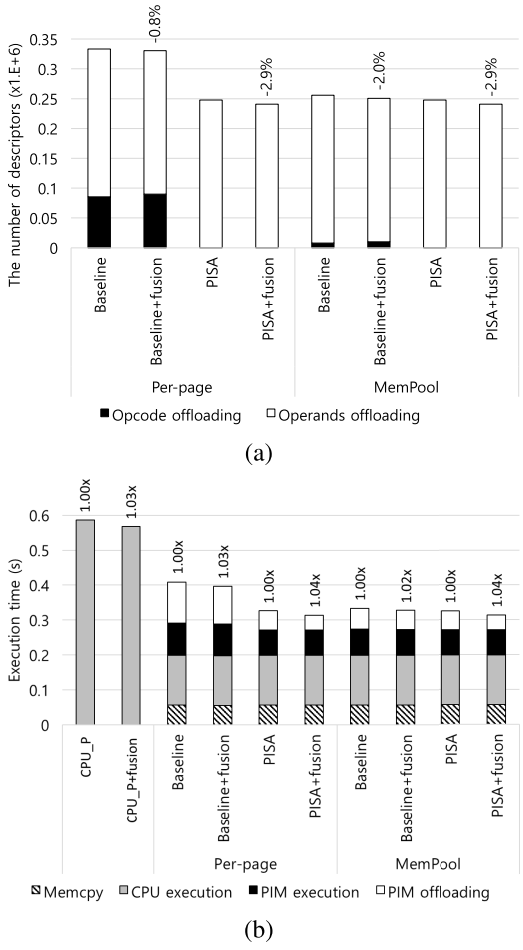

FIGURE 10. The execution with a sequence length of 128 in BERT without and with the operator fusion. (a) The number of descriptors. The percentage numbers represent the fusion execution's descriptor reduction from the unfused execution. (b) The speedup of the fused execution with respect to the unfused one with the execution time breakdown.

not needing descriptors for the device re-configuration; 4% and 4% in the per-page and MemPool executions, respectively, compared with the unfused PISA execution. Therefore, PISA directly improved the performance by reducing the operand descriptors, i.e., by removing spills. The fusion did not improve the performance noticeably since the multiplication required $O(N^3)$, and the fusion removed the spills $O(N^2)$ with $N \times N$ matrices.

F. AREA OF THE PIM INTERFACE UNIT

Table 3 compares the interface unit area of the decoupled PIM and PISA-DMA, consisting of control registers and address matching logics, as discussed in Section III-C. For the comparison, we used the 65nm logic process, similar to the DRAM fabrication characteristics [25].

PISA-DMA reduced the total area by about 29% compared to the decoupled PIM, from 36% and 26% reduction in control registers and address matching logics, respectively. The reduction is crucial since there is little space available for implementation in commercial DRAM. The area of control

TABLE 3. Area of the PIM interface unit (um^2). The percentage numbers represent the PISA-DMA's area reduction from the decoupled PIM.

	Decoupled PIM	PISA-DMA
Address matching logics	1425.61	912.32 (-36%)
Control registers	2626.56	1935.36 (-26%)
Total	4052.16	2847.68 (-29%)

registers occupies two times more than the address-matching logic.

V. RELATED WORK

Most PIM studies [11], [12], [22], [23], [30], [32] have proposed their own PIM ISAs. However, they neither fit the PIM concept, i.e., not involving the CPU execution, nor PIM-target application characteristics, i.e., expressing large-size operands with the same operation. Also, they require hardware modification and are incompatible with current commercial computing platforms.

PEI [11] and GraphPIM [12] needed to modify the core pipeline for their own PIM ISAs, thus lacking compatibility with commercial computing platforms.

In AiM [22] based on GDDR6, a host offloads the AiM instructions to the ISR register inside the memory controller. Then, the controller decodes them into DRAM commands called AiM commands to perform all-bank execution with the DMA-offloaded operands. It requires modifying the memory controller, thus incurring incompatibility with current commercial computing platforms. Also, the separate offloading of the opcode and operands incurs the execution overhead and makes the PIM programming difficult.

UPMEM [30] separates the PIM memory area (MRAM) from the main memory to avoid the memory controller modification and includes the accelerator inside the PIM memory. The DMA engine in the UPMEM device offloads instructions and operands from MRAM to IRAM (instruction memory) and WRAM (scratchpad memory), respectively. The limited resource of IRAM and WRAM incur frequent offloading. Its design and execution follow the traditional accelerators, not PIM handling large-size operands with the same operation.

Samsung FIM [23] also embedded a core to execute PIM-HBM instructions to support the all-bank execution and separated the PIM memory from the main memory. FIM stores the PIM-HBM instructions in the CRF instruction buffer, and the CPU load/store instructions trigger DRAM commands for the execution. Samsung-FIM issues the memory commands in a user-defined order, and the memory requests to PIM can be reordered while passing through the memory hierarchy and a memory controller. RNN-T [32] based on Samsung-FIM utilizes a DMA engine to guarantee memory ordering. However, it still uses separate opcode and operand offloading, thus incurring the execution overhead and increasing the program complexity.

VI. CONCLUSION

This paper proposed PIM ISAs that represent both the PIM opcode and operand in one data structure using the DMA descriptor while providing full compatibility with commercial platforms. Committing one PIM instruction is the same as completing one PISA-DMA transaction. Also, we can represent a sequence of PIM instructions using the DMA descriptor list. It allows a programmer to express the PIM codes intuitively and reduces the PISA-DMA execution overhead.

We measured the performance of PISA with BERT, RoBERTa, and GPT-2 in ONNX runtime on real machines. The PISA's opcode descriptor elimination allowed us to achieve speedups of 1.25x, 1.31x, and 1.29x in the models, respectively, from the decoupled PIM in the per-page memory layout. Also, we showed that PISA diminished the necessity of MemPool to provide large contiguous physical pages and incurred fewer DRAM row buffer misses. Additionally, we studied the performance variants when applying the operator fusion. We found that the PISA execution took a higher fusion advantage than the baseline PIM execution.

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] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, pp. 1–18, Aug. 2012.

[3] M. H. Munna, M. R. I. Rifat, and A. S. M. Badrudduza, "Sentiment analysis and product review classification in E-commerce platform," in *Proc. 23rd Int. Conf. Comput. Inf. Technol. (ICCIT)*, Dec. 2020, pp. 1–6.

[4] C. Wu and M. Yan, "Session-aware information embedding for E-commerce product recommendation," in *Proc. ACM Conf. Inf. Knowl. Manage.*, Nov. 2017, pp. 2379–2382.

[5] S. Rai, A. Gupta, A. Anand, A. Trivedi, and S. Bhadauria, "Demand prediction for e-commerce advertisements: A comparative study using state-of-the-art machine learning methods," in *Proc. 10th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2019, pp. 1–6.

[6] X. Niu, B. Li, C. Li, R. Xiao, H. Sun, H. Deng, and Z. Chen, "A dual heterogeneous graph attention network to improve long-tail performance for shop search in E-commerce," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2020, pp. 3405–3415.

[7] 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. Exp. Syst. (GRADES)*, Jun. 2014, pp. 1–6.

[8] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: SIAM, Jan. 2011.

[9] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "iPIM: Programmable in-memory image processing accelerator using near-bank architecture," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 804–817.

[10] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 655–668.

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

[12] 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.

[13] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian data engine," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 639–651.

[14] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A dram-based reconfigurable in-situ accelerator," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 288–301.

[15] 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 Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2613–2622, Oct. 2018.
[16] S. Cho, H. Choi, E. Park, H. Shin, and S. Yoo, "McDRAM v2: In-dynamic random access memory systolic array accelerator to address the large model problem in deep neural networks on the edge," *IEEE Access*, vol. 8, pp. 135223–135243, 2020.
[17] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Jan. 2017, pp. 273–287.
[18] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: A DRAM based accelerator for accurate CNN inference," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, Jun. 2018, pp. 1–6.
[19] Y. Long, T. Na, and S. Mukhopadhyay, "ReRAM-based processing-in-memory architecture for recurrent neural network acceleration," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 12, pp. 2781–2794, Dec. 2018.
[20] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Mar. 2018, pp. 1–14.
[21] B. Li, L. Song, F. Chen, X. Qian, Y. Chen, and H. H. Li, "ReRAM-based accelerator for deep learning," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 815–820.
[22] Y. Kwon et al., "System architecture and software stack for GDDR6-AiM," in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2022, pp. 1–25.
[23] S. Lee, S.-H. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, and J. Kim, "Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2021, pp. 43–56.
[24] W. J. Lee, C. H. Kim, Y. Paik, J. Park, I. Park, and S. W. Kim, "Design of processing-'inside'-memory optimized for DRAM behaviors," *IEEE Access*, vol. 7, pp. 82633–82648, 2019.
[25] C. H. Kim, W. J. Lee, Y. Paik, K. Kwon, S. Y. Kim, I. Park, and S. W. Kim, "Silent-PIM: Realizing the processing-in-memory computing with standard memory requests," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 251–262, Feb. 2022.
[26] Y. Paik, C. H. Kim, W. J. Lee, and S. W. Kim, "Achieving the performance of all-bank in-DRAM PIM with standard memory interface: Memory-computation decoupling," *IEEE Access*, vol. 10, pp. 93256–93272, 2022.
[27] D. Amodei et al., "Deep speech 2: End-to-end speech recognition in English and Mandarin," in *Proc. 33rd Int. Conf. Mach. Learn.*, Jun. 2016, pp. 173–182.
[28] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, Q. Liang, D. Bhatia, Y. Shangguan, B. Li, G. Pundak, K. C. Sim, T. Bagby, S.-Y. Chang, K. Rao, and A. Gruenstein, "Streaming end-to-end speech recognition for mobile devices," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 6381–6385.
[29] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," Oct. 2016, *arXiv:1609.08144*.
[30] F. Devaux, "The true processing in memory accelerator," in *Proc. IEEE Hot Chips 31 Symp. (HCS)*, Aug. 2019, pp. 1–24.
[31] *JEDEC Standard: DDR4 SDRAM*, Standard JESD79-4, JEDEC Solid State Technology Association, Arlington, VA, USA, Sep. 2012.
[32] S. Kang, S. Lee, B. Kim, H. Kim, K. Sohn, N. S. Kim, and E. Lee, "An FPGA-based RNN-T inference accelerator with PIM-HBM," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2022, pp. 146–152.
[33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," Nov. 2018, *arXiv:1810.04805*.
[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.
[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.
[36] Xilinx. *AXI Central Direct Memory Access V4.1*. Accessed: Feb. 24, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf
[37] Microsoft. *Microsoft Linear Algebra Subprogram*. Accessed: Apr. 2, 2021. [Online]. Available: https://github.com/microsoft/onnxruntime/tree/master/onnxruntime/core/mlas
[38] Microsoft. *ONNX Runtime*. Accessed: Apr. 2, 2021. [Online]. Available: https://onnxruntime.ai/
[39] The Linux Foundation. *Huge TLB Page*. Accessed: Sep. 16, 2022. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[40] Microsoft. *Graph Optimizations in ONNX Runtime*. Accessed: Sep. 16, 2022. [Online]. Available: https://onnxruntime.ai/docs/performance/graph-optimizations.html

WON JUN LEE received the B.S. degree in electrical engineering from Korea University, Jochiwon, Republic of Korea, in 2014. He is currently pursuing the integrated M.S. and Ph.D. degree with the Compiler and Microarchitecture Laboratory, Korea University. 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. He is currently pursuing the integrated M.S. and Ph.D. degree with the Compiler and Microarchitecture Laboratory, Korea University. His research interests include microarchitecture, memory designs, and SoC design.

YOONAH PAIK received the B.E. degree in electrical engineering and the Ph.D. degree in electrical and computer engineering from Korea University, Seoul, South Korea, in 2015 and 2022, respectively. She is currently working with the Samsung Advanced Institute of Technology. Her current research interests include microarchitecture and memory system design.

SEON WOOK KIM (Senior Member, IEEE) 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 at the Agency for Defense Development, from1990 to 1995, and a Staff Software Engineer at 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.