2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00013

Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology

Industrial Product

Sukhan Lee[§1], Shin-haeng Kang[§1], Jaehoon Lee[1], Hyeonsu Kim[2], Eojin Lee[1], Seungwoo Seo[2], Hosang Yoon[2], Seungwon Lee[2], Kyounghwan Lim[1], Hyunsung Shin[1], Jinhyun Kim[1], Seongil O[1], Anand Iyer[3], David Wang[3], Kyomin Sohn[1] and Nam Sung Kim[§1]

[1]Memory Business Division, Samsung Electronics
[2]Samsung Advanced Institute of Technology, Samsung Electronics
[3]Device Solutions America, Samsung Electronics

***Abstract*—Emerging applications such as deep neural network demand high off-chip memory bandwidth. However, under stringent physical constraints of chip packages and system boards, it becomes very expensive to further increase the bandwidth of off-chip memory. Besides, transferring data across the memory hierarchy constitutes a large fraction of total energy consumption of systems, and the fraction has steadily increased with the stagnant technology scaling and poor data reuse characteristics of such emerging applications. To cost-effectively increase the bandwidth and energy efficiency, researchers began to reconsider the past processing-in-memory (PIM) architectures and advance them further, especially exploiting recent integration technologies such as 2.5D/3D stacking. Albeit the recent advances, no major memory manufacturer has developed even a proof-of-concept silicon yet, not to mention a product. This is because the past PIM architectures often require changes in host processors and/or application code which memory manufacturers cannot easily govern. In this paper, elegantly tackling the aforementioned challenges, we propose an innovative yet practical PIM architecture. To demonstrate its practicality and effectiveness at the system level, we implement it with a 20nm DRAM technology, integrate it with an unmodified commercial processor, develop the necessary software stack, and run existing applications without changing their source code. Our evaluation at the system level shows that our PIM improves the performance of memory-bound neural network kernels and applications by 11.2× and 3.5×, respectively. Atop the performance improvement, PIM also reduces the energy per bit transfer by 3.5×, and the overall energy efficiency of the system running the applications by 3.2×.**

***Index Terms*—processing in memory, neural network, accelerator, DRAM**

I. Introduction

Deep neural networks (DNNs) are a class of machine learning (ML) algorithms, and it has been widely used for various disruptive applications with significant economic and societal impacts. Convolutional neural networks (CNNs) [18], [24], [38], [50] for computer vision (CV) and recurrent neural networks (RNNs) [4], [53] for natural language processing (NLP) are the most representative examples. New applications such as recommendation model (RM) [14], [40] are also popular these days.

The property of a neural network (NN) is determined by that of layers composing the network. Some key layers dominate not only the property but also the execution time of a given NN. Especially, the key layers of popular NN models, e.g., gate recurrent unit [4], [53] and embedding lookup [14], [40], and batch normalization [18], [24], have a common computational characteristic; these layers exhibit low temporal and/or spatial locality (i.e., low cache hit rates), thereby demanding high off-chip memory bandwidth. Meanwhile, the demand for improving the energy efficiency of systems has continuously increased, and the energy consumed for transferring data across the memory hierarchy constitutes an increasing fraction of the total energy consumption of systems with stagnant technology scaling. For example, transferring data from off-chip DRAM devices through the on-chip cache hierarchy to the register file of a processor consumes about two orders of magnitude more energy than performing a floating-point (FP) operation in a processor [31]. That is, the performance and energy efficiency of executing these layers are primarily governed by those of data transfers between the processor execution units and the DRAM devices.

The high bandwidth memory (HBM) [25]) was introduced to satisfy the increasing demand for high bandwidth and low energy per bit transfer. It can provide much higher bandwidth at lower energy per bit transfer than conventional DRAM, because it is tightly integrated with a processor die on a common substrate (e.g., silicon interposer [51]) in a package. For instance, such a tight integration at the package level allows 10∼13× more I/O interconnects between a processor and DRAM at 2∼2.4× lower energy per bit transfer than conventional DDR4 DRAM. Nonetheless, recent studies [9], [29] show that the rapidly growing model size and compute density create memory bottlenecks even with the use of HBM. Note that it is very costly to further increase the overall bandwidth by increasing bandwidth per I/O pin and/or the number

This paper is part of the Industry Track of ISCA 2021's program.

[§]Sukhan Lee and Shin-haeng Kang equally contributed to this work and Nam Sung Kim is the corresponding author.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00013

of I/O interconnects under stringent physical constraints of a package, such as signal integrity, power, temperature, and form factor [7], [36], [51].

To expose higher bandwidth to processors at lower energy per bit transfer, researchers began to reconsider past processing-in-memory (PIM) architectures (e.g., [10], [13], [17], [30], [44]) and further advance them, many of which exploit recent package- and chip-level integration technologies such as 2.5D and 3D stacking (e.g., [2], [12], [19], [33], [35], [45], [54]). Subsequently, many promising PIM architectures have emerged, but no major memory manufacturer has developed even a proof-of-concept silicon yet, not to mention a product. The primary reasons are two folds. First, the 3D or 2.5D integration itself cannot automatically offer significantly higher bandwidth for processors than standard DRAM. This is because the bandwidth of each DRAM bank is designed to match that of device I/O. To provide notably higher bandwidth for the processors, considerable customization of DRAM is required, and it significantly increases the cost [5]. Second, especially to orchestrate the communication between host and in-memory processors, the past PIM architectures often demand notable changes in the host processors and/or application code (e.g., [2], [3], [11], [12], [35], [45]), which memory manufacturers cannot easily govern.

In this paper, elegantly tackling the aforementioned challenges, we propose an innovative yet practical PIM architecture. First, it does not disturb the key components (i.e., subarray and bank) of commodity DRAM. Instead, it exploits bank-level parallelism to provide higher bandwidth and lower energy per bit transfer for processors in DRAM. That is, it can be easily and seamlessly integrated with any commodity DRAM. Second, it does not demand any change in any component of modern commercial processors including DRAM controllers, as it is architected for host processors to control PIM operations through standard DRAM interfaces. This can facilitate a drop-in replacement of current JEDEC-compliant DRAM with PIM-DRAM for any systems.

To demonstrate its feasibility and efficacy at the system level running the full software stack, we implement the proposed PIM architecture based on a HBM2 design [51], dubbing it PIM-HBM in this paper. After fabricating the implementation with a 20nm DRAM technology, we integrate it with an unmodified commercial processor. Concurrently, to run existing NN application code without any change, we also develop a software stack. To the best of our knowledge, this work is the first HBM-based PIM architecture that is fabricated by a major DRAM manufacturer and 2.5D-integrated with an unmodified commercial processor with the full software stack support. Our system-level evaluation shows that a processor with PIM-HBM can execute memory-bound NN kernels and applications 11.2× and 3.5× faster, respectively, than the same processor with commodity HBM. In addition to the performance improvement, PIM reduces the energy per bit transfer by 3.5× and improves the energy efficiency of the system running the applications by 3.2×.

The rest of this paper is organized as follows. Section II gives background on DNN and HBM. Section III presents a PIM architecture that can function as both standard DRAM and PIM-DRAM. Section IV details a PIM microarchitecture, focusing on the execution unit. Section V depicts a software stack to support PIM for a system. Section VI describes a chip implementation and integration with a commercial processor. Section VII evaluates performance and energy efficiency. Section IX discusses related work.

II. Background

A. Characteristics of Modern DNN Applications

A DNN comprises many layers between the input and output layers, and it can be used for various applications (e.g., CV, NLP, and RM) depending on the composition of layers. To develop DNN applications more efficiently, we often use popular ML programming frameworks such as TensorFlow [1] and PyTorch [43], which make use of many APIs such as MKL [23], oneAPI [21] (formerly known as MKL-DNN), CuDNN [6] and MIOpen [32].

Those APIs are often built on basic linear algebra subprograms (BLAS [41]), and it is well known that the level-1 BLAS (scalar-vector and vector-vector operations) and level-2 BLAS (matrix-vector operations) do not benefit from on-chip cache; data can be seldom reused especially when the entire data set does not fit into on-chip cache. Consequently, frequent off-chip memory accesses are needed to perform few computations, resulting in few operations per byte, and performance is ultimately bounded by the memory bandwidth (**memory-bound**). On the other hand, the level-3 BLAS (matrix-matrix operations) can reuse a considerable fraction of the data in the on-chip cache (i.e., many operations per byte). That is, the performance is determined by the compute capability (**compute-bound**).

Early DNN applications exhibit compute-bound characteristics, because they mainly perform compute operations on matrices (e.g., convolutional layers) [38], [50]. However, the number of memory-bound layers has gradually increased in recent popular DNN applications. For instance, the key functions of RNN are matrix-vector multiplications, where the main memory bandwidth determines the performance [4], [53]. In addition, batch normalization (BN) and skip connection layers in contemporary CNNs are memory-bound due to their low data reuse characteristics [18], [24]. The embedding lookup layer, a key component in RM, is also memory-bound because of a large number of memory accesses to the large embedding table for sparse-length-sum (SLS) operations [40].

B. High Bandwidth Memory

As the compute capability of processors has significantly increased, so has the demand for the off-chip memory bandwidth. To satisfy such demand, HBM was introduced. HBM 3D-stacks multiple DRAM dies with a buffer die that comprises I/O circuits, memory built-in-self-test (MBIST), and features to support testing and debugging. To provide high memory bandwidth, the DRAM dies communicate with the buffer die using through silicon vias (TSVs), and the buffer

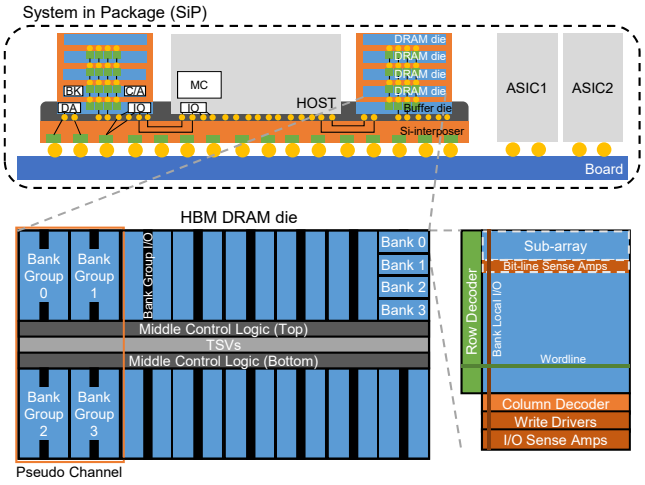

Fig. 2: A cross-section view of a system in package (SiP) comprising an ASIC and HBM devices (top) and an HBM DRAM die organization (bottom).

die is connected to a host processor by the silicon interposer in a package. As such, the integration of a host processor with one or more HBM devices in a package is often called SiP (System-in-Package). Fig. 2 depicts a typical SiP comprising an ASIC and HBM devices (top) and an HBM DRAM die organization (bottom).

A HBM2 DRAM die comprises 4 pseudo channels (pCHs), each consisting of 4 bank groups (BGs), middle control logic, data bus, and TSV I/O circuitry. A bank group comprises 4 banks that share datapath resources (e.g., bank control lines, and bank group I/O (BGIO)). The middle control logic receives a DRAM command and address pair (CA), decodes the CA to generate bank control signals and addresses, and delivers the generated signals to the target bank through bank control lines and BGIO. It also controls the data bus and TSV I/O circuit based on the CA. A bank consists of DRAM cell arrays, row and columns decoders, I/O sense amplifiers (IOSA), and write drivers. A DRAM cell array is composed of sub-arrays, which has sub-wordline drivers (SWD), bit-line sense amplifiers (BLSAs), and local I/O (LIO) lines. The sequence of data access operations is almost the same as that of DDR DRAM (i.e., activation (ACT), read (RD), write (WR), precharge (PRE)) except for the data access size; an access to HBM transfers a 256-bit data block over 4 64-bit bursts over one pCH. A group of 4 DRAM dies constitutes a rank, providing a total of 16 pCHs. An HBM device with more stacked DRAM dies can provide more ranks for larger capacity, but it does not offer higher bandwidth as 16pCHs are shared among ranks.

Even with HBM, modern DNNs still suffer from limited memory bandwidth, which in turn leads to poor utilization of processors and thus limited performance [9], [29]. One way to provide higher bandwidth is to integrate a processor with more HBM devices. However, it is hard to do so because not only the power and thermal budgets of an SiP but also the number of I/O connections with a silicon interposer are limited.

III. PIM-DRAM Architecture

In this section, we present a PIM architecture that exploits bank-level parallelism to expose high on-chip bandwidth of standard DRAM to processors in a practical way. Although it is illustrated based on HBM2 in this paper, it is applicable to any standard DRAM such as DDR, LPDDR, and GDDR DRAM with a few changes.

A. Overview

Fig. 1 overviews our PIM architecture: (a) a PIM-HBM DRAM die; (a) a PIM-DRAM die; (b) a bank coupled with a PIM execution unit comprising a single instruction multiple data (SIMD) floating-point unit (FPU), command, general and scalar register files (CRF, GRF, and SRF); and (c) datapath of the PIM execution unit. Two of the key design philosophies are (1) supporting both standard DRAM and PIM-DRAM modes for versatility, and (2) minimizing the high engineering cost of re-designing the DRAM sub-array and bank to support only PIM. As such, we place a PIM execution unit at the I/O boundary of a bank (Fig. 1(a) and (b)).

In PIM mode, PIM execution units across all the banks concurrently respond to a standard DRAM column (RD or WR) command from the host processor, executing one wide-SIMD operation commanded by a PIM instruction with *deterministic latency* in a lock-step manner.

PIM instructions are stored in the CRF serving as an instruction buffer, and a DRAM column command triggers the execution of a PIM instruction; see Section IV for the details. A PIM execution unit with 256-bit datapath may get necessary 16 16-bit operands from IOSAs, one of the registers, and/or the result bus (Fig. 1(c)). The column address of a given DRAM

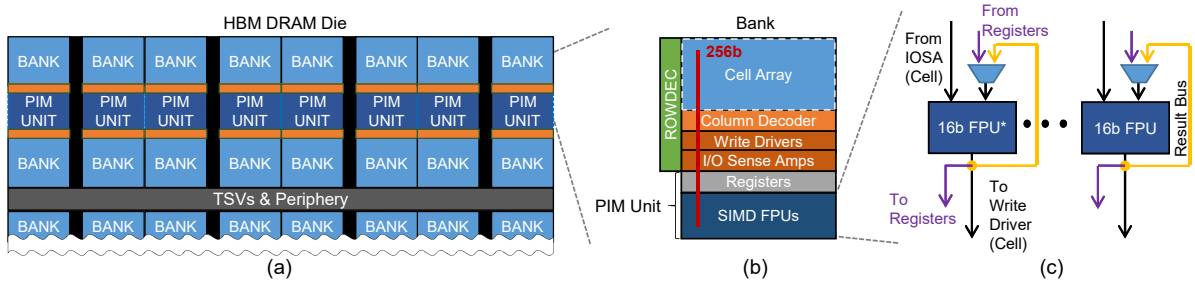

Fig. 1: (a) HBM DRAM die organization, (b) bank coupled with a PIM unit, (c) PIM datapath.

RD command determines the address of the operands from the IOSAs of a bank, pulling a total of 4096 bits (i.e., 256 bits per bank × 16 banks) across all the banks in parallel. A DRAM WR command operates in the same manner as a DRAM RD command except that the host processor pushes 256 bits to the write drivers or PIM registers of all the banks. That is, PIM can expose up to 16× (= the number of banks per pCH) higher bandwidth to PIM execution units than a processor connected with an off-chip standard DRAM interface. Note that the number of PIM execution units can be fewer than that of banks, i.e., trade-off between the cost and the on-chip compute bandwidth. Executions of PIM instructions with standard DRAM commands and deterministic latencies are essential to facilitate PIM-DRAM with unmodified JEDEC-compliant DRAM controllers.

Finally, the host processor can control execution of every PIM instruction one by one with its load (LD) and store (ST) instructions which are translated into standard DRAM commands to DRAM. That is, the host processor precisely knows the address, value and timing/order of each PIM operation since each PIM operation coupled with a DRAM command is executed with deterministic timing. As such, the host processor can independently control PIM operations of each memory channel. In addition, each PIM execution unit accesses the memory at the same data access granularity as the host processor. These make our PIM more agnostic to a physical address mapping scheme of the host processor and more elegantly and efficiently handle coherence/consistency challenges without any change in existing processors than prior PIM architectures.

B. Bank-Level Parallelism for PIM

Single Bank Mode: In standard DRAM devices, an ACT command targets a specific bank pointed by the bank and bank group addresses (BA/BG), and only the single bank is activated in response to the command. In this paper, we refer to this standard DRAM operation mode (Fig. 3(a)) as single bank mode (SB mode).

All Bank Mode: In contrast to the standard SB mode, we expose the on-chip bandwidth of DRAM to PIM execution units by allowing concurrent accesses to multiple banks with a single DRAM command (Fig. 3(b)). As such, we implement all-bank control logic to support all-bank operation mode (AB mode), where the BA and BG of a given column address are ignored and the same row and column of all the banks are concurrently accessed in a lock-step manner by a single DRAM command. In AB mode, PIM-HBM with 16 banks per pCH can provide 8× higher on-chip compute bandwidth than standard HBM. Note that each bank can operate at every tCCD_L (the delay between two back-to-back column commands to the same bank group) in AB mode, not tCCD_S (the delay between two back-to-back column commands to banks in different bank groups). Since tCCD_S (2 tCK) is typically a half of tCCD_L (4 tCK), the compute bandwidth improves by a half of the number of banks. For example, PIM-HBM operating at 1.0GHz can provide 128GB/s per pCH for

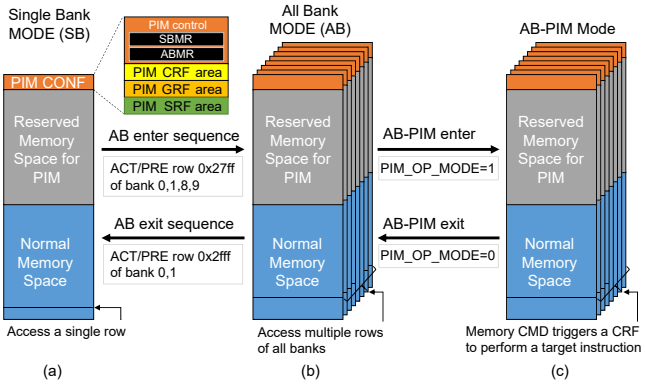

Fig. 3: PIM-HBM operation modes: (a) single bank (SB) mode, (b) all-bank (AB) mode, and (c) all-bank-PIM (AB-PIM) mode.

PIM execution units (i.e., a total of 2TB/s for all 16 pCH) while HBM2 can give 16GB/s per pCH to the host processor.

All Bank PIM Mode: In addition to AB mode, we support AB-PIM mode which is proceeded by the AB mode (Fig 3(c)). Similar to AB-mode, a given DRAM command is applied to all the banks in this mode, as well. However, in AB-PIM mode, a DRAM column command triggers execution of a PIM instruction in the CRF and then updates a PIM program counter (PPC) to get the next PIM instruction. Note that the AB-PIM mode does not consume power for transferring data from the bank I/O all the way to the I/O circuits that interface with the host processor. Such power consumption is substantial enough to offset the power increase by the PIM execution units operating concurrently (Section VII-C).

Transitions between Modes: To support transitions between SB and AB modes for an unmodified host processor using a standard DRAM interface, we may consider using a mode register set (MRS) command [19]. However, user processes cannot easily access mode registers because privileged supervisor processes typically control them. That is, to make a mode change, we need to invoke a privileged kernel. Thus, we choose not to use the MRS approach as the overhead of context switching incurs significant performance degradation.

To implement mode transitions between SB and AB modes with small performance overhead, we propose to use a sequence of standard DRAM commands. We implement two configuration registers, AB mode register (ABMR) and SB mode register (SBMR) that are mapped to a reserved memory space, PIM CONF. To enter the AB mode, the host processor sends a sequence of ACT and PRE commands to specific addresses in PIM CONF (Fig. 3). To exit the AB mode, the host processor precharges (closes) all the open rows of the banks so that there is no row-buffer conflict after the transition to the SB mode. This approach is compatible with any processors adopting JEDEC-compliant DRAM controllers because it relies on standard DRAM commands. To enter and exit the AB-PIM mode, the PIM_OP_MODE register mapped to an address in PIM CONF is set to 1 and 0, respectively.

TABLE I: The relative area and energy/op of MAC units in a DRAM 20nm technology. The numbers are normalized to that of an INT16 MAC with a 48-bit accumulative register.

Number format	Area	Energy/Op.
INT16 (w/ 48-bit Acc.)	1	1
INT8 (w/ 48-bit Acc.)	0.45	0.81
INT8 (w/ 32-bit Acc.)	0.35	0.77
FP16	1.32	1.21
BFLOAT16	1.15	1.04
FP32	3.96	1.34

Note that PIM mode, configuration, general, command scalar registers are mapped to specific reserved memory addresses that can be accessed by standard DRAM commands.

C. Instruction Set Architecture

Data and Operation Types: Our primary target is to accelerate memory-bound ML kernels. As mentioned in Section II-A, it is well known that level-1 BLAS (e.g., `AXPY` for CV) and level-2 BLAS (e.g., `GEMV` for NLP) kernels are often memory-bound. Although INT8 operations have been widely used especially for inference, FP16 operations have become more prevalent for both inference and training. Besides, the 16-bit brain floating-point format (BFLOAT16 [52]) optimized for ML applications has emerged. The BFLOAT16 preserves the number of exponent bits in FP32, then reduces the number of significant bits from 23 to 7 bits to represent a number with 16 bits. It allows a simple conversion from FP32 and provides a wider dynamic range than FP16. The rationale behind BFLOAT16 is to minimize the power and area costs of supporting FP32 while providing enough compute accuracy for ML applications. However, it is not an IEEE standard and is specialized for deep learning only.

Table I compares the area and energy/op of INT16, INT8, FP16, BFLOAT16 and FP32 multiply-accumulate (MAC) units in a DRAM 20nm technology. This shows that the area and energy/op of FP32 MAC units are too large to be implemented in DRAM, limiting the number of MAC units (or compute capability). FP16 and BFLOAT16 MAC units give a much wider dynamic range than INT16 MAC units while offering the area and energy/op comparable to INT16 MAC units. Between FP16 and BFLOAT16 MAC units, we see that the BFLOAT16 MAC unit is slightly smaller and more energy-efficient than the FP16 MAC unit. Nonetheless, FP16 has been natively supported by most processors and widely used for ML applications. Thus, we choose implementing FP16 MAC units rather than BFLOAT16 MAC units to efficiently support not only ML applications but also applications in other domains such as OpenCL, OpenCV and HPC applications [16], which are built on legacy FP16 libraries.

In addition, we support the ReLU operation since NN layers typically require an activation function at the end. ReLU returns zero if a given input value is a negative number. Otherwise, it returns the input value itself. Among various activation functions, ReLU is the most widely used one and has three advantages over sigmoid variants: (1) it is simple to implement and fast (i.e., a 2-to-1 multiplexer controlled by the sign bit of a given input value); (2) it is friendly to zero-skipping ML accelerators; and (3) it is known to give higher inference accuracies than sigmoid variants [38], [39]. Lastly, we support data movement among GRF, SRF, and BANK; see Section IV-A for more details on the GRF, SRF, and BANK.

TABLE II: The supporting operations, operand sources, and result destination.

Op. Type	Operand (SRC0)	Operand (SRC1)	Result (DST)	# of Combinations
MUL	GRF, BANK	GRF, BANK, SRF_M	GRF	32
ADD	GRF, BANK, SRF_A	GRF, BANK, SRF_A	GRF	40
MAC	GRF, BANK	GRF, BANK, SRF_M	GRF_B	14
MAD	GRF, BANK	GRF, BANK, SRF_M SRF_A (for SRC2)	GRF	28
MOV (ReLU)	GRF, BANK		GRF	24

Table II lists the types of all the PIM operations and their possible operand sources and result destination. In general, a source operand can come from GRF, SRF or BANK coupled with a PIM execution unit. As each operation type can have various combinations of operand sources, PIM supports a total of 114 operand combinations for computations, and 24 different ways of data movement. Note that we implement ReLU as part of data movement operation; a flag bit determines whether or not ReLU is performed during data movement; see in Section IV for more details.

Instruction Format: The PIM execution unit supports typical RISC-style 32-bit instructions. There are total of 9 instructions, which are divided into three types, as summarized in Table III: (1) NOP, JUMP, and EXIT are flow-control instructions; (2) ADD, MUL, MAD, and MAC are arithmetic instructions especially chosen for accelerating ML applications; (3) MOV and FILL are data movement instructions for loading/storing data from/to the registers in a PIM execution unit.

JUMP is an essential instruction to control a flow of a PIM microkernel[1] program stored in CRF. In particular, to reduce the overhead of control flow change, we support a zero-cycle JUMP instruction that does not require any computation to determine a jump target address by supporting only a pre-programmed number of iterations and pre-decoding a given JUMP instruction at the fetch and decode stages. In addition

[1] A kernel is executed by the host processor while a microkernel is executed by PIM execution units in this paper.

TABLE III: The PIM-HBM instruction format.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Control	OPCODE				U									IMM0								IMM1										
Data	OPCODE				DST			SRC0			U									R	U	DST #			U	SRC0 #			U	SRC1 #		
ALU	OPCODE				DST			SRC0			SRC1			SRC2			A	U			U	DST #			U	SRC0 #			U	SRC1 #		

to JUMP, we support special techniques, such as address aligned mode (AAM) (Section IV-C) and multi-cycle NOP to efficiently implement instruction-flow controls with a limited number of entries in CRF.

In arithmetic instructions, SRC0, SRC1, and SRC2 indicate where the instructions get operands (i.e., GRF_A, GRF_B, BANK, SRF_M, and SRF_A). DST #, SRC0 #, and SRC1 # represent register numbers (or indices) and they are used when the operands come from register files. MAC performs an operation like GRF_B += GRF_A $\times$ BANK where SRC2 (GRF_B) points to the same GRF register as DST. MAD does an operation like GRF_A = BANK $\times$ SRF_M + SRF_A where SRC1 # and SRC2 # point to the same register index but in different register files (e.g., SRF_M and SRF_A, respectively). 'R' denotes ReLU that determines whether or not MOV performs a ReLU operation during data movement; MOV(ReLU) indicates a MOV instruction with R set to 1. 'U' represents unused bit(s). Finally, 'A' stands for address aligned mode for arithmetic instructions.

IV. PIM Microarchitecture

In this section, we describe microarchitecture of the PIM execution unit, including components, pipeline and instruction ordering mechanisms.

A. Components

A PIM execution unit consists of three components: (1) a 16-wide SIMD FPU, (2) register files, and (3) a controller, as depicted in Fig. 4. To limit the power and area costs, we decide to place one PIM execution unit between two banks instead of one PIM execution unit per bank. As a PIM execution unit is shared by two banks, it can access either of the two banks (denoted by EVEN_BANK and ODD_BANK) at a time. The 16-wide SIMD FPU consists of a pair of 16 FP16 multipliers and adders, each with 5 stages. The register files include CRF, GRF, and SRF. The CRF serving as an instruction buffer comprises 32 32-bit registers. GRF has 16 256-bit registers that are evenly split into GRF_A and GRF_B for EVEN_BANK and ODD_BANK, respectively. SRF replicates a given 16-bit value by 16 times and supply them to the 16-wide SIMD FPU as one of source operands, and it consists of SRF_M and SRF_A, each with 8 registers, for scalar multiplications

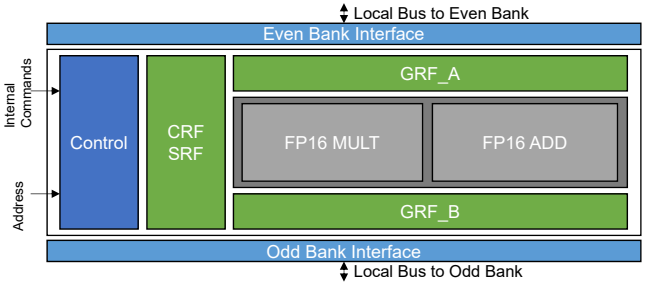

Fig. 4: The microarchitecture of PIM execution unit (1) an instruction sequence manager (blue color), (2) register files (green color), and (3) a 16-wide SIMD FPU (gray color).

and additions, respectively. The controller first fetches a PIM instruction from the CRF and decodes the instruction, and then controls the flows of PIM instructions, source operands to the SIMD FPU, and the result to the GRF.

B. Pipeline

We divide the PIM execution unit into up to five pipeline stages to satisfy the DRAM internal timing for reading/writing data. The first stage fetches and decodes a PIM instruction. The second stage loads 256-bit data from EVEN_BANK or ODD_BANK to either a GRF register or an input of the SIMD FPU. The address of the 256-bit data in the row buffer is determined by the column RD command sent by the host memory controller. Note that in AB-PIM mode, the column RD command does not transfer the data to the chip external I/O interface. Instead, it triggers an execution of a PIM instruction.

The PIM execution unit can skip the second stage when a given PIM instruction does not require any data from a bank (e.g., MAD GRF_B[0], GRF_A[0], GRF_B[1]). The source of operands are specified in a given instruction, but operands from a bank (row buffer) is not specified in the instruction. For instance, in GRF_B[m] = GRF_A[n] $\times$ BANK[row, column], the PIM instruction does not explicitly specify BANK[row, column]. Instead, the memory address of a PIM instruction is implicit (i.e., the row address of the currently open row and the column address of the column command that triggers the execution of the instruction pointed by PPC). The third and fourth stages are MULT and ADD, respectively. That is, MAC goes through both the third and fourth stages, while MULT and ADD skip the fourth and third stages, respectively. The last fifth stage writes back the result to a GRF register.

C. Instruction Ordering

Consider a PIM microkernel with 8 MAC operations stored in CRF as part of performing `GEMV`, as shown in Fig. 5(a). The MAC instructions get the column addresses of a bank from 8 column RD commands that also trigger the execution of these instructions; the column RD commands are sent by the host processor running a `GEMV` kernel. That is, each column is implicitly coupled with a specific MAC instruction, and thus the order of column commands from the PIM programmer's view needs to be the same as ones sent to PIM-DRAM. However, modern DRAM controllers often re-order DRAM commands to maximize performance [47] (Fig. 5(b)). This may result in associating the wrong column addresses with the MAC instructions which get the wrong values for the operands from the bank (Fig. 5(c)).

To guarantee the functional correctness of a given PIM microkernel, we need to warrant that DRAM commands are sent to PIM-DRAM in the order that the programmer intends (i.e., in-order DRAM accesses). However, forcing in-order memory accesses with fences [42] may incur a considerable performance penalty [37]. Thus, to minimize the performance penalty, we propose a simple mechanism that can tolerate out-of-order memory accesses in some degree, dubbed address aligned mode (AAM).

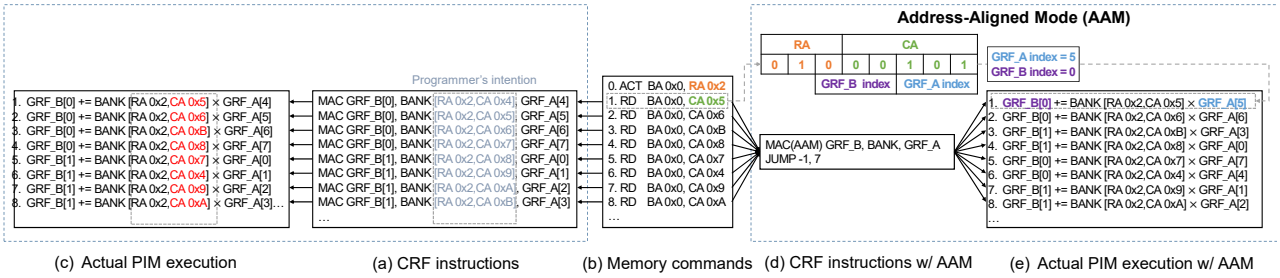

Fig. 5: Ordering MAC instructions in a GEMV microkernel. BA, RA and CA denote bank, row and column addresses.

To this end, we assign a 1-bit AAM flag to PIM arithmetic instructions ('A' in Table III). This allows us to program 8 MAC operations with two PIM instructions (Fig. 5(d)). When 'A' is set, the source and destination operand fields (SRC0 #, SRC1 #, and DST #) are ignored and replaced with sub-fields of the row and column addresses of a DRAM command (Fig. 5(e)). A constraint is that we need to assign the same register index to SRC0 #, SRC1 #, and DST # when programming a PIM microkernel. Nonetheless, such a constraint is not a problem or limitation because of the following two reasons. (1) Although the register number is the same, SRC0, SRC1, and DST can point to different operand sources (GRF_A, GRF_B, SRF_A, SRF_M, EVEN_BANK, and/or ODD_BANK). Thus, all three can point to unique operands and a destination. (2) PIM aims to accelerate vector-vector and vector-matrix operations. Such kernels often execute the same arithmetic instruction repeatedly while incrementing the source and destination register indices linearly. Moreover, they have vector (or data-level) parallelism. As such, as long as the correct operands are fed to the operations by the proposed mechanism, the PIM instructions can be executed out of order.

V. PIM Software Stack and Programming Model

In this section, we describe the software support for PIM-DRAM. Specifically, we first describe the PIM software stack that allows users to run unmodified source code based on a popular ML framework such as TensorFlow. Subsequently, we present a programming model with a code example.

A. Software Stack

To run existing ML applications on a system adopting PIM-DRAM without any change in their original source code, we adapt the existing ML software stack illustrated in Fig. 6; the blue ones are part of the original software stack while the orange ones are additionally implemented for PIM-DRAM. First, the PIM device driver reserves memory space for PIM operations (the gray region in Fig. 3) during the booting process. It also sets the reserved memory space to an uncacheable region so that the host processor sends a DRAM command for every memory access to the PIM memory space. Receiving a request from an upper software layer, the PIM device driver allocates physically contiguous memory blocks. This allows us not to worry about virtual-physical address translations for PIM kernels. Second, the PIM BLAS supports a set of

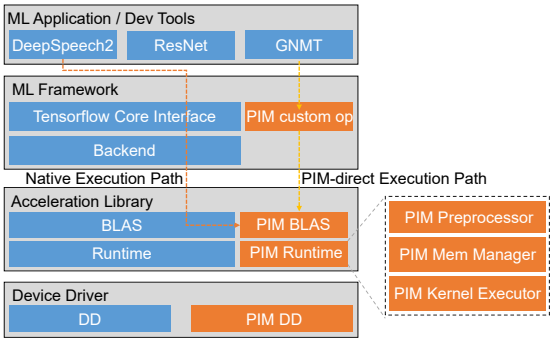

Fig. 6: PIM software stack. The native execution path does not require any modification of application source code (orange arrow). The PIM-direct execution path needs to explicitly use PIM TF custom ops (yellow arrow). 'DD' denotes device driver.

common linear algebra operations that can exploit PIM such as vector-scalar or vector-matrix multiplication. The PIM BLAS is implemented on the PIM runtime and it makes users access and utilize the PIM execution unit without knowing how to handle PIM. Third, the PIM runtime is a set of user-level modules used by PIM BLAS functions. Specifically, it consists of three modules: (1) preprocessor, (2) memory manager and (3) executor.

The PIM preprocessor analyzes the source code of applications and finds TensorFlow (TF) ops suitable for PIM acceleration at runtime. After identifying such TF ops, it maps associated operand data to memory space in a PIM-friendly way and generates PIM microkernel code. The PIM memory manager governs the memory allocated by the PIM device driver, and stores not only generated PIM microkernel code (or CRF commands) but also the operand data in cache area for later use. The PIM executor configures and invokes a PIM kernel.

Lastly, PIM BLAS functions can also be called directly by TF "PIM custom ops," which we implement as part of the TensorFlow framework for explicit and manual use of PIM operations. We currently support six custom TF operations (ADD, MUL, ReLU, LSTM, GEMV, and BN). Fig. 7 illustrates the execution stack starting with GEMV PIM custom op. The TF PIM custom op directly calls the corresponding function in the PIM BLAS library. Subsequently, the PIM BLAS function

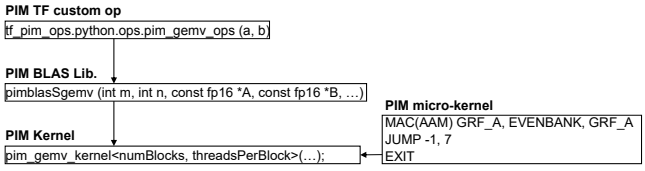

Fig. 7: PIM-direct execution path.

calls the PIM kernel, which sets up a microkernel in the CRF and starts to send DRAM commands to execute the PIM microkernel. For example, the `GEMV` PIM microkernel consists of only two PIM instructions: (1) MAC GRF_A, EVENBANK, GRF_A and (2) JUMP. MAC will be executed 8 times as JUMP is set up to repeat the loop 8 times.

B. Programming Model

A normal compute kernel performs all the operations in a device, as illustrated in Fig. 8(a). On the other hand, a PIM kernel needs to send memory requests to DRAM such that the PIM microkernel code in PIM-DRAM can be properly executed, as depicted in Fig. 8(b). Therefore, an important aspect of PIM kernel programming is to generate memory requests to the correct addresses, ensure the order of those memory requests, and fully utilize the on-chip compute bandwidth of PIM-DRAM.

To fully utilize the compute throughput of PIM-DRAM, it is necessary for a PIM kernel to send a sufficient number of memory requests to DRAM. Thus, it is natural to create multiple threads, each of which sends 8 memory requests to a contiguous memory region of 256 bytes (8 accesses $\times$ 32 bytes per access) to fully utilize the GRF storing operands for arithmetic operations. For example, if the maximum memory access size of the memory access APIs determined by a given processor ISA is 16 bytes, we need 16 threads to generate memory requests for accessing 256 bytes at a time, as illustrated in Fig. 8(c). All 16 threads are allocated to one thread group, which is executed in a lockstep manner, runs the same set of instructions, and follows the same control-flow path, as depicted in Fig. 8(d). Therefore, all threads in the same thread group concurrently send memory requests to the memory controller. To synchronize the threads and guarantee the order of memory requests, we use `barrier` APIs. Since `barrier` enforces an ordering constraint on memory requests issued by threads belonging to the same thread group, we let

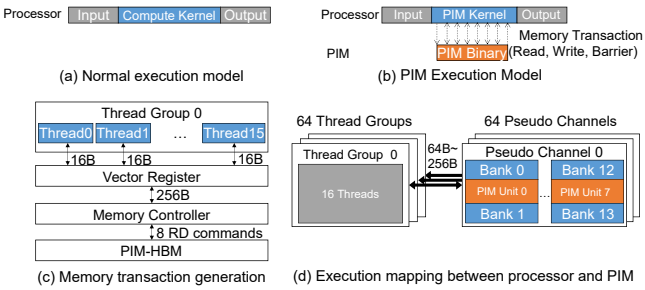

Fig. 8: A programming model for a processor exploiting PIM.

each thread group exclusively access single DRAM channel, minimizing unnecessary fence overhead between memory requests to different channels. For instance, we implement a PIM kernel that allocates 64 thread groups for PIM-HBM because there are 64 pCHs in 4 HBM2 cubes (16 pCHs each). As described earlier, each thread group comprises 16 threads, resulting in a total of 1,024 threads.

VI. Chip Implementation and Integration with a System

We design the proposed PIM architecture based on HBM2 [51], fabricate it with a 20nm DRAM process, and rigorously go through every post-manufacturing step to meet a product quality standard as a commercial engineering sample. Tables IV and V depict the specification of the PIM execution unit and PIM-HBM device, respectively. First, we first start with designing a PIM execution unit. The PIM execution unit consists of 16 16-bit SIMD lanes, each with a FP multiplier, a FP adder, a 32-entry CRF, a 16-entry GRF and a 16-entry SRF. It is implemented with approximately 200,000 logic gates consuming $0.712\,\mathrm{mm}^2$ space. It is designed to operate at the same frequency as the HBM2 DRAM (250MHz$\sim$300MHz), delivering up to 9.6GFLOPS of throughput. Note that the operating frequency of HBM2 DRAM is 4$\times$ slower than the memory bus frequency (1.0GHz$\sim$1.2GHz).

As a drop-in-replacement of HBM, PIM-HBM needs to keep the same physical dimension as HBM (10.75mm $\times$ 9.75mm for HBM2) and preserve the mechanical compatibility with the existing 2.5D SiP. To make space for PIM execution units, we reduce the number of DRAM sub-arrays by half. Besides, we have a PIM execution unit shared by two banks. These allow us to place a total of 32 PIM execution units as a PIM-HBM DRAM die has 4 pCHs and a pCH is connected to 16 banks like the standard HBM DRAM die (8 PIM execution units per pCH $\times$ 4 pCHs per PIM-HBM DRAM die). A photo of a fabricated PIM-HBM DRAM die is shown in Fig. 9(a).

Subsequently, the fabricated PIM-HBM DRAM dies are 3D-stacked with a buffer die, as illustrated in Fig. 9(b) where 4

TABLE IV: Specification of PIM execution unit.

# of MUL/ADD FPUs	16/16
Datapath Width	256 bits (16 bits $\times$ 16 lanes)
Operating Frequency	250MHz $\sim$ 300MHz
Throughput	9.6 GFLOPs at 300MHz
Equivalent Gate Count	200,000 (only logic)
Instruction Registers	32b $\times$ 32 (CRF)
Vector and Scalar Registers	256b $\times$ 16 (GRF), 16b $\times$ 16 (SRF)
Area	$0.712\,\mathrm{mm}^2$

TABLE V: Specification of PIM-HBM Device.

Ext. Clocking Frequency	1$\sim$1.2GHz
Timing Parameters	Same as HBM2
# of pCHs	16
# of banks per pCH	16
# of PIM exe. units per pCH	8
On-Chip (Compute) Bandwidth	1TB/s$\sim$1.229TB/s
Off-Chip (I/O) Bandwidth	256GB$\sim$307.2GB/s
Capacity	6GB
Area of DRAM Die	$84.4\,\mathrm{mm}^2$

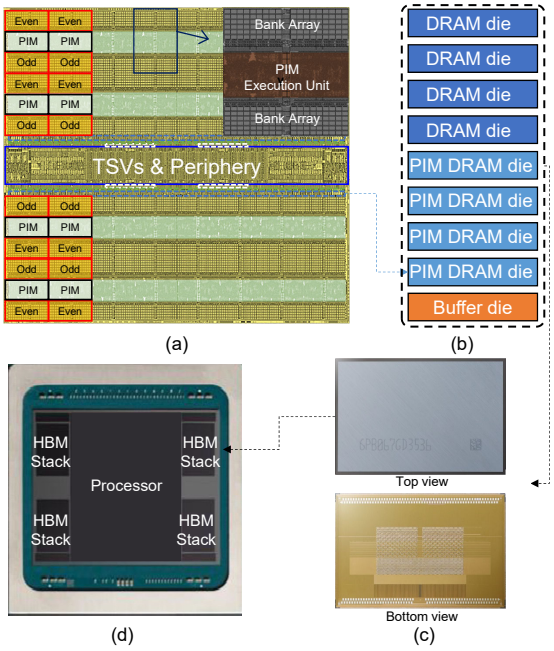

Fig. 9: PIM-HBM implementation and SiP 2.5D-integrating PIM-HBM devices with a processor: (a) a photo of a fabricated PIM-HBM die, (b) a cross-section view of 3D-stacked PIM-HBM, (c) a photo of a manufactured PIM-HBM package, (d) a photo of a processor SiP.

PIM-HBM DRAM dies are 3D-stacked on a buffer die, and then four HBM DRAM dies were 3D-stacked atop the PIM-HBM dies. This gives a total of 6GB capacity (4 $\times$ 4Gb PIM-HBM dies + 4 $\times$ 8Gb HBM dies), 1.229TB/s on-chip compute bandwidth (1.2Gbps $\times$ 64 bits per bank $\times$ 8 operating banks per pCH $\times$ 16 pCHs per device), and 307GB/s I/O bandwidth (2.4 Gbps $\times$ 64 bits per bank $\times$ 1 operating banks per pCH $\times$ 16 pCHs per device).

Fig. 9(c) shows a photo of a fully assembled and packaged PIM-HBM device (or stack) in a micro pillar grid array (MPGA) package. The PIM-HBM's technical specifications seen by the host processor such as ball map, signaling, and DRAM timing parameters are precisely the same as conventional HBM2 [51]. Before 2.5D-integrating PIM-HBM devices with the processor, we also design and build an FPGA-based system to verify whether or not PIM-HBM is compatible with a JEDEC-compliant memory controller and functionally correct. The FPGA-based system comprises a Xilinx Zynq Ultrascale+ FPGA and a socket to mount a package that encapsulates an HBM MPGA package. Note that a stand-alone HBM device in an MPGA package cannot be directly mounted on a PCB due to its small ball size. As such, it needs to be integrated on a silicon interposer and packaged in a conventional package before it is mount on a PCB. As we can precisely control the operation of PIM-HBM with this system, we also use it to measure power and energy consumption of PIM-HBM.

Lastly, PIM-HBM devices are 2.5D-integrated with a processor die, as shown in Fig. 9(d). For evaluations, we integrate four PIM-HBM devices with an *unmodified* high-end processor with 60 compute units, each operating at 1.725GHz. That is, the total off-chip memory bandwidth for the processor is 1.229TB/s while the total on-chip compute bandwidth for PIM execution units is 4.915TB/s.

VII. Evaluation

A. Benchmark

To evaluate performance and energy efficiency of PIM-HBM-based systems, we first run two types of microbenchmarks: vector-matrix multiplication (`GEMV`) and element-wise addition (`ADD`) frequently used for computing residual connections. For evaluations of the microbenchmarks, we apply various input sizes obtained from the real-world applications to the systems (Table VI). Furthermore, for end-to-end evaluations of performance and energy consumption, we choose five representative ML applications: three NLP applications (Baidu's DeepSpeech2 (`DS2` [4]), Google's RNN Transducer (`RNN-T` [20]) and Google's Neural Machine Translation System (`GNMT` [53])) and two popular CV applications (`AlexNet` [38] and `ResNet` [18]). Note that the embedding look-up layer of recommendation models is memory-bound but it also requires a large memory capacity (e.g., 256GB [15]). Thus, processors integrated with HBM are not suitable for running such applications as they provide limited memory capacity (e.g., 32GB with 4 HBM devices). For evaluation, we use a batch size of 1, 2, and 4. Note that a larger batch size gives more data reusability (i.e., higher LLC hit rates), which makes a given application more compute-bound and increases throughput. Nonetheless, it also increases response time. Hence, we focus on discussing the evaluation of applications with a batch size of 1 as we consider the target use of PIM-HBM systems for memory-bound, latency-sensitive applications such as commercial online services.

`DS2` consists of 2 convolution layers, 6 bidirectional LSTM layers, and a fully connected layer. `RNN-T`, another speech recognition model, is a combination of three types of networks: 5 LSTM encoder layers with dropout, 2 LSTM prediction layers with dropout, and 2 fully connected joint-network layers with ReLU/dropout; we use a variant of the model included in the `MLPerf` benchmark suite [46]. For input data of two speech recognition models, we use a linear spectrogram extracted from a voice clip of 2 seconds. `GNMT` consists of 8 LSTM encoders, 8 LSTM decoders, and an attention layer. In our evaluation, we use sentences, each with approximately 50 words, as input. `AlexNet` comprises 5 convolution layers and 3 fully connected layers. `ResNet-50` has 50 layers, most of which perform 3$\times$3 and 1$\times$1 convolution operations,

TABLE VI: Microbenchmark.

Name	GEMV Dim.	Name	ADD Dim.
GEMV1	1k $\times$ 4k	ADD1	2M
GEMV2	2k $\times$ 4k	ADD2	4M
GEMV3	4k $\times$ 8k	ADD3	8M
GEMV4	8k $\times$ 8k	ADD4	16M

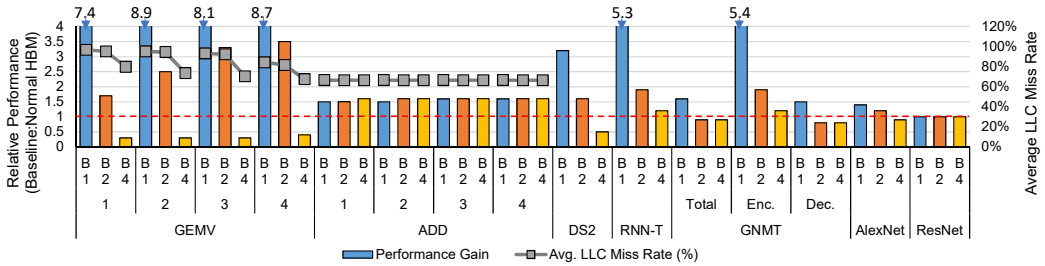

Fig. 10: Relative performance and average LLC miss rates with batch size of 1, 2, and 4 (denoted by B1, B2, and B4); the LLC miss rates for DS2, RNN-T, GMNT, AlexNet, and ResNet consisting of multiple kernels cannot be reported because the performance monitoring tool can report a LLC miss rate of only a single kernel at a time.

with a identify shortcut connection which skips one or more layers. Although most layers of both AlexNet and ResNet are compute-bound, which are not a target for PIM, we still evaluate them for the completeness of evaluation. For input to CV models, we use images, each with 224×224×3 size. Finally, we accelerate the LSTM layers of DS2, RNN-T, and GNMT, and the fully connected layers of AlexNet.

B. Performance

Fig. 10 compares the performance of the HBM- and PIM-HBM-based systems running the microbenchmarks and the ML applications with various batch sizes. First, we analyze the performance of the microbenchmarks with a single batch size. The PIM-HBM-based system gives 1.4∼11.2× higher performance than HBM-based one for the microbenchmarks. Especially, PIM-HBM improves the performance of GEMV by up to 11.2× compared to HBM. The performance improvement is significantly greater than the on-chip compute bandwidth increase with PIM-HBM because GEMV provided by the software stack of the processor is not optimized to fully utilize the off-chip memory bandwidth of HBM. On the other hand, PIM-HBM improves the performance of ADD by only 1.6× over HBM. This is because the ADD kernel uses both operands only once and the computed result should be stored to the bank after 8 ADD instructions, which is limited by the number of GRF registers. That is, the host processor needs to execute a fence instruction after PIM-HBM executes 8 PIM instructions; as discussed in Section IV-C, and the number of PIM instructions, which can be executed out of order, is limited to the number of registers in GRF. We discuss how we can reduce the overhead of frequently executing fence instructions later in this section.

For DS2, GNMT, and AlexNet, PIM-HBM gives 3.5×, 1.5×, and 1.4× higher performance than HBM, respectively. PIM-HBM considerably improves the performance of DS2 compared to HBM, as DS2 spends most of the time to execute the LSTM layers which are very suitable for PIM acceleration. For GNMT, the LSTM decoder is required to invoke the PIM kernel at every step and every layer because the output of the last layer of the previous step becomes the input of the first layer of the next step. Hence, the overhead caused by many kernel calls limits the performance improvement of GNMT. On the other hand, the inputs of all steps of the LSTM encoder are available at the beginning, which can reduce the number of kernel calls for the encoder. As a result, we observe that PIM-HBM improves the LSTM encoder performance by 6.2× compared to HBM. Although the convolutional layers dominate the execution time of AlexNet, PIM-HBM still improves the performance by accelerating the fully connected layer. For ResNet-50, PIM-HBM gives the same performance as HBM because the execution time of ResNet-50 is dominated by that of convolution layers, which are compute-bound. This is to demonstrate the PIM-HBM does not hurt the performance of compute-bound applications.

As the batch size increases, we observe that the performance improvement with PIM-HBM decreases. For example, PIM-HBM improves the performance of GEMV by 11.2× and 3.2× for batch size of 1 and 2, respectively, compared to HBM. However, for batch size of 4, the processor with HBM begins to outperform one with PIM-HBM as it becomes less memory-bound with more reused data in LLC. We can confirm this with the LLC miss rates that decrease from almost ∼100% to 70–80% (Fig. 10). In general, batching transforms the (memory-bound) level-2 BLAS (GEMV) to the (compute-bound) level-3 BLAS (GEMM). That is, PIM-HBM does not improve performance if the batch size is 4 or larger in our evaluation. Meanwhile, ADD, which is the level-1 BLAS, is still memory-bound regardless of the batch size as it becomes the level-2 BLAS with batching. In summary, for batch size of 2, PIM-HBM still gives 1.6× and 1.9× higher performance than HBM for DS2 and RNN-T, respectively, but it provides lower performance for other ML applications.

In our evaluation above with an unmodified processor, we need to use a barrier for every 8 DRAM commends to guarantee the correct execution order of PIM instruction because our AAM can handle out-of-order execution of only up to 8 PIM instructions at a time (Section IV-C). The overhead of using fence instructions prevents us from exploiting the full potential of PIM-HBM. If a processor can guarantee the order of DRAM commands in PIM mode, our evaluation after removing fence instructions shows that PIM-HBM can offer 2.2×, 1.9×, and 2.0× higher performance than HBM for microbenchmarks with batch size of 1, 2, and 4, respectively. Note that a processor manufacturer confirms that the order

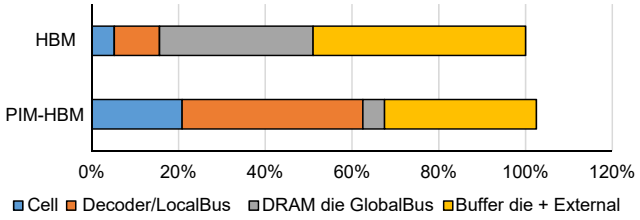

Fig. 11: Power breakdown of HBM and PIM-HBM over back-to-back DRAM RD commands. Both HBM and PIM-HBM are operating at 2.4Gbps and 85°C with random FP16 numbers.

of DRAM commands can be preserved only in PIM mode at negligible hardware and performance costs.

Lastly, FPGA-based RNN accelerators such as Brainwave [8] provide high performance, but they evaluate microbenchmarks with data sets that fit into on-chip SRAM in FPGA (e.g., ARRIA-10 with total 67Mb SRAM). Graphcore also relies on large capacity and high bandwidth of SRAM that stores all the data necessary for processing. Meanwhile, PIM-HBM based on larger capacity DRAM targets applications and kernels with data sets that do not fit into SRAM.

C. Power and Energy

At first, we suppose that the PIM execution units and their parallel accesses of banks may significantly increase the power consumption of PIM-HBM compared to HBM. Measuring power consumption of the fabricated PIM-HBM chips, however, we observe that the PIM-HBM consume only 5.4% higher power even with 4× higher (on-chip) bandwidth than HBM, as shown in Fig. 11. In PIM-HBM, since multiple PIM execution units, each coupled with a bank, operate at the same time, the power consumption of DRAM internal components such as cell and IOSA/decoders, which are depicted by the blue and orange colors in Fig. 11, increases proportionally. Nonetheless, the power consumption of internal global I/O buses, most of which is represented by the gray color, and I/O PHYs, some of which is represented by the yellow color, considerably decreases. In summary, the power consumption of PIM-HBM is slightly higher than that of HBM, staying within the thermal design power (TDP) limit set by the original HBM-based system. Note that we could have made the power consumption of PIM-HBM ∼10% lower than that of the HBM if we implemented a feature eliminating unnecessary power consumption by the buffer die's 1024-bit data I/O circuit that does not need to toggle in PIM mode (the orange color in Fig. 11). Therefore, PIM-HBM can also offer a thermal advantage over HBM.

Fig. 12 shows power and energy consumption of PIM-HBM, PROC-HBM, and PROC-HBM×4. PIM-HBM and PROC-HBM represent a processor integrated with 4 PIM-HBM and 4 HBM devices, respectively. PROC-HBM×4 denotes a hypothetical processor with 4 times more HBM devices than PROC-HBM. Using the system described in Section VI, we measure the power consumption and execution time of PIM-HBM and PROC-HBM. We estimate the power consumption

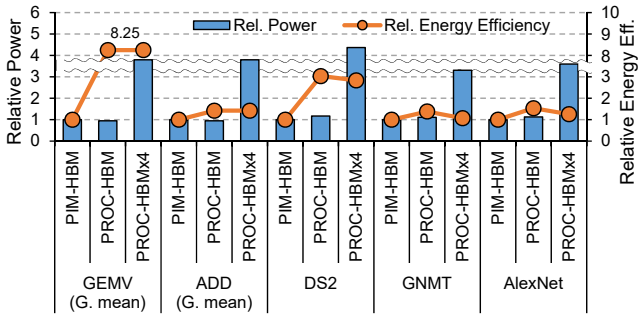

Fig. 12: The relative power and energy consumption of processors with HBM, PIM-HBM, and HBM×4.

and execution time of PROC-HBM×4 after breaking down those of PROC-HBM and considering the effect of increasing the number of HBM devices by 4 times.

For `GEMV`, PIM-HBM gives 8.25× higher energy efficiency than PROC-HBM. PROC-HBM×4 shows energy efficiency similar to PROC-HBM, as the system's power consumption and performance increase proportionally with higher bandwidth for memory-bound applications. PIM-HBM performing `ADD` shows relatively lower energy efficiency than performing `GEMV` operations, 1.4× improvement, because it provides only 1.6× performance improvement for `ADD`.

For end-to-end execution of applications, the improvement in energy efficiency is smaller than microbenchmarks because PIM operations cannot be applied to all the layers and other essential parts of the software stack. For `DS2`, `GNMT`, and `AlexNet`, PIM-HBM gives 3.2×, 1.38×, and 1.5× higher energy efficiency than PROC-HBM, respectively. For the three ML applications, PIM-HBM gives 2.8×, 1.1×, and 1.3× higher energy efficiency than PROC-HBM×4, respectively. Lastly, Fig. 13 shows the measured average power consumption of `DS2` over time. This demonstrates that PIM-HBM improves energy efficiency not only with shorter execution time but also with lower average power consumption.

D. Design Space Exploration

As part of design space exploration, we also evaluate other PIM (micro)architectures that could not be implemented due to constraints such as die size, pin compatibility, timing, and use of a JEDEC-compliant DRAM controller for the time being. For evaluation of such PIM (micro)architectures, we perform simulations with a modified version of DRAMSim2 [48]. Since simulations using DRAMSim2 do not model the per-

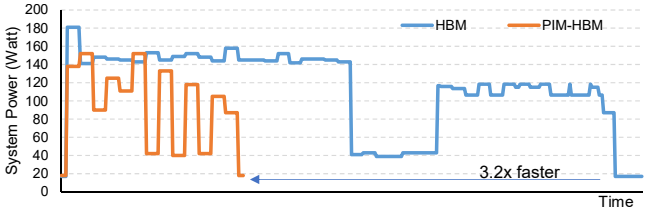

Fig. 13: Average system power of `DS2` over time.

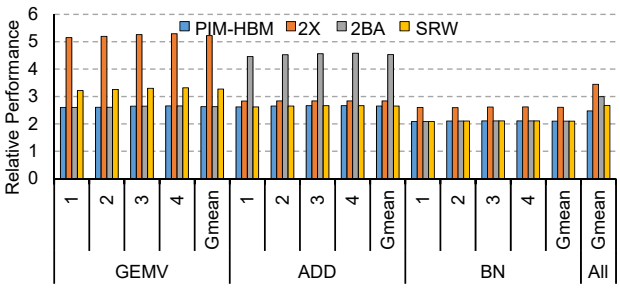

Fig. 14: The performance improvement of a processor with PIM-HBM-2×, - 2BA, and -SRW over the processor with HBM.

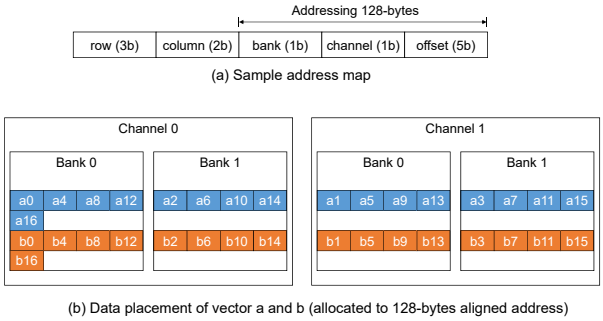

Fig. 15: Data layout for PIM ADD.

formance of the host processor, estimated performance values are theoretical upper-bound but close to the true performance for very memory-bound kernels. Specifically, we simulate the performance of three more enhanced PIM architectures: a PIM execution unit that can (1) provide 2× more resources (denoted by PIM-HBM-2×); (2) access EVEN_ BANK and ODD_ BANK at the same time to get two operands for one PIM instruction (PIM-HBM-2BA where 2BA stands for 2 bank access); and (3) get a 32-byte data sent over the DRAM write datapath by a WR command and another 32-byte data from the column address of the EVEN_BANK or ODD_BANK (PIM-HBM-SRW where SRW stands for simultaneous DRAM column RD and WR commands).

Fig. 14 shows the performance improvement of a processor with PIM-HBM-2×, - 2BA, and -SRW over the processor with HBM, where we also evaluate a batch-normalization kernel (BN) with the same input size as ADD. PIM-HBM-2× gives ∼40% higher geo-mean performance than PIM-HBM, but it increases the die size by 24%. PIM-HBM-2BA offers ∼20% higher geo-mean performance. It is useful especially for ADD because it reduces the performance bottleneck by the limited number of GRF registers. It does not notably increase the die size, but it consumes 60% more power than PIM-HBM. PIM-HBM-SRW provides only ∼10% higher geo-mean performance than PIM-HBM, but it offers ∼25% higher performance especially for GEMV because it does not need to write the vector to GRF registers first with a DRAM column WR command and then execute the operation with a subsequent DRAM column RD command.

VIII. Discussion

In this section, we discuss various challenges, how we tackled some of them in this work, and how we can better handle them in the future.

Cache Bypassing: PIM requires data to be located in memory. Thus, we need to make memory regions that PIM operates on uncacheable or flush cached data associated with the memory regions to memory before PIM starts to operate on the data. These approaches, however, can lead to notable performance degradation. To reduce such performance degradation, we use cache bypass instructions (e.g., LDNP/STNP in ARMv8) that directly send write requests to memory through a write-combining buffer. Note that PIM targets to accelerate applications with large data sets that do not fit into caches. As such, making such memory regions uncacheable in fact reduces interference and contention at caches and thus improves the performance when such data can be processed by PIM.

ECC, Virtualization and Multi-tenancy: Our current PIM-HBM does not support ECC yet. However, future PIM based on the proposed architecture can easily support ECC as each PIM execution unit reads and writes data at the same the data access granularity as a host processor. In addition, DRAM began to have on-die ECC including HBM3. Thus, PIM may leverage the on-die ECC engine to generate and check the ECC parity bits even in PIM mode. Currently, we developed an ECC scheme for an HBM3-generation PIM-HBM with the company that integrated PIM-HBM with its unmodified processor. Lastly, PIM-HBM can support virtualization and multi-tenancy at some degrees since it allows a processor to independently controls PIM operations of each memory channel.

Memory Interleaving and Data Layout: As discussed earlier, (1) the host processor can independently control PIM operations of each memory channel, and (2) a PIM execution unit accesses the memory at the same data access granularity as a host processor. These two aspects make our PIM architecture more agnostic to a physical address mapping scheme of the host processor than prior PIM architectures. For example, suppose a physical address mapping scheme depicted in Fig. 15(a). In case of ADD, two operands can be allocated at the 128-byte aligned boundary as illustrated in Fig. 15(b). This is the same data layout as a standard processor except that the size of vectors should be multiple of 128 bytes. If the size of vectors is not multiple of 128 bytes, we can concatenate dummy values to the end of the vectors. Even so, the overhead or inefficiency is negligible as we target very large vectors that do not fit into cache. To maximize the performance of GEMV, however, our PIM architecture may still require some changes in data layout and/or software needs to be aware of the physical address mapping scheme; currently PIM BLAS APIs automatically rearrange data layout when the host processor brings weight matrix values to memory. Note that we expect to shun such data rearrangement for GEMV in

a HBM3-generation PIM-HBM architecture that will support fine-grained interleaving between SB and AB-PIM modes. In such a PIM architecture, we see an opportunity that both the host processor and PIM can perform GEMV in a collaborative way and eliminate the need for data layout rearrangement. We will leave such an approach as future work.

IX. Related Work

Many researchers have explored diverse PIM architectures so far (e.g., [2], [12], [19], [33]–[35], [49], [54]). These architectures share a common concept, exploiting the array- and/or bank-level parallelism to expose the abundant on-chip bandwidth of DRAM to processors. However, the past PIM architectures including the most recent one exploiting bank-level parallelism (dubbed Newton [19]) lack the following considerations for practicality and wide industry adoption.

Interface between Host and DRAM: The interfaces between commercial processors and DRAM devices are defined by JEDEC standards [25]–[28]. To adopt custom DRAM devices that do not follow the standard interfaces, host processors also need to be customized. The engineering cost and time of not only customizing both the processors and the DRAM devices but also building necessary ecosystems will inevitably hinder wide adoption by the industry which is often driven by the cost and time to market. That is, the compatibility with the current standard DRAM interfaces is essential to successfully and quickly deploy PIM-DRAM devices to commercial systems.

Software Stack: Prior PIM work [2], [12], [19], [35], [49] discusses necessary programming models and software stacks, but it only approximates the effects of the programming models and software stacks on performance without implementing them. In contrast, the software stack for our PIM architecture is fully implemented to precisely evaluate the performance effects of (1) performing non-cacheable memory accesses; (2) translating a virtual address to a physical one [22] and correctly accessing a target DRAM bank, row, and column of the (interleaved or scrambled) physical address; (3) maintaining the ordering of DRAM commands to ensure the functional correctness [47]; and (4) executing existing applications without or with minimal modifications of the source code.

Lastly, comparing our work with Newton in depth, we see that Newton accelerates only matrix-vector multiplication with MAC units that are placed near DRAM banks. This limits applications that can benefit from the architecture. Furthermore, it requires special DRAM commands that are not part of the JEDEC standard, which increases the complexity of DRAM controller and DRAM designs due to more CA pins and associated states than the standard DRAM interfaces. As such, it will lose the compatibility with existing JEDEC-compliant DRAM controllers. More importantly, unlike our work, Newton has been neither implemented with a commercial DRAM technology, integrated with a commercial processor, nor supported by a necessary software stack for full system-level evaluations yet.

X. Conclusion

In this paper, we proposed a practical yet innovative PIM architecture that can seamlessly work with unmodified commercial processors as a drop-in-replacement of standard DRAM. To demonstrate its practicality and efficacy at the system level, we implemented the proposed PIM architecture based on a commercial HBM2 DRAM die design, fabricated it with a 20nm DRAM technology, integrated the fabricated PIM-HBM with an unmodified commercial processor, and developed the necessary software stack. Our system-level evaluation showed that PIM reduced the end-to-end execution time of memory-bound neural network kernels and applications by up to 11.2× and 3.5×, respectively, and improved the energy efficiency of the system running the applications by up to 3.2×. This work is the first silicon implementation, software stack development, and system-level evaluation by a major memory manufacturer, and it opens the door for wider adoption and further development of PIM by other industry players.

References

[1] "TensorFlow," https://github.com/tensorflow.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Int'l Symp. on Computer Architecture (ISCA)*, 2015.

[3] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim, "Application-Transparent Near-memory Processing Architecture with Memory Channel Network," in *Int'l Symp. on Microarchitecture (MICRO)*, 2018.

[4] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen, J. Chen, Z. Chen, M. Chrzanowski, A. Coates, G. Diamos, K. Ding, N. Du, E. Elsen, J. Engel, W. Fang, L. Fan, C. Fougner, L. Gao, C. Gong, A. Hannun, T. Han, L. Johannes, B. Jiang, C. Ju, B. Jun, P. LeGresley, L. Lin, J. Liu, Y. Liu, W. Li, X. Li, D. Ma, S. Narang, A. Ng, S. Ozair, Y. Peng, R. Prenger, S. Qian, Z. Quan, J. Raiman, V. Rao, S. Satheesh, D. Seetapun, S. Sengupta, K. Srinet, A. Sriram, H. Tang, L. Tang, C. Wang, J. Wang, K. Wang, Y. Wang, Z. Wang, Z. Wang, S. Wu, L. Wei, B. Xiao, W. Xie, Y. Xie, D. Yogatama, B. Yuan, J. Zhan, and Z. Zhu, "Deep Speech 2: End-to-end Speech Recognition in English and Mandarin," in *Int'l Conf. on Machine Learning (ICML)*, 2016.

[5] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "CuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, 2014.

[7] K. Cho, H. Lee, and J. Kim, "Signal and Power Integrity Design of 2.5 D HBM (High Bandwidth Memory Module) on SI Interposer," in *Pan Pacific Microelectronics Symp.*, 2016.

[8] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, no. 2, 2018.

[9] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating Reduction and Scan Using Tensor Core Units," in *Int'l Conf. on Supercomputing (ICS)*, 2019.

[10] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Design & Test of Computers*, vol. 16, no. 1, 1999.

[11] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-data Processing for In-memory Analytics Frameworks," in *Int'l Conf. on Parallel Architecture and Compilation (PACT)*, 2015.

[12] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[13] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *Computer*, vol. 28, no. 4, 1995.

[14] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference," in *Int'l Symp. on Computer Architecture (ISCA)*, 2020.

[15] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation," in *Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2020.

[16] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-precision Iterative Refinement Solvers," in *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018.

[17] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-intensive Architecture," in *Conf. on Supercomputing (SC)*, 1999.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[19] 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 *Int'l Symp. on Microarchitecture (MICRO)*, 2020.

[20] 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 *Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2019.

[21] Intel, *oneAPI Deep Neural Network Library*, https://github.com/oneapi-src/oneDNN.

[22] Intel, "Intel Virtualization Technology for Directed I/O Architecture Specification," 2006.

[23] Intel, *Intel® oneAPI Math Kernel Library.*, 2020, https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[24] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv preprint arXiv:1502.03167*, 2015.

[25] JEDEC, "High Bandwidth Memory (HBM) DRAM," JESD235, 2013.

[26] JEDEC, "Graphics Double Data Rate 6 (GDDR6) SGRAM," JESD250B, 2018.

[27] JEDEC, "DDR5 SDRAM," JESD79-5, 2020.

[28] JEDEC, "Low Power Double Data Rate (LPDDR5)," JESD209-5A, 2020.

[29] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[30] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *Int'l Conf. on Computer Design (ICCD)*, 1999.

[31] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, 2011.

[32] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, "MIOpen: An Open Source Library For Deep Learning Primitives," 2019.

[33] B. Kim, J. Chung, E. Lee, W. Jung, S. Lee, J. Choi, J. Park, M. Wi, S. Lee, and J. Ahn, "MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks," *IEEE Transactions on Computers*, vol. 69, no. 7, 2020.

[34] B. Kim, J. Park, E. Lee, M. Rhu, and J. H. Ahn, "TRiM: Tensor Reduction in Memory," *IEEE Computer Architecture Letters*, 2020.

[35] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-density 3D Memory," in *Int'l Conf. on Computer Architecture (ISCA)*, 2016.

[36] J. Kim and Y. Kim, "HBM: Memory Solution for Bandwidth-Hungry Processors," in *Hot Chips Symp. (HCS)*, 2014.

[37] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Int'l Conf. on Neural Information Processing Systems (NIPS)*, 2012.

[39] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Int'l Conf. on Machine Learning (ICML)*, 2010.

[40] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *arXiv preprint arXiv:1906.00091*, 2019.

[41] Netlib, *Basic Linear Algebra Subprograms*, http://www.netlib.org/blas.

[42] NVIDIA, "CUDA C++ Programming Guide," 2020.

[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*, 2019.

[44] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, vol. 17, no. 2, 1997.

[45] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-stacked Memory+Logic Devices on MapReduce Workloads," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[46] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "MLPerf Inference Benchmark," in *Int'l Symp. on Computer Architecture (ISCA)*, 2020.

[47] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, 2000.

[48] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, 2011.

[49] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, 2018.

[50] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[51] 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 Journal of Solid-State Circuits*, vol. 52, no. 1, 2016.

[52] S. Wang and P. Kanwar, "BFloat16: The secret to high performance on Cloud TPUs," https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, 2019.

[53] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[54] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. S. Kim, "In-DRAM near-Data Approximate Acceleration for GPUs," in *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2018.