IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 23 May 2022, accepted 24 August 2022, date of publication 31 August 2022, date of current version 12 September 2022.
Digital Object Identifier 10.1109/ACCESS.2022.3203051

RESEARCH ARTICLE

Achieving the Performance of All-Bank In-DRAM PIM With Standard Memory Interface: Memory-Computation Decoupling

YOONAH PAIK, CHANG HYUN KIM, WON JUN LEE, 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 by the Samsung Research Funding and Incubation Center of Samsung Electronics under Project SRFC-IT2002-01.

ABSTRACT Processing-in-Memory (PIM) has been actively studied to overcome the memory bottleneck by placing computing units near or in memory, especially for efficiently processing low locality data-intensive applications. We can categorize the in-DRAM PIMs depending on how many banks perform the PIM computation by one DRAM command: per-bank and all-bank. The per-bank PIM operates only one bank, delivering low performance but preserving the standard DRAM interface and servicing non-PIM requests during PIM execution. The all-bank PIM operates all banks, achieving high performance but accompanying design issues like thermal and power consumption. We introduce the memory-computation decoupling execution to achieve the ideal all-bank PIM performance while preserving the standard JEDEC DRAM interface, i.e., performing the per-bank execution, thus easily adapted to commercial platforms. We divide the PIM execution into two phases: memory and computation phases. At the memory phase, we read the bank-private operands from a bank and store them in PIM engines' registers bank-by-bank. At the computation phase, we decouple the PIM engine from the memory array and broadcast a bank-shared operand using a standard read/write command to make all banks perform the computation simultaneously, thus reaching the computing throughput of the all-bank PIM. For extending the computation phase, i.e., maximizing all-bank execution opportunity, we introduce a compiler analysis and code generation technique to identify the bank-private and the bank-shared operands. We compared the performance of Level-2/3 BLAS, multi-batch LSTM-based Seq2Seq model, and BERT on our decoupled PIM with commercial computing platforms. In Level-3 BLAS, we achieved speedups of 75.8×, 1.2×, and 4.7× compared to CPU, GPU, and the per-bank PIM and up to 91.4% of the ideal all-bank PIM performance. Furthermore, our decoupled PIM consumed less energy than GPU and the per-bank PIM by 72.0% and 78.4%, but 7.4%, a little more than the ideal all-bank PIM.

INDEX TERMS Memory-computation decoupling, in-memory processing, standard memory interface, all-bank execution.

I. INTRODUCTION

Emerging data-intensive applications such as natural language processing deploy Recurrent Neural Networks (RNNs) such as Long Short-Term Memory (LSTM) [1] and Transformer-based models [2]. Their primary characteristic is to process a large amount of data with a very low locality [3], [4], [5], [6]. For example, Bidirectional Encoder Representations from Transformers (BERT) [7] and Generative Pre-trained Transformer (GPT) [8], [9], the transformer-based models for understanding and generating human-like texts, process 110∼335 million and up to billions of parameters, being used only within one layer. Therefore, the von Neumann architecture suffers from a severe data transfer bottleneck from main memory in these workloads, limiting the system performance [10], [11].

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

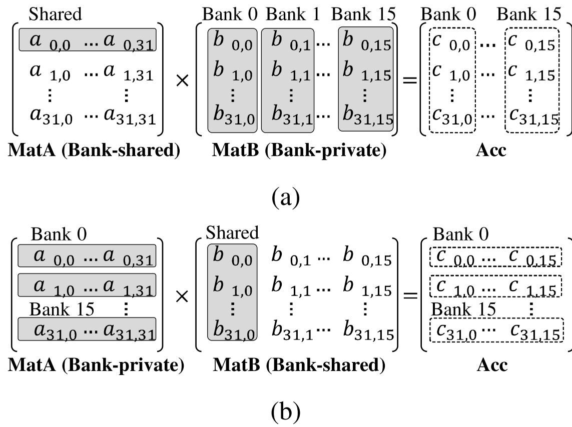

FIGURE 1. Execution flows of the matrix-matrix multiplication. (a) A row of *MatA* is shared by the columns of *MatB*. (b) A column of *MatB* is shared by the rows of *MatA*.

Processing-in-Memory (PIM) has been proposed to alleviate the problem by placing the computing units in the memory [3], [4], [5], [6], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. The in-bank PIM architecture, typically based on DRAMs, places the PIM engines in the bank peripherals, but the space for the engine implementation is very constrained [3], [4], [5], [6], [12], [21]. Also, we should carefully consider the engine operation's power consumption since the high-temperature results in increased unreliable cells, refresh rates, and traffic throttling, leading to performance degradation [22], [23]. Therefore, the PIM engine consists of simple logic, including Multiply-Accumulate (MAC) units and a few registers. The engine fetches the operands only from its own bank; if the operands are unavailable, we need to perform explicit data copy from other banks, incurring a significant overhead in performance.

We can easily perform the element-wise computations by aligning their operands in the same bank since all the computations are independent. On the other hand, for Level-2 Basic Linear Algebra Subprograms (BLAS) (i.e., vector-matrix (VM) multiplication), we map each column of the matrix to a bank (*bank-private*) and compute with the vector shared in all banks (*bank-shared*). *We call an operand used by only one bank as a bank-private operand and an operand commonly used by all banks as a bank-shared operand.* Before the computation, we should copy the shared vector to all banks. The Level-3 BLAS (i.e., matrix-matrix (MM) multiplication) is more complicated than VM: either a multiplicand row is shared by multiplier columns or a multiplier column is shared by multiplicand rows, as depicted in Figure 1. However, most existing in-bank PIMs are designed for Level-2 BLAS, so they execute Level-3 BLAS by repeating VM multiplications by the number of rows of the multiplicand [3], [4], [5], [6], [12], [21].

The recent in-bank PIM can be categorized by whether it targets per-bank execution [4], [21] or all-bank execution [3], [5], [6], [12] depending on how many banks one DRAM command activates for the PIM execution. The per-bank execution triggers only one bank at a time, while the all-bank execution invokes all or multiple banks. The number of the commands required for a PIM kernel directly determines the PIM performance. The memory requests for the per-bank PIM execution are much higher than the all-bank one, thus delivering much lower performance [4], [21]. The all-bank PIM achieves high computation throughput by exploiting the bank-level computation parallelism and using the full internal bandwidth. However, it accompanies the following design difficulties and potential performance degradation: 1) data alignment issue from its computation granularity, 2) synchronization overhead for syncing the bank states before starting the all-bank execution, 3) modification of memory interface/controller, 4) implementation overhead such as power consumption and thermal dissipation, and 5) PIM mode switching which prevents the non-PIM request service during PIM operations [3], [5], [6], [12], [13]. Even though the per-bank PIM suffers the lower performance, it preserves the standard DRAM interface and supports non-PIM request service during the PIM execution as a standard DRAM. Both per-bank and all-bank PIMs suffer from replicating bank-shared operands to all banks, resulting in multiple copies of the same data stored in different memory addresses. Several PIMs provide a global/local buffer to avoid the replication but incur significant overhead in size, much worse, especially in the DRAM fabrication [3], [5], [6], [12].

For taking only advantages from both PIMs, this paper introduces the memory-computation decoupling architecture to achieve the ideal all-bank performance of in-DRAM PIM with standard memory interface, i.e., the per-bank execution. Also, to maximize the decoupled execution performance, we introduce a compiler analysis and code generation technique.

The decoupled architecture uses two PIM execution phases: *a memory phase* for the per-bank memory operation and *a computation phase* for the all-bank computation. At the memory phase, we read bank-private operands from a memory array and store them to the bank's PIM registers bank-by-bank. At the computation phase, we decouple each PIM engine datapath from its bank and broadcast bank-shared operands read from one bank or written from a host to all banks' engines without operating all banks' memory arrays. It allows us to compute the PIM engine in all banks in parallel with operating only one bank of the memory array and achieve the *ideal* performance of the all-bank execution while preserving the standard DRAM interface and satisfying the standard power budget. In reality, only half of all banks could perform concurrently due to power and thermal issues [5], [6]; thus, our performance would be more attractive to users than the all-bank execution in the real world. Also, since we preserve the standard interface, we can serve non-PIM requests during our computation phase, i.e., while performing the all-bank computation. As we operate only one bank array at both phases and conform to the standard interface, we implement our architecture based on the per-bank PIM [4].

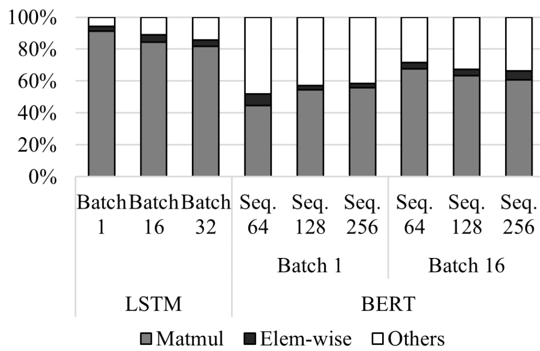

FIGURE 2. Ratio of the execution time for the matrix multiplication and element-wise operations to the total execution in the DNN applications, LSTM [1] and BERT [7].

Although there have been many studies about broadcasting the same data to computing units to reduce memory operations [3], [6], [12], [13], [24], [25], [26], we are the first to combine the broadcast with the decoupling of datapath from a memory array for realizing the all-bank PIM execution. Our PIM computation throughput is also decoupled from the internal bandwidth: The broadcast has the same effect of using the full internal bandwidth on all-bank PIMs to provide operands to all their engines at once. Therefore, our decoupled PIM's performance can become closer to the all-bank PIM when the computation phase lasts longer; thus, to achieve the highest computation throughput, we developed a compiler technique to identify the bank-private and the bank-shared operands and find out a tiling factor for maximizing the bank-private operands' reuse and the Arithmetic Logic Unit (ALU) utilization.

There is great potential to apply our decoupled approach to recent applications. Figure 2 shows that the Deep Neural Network (DNN) applications with batching spent about 67.1% on average of the total execution time for the matrix-matrix multiplication to involve bank-private and bank-shared operands as shown in Figure 1, giving the significant opportunity of the two-phase execution of our decoupled PIM. On the other hand, the element-wise operations consist of only bank-private operands, disallowing our computation phase. However, since its computation ratio was only 4.2% of the total execution time on average, the disadvantage has little effect on the overall execution time. Compared to our decoupled PIM, the all-bank PIM can achieve the performance benefit when all banks compute different data only, i.e., bank-private operands. Interestingly, the opportunity is minimal in the PIM-targeted DNN applications, and therefore, we would conclude that the all-bank PIM spends unnecessary computation resources.

We developed the PIM-emulated Field Programmable Gate Array (FPGA) platform, evaluated its performance by running microbenchmarks of Level-2/3 BLAS, a multi-batch LSTM-based Seq2Seq model, and BERT, and compared it to AMD Ryzen-5 CPU with OpenBLAS [27] and Nvidia Titan Xp GPU with cuBLAS [28]. Compared to CPU and GPU, we achieved speedups of 75.8× and 1.2× in Level-3 BLAS, 8.4× and 1.5× in LSTM-based Seq2Seq, and 3.1× and 15.5× in BERT, respectively. In addition, we outperformed the per-bank PIM by 4.7×, 4.4×, and 1.4× and reached up to 91.3%, 97.8%, and 86.6% of the ideal all-bank PIM performance in Level-3 BLAS, LSTM-based Seq2Seq, and BERT, respectively. The energy consumption of our decoupled PIM was 72.0% and 78.4% lower in Level-3 BLAS, 80.8% and 77.0% lower in LSTM-based Seq2Seq, and 98.3% and 33.5% lower in BERT than GPU and the per-bank PIM. Also, our PIM consumed only 7.4%, 0.3%, and 3.1% more energy than the ideal all-bank PIM in the applications.

Besides our remarkable performance with the standard memory interface, it should be noted that 1) since we use the standard JEDEC DRAM interface [29], including the power budget, we can easily apply our approach to commercial computing platforms. 2) Our approach allows the memory requests from non-PIM applications to be serviced during even all-bank PIM execution, thus not impacting any of their execution behavior running on a host [4], [21], [30]. 3) Finally, when we modeled the ideal all-bank PIM for the performance comparison, we assumed it could simultaneously compute all banks. In reality, only half of all banks could perform concurrently; thus, our approach would outperform currently available all-bank PIMs [5], [6].

The rest of the paper is organized as follows: Section II introduces existing in-bank PIMs and our baseline per-bank PIM [4] for this work. Section III proposes the memory-computation architecture and its compiler techniques by applying our work to Level-3 BLAS as an example. Section IV evaluates the performance, and Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

A. PER-BANK VS. ALL-BANK PIMs

The PIM architecture places the computing unit inside memory devices to use the full internal memory bandwidth. The architecture can be classified by where the computation is performed; in-cell, i.e., computing on memory cells, and in-bank, i.e., on bank peripherals.

The in-cell PIM architecture utilizes the analog properties of Non-Volatile Memory (NVM) cells such as ReRAM and MRAM to use them as both storage and computing devices [16], [17], [18], [20], [31]. However, the analog-based computation drops accuracy due to the limited precision and the error vulnerability in analog-to-digital conversion. Analog-to-Digital (ADC) and Digital-to-Analog Converter (DAC) also result in a large area overhead [3], [32]. For alleviating such problems, some in-cell PIM architectures proposed arithmetic computation in NVM and DRAM, based on the logic operations implemented upon their analog characteristics such as resistance of ReRAM and charge sharing of DRAM [14], [15], [33], [34], [35], [36], [37], [38], [39], [40], [41].

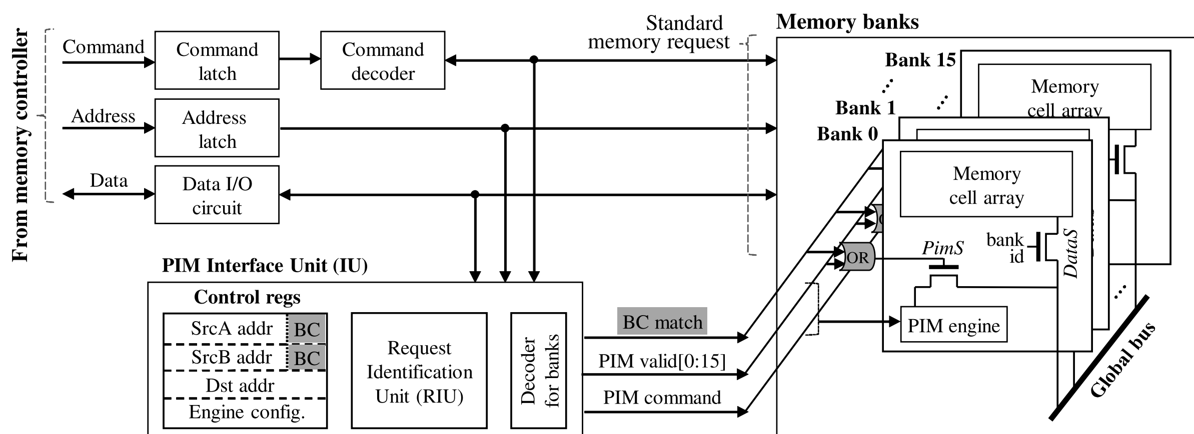

FIGURE 3. The per-bank Silent-PIM [4], the baseline architecture for our work. The graded components are added for our decoupled execution.

This paper focuses on in-bank DRAM-based PIM architectures that utilize the digital arithmetic computing units placed in bank peripherals and can be used as the main memory. The existing in-bank PIM architectures can be categorized by the execution granularity of a PIM operation: per-bank and all-bank PIMs. One DRAM request enables only one bank's PIM engine in the per-bank PIM [4], [21] and all banks' engines in the all-bank PIM [3], [5], [6], [12], [13]. Silent-PIM [4], a representative work of the per-bank PIM, preserves the standard memory behaviors and timing constraints of DRAM under the general supply voltage, complying with the power budget of a standard DRAM. Each bank is scheduled independently using the standard memory request, and the memory requests from non-PIM applications can also be serviced while executing a PIM kernel [30]. However, one DRAM command triggers a PIM operation with one burst of data per bank, resulting in a large number of memory requests to execute a kernel and lower performance than all-bank execution. Also, as only one bank can perform the PIM operation at a time, it wastes the opportunity for the bank-level computation parallelism.

The all-bank execution PIM is a compute-centric architecture that provides the highest computation throughput by leveraging the bank-level computation parallelism and fully utilizes the DRAM's internal bandwidth [3], [5], [6], [12], [13]. One DRAM request enables all or multiple banks to read source operands from DRAM cells, store the results to DRAM, or concurrently perform PIM operations. The bank-shared operands must be read from all banks by one read request, so the operands should be available and carefully aligned in all the banks before the execution [5]. Also, the simultaneously enabled bank execution imposes a power burden, resulting in exceeding the standard power budget and worsening the thermal problem [22], [23]. Therefore, the existing all-bank architectures support only a part of the banks to operate at once, sacrificing the bank-level computation parallelism to deal with the problem. For example, in [6] and [12], only four banks per channel perform the PIM operations concurrently. In [5], the PIM unit is shared among two banks, thus allowing only half of the banks to perform at once. For supporting the lockstep-style behavior of all banks, [3], [6], [12], [13] requires a customized memory controller violating the JEDEC standard, and [5], [12] requires the mode switching before and after the PIM operations. Besides, memory requests of non-PIM applications cannot be serviced during PIM operations [3], [5], [6], [12], [30].

B. SILENT-PIM: OUR BASELINE ARCHITECTURE

Figure 3 illustrates the architecture of a representative per-bank PIM named Silent-PIM [4], based on which we implemented our memory-computation decoupled architecture.

Each bank has its memory cell array and a PIM engine. The memory banks receive `Command`, `Address`, and `Data` signals from standard memory requests as conventional DRAM devices do and accept the PIM signals generated from the PIM Interface Unit (IU). Before executing a PIM kernel, a programmer stores the PIM operands' start addresses of the uncacheable physical pages and the engine configuration information in the control registers in PIM IU. The PIM Request Identification Unit (RIU) compares the input address with the operand addresses stored in the control registers to determine if the incoming memory request is the PIM command. If the addresses are matched, the `PIM valid` signal is generated and delivered to a target bank with the other signals for providing data from the bank to its PIM engine.

Silent-PIM has the 4-stage pipelined datapath for bfloat16, as shown in Figure 4, performing all the computations without violating the DRAM timing of the data burst, i.e., in 4 cycles. The first stage fetches the source operands into two vector registers, `vecA` and `vecB`. A switch (`PimS`) lies in between the global data bus and the register file bus interface of the engine. The `PIM valid` signal enables the switch to connect the global data bus to the source or the destination register for the PIM command or disconnected for typical

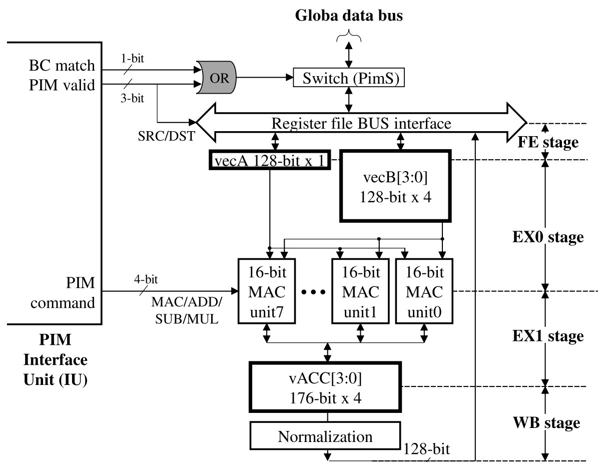

FIGURE 4. PIM engine in Silent-PIM [4]. The shaded OR gate is the only added component for our decoupled execution.

memory requests from non-PIM applications. That is, if RIU recognizes the in-flight read request to be the source of the PIM operand, the unit turns on the bank's `PimS` switch; thus, the data on the global data bus is stored in either `vecA` or `vecB`. The data is delivered through a 4-cycle burst, and `vecB` stores the whole burst data (i.e., 128-bit $\times$ 4) of the request. On the other hand, `vecA` only stores 1-cycle burst (i.e., 128-bit $\times$ 1) from the whole burst cycle-by-cycle since we could implement a small size of `vecA` due to the available space constraint in DRAM for the PIM engine design.

The second and the third stages consist of 2-stage pipelined 8-way MAC unit array that performs the PIM arithmetic operations - MAC/ADD/SUB/MUL - according to the configuration information in the control registers of PIM IU. The operation is performed whenever data is stored to `vecA`. After executing the operation, the result of the operation is stored to `vACC`. The register `vACC` consists of 22-bit registers for each element, resulting in 176-bit per 1-cycle burst, 176-bit $\times$ 4 for the whole burst data. Saving the result in 22-bit improves the accuracy by holding more fraction bits. In the last stage, the results in `vACC` are stored to the bank by a standard write request after the bfloat16 normalization when the incoming address matches the address of the destination operand in the control registers of PIM IU.

III. MEMORY-COMPUTATION DECOUPLING

This section introduces our memory-computation decoupled PIM architecture and its compiler analysis and code generation technique.

We perform the PIM execution using two phases: a memory phase and a computation phase. At the memory phase, we fetch the bank-private operands from memory arrays to PIM engines' registers in a per-bank manner. On the other hand, all banks' PIM engines execute the computation phase without accessing their memory array but using the broadcast bank-shared operand from one bank or a host. Since both phases operate the memory array of only one bank, our decoupled PIM preserves the standard DRAM interface, i.e., the standard DRAM power budget, commands, timings, and so on.

The decoupled execution outperforms the per-bank PIM in speedup and energy consumption while slightly increasing the power consumption since the PIM engines of all banks operate simultaneously in the computation phase. On the other hand, the proposed decoupled PIM shows less performance than the all-bank PIM in the memory phase. However, the total power of the decoupled PIM remains within the standard power budget, unlike the all-bank PIM. Consequently, the proposed decoupled PIM architecture becomes more attractive and acceptable than the prior all-bank PIMs when there is a higher opportunity for broadcast in applications.

A. MODIFYING THE PER-BANK PIM HARDWARE

The memory phase execution is the same as the per-bank execution of Silent-PIM, whose memory request turns on/off `PimS` and `DataS` switches of only a target bank. However, for supporting the computation phase, i.e., broadcasting bank-shared data to all banks' engines, we added one attribute to source operands and modified the decoder for the `PimS` switch from Silent-PIM, marked in grey in Figs. 3 and 4.

For the PIM engine to identify bank-shared data during the PIM execution, we added the broadcast attribute (`BC`) to source operands in the control registers of the PIM IU. A programmer provides the address of the broadcast target (i.e., the bank-shared operand) with setting the associated `BC`. The matching of the incoming request's address with the broadcast target while ignoring their bank addresses generates the `BC match` signal to notify the broadcast to all banks' engines. Therefore, we modified the decoder for the `PimS` switch of each bank by ORing `PIM valid` and `BC match` signals. We turn on all the `PimS` switches so that all banks' engines receive the broadcast (bank-shared) data from the global data bus and store them in registers. The broadcast data can either be read from a bank or provided outside DRAM, such as a host's write request. A standard memory request performs the data broadcast.

Suppose the BC attribute is unset and the address of the incoming request is matched with the source operand address while considering their bank addresses. In that case, the `PIM valid` signal is generated and delivered to only the target bank, i.e., performing as per-bank. All the memory requests turn on their target banks' `DataS` switch for accessing their data array as usual, and the PIM memory request among them also controls the `PimS` switches.

Figure 5 shows an example of how to control the switches at the decoupled execution phases. We use `vecB` for bank-private operands and `vecA` for bank-shared operands. At the *memory phase*, we turn on the `DataS` and `PimS` switches of the target bank to read the bank-private operand from its data array and store the operand to `vecB` bank-by-bank. On the other hand, we turn on the `DataS` of only one bank (Bank 0) and *all* the `PimS` switches at the *computation*

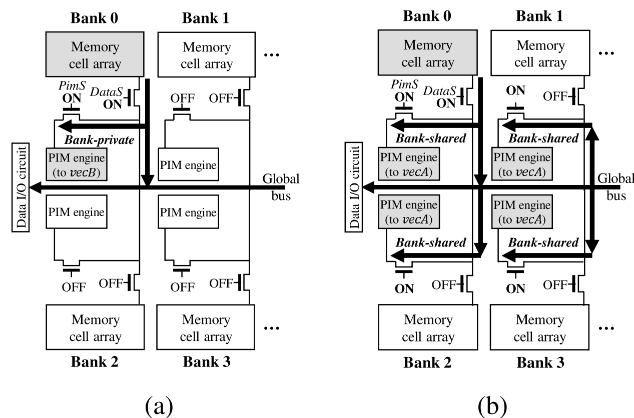

FIGURE 5. Two execution phases of our decoupled PIM. (a) A memory phase and (b) a computation phase (active components are marked in grey).

phase and store the broadcast data in vecA's, thus triggering the PIM computation for every cycle within a burst. The computation phase adheres to the standard memory power as it only operates at most one bank of memory array while using the PIM engines of all banks in parallel. We discuss the power consumption of memory arrays and PIM engines in Section IV-B4. Since vecA only stores a 1-cycle burst of a request (i.e., $\frac{1}{4}$ of a whole burst), the broadcast data is not reused within a bank. However, broadcasting the data has the same effect as reusing it across all banks since it initiates all-bank PIM execution by accessing only one memory array bank.

B. DETERMINING BANK-PRIVATE AND BANK-SHARED OPERANDS

Determining which operand to be bank-private or bank-shared directly impacts the performance. The computation phase performs the all-bank execution by broadcasting the bank-shared operand, thus taking full advantage of the bank-level computational parallelism, i.e., the same as the all-bank execution. Thus, the longer the computation phase, the higher the performance.

For maximizing the opportunity of the all-bank execution by extending the computation phase, it is essential to select a high reusable operand from the code as the bank-private. Otherwise, the computation phase cannot continue whenever requiring a new bank-private operand. Therefore, the bank-private operands stored in vecB should be reused as much as the program allows. On the other hand, the highly shared and low reusable data should be recognized as bank-shared to be broadcast. The reusability degree of the broadcast bank-shared operand is the same as the number of banks. Our compiler uses the conventional iteration space analysis and a code tiling from our cost model for identifying the bank-private and the bank-shared operands.

Consider the Level-3 BLAS code and its dependence table in Figure 6(a). The values 0 and 1 in the table represent

```
for(int i = 0; i < I; i++)
 for(int j = 0; j < J; j++)
  for(int k = 0; k < K; k++)
   Acc[i][j] +=
    MatA[i][k] * MatB[k][j];
```

	i	j	k
MatA	1	**0**	1
MatB	**0**	1	1

(a)

```
for(int b = 0; b < B; b++)
 for(int r = 0; r < R; r++)
  for(int c = 0; c < C; c++)
   for(int m = 0; m < M; m++)
    for(int n = 0; n < N; n++)
     out[b][m][r][c] +=
         inp[b][n][r][c] *
                    wgt[m][n];
```

	b	r	c	m	n
inp	1	1	1	**0**	1
wgt	0	0	**0**	1	1

(b)

FIGURE 6. Dependence of matrices to loop dimensions. (a) Level-3 BLAS code and its dependence of matrices to dimensions. (b) Convolution code and its dependence of matrices to dimensions.

independence and dependence on each dimension of the matrices, respectively. For example, *MatA* and *MatB* are independent of the *j* and the *i* dimensions, respectively. Each matrix is independent of one of the three dimensions and reused at its lower iteration space. That is, *MatA* is reused (i.e., shared) within the *j* dimension, and *MatB* is shared within the *i* dimension. We select *MatB*, which is reused in a larger iteration space, as a bank-private operand since we can reuse the matrix maximally within the *i* dimension, i.e., repeatedly using $K \times J$ elements. Also, we choose *MatA*, which is reused in a smaller iteration space, as a bank-shared operand by considering the operand reuse across the banks for the *j* dimension. If the *i* and *j* dimensions are interchanged, *MatB* becomes the bank-shared since the *i* dimension will be the lower dimension, and *MatA* turns into the bank-private operand.

We can consider a convolution algorithm in Figure 6(b) in the same way and determine *inp* as bank-shared and *wgt* as bank-private operands.

C. MAKING THE COMPUTATION PHASE LONGER: REGISTER TILING

We fetch the bank-private operands and store them into the PIM engine registers at the memory phase in a per-bank manner. The higher the reuse of bank-private operands in the registers, the longer the computation phase, and the higher the performance due to the decoupled all-bank execution. For maximizing the reusability, we apply the loop tiling and develop a cost model to derive a tiling factor for the code generation. Our compiler technique is different from the conventional compiler's approach to employ the tiling for cache hierarchy. Our PIM does not include a cache, so we match the tiling to two PIM resources: registers for maximizing the bank-private operand reuse and an ALU width for maximizing the computation utilization.

1) TILING WITH COST MODEL

Figure 7 shows a loop tiled code of Level-3 BLAS. The ranges of the dimensions *i*, *j*, and *k* are I, J, and K, and the tiling factors of each dimension are *p*, *r*, and *q*, respectively. As

```
1   bfloat16 MatA[I][K], MatB[K][J], Acc[I][J];
2
3   for(int io = 0; io < I; io += p)
4     for(int jo = 0; jo < J; jo += r){
5       Load_tile_Acc();
6       for(int ko = 0; ko < K; ko += q){
7         Load_tile_MatB();
8         for(int ii = 0; ii < p; ii += 1){
9           for(int ji = 0; ji < r; ji += 1)
10            for(int ki = 0; ki < q; ki += 1)
11              Acc[io+ii][jo+ji] +=
12                  MatA[io+ii][ko+ki] *
13                        MatB[ko+ki][jo+ji];
14        }
15      }
16    Store_tile_Acc();
17    }
```

FIGURE 7. Loop tiled code of Level-3 BLAS.

$Acc[io+ii][jo+ji]$ is independent of the dimension ko, its tiles are loaded and stored in the jo dimension (Lines 5 and 15). The tiles of *MatB* are loaded in the ko dimension (Line 7). To reuse *MatB* and *Acc*, the tile sizes of *MatB* and *Acc* should not exceed the register `vecB` and `vAcc`, respectively, as represented by Equation (1).

$$\begin{aligned} TileSize_{MatB} &= q \times r \leq Size_{vecB} \\ TileSize_{Acc} &= p \times r \leq Size_{vAcc} \end{aligned} \tag{1}$$

From the decision that *MatA* is to be bank-shared and *MatB* is to be the bank-private, the load/store cost of the tiled code is defined by Equations (2), (3), and (4) in a unit of a matrix element. Minimizing the total cost, i.e., the sum of all the equations, implies maximizing the reusability, thus maximizing the all-bank execution opportunity.

Since broadcasting the *MatA* has the same effect as reusing it for all banks by performing the all-bank PIM execution while accessing only one memory array, the cost of the bank-shared operand *MatA* becomes the number of total load instructions divided by the number of banks. Since the load instruction is executed in the inner-most loop, its total number is $I \times J \times K$. We can measure the reusability by the cost of the bank-private operand *MatB*, i.e., the number of load instructions for the operand divided by the number of reuses of an element. Since an element of *MatB* is reused in the i dimension for p times within the intra-tile, the number of reuses is p. Each tile of *Acc* is loaded and stored in the jo dimension. Then, the cost is calculated by multiplying the tile size and the number of the tile load and stores. From Equations (1)∼(4) where $Size_{vecB} = Size_{vAcc} = 32$ and the number of banks is 16, we acquire the optimal tiling factor as $(p, q, r) = (32, 1, 1)$, which minimizes the total cost.

$$\begin{aligned} Cost_{MatA} &= \frac{\#\text{ of LD}}{\#\text{ of banks}} \\ &= \left(\frac{I}{p} \times \frac{J}{r} \times \frac{K}{q} \times p \times r \times q\right) \times \frac{1}{\#\text{ of banks}} \\ &= \frac{I \times J \times K}{\#\text{ of banks}} \end{aligned} \tag{2}$$

$$\begin{aligned} Cost_{MatB} &= \frac{\#\text{ of LD}}{\#\text{ of reuse per elem}} \\ &= \left(\frac{I}{p} \times \frac{J}{r} \times \frac{K}{q} \times p \times r \times q\right) \times \frac{1}{p} \\ &= \frac{I \times J \times K}{p} \end{aligned} \tag{3}$$

$$\begin{aligned} Cost_{Acc} &= TileSize_{Acc} \times \#\text{ of tile LD/ST} \\ &= (p \times r) \times \left(\frac{I}{p} \times \frac{J}{r} \times 2\right) \\ &= I \times J \times 2 \end{aligned} \tag{4}$$

When the i and j dimensions are interchanged, *MatA* and *MatB* become the bank-private and the bank-shared, respectively. In that case, their costs become $Cost_{MatA} = \frac{I \times J \times K}{r}$ and $Cost_{MatB} = \frac{I \times J \times K}{\#ofbanks}$, resulting in the optimal tiling factor of $(p, q, r) = (1, 1, 32)$.

2) ALL-BANK EXECUTION TILE BY TILE

We found an optimal tiling factor of $(p = 32, q = 1, r = 1)$ to maximize the all-bank execution opportunity by reusing a bank-private operand as much as possible, i.e., avoiding frequent reloading of the bank-private operand. Therefore, the optimal tiling factor prefers to perform $(p \times q) \times (q \times r) = (p \times r)$, i.e., $(32 \times 1) \times (1 \times 1) = (32 \times 1)$.

Both tile sizes of *MatA* and *Acc* are 32×1, and that of *MatB* is 1×1. However, since the DRAM access granularity is 64B (i.e., 32 elements), we regard 32 elements as a tile for *MatB* for one bank; thus, the optimal tiling factor becomes (32, 32, 1). Also, we concurrently execute the 16 banks by the broadcast in the j dimension; thus, the optimal tiling factor finally becomes (32, 32, 16). Therefore, we store the interleaved 32 ($= q$) columns of *MatB* and *Acc* across 16 ($= r$) banks and broadcast 32 ($= p$) elements of *MatA* to all banks 32 ($= q$) times. *We call this a register-sized window in the rest of the paper.*

Figure 8(a) illustrates the register-sized window matrix multiplication at bank 0, i.e., $(32 \times 32) \times (32 \times 16)$, and Figure 8(b) shows the timeline of DRAM commands executing the multiplication. For the timeline, we assume that all matrices are stored in the same row of a DRAM so that only one activation is required per bank. Also, standard memory requests perform all reads, writes, and broadcasts with the PIM computation. Each bank i multiplies pairs of $(a_{0:31,0}, b_{0,i})$, $(a_{0:31,1}, b_{1,i})$, $\cdots$, $(a_{0:31,31}, b_{31,i})$ and accumulates the multiplication results one-by-one to calculate $c_{0:31,i}$.

The execution performs the following phases:

- *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`.
- *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` ($b_{0,j}$ where j indicates the bank number) is multiplied with

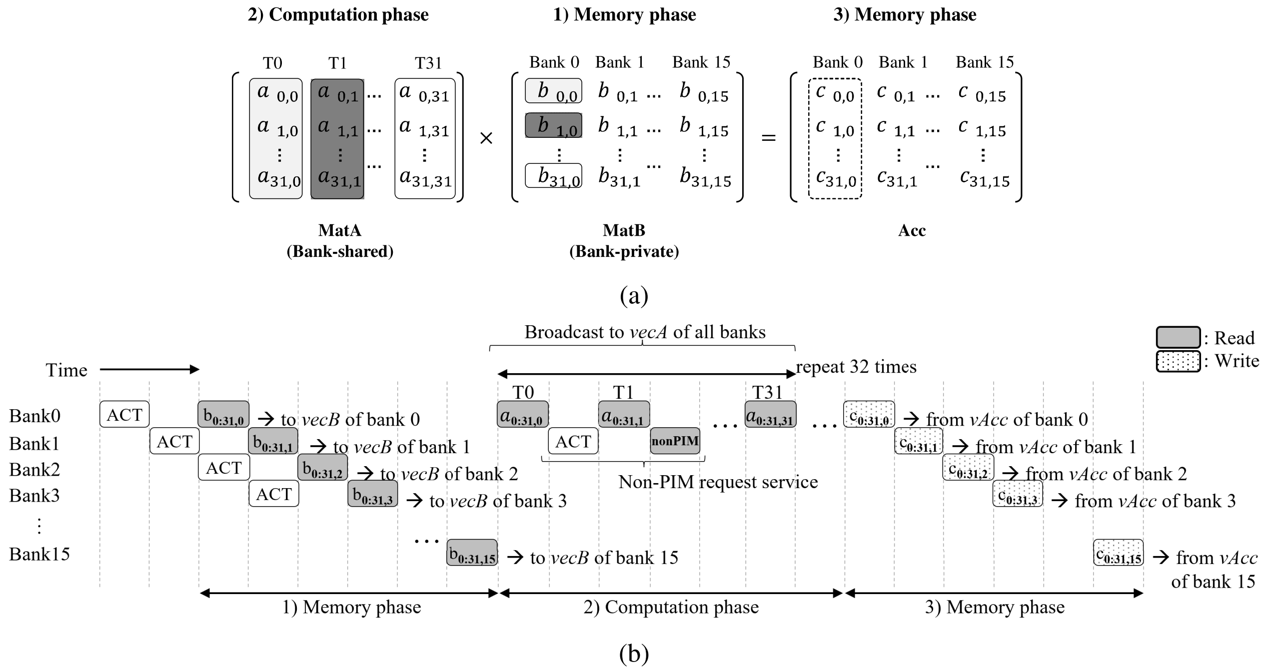

FIGURE 8. (a) Matrix multiplication for bank 0 and (b) DRAM command timeline of the (32 × 32) × (32 × 16) Level-3 BLAS, i.e., in one register-sized window. The timing scales are simplified for a clear explanation.

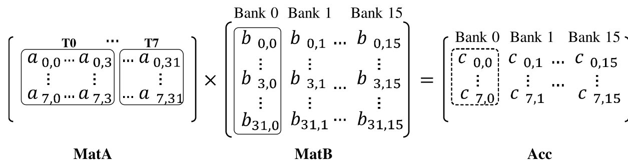

FIGURE 9. Optimized tiles of the (8 × 32) × (32 × 16) Level-3 BLAS.

$a_{0:31,0}$ 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*, i.e., in the case of $K > 32$.

- *Memory phase for storing the results into a memory*: Each bank stores a 64-byte `vAcc` to the memory cell array.

D. MAXIMIZING THE ALU UTILIZATION: ALU-WIDTH TILING

When I of *MatA* is smaller than 32, ALUs become underutilized, resulting in a waste of time and power. For example, when $I = 8$, MAC operations only with $a_{0:7,i}$ are performed during Ti where $i = 0, 1, \ldots, 31$ of Figure 8(b), thus wasting 3 cycles of every burst. Therefore, we further optimize the PIM execution by employing the second-level tiling to fit an ALU width. We tile the matrix multiplication by $(32 \times 32) \times (32 \times 16)$ in the first step, as described in the previous subsection, and then tile each (32×32) tile of *MatA* by the ALU-width size, 8.

Figure 9 depicts the optimized tiles processed in bank 0. Our 8-way ALUs process 8 elements of *MatA* simultaneously. Therefore, we provide 8×4-sized tiles ($a_{0:7,4\times i:4\times i+3}$) of *MatA* through $T0$ to $T7$. Each burst calculates 8 partial sums instead of 32. For example, at $T0$, $b_{0:3,0}$ is multiplied with $a_{0:7,0:3}$ to generate the partial sum of $c_{0:7,0}$; ($a_{0:7,0}$, $b_{0,0}$), ($a_{0:7,1}$, $b_{1,0}$), ($a_{0:7,2}$, $b_{2,0}$) and ($a_{0:7,3}$, $b_{3,0}$) at each cycle. In this way, when $I = 8$, we read *MatA* 8 times to finish the task instead of 32 times.

E. CORRECTNESS OF THE PIM COMPUTATION

To guarantee the correctness of the PIM computation regarding Figure 8, the following conditions should be satisfied. 1) Each phase should start after the previous phase is finished, and 2) at the computation phase, a PIM engine should correctly determine which element of `vecB` is multiplied with the in-flight broadcast data.

When each phase is not separated, e.g., the computation phase starts before finishing the memory phase, some banks start the PIM computation with its `vecB` filled with garbage or empty, thus violating the correctness. We prevent such situations by offloading the PIM requests using Direct Memory Access (DMA). Since our PIM architecture conforms to

the standard DRAM interface, we offload the PIM requests using a conventional DMA engine. Each DMA transaction invokes each phase, and the DMA engine requests the next transaction only after the previous transaction is finished. Therefore, we ensure that each phase starts after the earlier phase is completed.

However, the memory requests within a DMA transaction can be reordered by the memory controller scheduling. The scheduling does not affect the correctness at the memory phase since each bank receives only one request to fill `vecB` and filling `vecB` of a bank is independent of the other banks. On the other hand, at the computation phase, a PIM engine should correctly determine which element of `vecB` to be multiplied with the in-flight broadcast data despite the scheduling. For example in Figure 8(b), when the memory controller broadcasts the sixteenth column $a_{0:31,16}$ at $T0$, each bank should multiply the sixteenth element $b_{16,j}$ with the broadcast data, where j indicates the bank number. That is, the column number of $MatA$ becomes the `vecB` index.

Therefore, we determine the `vecB` index of the ALU input by using the physical address of the in-flight broadcast data. We assume that the columns of $MatA$ are stored in a contiguous address of 2KB where $Size_{MatA} = 32 \times 32 \times 2B = 2KB$. As each column size is the DRAM access granularity 64B (32 elements) occupying the 6-bit LSB of the physical address $PA_{MatA}[5:0]$, the column number is determined by $PA_{MatA}[10:6]$. Therefore, we determine the `vecB` index by $PA_{MatA}[10:6]$ of the in-flight broadcast data and extend `PIM valid` to include the information. Consequently, we guarantee the correctness of the PIM computation despite the memory controller scheduling.

F. NON-PIM REQUEST SERVICE DURING THE COMPUTATION PHASE

Our decoupled PIM adopts the standard memory request for PIM operations, so a memory controller does not differentiate between PIM and non-PIM memory requests and schedules them together. As shown in Figure 8(b), we can service non-PIM requests from other processes in the middle of the computation phase since all the `DataS` and `PimS` switches are turned off at the end of every RD/WR command.

The figure shows a situation where a read request for bank 1 is serviced during the computation phase. Right after $T0$, a memory controller activates the requested row in bank 1. For the PIM computation at $T1$, bank 0's `DataS` and all banks' `PimS` are turned on so that the bank-shared operand is delivered to all PIM engines. At the end of $T1$, all the switches associated with the previous PIM request are turned off, and only the `DataS` switch of bank 1 is turned on for the non-PIM read. Since all `PimS` are turned off, any non-PIM memory operation does not corrupt the PIM engine states, and the engines continue the PIM operation whenever the next PIM request arrives. Any requests to other banks, including bank 0, can be handled in the same way by a conventional memory controller.

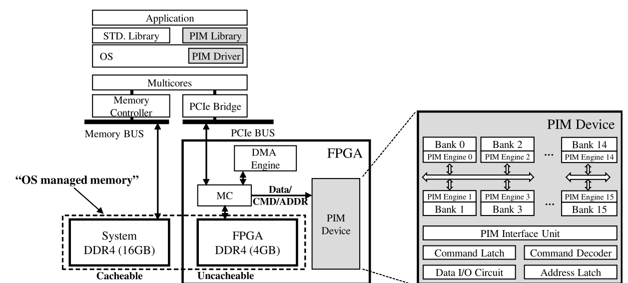

FIGURE 10. Overall architecture of the experimental platform supporting PIM emulation [4].

IV. PERFORMANCE EVALUATION

A. EXPERIMENTAL ENVIRONMENT

Figure 10 illustrates our experimental platform for the PIM emulation using an FPGA board (Xilinx Virtex UltraScale board XCVU190). The platform is developed based on Silent-PIM [4]. We implemented the software layers on the host platform. The FPGA board is connected to the host platform through PCIe. The two DDR4 [29] placed on the host platform and the FPGA compose the OS-managed main memory. Since we used the FPGA DDR4 for PIM, we configured it uncacheable. The memory controller (MC) of the FPGA was regarded to be equivalent to the host memory controller. We verified all operations at the system level.

Since our PIM architecture complies with the standard DRAM interface, we did not modify the Xilinx DDR4 memory controller IP, and a conventional DMA invoked the PIM requests to the FPGA-based PIM [4]. The number of banks in a PIM device is 16, and each bank has 8-way MAC units. As discussed in Section II, the data is fetched by 128-bit $\times$ 4 cycles. The proposed architecture can be considered as one die of 3D-stacked memory. When the memory controller captures a PIM request by scanning the requested address, the PIM device module placed between the memory controller and the DRAM emulates the DRAM access, broadcast, and PIM operations while obeying the DDR4 timings mounted on the FPGA.

While the host system memory controller operated at 1,200MHz, the Xilinx memory controller operated at 156.25MHz. The DRAM execution time is determined by the timing constraints (e.g., tCL-tRCD-tRP-tRAS: 17-17-17-39 tCK). We adjusted the timing constraints of the Xilinx memory controller equal to the host memory controller *in cycles* to make the execution time directly proportional to the operating frequency [4]. Therefore, we estimated our FPGA-based PIM's performance by multiplying the execution time with the frequency difference.

Also, to estimate the performance of the *ideal* all-bank PIM, we modified the Xilinx memory controller to simulate the all-bank execution behavior (i.e., one command operates all banks at once). *We assumed that all banks operate by one DRAM command; thus, we named "ideal".*

We ran microbenchmarks of Level-2/3 BLAS, a multi-batch LSTM-based Seq2Seq model [42], and BERT [7] on

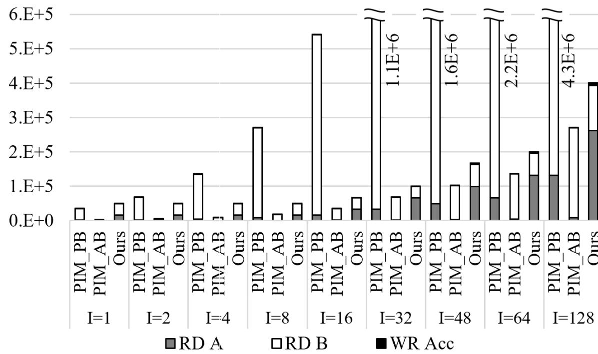

FIGURE 11. The number of memory requests in PIMs: PIM_PB, PIM_AB, and our decoupled PIM at (I × 512) × (512 × 2048) Level-2/3 BLAS.

CPU, GPU, and the PIM-emulated FPGA. We used AMD Ryzen 7 1700 (3,000MHz) for CPU and NVIDIA TITAN Xp (1,582MHz) for GPU. We used OpenBLAS [27] on CPU and cuBLAS [28] on GPU for the BLAS algorithms. We executed the inference of the LSTM-based Seq2Seq model on PyTorch v1.5.0 [43] using `de_core_news_sm` and `en_core_web_sm` language models from Spacy v2.3.0 [44]. We ran the BERT model on ONNX Runtime v1.6.0 [45] with MKL [46] and MLAS [47] instead of OpenBLAS unsupported by ONNX Runtime. We determined the matrix sizes of the BLAS algorithms to (I × 512) × (512 × 2048) based on the matrix sizes used in the Seq2Seq model.

We compared the performance of our decoupled PIM using the 8×4 tile for *MatA* to CPU serial execution (CPU_S), CPU parallel execution using OpenMP (CPU_P), GPU, per-bank PIM (PIM_PB), and ideal all-bank PIM (PIM_AB). OpenMP utilizes 16 logical CPU cores, the same as the number of banks of our PIM device. We also analyzed the performance impact of the ALU-width tiling on our decoupled PIM in Section IV-D.

Both PIM_PB and PIM_AB require *MatA* to be copied to all banks. The average time for the copy accounted for 11.2% of the total execution time. However, we excluded the overhead and measured only the PIM execution time of PIM_PB and PIM_AB since data may be in cache or memory in PIM's execution environment, so including the copy overhead could lead the reader to erroneous conclusions. That is, we evaluated the performance of our decoupled PIM conservatively.

B. LEVEL-2/3 BLAS

1) MEMORY REQUESTS

The number of memory requests in the PIM execution determines the execution time because the requests trigger the computation and the operand load/store. Therefore, we compare them using Level-2/3 BLAS, (I × 512) × (512 × 2048), and Figure 11 shows the result.

PIM_PB PIM fully reuses *MatA*, i.e., reads only once, but the decoupled PIM reads *MatA* by $J/\#ofbanks$ times; thus, the memory requests of `RD A` of PIM_PB are lower than those of our decoupled PIM. Both PIM_PB and PIM_AB repeat the VM multiplications for I times, resulting in the proportional increase in the memory requests of `RD B`. The decoupled PIM maximizes the reuse of `RD B`, resulting in the `RD B` requests much lower than PIM_PB and even lower than PIM_AB. PIM_AB operates 16 banks by one command, and thus the memory requests for `RD A` and `RD B` are $\frac{1}{16}$ of PIM_PB.

Although we applied the ALU-width tiling, we underutilize the 8-way ALUs for $I < 8$, thus the number of memory requests on our PIM for $I < 8$ is the same as $I = 8$. Therefore, the memory requests increase at every I multiple of 8. When I is a multiple of 32, the proposed PIM requires only 9.3% of memory requests of PIM_PB due to maximizing the reuse of the bank-private operands and sharing the bank-shared operands by broadcast.

The number of RD A requests on our approach is the same as RD B on PIM_AB when I is a multiple of 32 since both the requests trigger the all-bank computations and their number of computations are the same. However, as our decoupled PIM uses the memory phase for the bank-private operands (i.e., RD B) by per-bank, it needs 48% more total memory requests than PIM_AB.

2) EXECUTION TIME AND SPEEDUP

Figure 12 illustrates the execution time in a log scale and the speedup normalized to CPU_S running Level-2/3 BLAS algorithms on each platform by varying I. The performance was measured assuming that all the matrices are stored in main memory, i.e., not yet brought into any cache at the start.

The execution time of CPU_S and CPU_P grew slowly when I was small because of the data reuse in a cache, but they became proportional to I as the data size increased. The speedup of CPU_P using 16 logical cores increased slightly from 4.5× at $I = 1$ to 5.8× at $I = 32$ and slightly degraded as I increases due to cache misses. GPU spent over 90% of the time for copying the input/output data to/from the device; therefore, its execution time was longer than CPU_P when $I < 8$. However, the execution time hardly increased thanks to the massive parallelism supported by its numerous stream multiprocessors, and its speedup continuously grew as I increased, up to 65.9×.

Since it was observed in [4] that using DMA as the PIM offloading engine and applying both DMA and DRAM-friendly data layout for PIM operands improves performance, we adopted the same approach for PIM_PB, PIM_AB, and our decoupled PIM. Such an approach allowed the performance of all PIM platforms to outperform CPU in all cases despite the relatively high data reuse in batching. PIM_PB and PIM_AB repeat the VM multiplication for I times without exploiting the reuse opportunities, and their number of memory requests determines the execution time, as discussed in the previous section. They demonstrated the highest speedup of 37.1× and 169.1× at $I = 1$ and an almost constant speedup of 16× and 86× due to CPU_S's cache effect at larger I's. Although PIM_AB demonstrates

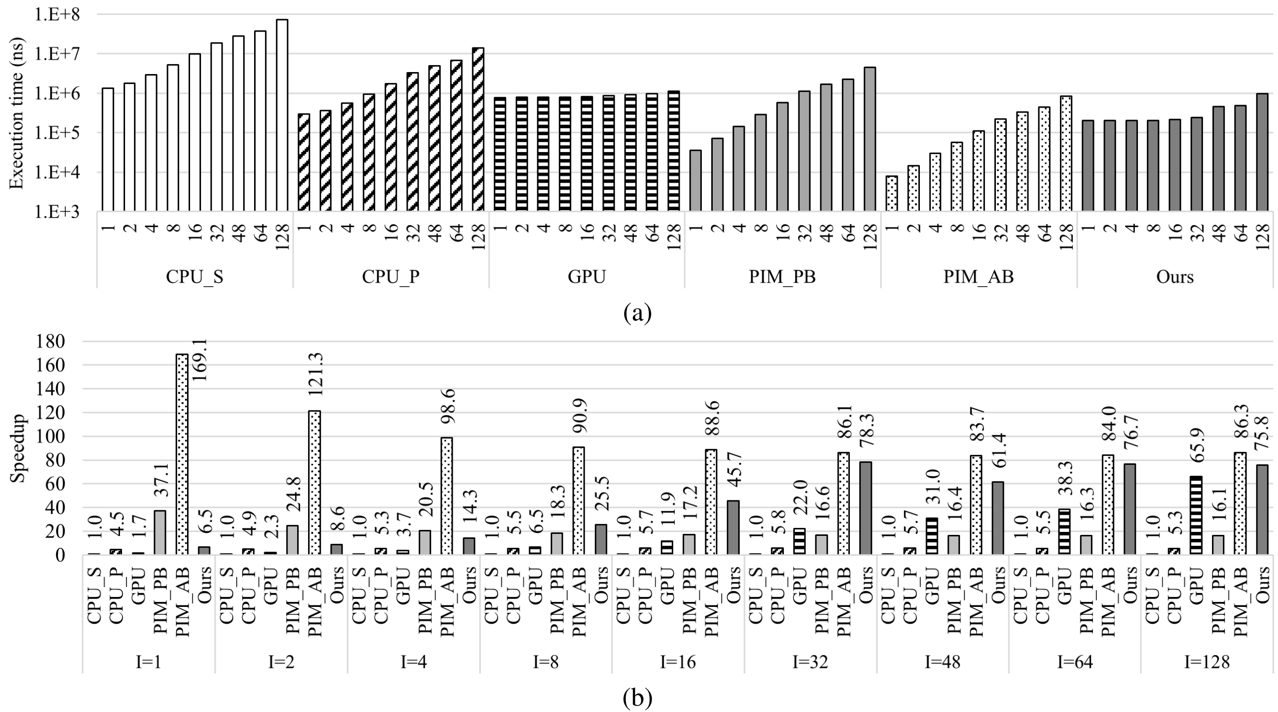

FIGURE 12. **(a) The execution time (in a log scale) and (b) the speedup of CPU_P, GPU, PIM_PB, PIM_AB, and our decoupled PIM with respect to CPU_S running Level-2/3 BLAS.**

16× computation parallelism than PIM_PB, they showed less speedup difference due to their PIM offloading overhead. The reason was that about 10% of PIM_PB's execution time was spent on the DMA interface overhead to offload the PIM execution; therefore, only the rest was accelerated by the all-bank execution.

Our decoupled PIM effectively utilizes the highly reusable data and broadcasts the shared data across banks through memory-computation decoupling and eventually realizes the all-bank execution of PIM. As discussed in the previous section, we underutilize the 8-way ALUs for $I < 8$; thus, the execution time of $I < 8$ was the same as $I = 8$ and increased at every I multiple of 8. The speedup at $I < 8$ increased as I grows since the execution time of CPU_S increased but showed lower speedup than PIM_PB because of the under-utilization. At $I = 8$ and $I = 16$, we fully utilized the ALU width but poorly reused the bank-private operands. At I multiple of 32, we reached the highest performance over 75×, as we fully utilized the ALUs and reused the bank-private operands at best.

Our PIM performance outperformed PIM_PB by 4.7× and reached up to 91.4% of PIM_AB performance. *It should be noticed that the conventional all-bank PIMs [5], [6], [12] would provide about half of the modeled PIM_AB performance since they activate up to half of all banks due to the power and thermal issues.* In all the cases, GPU performed worse than our decoupled PIM since it spent a substantial time copying the data to/from the device, whereas ours did not require any memory copies.

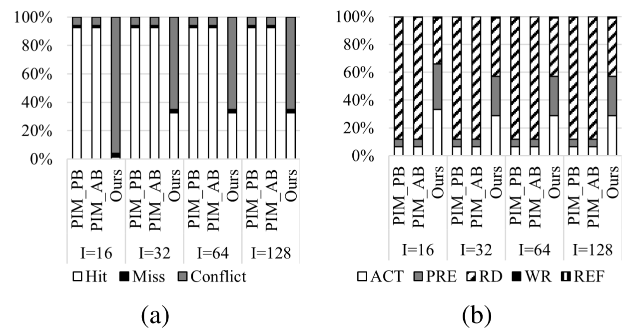

FIGURE 13. **(a) The breakdown of row buffer hit/miss/conflict ratio and (b) the breakdown of DRAM commands of PIM_PB, PIM_AB, and our decoupled PIM executing Level-3 BLAS, (I × 512) × (512 × 2048).**

3) DRAM BEHAVIOR

Figure 13 illustrates the breakdown of the row buffer hit/miss/conflict and DRAM commands of PIM_PB, PIM_AB, and our decoupled PIM. We implemented the performance counter inside the Xilinx memory controller for profiling the DRAM behaviors.

PIM_PB and PIM_AB read *MatA* once and perform MAC by 32 consecutive RDs for *MatB* in all 16 banks, resulting in $32 \times 16 = 512$ row buffer hits. Their DRAM commands are mainly RDs since they read *MatB* for I times, i.e., repeating the VM multiplication. Also, as the RDs rarely encounter row conflicts, the ratio of ACT/PRE is low.

At $I = 16$, with the register-sized window of (16, 32, 16), our decoupled PIM reads 16 tiles of *MatB* bank-by-bank at the memory phase and then reads 16 tiles of *MatA* from banks

at the computation phase, as described in Section III-D. Since the two matrices reside in different DRAM rows, reading *MatB* and *MatA* always encounter row conflicts, resulting in close to 100% row conflicts and a high ACT/PRE ratio as illustrated in Figure 13.

When I is a multiple of 32, the register-sized window becomes (32, 32, 16). 16 banks sequentially read a tile of *MatB* at the memory phase and perform MAC by broadcasting 32 tiles of *MatA* at the computation phase; thus, encounters the 16 row conflicts for *MatB* after 32 RDs for *MatA*. As a result, for computing each tile, 16 row conflicts occur at the memory phase, and 16 conflicts and 16 row hits at the computation phases.

4) POWER AND ENERGY CONSUMPTION

We evaluated the average power and energy consumption of each platform in Level-2/3 BLAS, $(I \times 512) \times (512 \times 2048)$. The power of each platform was estimated by aggregating the power of the components as follows: CPU (CPU_{pow} + $DRAM_{pow}$), GPU (GPU_{pow} + CPU_{pow} + $DRAM_{pow}$), and PIM (PIM $engine_{pow}$ + CPU_{pow} + $DRAM_{pow}$). As we assume our PIM architecture is part of the system memory, we did not consider the power consumption of the FPGA.

The CPU power was measured using the Running Average Power Limit (RAPL) interface, and the GPU power was measured using the Nvidia System Management Interface [48], [49]. The DRAM power was estimated using the DRAM-Power tool since measuring the DRAM power through RAPL on the AMD processor was not supported [50]. For measuring the DRAM power of CPU and GPU platforms, we assumed that the data is read from the main memory only once, and the following access always hits in the cache. On the other hand, to measure the DRAM power of PIM platforms, we extracted the address trace of the actual memory accesses from our PIM kernel. Also, we implemented the PIM engine in Verilog with 65nm PDK under the worst case, which has similar characteristics to the current DRAM process [4], [21], [51], [52], [53], and estimated the power consumption using the Synopsys Design Compiler. The design satisfied the DRAM internal frequency of 800MHz and the available space near each bank of about 40,000μm^2. The power of the PIM engines was at 0.03W by the logic synthesis, the same as Silent-PIM [4] due to little additional logic for this work.

The total power consumption of CPU_S, CPU_P, and GPU was 23.3W, 57.0W, and 83.4W, respectively. The DRAM power on those platforms was under 1W. All the PIMs used the same CPU, so the same power of 23.4W and the average DRAM power on PIM_PB, PIM_AB, and ours were 3.4W, 4.1W, and 3.6W. The different tile sizes of our decoupled PIM did not affect the power consumption. PIM_AB showed the highest power consumption since it performs the same amount of operations as PIM_PB in a shorter time. Ours consumed about 6% more power than PIM_PB since we encounter more row buffer conflicts as discussed in Section IV-B3, thus resulting in increased activation and precharge power.

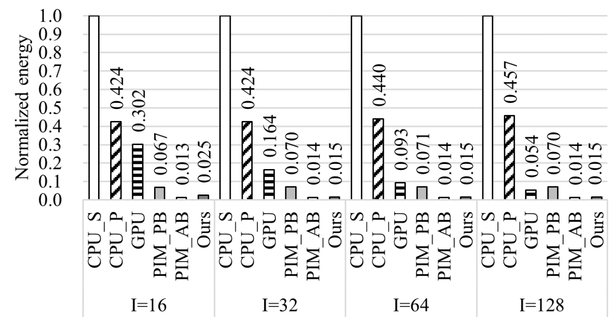

FIGURE 14. The normalized energy consumption of CPU_S, CPU_P, GPU, PIM_PB, PIM_AB, and our decoupled PIM on Level-3 BLAS, (I × 512) × (512 × 2048).

We compared the PIM power consumption with the conventional DDR4 peak power consumed by back-to-back RDs; 5.95W [12]. PIM_PB obeys the standard DRAM constraints as it operates at most one bank. Our PIM also adheres to the standard memory power as it operates in a per-bank manner in the memory phase, and also operates at most *one* bank of memory array to broadcast the bank-shared operand to the PIM engines of all banks in the computation phase. Therefore, PIM_PB and our PIM's worst-case peak powers remain close to 5.95W considering 0.03W of the engine power, i.e., 5.98W at the worst case. PIM_AB consumed a peak power of 21.58W when performing RDs for all 16 banks, far exceeding the conventional peak power. Only four banks could perform the computations simultaneously within the conventional peak power [6], [12]. In [5], the authors reduced the power consumption by limiting the concurrently operating banks to half and avoiding the data transfer to external I/O at all-bank PIM mode. Their back-to-back RDs consumed 105.4% of the normal HBM2 power [54].

Figure 14 illustrates the energy consumption normalized to CPU_S in a log scale. The normalized energy consumption of CPU_P and PIM_PB did not vary much by the I size. The normalized energy consumption of GPU became less as I increased but always worse than our decoupled PIM. At I = 128, the energy consumption of GPU was 94.6% and 33.0% less than CPU_S and PIM_PB, respectively. The energy of our PIM was less consumed than CPU, GPU, and PIM_PB by 98.5%, 72.0%, and 78.4%, respectively. PIM_AB showed the lowest energy consumption among all platforms due to the fastest execution time, and our decoupled PIM consumed only 7.4% higher than PIM_AB.

C. APPLICATIONS: MULTI-BATCH LSTM-BASED Seq2Seq AND BERT

1) EXECUTION TIME AND SPEEDUP

The execution time and the speedup of the multi-batch LSTM-based Seq2Seq model to process 1000 input data on each platform are depicted in Figure 15. The larger batch size implies a lower framework overhead for the python to C++ interface and a higher opportunity for weight reuse since the

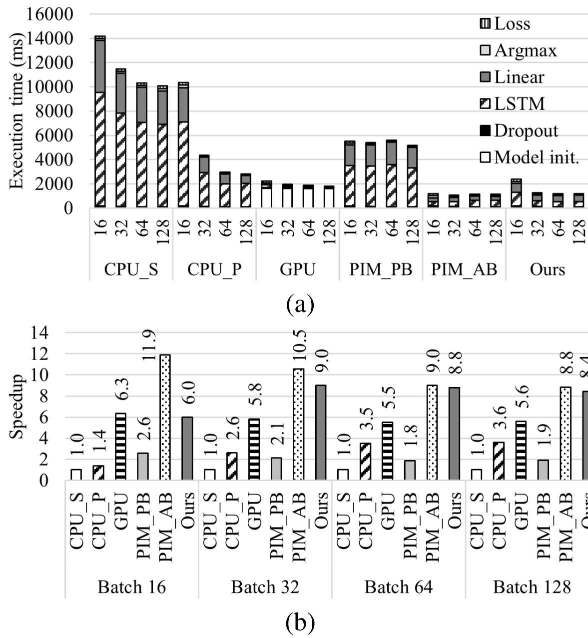

FIGURE 15. (a) The execution time breakdown and (b) the speedup of CPU_P, GPU, PIM_PB, PIM_AB, and our decoupled PIM with respect to CPU_S on the multi-batch LSTM-based Seq2Seq model.

batch size indicates I in the matrix operations of the model. Therefore, the execution time of CPU_S decreased as the batch size increased.

CPU_P showed higher speedup for larger batch size due to the weight reuse and the parallelism exploitation, reaching 3.6× at batch size 128. The model initialization that consists of memory copy from system memory to the device took most of the GPU execution time. The speedup was from 6.3× to 5.5×, and it was highest at batch size 16 due to the lowest performance of CPU_S. The execution time of PIM_PB and PIM_AB remains unchanged by the batch size since they execute Level-3 BLAS by repeating the VM multiplication for the input data, i.e., 1000 times, without reusing the weight matrices. Thus, their speedup decreased for larger batch sizes to 1.9× and 8.8×, respectively, due to the higher performance of CPU_S.

Our PIM finds lower reuse opportunities of the bank-private operands at batch size 16, and it fully reuses them at batch sizes multiple of 32. Therefore, its execution time for batch size 16 was higher than the other batch sizes, and the execution time for batch sizes multiple of 32 was the same since the input data size was fixed to 1000. The speedup was 9.0× at batch size 32 and decreased to 8.4× at 128. As a result, the PIM_AB and ours at batch size multiple of 32 outperformed GPU. Since the execution time ratio of the MM multiplication to the total execution on the CPU_S ranged 84.2∼85.5% where our PIM took advantage of the decoupled execution, our PIM outperformed PIM_PB by 4.3× and reached 94.3% performance of the PIM_AB.

Figure 16 shows the execution time breakdown and speedup of BERT with sequence lengths 8, 16, and 32 on

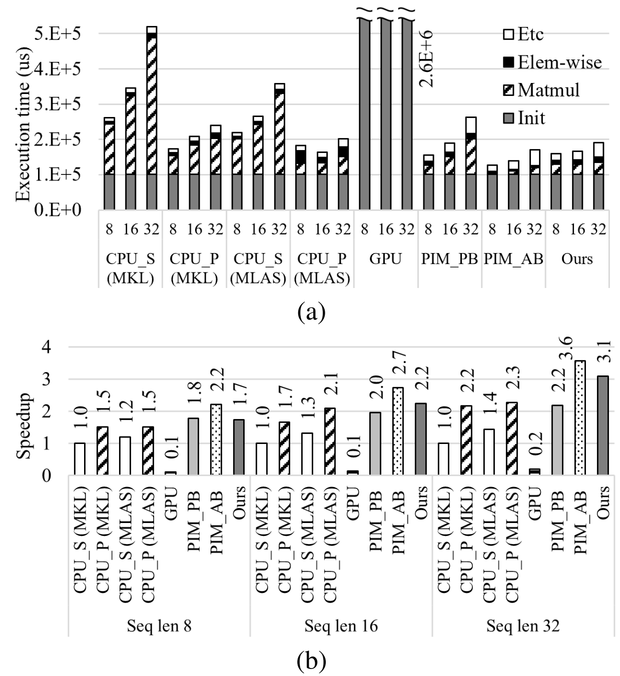

FIGURE 16. (a) The execution time breakdown and (b) the speedup of CPU_P, GPU, PIM_PB, PIM_AB, and our decoupled PIM with respect to CPU_S (MKL) on the BERT model.

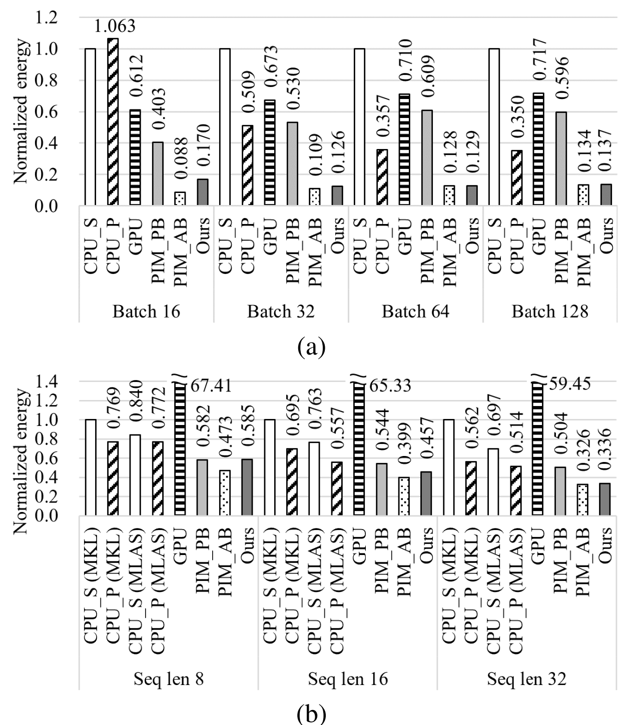

FIGURE 17. The normalized energy consumption of CPU_P, GPU, PIM_PB, PIM_AB, and our decoupled PIM to CPU_S on (a) multi-batch LSTM-based Seq2Seq and (b) BERT.

each platform. The sequence length indicates the I in the matrix operations. The execution time of both CPU_S (MKL, MLAS) increased as the sequence length grew since the

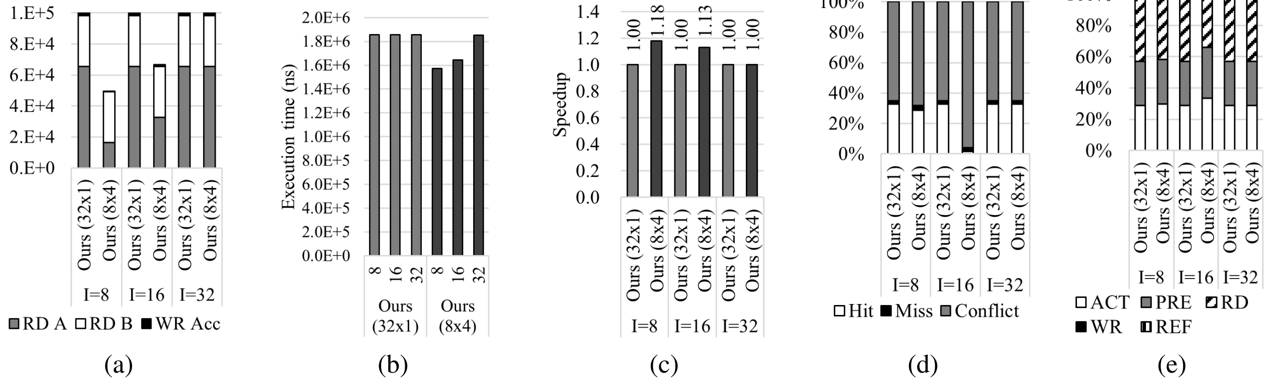

FIGURE 18. (a) The number of memory requests (b) execution time (c) the speedup (d) breakdown of row buffer hit/miss/conflict ratio and (e) DRAM command ratio of our decoupled PIM with different tile sizes executing Level-3 BLAS, (I × 512) × (512 × 2048).

longer sequence length implies the larger input size. As in the Seq2Seq model mentioned above, CPU_P (MKL) and CPU_P (MLAS) performed better by data reuse and parallel computation for longer sequence length, reaching 2.2× and 2.3× speedup, respectively. The session initialization, including the memory copy from the host to GPU, consumed over 99% of the GPU execution time, thus resulting in under 0.2× speedup.

The execution time ratio of the MM multiplication on CPU_S (MKL) was 54.6∼74.7%, which was much less than the Seq2Seq model. Therefore, the performance improvement achieved by the per-bank, all-bank and our PIM was less than the Seq2Seq model. The speedup of PIM_PB and PIM_AB increased up to 2.2× and 3.6×, respectively. The execution time of the MM multiplication on PIM_PB and PIM_AB increased in proportion to the sequence length. However, since the execution time ratio of MM multiplication is relatively small, the overhead of repeating VM did not significantly affect the overall performance. Therefore, the performance of PIM_PB and PIM_AB improved with the sequence length.

The speedup of our PIM was 1.7× at sequence length 8 and 3.1× at sequence length 32. Our PIM performed a little worse than PIM_PB at sequence length 8 and outperformed PIm_PB by 1.4× at sequence length 32, reaching up to 86.5% of the PIM_AB performance.

2) ENERGY CONSUMPTION

Figure 17 illustrates the energy consumption normalized CPU_S on both applications on a log scale. On the multi-batch LSTM-based Seq2Seq in Figure 17(a), the normalized energy consumption of CPU_P decreased by the batch size because the higher weight reuse improved the speedup as the batch size increased, as shown in Figure 15. The energy consumption of GPU increased slightly as its performance degraded by the batch size. Since PIM_PB and PIM_AB repeat the VM multiplication without reusing weights, the batch size increased their normalized energy consumption. However, because of the highest speedup of PIM_AB, its energy consumption was the lowest in all cases. At the batch size 128, the normalized energy consumption of GPU was 28.3% less than CPU_S and 20.2% higher than PIM_PB, respectively. The normalized energy consumption of our PIM was 86.3%, 80.8%, and 77.0% less than CPU_S, GPU, and PIM_PB. Compared to PIM_AB, our PIM consumed only 0.3% and 2.7% higher energy at the batch sizes of 64 and 128, respectively.

Figure 17(b) shows that the normalized energy consumption of BERT on all the platforms became lower by the sequence length since their speedup increased, as shown in Figure 16. The normalized energy of GPU was significantly higher than all platforms because of the excessive initialization overhead. At the sequence length 32, the energy consumption of CPU_S (MLAS) was 30.3% less than CPU_S (MKL), and both CPU_P showed similar energy consumption, which was 43.8% and 48.6% lower than CPU_S (MKL), respectively. The normalized energy consumption of PIM_PB were 50.6% less than CPU_S (MKL), and our PIM consumed 40.3%, 98.3%, and 33.5% less energy than CPU_P (MKL), GPU, and PIM_PB, respectively. PIM_AB consumed the lowest energy in all cases, and the energy consumption of our decoupled PIM was 3.1% higher than PIM_AB.

D. PERFORMANCE IMPACT OF ALU-WIDTH TILING

We also analyzed the performance of our decoupled PIM using one more tile (32 × 1) for *MatA* to underutilize ALUs. Figure 18 compares the number of memory requests, execution time, speedup, and DRAM behavior of Level-3 BLAS, (I × 512) × (512 × 2048) on our decoupled PIM of two tile sizes (i.e., 32 × 1 and 8 × 4). At $I = 32$, both tile sizes fully utilized the ALUs and exploited the maximum opportunity for reusing the bank-private operands. Therefore, the performance of both tile sizes was the same.

The (32 × 1) tile operates in the same way for all the cases $I \leq 32$, i.e., underutilizes ALUs for $I = 8$ and $I = 16$. Therefore, the execution time for all the cases was the same. On the other hand, the (8 × 4) tile fully utilized the ALUs

and reduced the number of RD A requests by 75% and 50%, respectively; thus, the (8 × 4) tile showed 18% and 13% speedup over the (32 × 1) tile execution.

The (32 × 1) tile with a register-sized window of (32, 32, 16) reads 16 tiles of *MatB* at the memory phase and 32 tiles of *MatA* at the computation phase, incurring a total of 32 row conflicts and 16 row hits, respectively. At $I = 8$, the (8 × 4) tile with a register-sized window of (8, 32, 16) reads 16 *MatB* tiles at the first memory phase and then reads 8 tiles of *MatA* from 8 banks at the computation phase with 8 row conflicts. At the next memory phase, the (8 × 4) tile encounters 8 conflicts in the banks whose rows were open for *MatA* and 8 row hits in the other banks. At $I = 16$, the (8 × 4) tile results close to 100% in row conflicts, as discussed in Section IV-B3. The power consumption was hardly affected since the different tile sizes changed the number of RD commands, while the number of ACT/PRE remained the same.

V. CONCLUSION

This paper proposed the memory-computation decoupled PIM architecture to provide the performance comparable to the all-bank PIM while preserving the standard DRAM interface, i.e., DRAM commands, power budget, timing constraints, etc. For achieving our goal, we introduced two PIM execution phases: memory and computation. The memory phase follows the per-bank PIM execution method. The computation phase broadcasts the bank-shared operands to all banks, thus making their PIM engines perform the computation simultaneously without accessing the bank data arrays. Also, we developed the compiler techniques to maximize the decoupled execution opportunity by increasing the bank-private operands' reusability.

We evaluated the performance of Level-2/3 BLAS, a multi-batch LSTM-based Seq2Seq model, and BERT on real machines; CPU, GPU, FPGA-based PIM to provide an accurate performance analysis. The performance of the proposed PIM compared to CPU and GPU was 75.8× and 1.2× faster in Level-3 BLAS and 8.4× and 1.5× faster in LSTM-based Seq2Seq and 3.1× and 15.5× in BERT, respectively. We also reached the performance of the ideal all-bank PIM up to 91.3%, 97.8%, and 86.6% in the applications, respectively. Compared to GPU and the per-bank PIM, the energy consumption of our decoupled PIM was lower by 72.0% and 78.4% in Level-3 BLAS, 80.8% and 77.0% in LSTM-based Seq2Seq, and 98.3% and 33.5% in BERT. Also, our PIM consumed only 7.4%, 0.3%, and 3.1% more energy than the ideal all-bank PIM in the applications.

ACKNOWLEDGMENT

The authors would like to note that this article has been published in the doctoral thesis: Yoonah Paik, "Accelerating DRAM Verification by Test-oriented Command Scheduling and In-DRAM PIM Execution by Memory-computation Decoupling," Ph.D. thesis, Department of Electrical and Computer Engineering, Korea University, Seoul, Korea, 2022. [Online]. Available: http://www.dcollection.net/handler/korea/000000269250.

REFERENCES

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, Long Beach, CA, USA, Dec. 2017, pp. 5998–6008.

[3] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Athens, Greece, Oct. 2020, pp. 372–385.

[4] 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.

[5] 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.

[6] B. Kim, J. Chung, E. Lee, W. Jung, S. Lee, J. Choi, J. Park, M. Wi, S. Lee, and J. Ho Ahn, "MViD: Sparse matrix-vector multiplication in mobile DRAM for accelerating recurrent neural networks," *IEEE Trans. Comput.*, vol. 69, no. 7, pp. 955–967, Jul. 2020.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[8] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. *Improving Language Understanding by Generative Pre-Training*. Accessed: Jul. 22, 2021. [Online]. Available: https://s3-us-west-2.amazonaws.com/openaiassets/researchcovers/language-unsupervised/language_understanding_paper.pdf

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI Blog, Feb. 2019.

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

[11] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "ABC-DIMM: Alleviating the bottleneck of communication in DIMM-based near-memory processing with inter-DIMM broadcast," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 237–250.

[12] 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, Oct. 2018.

[13] 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.

[14] 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)*, Boston, MA, USA, Oct. 2017, pp. 273–287.

[15] 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)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.

[16] 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.

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

[18] 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)*, Dresden, Germany, Mar. 2018, pp. 815–820.

[19] 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. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, Jun. 2016, pp. 27–39.

[20] 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. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 14–26.

[21] 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.

[22] S. Liu, B. Leung, A. Neckar, S. O. Memik, G. Memik, and N. Hardavellas, ''Hardware/software techniques for DRAM thermal management,'' in *Proc. 17th Int. Symp. High Perform. Comput. Archit.*, San Antonio, TX, USA, Feb. 2011, pp. 515–525.

[23] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, ''Adaptive-latency DRAM: Optimizing DRAM timing for the common-case,'' in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, Feb. 2015, pp. 489–501.

[24] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, ''Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,'' *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[25] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, ''Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,'' *IEEE J. Emerging Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

[26] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, ''HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation,'' in *Proc. 57th ACM/IEEE Design Automat. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[27] Z. Xianyi. *OpenBLAS—An Optimized BLAS Library*. Accessed: Nov. 22, 2021. [Online]. Available: https://www.openblas.net/

[28] NVIDIA Corporation. *CUDA Toolkit Documentation*. Accessed: Nov. 22, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[29] *JESD79-4: JEDEC Standard: DDR4 SDRAM*, JEDEC Solid State Technol. Assoc., Arlington, VA, USA, Sep. 2012.

[30] A. Nag and R. Balasubramonian, ''OrderLight: Lightweight memory-ordering primitive for efficient fine-grained PIM computations,'' in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 298–310, doi: 10.1145/3466752.3480103.

[31] G. Yuan, P. Behnam, Z. Li, A. Shafiee, S. Lin, X. Ma, H. Liu, X. Qian, M. N. Bojnordi, Y. Wang, and C. Ding, ''FORMS: Fine-grained polarized ReRAM-based *in-situ* computation for mixed-signal DNN accelerator,'' in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 237–250.

[32] Y. Long, E. Lee, D. Kim, and S. Mukhopadhyay, ''Flex-PIM: A ferro-electric FET based vector matrix multiplication engine with dynamical bitwidth and floating point precision,'' in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Glasgow, U.K., Jul. 2020, pp. 1–8.

[33] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, ''MAGIC—Memristor-aided logic,'' *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[34] J.-P. Wang and J. D. Harms, ''General structure for computational random access memory (CRAM),'' U. S Patent 9 224 447, Dec. 29, 2015.

[35] M. Imani, S. Gupta, Y. Kim, and T. Rosing, ''FloatPIM: In-memory acceleration of deep neural network training with high precision,'' in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Archit. (ISCA)*, Phoenix, AZ, USA, Jun. 2019, pp. 802–815.

[36] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, ''DUAL: Acceleration of clustering algorithms using digital-based processing in-memory,'' in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Athens, Greece, Oct. 2020, pp. 356–371.

[37] S. Gupta, M. Imani, and T. Rosing, ''FELIX: Fast and energy-efficient logic in memory,'' in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, San Diego, CA, USA, Nov. 2018, pp. 1–7.

[38] O. Leitersdorf, R. Ronen, and S. Kvatinsky, ''MultPIM: Fast stateful multiplication for Processing-in-Memory,'' *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 3, pp. 1647–1651, Mar. 2022.

[39] X. Xin, Y. Zhang, and J. Yang, ''ELP2IM: Efficient and low power bitwise operation processing in DRAM,'' in *Proc. IEEE 26th Int. Symp. High Perform. Comput. Archit. (HPCA)*, San Diego, CA, USA, Feb. 2020, pp. 303–314.

[40] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. A. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, ''SIMDRAM: A framework for bit-serial SIMD processing using DRAM,'' in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Apr. 2021, pp. 329–345.

[41] 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)*, Boston, MA, USA, Oct. 2017, pp. 288–301.

[42] I. Sutskever, O. Vinyals, and Q. V. Le, ''Sequence to sequence learning with neural networks,'' in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.

[43] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, ''PyTorch: An imperative style, high-performance deep learning library,'' in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates, 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[44] spaCy. *spaCy: Industrial-Strength Natural Language Processing in Python*. Accessed: Nov. 22, 2021. [Online]. Available: https://spacy.io

[45] ONNX Runtime Developers. *ONNX Runtime*. Accessed: Jan. 1, 2022. [Online]. Available: https://onnxruntime.ai/

[46] Intel Corporation. *Intel OneAPI Math Kernel Library*. Accessed: Nov. 22, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[47] Microsoft. *Microsoft Linear Algebra Subprogram*. Accessed: Jan. 1, 2022. [Online]. Available: https://github.com/microsoft/onnxruntime/tree/master/onnxruntime/core/mlas,

[48] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, ''Measuring energy and power with PAPI,'' in *Proc. 41st Int. Conf. Parallel Process. Workshops*, Pittsburgh, PA, USA, Sep. 2012, pp. 262–268.

[49] NVIDIA Corporation. *NVIDIA System Management Interface*. Accessed: Nov. 22, 2021. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[50] K. Chandrasekar, C. Weis, Y. Li, B. Akesson, N. Wehn, and K. Goossens. *DRAMPower: Open-Source DRAM Power & Energy Estimation Tool*. Accessed: Nov. 22, 2021. [Online]. Available: http://www.drampower.info

[51] M. Sung *et al.*, ''Gate-first high-K/metal gate DRAM technology for low power and high performance products,'' in *IEDM*, Washington, DC, USA, Dec. 2015, p. 26.

[52] M. M. Ghaffar, C. Sudarshan, C. Weis, M. Jung, and N. Wehn, ''A low power in-DRAM architecture for quantized CNNs using fast Winograd convolutions,'' in *Proc. Int. Symp. Memory Syst.*, Washington, DC, USA, Sep. 2020, pp. 158–168.

[53] C. Sudarshan, T. Soliman, C. De la Parra, C. Weis, L. Ecco, M. Jung, N. Wehn, and A. Guntoro, ''A novel DRAM-based process-in-memory architecture and its implementation for CNNs,'' in *Proc. 26th Asia South Pacific Design Automat. Conf.*, Tokyo, Japan, Jan. 2021, pp. 35–42.

[54] K. Sohn, W.-J. Yun, R. Oh, C.-S. Oh, S.-Y. Seo, M.-S. Park, D.-H. Shin, W.-C. Jung, S.-H. Shin, J.-M. Ryu, H.-S. Yu, J.-H. Jung, H. Lee, S.-Y. Kang, Y.-S. Sohn, J.-H. Choi, Y.-C. Bae, S.-J. Jang, and G. Jin, ''A 1.2 V 20 nm 307 GB/s HBM DRAM with at-speed wafer-level IO test scheme and adaptive refresh considering temperature distribution,'' *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 250–260, Sep. 2016.

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. Currently, she is with the Samsung Advanced Institute of Technology. Her current research interests include microarchitecture and memory system design.

CHANG HYUN KIM received the B.E. degree in electrical engineering from Korea University, Seoul, South Korea, in 2016, and 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.

WON JUN LEE received the B.S. degree in electrical engineering from Korea University, Jo Chi Won, South 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.

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 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, from 1990 to 1995 and a Staff Software Engineer at Inter/KSL, from 2001 to 2002. Currently, he is 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.