Energy-efficient In-Memory Address Calculation

Yousefzadeh, Amirreza; Stuijt, Jan; Hijdra, Martijn; Liu, Hsiao-Hsuan; Gebregiorgis, Anteneh; Singh, Abhairaj ; Hamdioui, Said; Catthoor, Francky

DOI
10.1145/3546071

Publication date
2022

Document Version
Final published version

Published in
ACM Transactions on Architecture and Code Optimization

Energy-efficient In-Memory Address Calculation

AMIRREZA YOUSEFZADEH, JAN STUIJT, and MARTIJN HIJDRA, IMEC, Netherlands
HSIAO-HSUAN LIU, IMEC, Belgium
ANTENEH GEBREGIORGIS, ABHAIRAJ SINGH, and SAID HAMDIOUI, Delft University of Technology, Netherlands
FRANCKY CATTHOOR, IMEC, Belgium

Computation-in-Memory (CIM) is an emerging computing paradigm to address memory bottleneck challenges in computer architecture. A CIM unit cannot fully replace a general-purpose processor. Still, it significantly reduces the amount of data transfer between a traditional memory unit and the processor by enriching the transferred information. Data transactions between processor and memory consist of memory access addresses and values. While the main focus in the field of in-memory computing is to apply computations on the content of the memory (values), the importance of CPU-CIM address transactions and calculations for generating the sequence of access addresses for data-dominated applications is generally overlooked. However, the amount of information transactions used for "address" can easily be even more than half of the total transferred bits in many applications. In this article, we propose a circuit to perform the in-memory Address Calculation Accelerator. Our simulation results showed that calculating address sequences inside the memory (instead of the CPU) can significantly reduce the CPU-CIM address transactions and therefore contribute to considerable energy saving, latency, and bus traffic. For a chosen application of guided image filtering, in-memory address calculation results in almost two orders of magnitude reduction in address transactions over the memory bus.

CCS Concepts: • **Hardware** → *Semiconductor memory*; **Memory and dense storage**; *Power estimation and optimization*; *Emerging architectures;*

Additional Key Words and Phrases: In-memory processing, address calculation unit, energy optimization

ACM Reference format:
Amirreza Yousefzadeh, Jan Stuijt, Martijn Hijdra, Hsiao-Hsuan Liu, Anteneh Gebregiorgis, Abhairaj Singh, Said Hamdioui, and Francky Catthoor. 2022. Energy-efficient In-Memory Address Calculation. *ACM Trans. Archit. Code Optim.* 19, 4, Article 52 (September 2022), 16 pages.
https://doi.org/10.1145/3546071

This work was funded in part by EU H2020 grant 780215 "MNEMOSENE" and in part by DAIS (KDT JU under grant agreement No. 101007273). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, Spain, Portugal, Belgium, Germany, Slovenia, Czech Republic, Netherlands, Denmark, Norway, and Turkey.
Authors' addresses: A. Yousefzadeh, J. Stuijt, and M. Hijdra, IMEC, Eindhoven, Netherlands; email: Amirreza.Yousefzadeh@imec.nl; H.-H. Liu and F. Catthoor, IMEC, Leuven, Belgium; A. Gebregiorgis, A. Singh, and S. Hamdioui, Delft University of Technology, Delft, Netherlands.

1544-3566/2022/09-ART52
https://doi.org/10.1145/3546071

1 INTRODUCTION

The conventional von-Neumann architecture suffers from the memory access bottleneck as a result of the separation between memory, and the processor [8, 21, 30]. Additionally, access to a distant memory from the processor costs a considerable amount of energy and time (10 to 100 times more than an ALU operation). This work proposes a solution to optimize memory access's energy/latency cost by reducing the non-negligible overheads of data communications.

In general for conventional systems, the communicated information between memory and the processing units consists of three main parts: *1-Instruction or Control word*, *2-Target word address*, and *3-Data Operand* where the instruction and address are considered to be overheads for each data transfer.[1] Communication of these overheads (*1-Instruction or Control word* and *2-Target word address*) can account for more than half of the total amount of transferred bits. Therefore methods to reduce this massive amount of overheads are worth investigating.

In many loop and data-dominated applications, it is required to access (near-)consecutive words in the memory (burst access). In this case, it is more efficient to send the first address and number of accesses and calculate the absolute addresses locally in the memory. Today this happens with local address generation units, including **Direct Memory Access (DMA)** engines that are located near the processor and outside of memories [1, 9, 24]. General DMA engines can perform some simple address calculations (mostly consecutive burst access). Additionally, application-specific **Address Generation Units (AGUs)** are also proposed that can perform more complex address calculations for specific applications [12, 14, 16]. Some existing approaches have also proposed to optimize the area footprint of such address calculation units for a large number of distributed memories [19, 20]. However, since all the mentioned units are located outside of memory, none can considerably reduce the overhead of address transactions over the memory bus (and therefore latency and energy consumption). However, several prior arts that cover in-memory processing have focused only on processing data inside the memory crossbar and sense amplifiers to reduce data movement [5, 25] and ignored the overhead of address transactions. In Reference [12] the authors introduced an in-memory pointer chasing accelerator to reduce inefficient and high-latency data transfers between main memory and the CPU. Therefore, architectures with a focus on the address calculation in the memory are yet to be explored.

To further facilitate this overhead, we propose to integrate **Address Calculation Accelerator (ACA)** into the memory periphery as shown in Figure 1.[2] By moving part of the functionality of the DMA into the memory, only compressed high-level instructions need to travel over the memory bus. ACA architecture is the result of a cross-layer optimization at the micro-architecture and circuit level that replaces the conventional address decoders in the memory and can generate a non-consecutive pattern of target word addresses for the pre- and post-decoders selecting the desired word- (respectively bit-) lines from the memory matrix. When needed, it can also address multiple bit-lines and word lines simultaneously, which is essential for some in-memory array computations (when multiple arguments are used in one cycle, like binary AND between two words of memory). This disruptive approach allows us to potentially remove all the address-related time and energy overhead, as shown in the experimental results. Our contributions are as follows:

[1] For example, to write a value X into the memory with address Y, the instruction is *write*, the target word address is Y and the data operand is X.

[2] In this article, CIM refers to Compute In Memory and Compute In Memory Periphery is a special kind of CIM architectures where the computation is happening in the **Periphery** of the memory without the need to modify the memory cells in the crossbar array (Figure 3) [6, 22, 29].

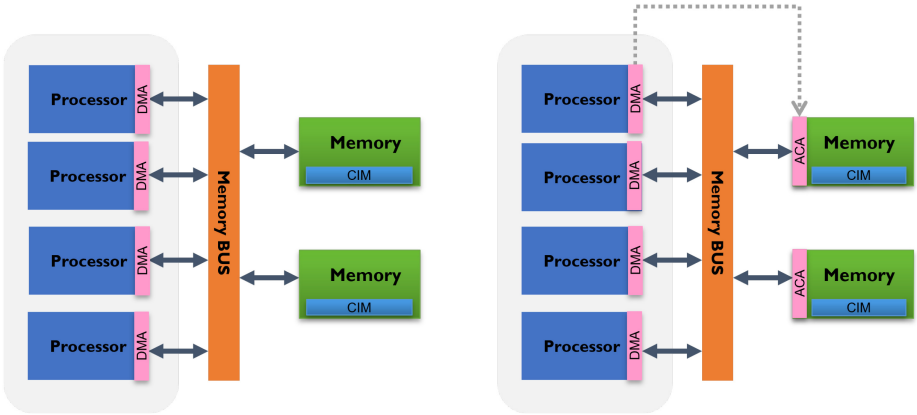

Fig. 1. Position of ACA vs. the position of the traditional DMAs in the compute platform to optimize the address transactions over the memory bus. ACA takes over part of the DMA's tasks related to address generation and replaces the conventional address decoders in the memory block. The memory chips in this figure are separated from the processor but may have some processing capabilities. CIM denotes Compute In Memory, which in this figure performs computations on the data inside the memory die [6].

- Propose a first-of-its-kind *in-situ* ACA as a **Compute In Memory Peripheral (CIM-P)** architecture, designed for embedded processor platforms, which removes the address decoder and replaces it with a direct word- and bit-line activation engine.
- Design a compiler for the proposed ACA to find the pattern of target word addresses automatically.
- Design a modular and plug-in hardware-aware simulator (nano-simulator) to include our **Compute In-Memory (CIM)** solution in a larger-scale processing platform.
- Validating the performance improvement of ACA in a realistic image-processing application.

It is worth emphasizing that, unlike several other CIM-P accelerators, which are designed to be used in the servers (e.g., References [3, 6, 17]), ACA is specially designed for embedded platforms. In embedded applications, circuit energy and area consumption is very restricted, and therefore it is not desirable to design complex processing blocks in the memory periphery.

This article starts with a brief background explanation in Section 2. Section 3 explains our architectures for Address Calculation Accelerator. Section 4 explains our compiler to be able to efficiently use ACA in a practical application and our nano-simulator platform to perform the hardware-aware simulations. The experimental results for guided image filtering application are presented in Section 5 and a brief conclusion follows it in Section 6.

2 BACKGROUND

Figure 2(a) shows a simplified computer architecture where the memory bus is the bottleneck for memory-intensive applications. One suggested solution to relax this bottleneck is to use **High Bandwidth Memory (HBM)** [2] technology. HBM uses a wide memory bus and places the memories as close as possible to the processing unit in the same package. However, since the memory is off-chip, it is yet far away (in the nanometer scale), and therefore each transaction is still quite expensive from energy and latency point of view (about 7 pJ/bit [2] compared to less than 0.04 pJ/bit for ALU operations in the same technology [23]).

As shown in Figure 2(b), allocating small memories inside the processors in the form of cache memory or **Tightly Coupled Memory (TCM)** is a widely used technique to reduce the number of

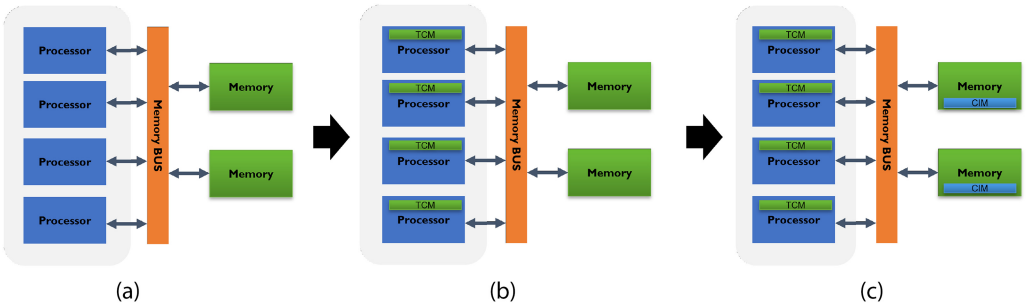

Fig. 2. Simplified architectures of a multi-processor system where the memory bus connects the memories to the processors (memory units are either off-chip or on-chip). (a) Traditionally separated memory and processors. (b) Bringing memory closer to the processor by using Tightly Coupled Memories (TCM). TCM is typically made from small blocks of on-chip SRAM next to the processor. (c) Bringing processor closer to the memory by using CIM circuit.

transactions over the memory bus. This way, data are read once from the memory blocks and stored in TCM to be reused several times. The performance increase by TCM depends on its size and the application that is executing in the processor. Since TCM is typically built from SRAM (which is not area efficient), the size of TCM has to remain relatively small. In addition, by increasing the size of TCM, its access energy consumption grows further, which is also undesirable.

Figure 2(c) illustrates a more recent idea to further relax the memory bus bottleneck by incorporating a CIM [6, 7, 18]. CIM is a specific purpose computation unit that can reduce memory bus transactions by performing the more common computations directly inside the memory. This way, it is possible to reuse a value in the memory several times without the need to bring it inside the processor [22]. When using CIM, the memory block can be considered as a co-processor and receive high-level instructions. The most efficient type of in-memory computation is done directly in the memory array, which is called **Compute In Memory-Array (CIM-A)** and possibly by modifying the sense amplifiers. In this case, CIM only supports binary (bitwise) or low-resolution computations [13, 15, 27]. However, these methods require delicate modification of the memory array and are highly dependent on the target memory technology [7]. This work focuses on another category of CIM by CIM-P. Therefore our proposed solution applies to any memory technology. Additionally, the IP can be ported into different technologies with minimal effort, since it is digitally designed.

As it is mentioned, the communicated information between memory and the processing units consists of three main parts: *1-Instruction or Control word*, *2-Target word address,* and *3-Data Operand*. For example, in a conventional system to write in a memory line, the CPU should provide the "target word address," the instruction, which is the "write instruction," and the operand, which is the "write data." When performing an in-memory process, it is possible to give higher-level instructions to the memory block. For example, an instruction can be accumulating the content of the "target word address" with the "Operand" [33]. Since, in this case, the accumulation happens inside the memory macro, it skips transferring the content of the "target word address" to the processor and the accumulated result back to the memory. Therefore, it considerably reduces the amount of memory-processor data transfer.

To the best of our knowledge, the majority of the previous CIM-P architectures are designed to execute arithmetic operations on the content of the memory [6, 11, 17] while keeping the conventional address decoder. Few other works [26, 28] tried to modify the address decoder to perform particular optimizations. However, no previous work optimized the address communication over

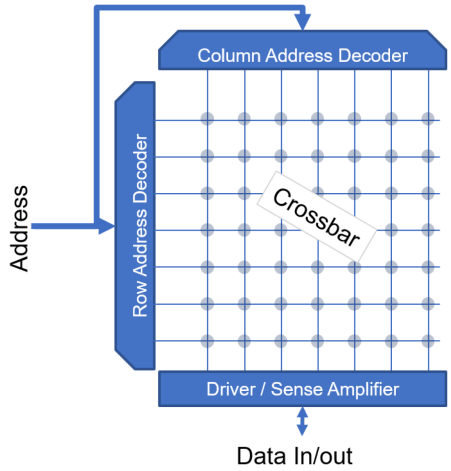

Fig. 3. Components of a simple memory block. Each crossing point in the crossbar may be one memory cell (in a 2D memory) or multiple memory cells (in a 2.5D memory architecture). A memory cell may contain 1-bit of data (binary cell) or multiple bits (in a multi-level cell, required analog to digital conversion after a sense amplifier).

the memory bus by replacing the conventional address decoder with a lower area-delay-energy overhead pointer-based address calculator. In the previous example, imagine the processor should repeat the same accumulate operation for 1,024 consecutive addresses. In this case, 1,024 transactions with similar "instructions," consecutive "target addresses," and optionally different "operands" need to be initiated from the processor. In this case, the overhead of the repeated "instructions" and "target address word" in every transaction can be considerable (especially for small word sizes, which is a new fashion in edge applications). For example, to access an 8-bit word in a relatively small 1-MB memory, it is required to transmit a 20b address (the number of address bits scales up with the number of words in the memory.). Moreover, new addresses have to be calculated for every transaction.

Offloading the address calculations from the processor is possible by using DMA or AGU for simple address calculations. However, as mentioned before, the raw addresses need to be communicated with the memory, which costs one to two orders of magnitude (depending on the location and type of memory) more energy and latency than the address calculations.

As shown in Figure 3, a typical memory contains three main parts: (1) Memory crossbar (array), (2) Input/Output interface for word-line/bit-line drivers, sense amplifiers, possibly analog-to-digital converters for multi-level cells, and (3) Pre- and Post-Address decoders. Using a conventional address decoding scheme would require continuous communication over the memory bus to fetch subsequent addresses. In this article, we proposed to replace the address decoders with a customized DMA circuit called ACA to remove the need for raw address transactions over the memory bus. Section 3 explained the architecture of our proposed ACA circuit.

3 ARCHITECTURE OF ADDRESS CALCULATION ACCELERATOR

ACA allows for a reasonably regular access pattern to the memory, which is not only usable for the purely consecutive burst access. This type of memory access is used in most loop nests and tensor processing operations. An example of this type of access is 2D convolution. As illustrated in Figure 4, applying a 3×3 kernel filter on a 2D image that is mapped linearly in the memory, requires access to three separated consecutive addresses in the memory. But this is also directly

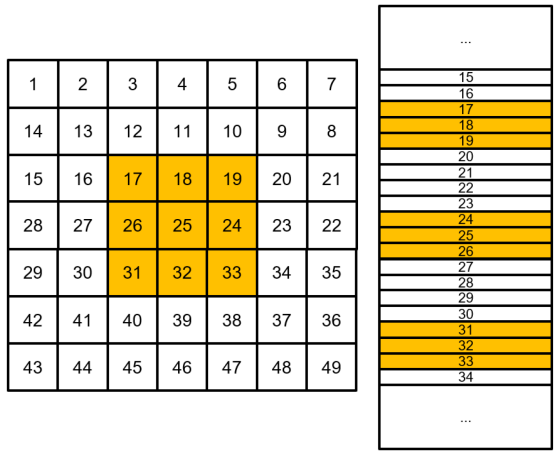

Fig. 4. A 2D image (left) and its mapping in the memory (right). As it can be seen, to apply a 3×3 filter to a region of the image, a repetitive (but non-consecutive) access pattern is required. It is more challenging during filtering the edges of the image.

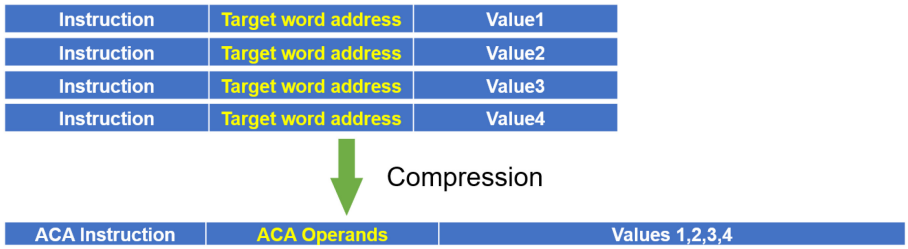

Fig. 5. Compression of address overhead by ACA. Please note that ACA only reduces the addressing overhead for a list of data and does not compress the data itself.

applicable to, e.g., 2D image filter kernels that contain gaps in their mask (for example in Dilated-Convolution [32]).

Our goal in designing the ACA is to provide higher-level instructions to the memory block to compress the number of address transactions. When using ACA, the conventional format of "*Instruction, target word address, data*" for processor-memory communications will be modified to the compressed form of "*ACA instruction, ACA operands, list of data*" as shown in Figure 5. Therefore, to use ACA, the processor should pack several memory access patterns in the form of an ACA instruction. In our experiments, we assume this is happening offline during compile time; however, this process can also happen dynamically during the runtime. In this case, the compiler is aware of the ACA instructions. Therefore there is no runtime process required to pack the memory accesses.

The main operands when using ACA are listed in Table 1. There is always only one "num_loops" in each ACA transaction, and its value defines the number of nested loops to be executed by the ACA controller. Then each individual loop will have its own row/col "Start(s)," "Stride," and "Cycles."[3] Therefore the size of "ACA Operands" in Figure 5 is variable. ACA nested loops execute

[3]Cycles in this figure are representing the number of iterations/repetitions through the rows/columns.

Table 1. Main ACA Operands

ACA Operand	Description
Num_Loops	Number of nested loops
Row_Start(s)	The start position(s) of the row shift register
Row_Stride	The amount of shift on the row shift register
Row_Cycles	The number of shift cycles for the row shift-register
Col_Start(s)	The start position(s) of the column shift register
Col_Stride	The amount of shift on the column shift register
Col_Cycles	The number of shift cycles for the column shift-register

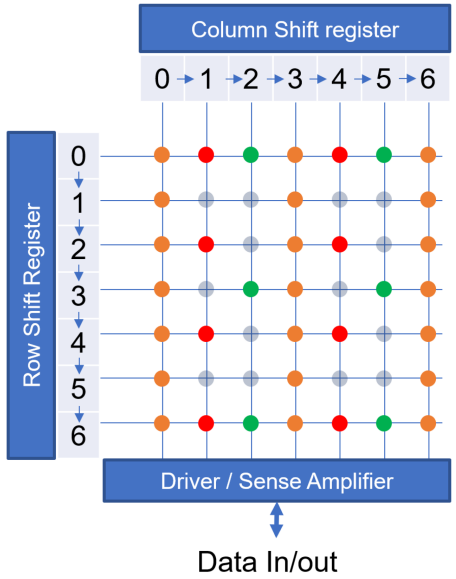

Fig. 6. An example of using ACA to generate memory address patterns. Each colored dot in the memory cross-bar is a word of memory (can be a bit, a byte, etc.).

Instruction	Num Loops	Row Start 1	Row Stride 1	Row Cycles 1	Col Start 1	Col Stride 1	Col Cycles 1	Row Start 2	Row Stride 2	Row Cycles 2	Col Start 2	Col Stride 2	Col Cycles 2	Row Start 3	Row Stride 3	Row Cycles 3	Col Start 3	Col Stride 3	Col Cycles 3
ACA-Read	3	0	1	7	0	3	3	0	2	4	1	3	2	0	3	3	2	3	2

Fig. 7. Instructions and operands for ACA to access the 1-orange, 2-red, and 3-green words in Figure 6.

from the first defined loop in the instruction operands to the last one. In each loop, it iterates first over columns and then over rows.

To explain more about ACA functionalities, imagine in Figure 6, we read the memory words in the order of 1-orange words, 2-red words, and 3-green words. As it can be seen, the access addresses would follow a pattern, but it is not a simple sequential pattern. To read all the words in a conventional method, "35" address transactions are required. However, by using ACA, only one transaction with three nested loops is enough. Figure 7 shows the ACA instruction and operands to read the orange/red/green words in order. The number of loops at the beginning of the instruction informs the **Finite State Machine (FSM)** in Figure 9 that it should expect three different sets of row/column instructions (yellow, red, and green loops in Figure 7).

ACA is a low-overhead and useful replacement for the address decoders, as shown in Figure 8, which can compute a complex pattern of addresses locally inside the memory. Figure 9 illustrates the schematic of our proposed ACA architecture. ACA contains at least two shift registers (row and

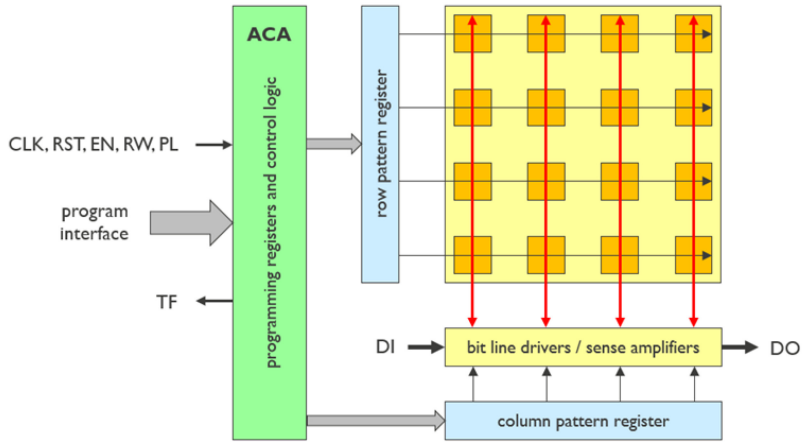

Fig. 8. Address Computation Accelerator replaces row/column address decoders in the full memory macro.

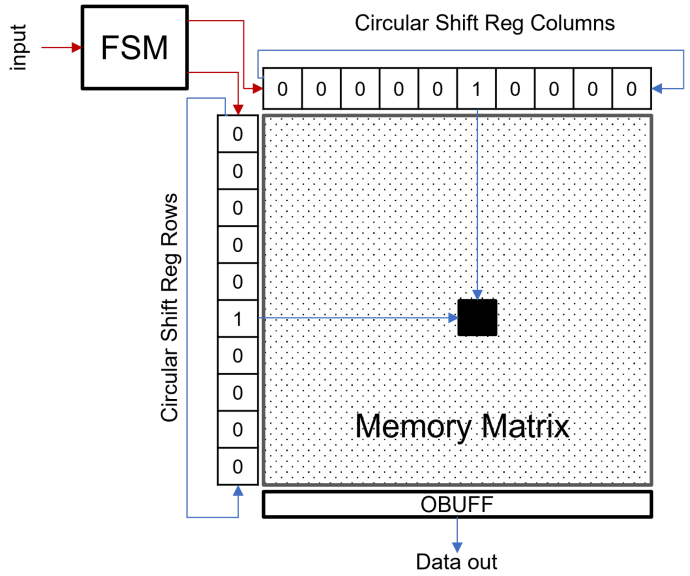

Fig. 9. In memory ACA block diagram.

column shift registers) and a control logic block implemented with a FSM. The FSM decodes high-level ACA instructions to generate a list of "target word addresses." Then the FSM initializes both of the shift registers and commands them to shift left or right. The row and column shift registers select one (or several if CIM-A functionality is implemented) words of the memory matrix at each cycle and therefore ACA can generate address patterns in rows and columns simultaneously. Even though we only used two shift registers (one for row dimension and one for column dimension), it is possible to use multiple of them for each dimension to implement the nested loops more efficiently.

In the current implementation, we have a single start position for rows and columns. When selecting several row/columns, it is required to have several row_start/col_start operands. It is

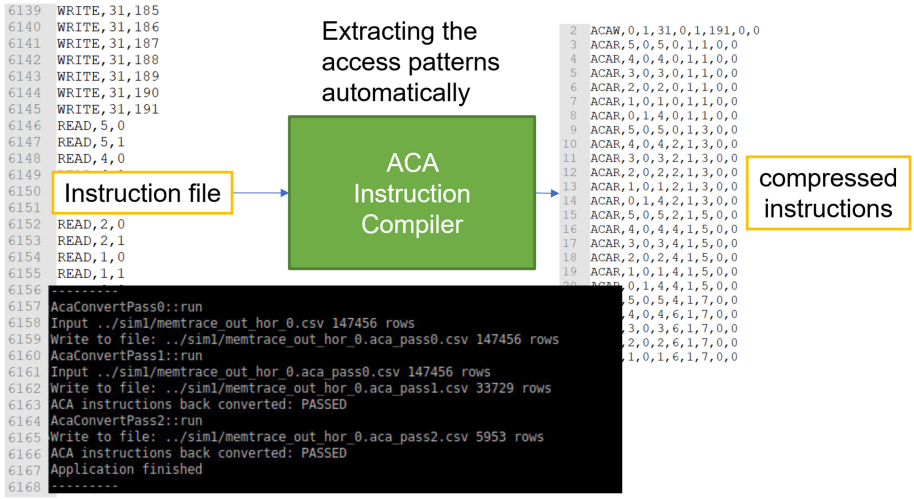

Fig. 10. ACA compiler receives the standard memory instructions and extract repetitive access pattern to generate compressed ACA instructions.

also essential that the memory core can accept such a configuration. For example, to perform in-memory binary operations between two rows of the memory, we should have two row_start active bits.

4 COMPILER AND SIMULATOR

4.1 ACA Compiler

When using in-memory processing capability (for example, in a platform same as Figure 2(c)), the processor can outsource some part of the computation to the CIM block. In this case, the CIM block accepts higher-level instructions. To perform this kind of computation and processor-CIM communication, the program compiler of the processor needs to be aware of the specific CIM features.

As we introduced the ACA logic block in the CIM, we also needed to compile the application with an ACA-aware compiler. Rather than directly modifying the existing compilers, we have made a separate ACA compiler that operates as a post-processing stage after a conventional compiler (Figure 10). This solution is more software maintenance friendly, and it broadens the applicability. The responsibility of the ACA compiler is to detect the access patterns to the memory and packed them by using ACA instructions.

Figure 10 shows an example of the input and output of the ACA compiler. In this example, we only use read/write instructions, but ACA is not limited to these instructions. ACA compiler only searches for memory access patterns and does not interfere with the instructions supposed to be executed inside the memory. The current version of the ACA compiler is performing a simple search. Therefore, it may be slow for big applications and may miss some more complex patterns. Further optimization of this compiler should be done in future work.

4.2 Nano-Simulator

Hardware-aware simulation provides valuable insight for architectural exploration at different abstraction levels. As part of the European Mnemosene project,[4] we have developed a

[4]http://www.mnemosene.eu/.

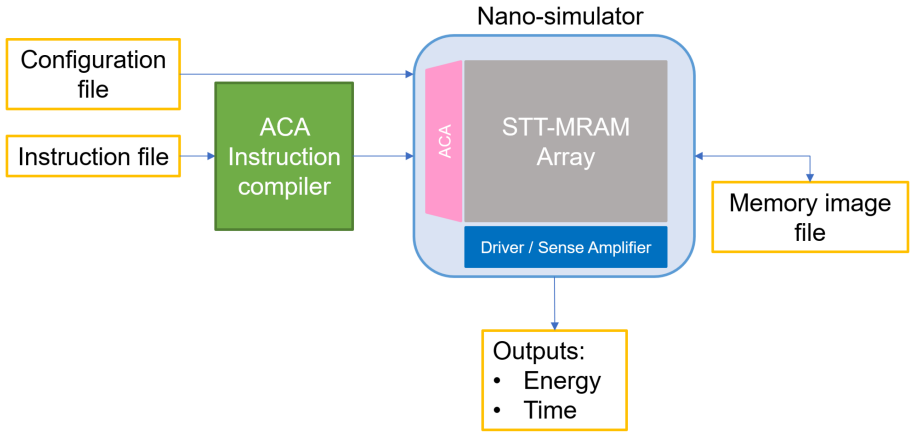

Fig. 11. Our nano-simulator receives a configuration file, instruction file (after being post-processed by the ACA compiler), and the initial memory image. After the simulation, it updates the memory image file and also provides consumed energy and time metrics as a result of the simulation.

Table 2. Input/outputs Files of Nano-simulator

Instruction file	Input	Contains the instruction list (without ACA extension)
Configuration file	Input	Contains the dimensions of the STT-MRAM memory, the path to the memory image file, instruction file, and result file
Memory image file	In/Out	Stores the content of the memory before, after, and during the simulation.
Results file	Output	Contains the detailed results of the simulation, including energy/power/time consumption of each element in the simulation

hardware-aware simulator with a close to technology focus. This simulator is called "nano-simulator," because it is part of a more extensive micro-instruction level system simulator in the Mnemosene project (nano-simulator emulates a CIM memory block in a processing system in Figure 2). We have used the nano-simulator in this article to report experimental results about the performance of ACA.

This simulator abstracts the functionality of the memory array along with its peripherals (including ACA). The nano-simulator mainly wraps the behavior of the memory cells, sense amplifiers, row/column driver lines, ACA, or conventional address decoder. Our nano-simulator is accurate and, at the same time, faster than low-level circuit simulations.

We can simulate the memory cells and peripherals with nano-simulation to explore metrics like energy consumption and latency for any application. It is possible to add other relevant metrics if it is required. We extract the data and equations from the low-level analog simulations to parameterize the memory cells and the sense amplifiers. Currently, those data are limited to read/write instructions. In future works, we can also add energy/latency data for CIM-A instructions (like in-memory binary operations). It is worth mentioning that ACA is categorized as a CIM-P technology and therefore can be used with all types of memory arrays. Figure 11 shows the input/output files of the nano-simulator. The Table 2 explains the input/out files.

Fig. 12. Assumed widths of each field of the communication packet between the processor and the memory in our experiments with/without using ACA. Some operations do not require a value (like reading). For our results, we do not need to assume the size of the value field.

5 EXPERIMENTAL RESULTS

5.1 Setup

To illustrate the functionality with real memory technology and to provide quantitative numbers for the delay, energy, and area, we have instantiated the realistic IMEC **Spin-Torque-Transfer Magnetic Random Access Memory (STT-MRAM)** macro along with its peripherals (including ACA) in the nano-simulator. STT-MRAM is an emerging non-volatile memory technology used for efficient analog CIM-A. All the critical components are parameterized, and the delay, energy, and area have been calibrated based on the measurement of fabricated test structures.

As mentioned before, ACA can reduce the number of individual transactions over the bus by packing/unpacking the addresses. We run the application on the Nano-simulator, which provides the energy consumption and latency for the read/write operations of the STT-MRAM memory block. However, Nano-simulator cannot offer the absolute energy numbers for the memory bus, since this number is dependent on the distance between the processor and the memory block. Therefore the relative energy saving on bus based on the reduced number of bits is provided as a metric in this article. As shown in Figure 12, we assume 8b for the instructions[5] and 64b for the target address. As shown in Table 1, for ACA operands in addition to row/col starts (64b for both), we need row/col increment (each limited to 4b, which results in a maximum shift of "16" in one cycle), and row/col cycles (each limited to 10b). In the current version of the ACA compiler, we did not use the nested loops (Num_Loops = 1) and therefore did not allocate any bits for "Num_Loops." Therefore we have allocated 92b for ACA operands as shown in Figure 12. These numbers are based on our assumption for a practical system and not yet based on any optimization.

5.2 Guided Image Filtering with ACA

It is always required to access regular or semi-regular repetitive accesses to one- or more-dimensional arrays and other composite data types for the important domain of streaming applications. To obtain a realistic case study from the streaming data and signal processing domain, we have chosen an image processing technique called "guided image filtering" [4, 10]. This application uses two images as input and guide and performs repetitive operations on the input image using the guided filter. In our case, the input image sizes, guided filter, and output are the same. This application follows the pipeline shown in Figure 13 to process an input image.

Since the input image size affects the energy consumption and the rate of address compression, we have used two input image sizes: 32×32 and 256×256. We have selected "RGB" images with single-precision floating-point (FP32) format for each pixel's color in the input images. "Guide" image has the same size as input but with integer 8b format for each color of its pixels. All the other variables are encoded in FP32.

Tables 3 and 4 reports our experimental results with 32×32 and 256×256 input image sizes. In these tables, we reported the occupied memory area for each buffered matrix (input, guide, tmp0, tmp1, tmp2, and output, as shown in Figure 13), the number of individual transactions, and the

[5]We assume that adding a few ACA instructions does not increase the size of instruction fields.

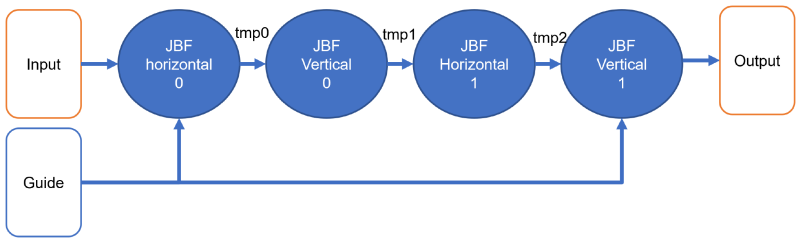

Fig. 13. The pipeline of the guided image filtering application. JBF denotes "Joint Box Filter".

number of transferred overhead bits (instructions and addresses/ACA operands).[6] In our experiments based on Figure 12 overhead size is 8+64 bits for each conventional transaction and 8+92 bits for each compressed ACA transaction. before and after using ACA over the memory bus along with their compression ratios, and total energy consumption in the memory array,[7] ACA unit and the time that is consumed by memory to finish the read/write operations for the corresponding matrix. The compression ratio is the ratio of overhead bits after compression with ACA over the normal one (lower is better).

As expected, ACA does not reduce the energy and time consumption of the memory array (STT-MRAM in this case), because it does not focus on data access. However, the amount of overhead bits reduces to 11% and 6.4% for small and big input image sizes. This reduction results in a high amount of energy saving for off-chip memories.

When we scale up the memory sizes, the compression ratio increases; however, the energy consumption by ACA also grows; since a bigger shift register consumes more energy (same as a bigger address decoder). We have implemented a shift register that can do a single shift in one cycle (<<1). As the digital peripheral speed is typically faster than the memory access time, it is possible to implement an "N" bit shift instruction (*Next address* = (*previous address*) << *N*) in several digital clock cycles equal to one memory access cycle. For example, suppose the digital clock frequency is 1 GHz and memory access time is 10 ns. In that case, in between two memory accesses, the shift register has ten digital clock cycles to perform ten shifts. It is also possible to have a shift register that can perform a shift with a variable amount. The optimum solution depends on the relative speed of memory and peripherals. It may be somewhere between complete time-multiplexing (this work) and full flexible shift registers (shift with an arbitrary amount in one digital clock cycle).

From Table 3, we can see that for a memory with $32 \times 32 \times 3$ words, ACA consumes around 355 fJ per memory access (12 nJ for 33,793 read access). From our analysis of another SRAM technology (IMEC, 3-nm FinFET technology[8]) and considering the linear scaling factor (our reported results is in TSMC40-nm technology), we calculated that a conventional address decoder with the same size ($32 \times 32 \times 3$) consumes at least[9] 640 fJ per each access. So, all the savings on the memory bus transferred bits are gained with little to no logic energy overhead. However, as mentioned before, the bus interconnects related energies are the dominant energy consumption in practical systems, and the ACA logic energy consumption is less significant.

[6]Overhead bits are defined as the non-data bits in each transaction.

[7]The simulation results for STT-MRAM energy varies a bit by changing the initial memory state (for example, input image in this case), since the energy/time consumption is different when switching from "1" to "0" and from "0" to "1" in an STT-MRAM memory cell.

[8]The reported energy report is just for the comparison purpose and cannot be considered accurate, since it is only covering logic, not the wiring and interconnects.

[9]Linear scaling factor that is used results in the minimum energy consumption in 40-nm technology. In practice, it should scale super linear and at most quadratic.

Table 3. Results of Simulation "Guided Image Filtering" Application Using the Nano-simulator for 32 × 32 Input Image Sizes

Kernel	buffer size	The normal number of transactions - overhead bits	Compressed number of transactions (ACA) - overhead bits	Compression Ratio (transactions - overhead bits)	STT-MRAM Energy (nJ)	ACA Energy (nJ)	Total time (μs)
Input	32 × 32 × 3–32b	33,793 T–2,376 kb	5,953 T–581 kb	18%–24%	496 nJ	12 nJ	109 μs
Guide	32 × 32 × 3–8b	36,865 T–2,592 kb	5,954 T–581 kb	16%–22%	289 nJ	13 nJ	118 μs
Tmp0 (Hor0)	32 × 32 × 12–32b	147,457 T–10,368 kb	5,954 T–581 kb	4.0%–6%	2,456 nJ	176 nJ	551 μs
Tmp1 (Ver0)	32 × 32 × 6–32b	73,729 T–5,184 kb	5,954 T–581 kb	8.0%–11%	1,228 nJ	46 nJ	275 μs
Tmp2 (Hor1)	32 × 32 × 6–32b	73,729 T–5,184 kb	5,954 T–581 kb	8.0%–11%	1,224 nJ	46 nJ	275 μs
Output (Ver1)	32 × 32 × 3–32b	3,073 T–216 kb	1 T–0.1 kb	0.03%–0.04%	118 nJ	1 nJ	30 μs
Total		368,646 T–25,920 kb	29,770 T–2,907 kb	8%–11%	5,811 nJ	294 nJ	1,358 μs

Overhead bits include instructions and target address/ACA operands.

Table 4. Results of Simulation "Guided Image Filtering" Application Using the Nano-simulator for 256 × 256 Input Image Sizes

Kernel	buffer size	The normal number of transactions - overhead bits	Compressed number of transactions (ACA) - overhead bits	Compression Ratio (transactions - overhead bits)	STT-MRAM Energy (nJ)	ACA Energy (nJ)	Total time (μs)
Input	256 × 256 × 3–32b	2,162,689T–148 Mb	219,649T–21 Mb	10%–14%	31,738 nJ	5,615 nJ	6,910 μs
Guide	256 × 256 × 3–8b	2,359,297T–162 Mb	219,650T–21 Mb	10%–13%	18,482 nJ	6,069 nJ	7,539 μs
Tmp0 (Hor0)	256 × 256 × 12–32b	9,437,185T–648 Mb	219,650T–21 Mb	2.3%–3.2%	157,170 nJ	88,838 nJ	35,248 μs
Tmp1 (Ver0)	256 × 256 × 6–32b	4,818,593T–331 Mb	219,650T–21 Mb	4.5%–6.3%	78,584 nJ	22,762 nJ	17,624 μs
Tmp2 (Hor1)	256 × 256 × 6–32b	4,718,593T–324 Mb	219,650T–21 Mb	4.6%–6.4%	78,586 nJ	22,751 nJ	17,724 μs
Output (Ver1)	256 × 256 × 3–32b	196,609T–13 Mb	1T–0.1 kb	0%–0%	7,553 nJ	456 nJ	1,901 μs
Total		23,692,966T–1,627 Mb	109,8250T–105 Mb	4.6%–6.4%	372,113 nJ	14,6491 nJ	86,946 μs

Overhead bits including instructions and target address/ACA operands.

In our setup, we assumed that the address bitwidth is 64b. For smaller address widths, and if we assume that other fields remain the same, ACA operands in Figure 12 will have higher overhead bits. For example, for 32b address bitwidth, the compression ratio for overhead bits changes from 11% to 13.6% for an input image size of 32 × 32 (Table 3) and from 6.4% to 7.8% for an input image size of 256 × 256 (Table 4).

5.3 Software Pipelining

For the guided image filtering application, we processed each kernel sequentially. However, as it is clear from Figure 13, it is possible to run all filters in the pipeline in parallel by exploiting a software-pipelining concept in memory with a long word line. In software pipelining, as all the kernels execute simultaneously with the unique access pattern to the memory, we can use a longer word line in the memory (200b in our experiments) to feed all the processes in parallel (as illustrated in Figure 14). Processing one long word can take one or several cycles, dependent on the target compute architecture and independent of the CIM unit. In this case, ACA only needs to generate one address per line, which results in a reduced cycle count and energy efficiency for the address generation and the address and data communication network. However, the energy consumption for the memory access itself and the arithmetic instructions on the processor cores mainly remains the same.

Table 5 shows that software pipelining with long word sizes results in almost two orders of magnitude reductions in the number of overhead bits over the memory bus (skipping of almost 1.6 Gb data transfer).

A more conventional computing architecture like a GPU can easily exploit software-pipelining due to a high level of parallelism with **Single Instruction Multiple Data (SIMD)** structures. However, irregular memory access (same as most advanced image and video processing kernels) will cause inefficiency in SIMD processing and reduces processor utilization. Hence, in the current hardware and compiler support for the GPUs, an ACA-like address decoding scheme will address this problem. We expect that compared to state-of-the-art GPUs, the guided filtering application can have at least 10× fewer address instructions. This compression will hence increase performance significantly. It also saves address instruction execution and bus communication energy.

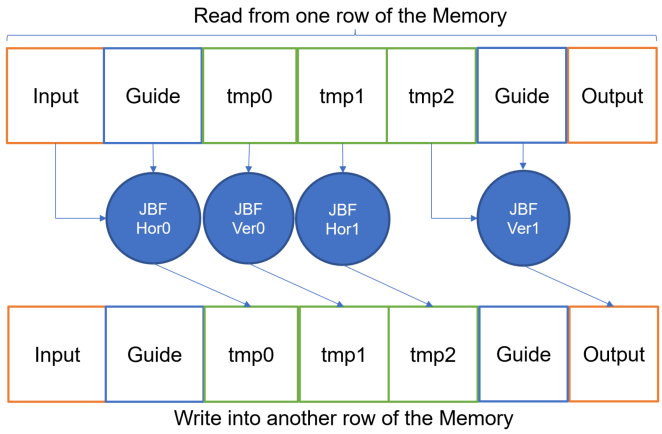

Fig. 14. Parallel read/write from a wide memory in guided image filtering. In the software-pipelining format, one long word of the memory is read, processed, and written back to the memory. In our experiment, one long word line consists of $6 \times 32 + 8$ bits.

Table 5. Results of Simulation "Guided Image Filtering" Application with Software Pipelining

Input Image size	Total number of transactions - overhead bits	Compressed number of (ACA) transactions - overhead bits	Compression Ratio (transactions - overhead bits)
32×32	368,646T–25.3 Mb	5,958T–0.57 Mb	1.6%–2.2%
256×256	23,692,966T–1627 Mb	219,654T–21 Mb	0.92%–1.3%

Still, we do not have access to a sufficiently detailed model of the entire GPU micro-architecture to calculate that energy-saving accurately.

6 CONCLUSION AND OUTLOOK

This article has presented our methods and results in performing the in-memory process to reduce data movements between memory blocks and processors in address transfers. Our nano-simulator makes it feasible to run fast experiments with different memory technologies. Using our nano-simulator, we have demonstrated that ACA can reduce address transactions when the application uses a predictable addressing scheme. Besides, our ACA approach can be combined in a hybrid CIM-P/CIM-A implementation by merging it with published analog and bit-level CIM-A concepts from literature.

Evolving the new architectures by using compute in memory is a breakthrough in computer architecture. As most of the energy in STOA computer systems is consumed by data movement, computing in memory reduces total energy for a given performance target. Several previous art architectures [6, 6, 11, 17, 27] are addressing the data processing inside the memory that are fully complementary to our proposal. Combining those architectures with ACA makes sense to improve saving on both data and address bandwidth in the memory bus.

ACA is developed as part of two European projects[10] where we demonstrated a complete processing platform to test several CIM-A and CIM-P accelerators and used in a RISC-V based neuromorphic processor [31]. Even though, as a digital CIM-P architecture, applications of ACA are not limited to any specific CPU or memory technology, it should be emphasized that its main target domain focuses on embedded processor platforms and not on high-performance server-oriented

[10]http://www.mnemosene.eu/, https://dais-project.eu/.

processors. ACA is easier to use with scratchpad or software-controlled caches than for the HPC type memory organizations. In this work we showed that ACA can reduce the amount of overhead bits for data transfer between the processor and the memory by almost two orders of magnitude. In future work, we will focus on the detailed micro-architecture level of CIM-A and CIM-P and the more application demonstrators. In this way, we will explore different micro-architecture-circuit-technology choices (STCO), including promising emerging memory options (especially MRAM and IGZO-DRAM) and global design PPAC tradeoff exploration space.

ACKNOWLEDGMENTS

The authors thank Bart Goossens and his team at the University of Gent for their support in the implementation of the optimized "guided image filtering" application. The authors thank the reviewers for their contribution to the improvement of this article.

REFERENCES

[1] 1993. IEEE standard for communicating among processors and peripherals using shared memory (direct memory access - DMA). IEEE Std 1212.1-1993 (1993), 1–134. https://doi.org/10.1109/IEEESTD.1993.8686399

[2] 2020. Product Brief: Virtex UltraScale+ HBM FPGA. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/virtex-ultrascale-plus-hbm-product-brief.pdf.

[3] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near data acceleration with concurrent host access. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. IEEE, 818–831.

[4] Thomas Faict, Erik H. D'Hollander, and Bart Goossens. 2019. Mapping a guided image filter on the HARP reconfigurable architecture using OpenCL. *Algorithms* 12, 8 (2019). https://doi.org/10.3390/a12080149

[5] Iason Giannopoulos, Abhairaj Singh, Manuel Le Gallo, Vara Prasad Jonnalagadda, Said Hamdioui, and Abu Sebastian. 2020. In-memory database query. *Adv. Intell. Syst.* 2, 12 (2020), 2000141.

[6] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware. In *Proceedings of the 12th International Green and Sustainable Computing Conference (IGSC'21)*. IEEE, 1–7.

[7] S. Hamdioui, H. A. Du Nguyen, M. Taouil, A. Sebastian, M. L. Gallo, S. Pande, S. Schaafsma, F. Catthoor, S. Das, F. G. Redondo, G. Karunaratne, A. Rahimi, and L. Benini. 2019. Applications of computation-in-memory architectures based on memristive devices. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'19)*. 486–491.

[8] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels. 2017. Memristor for computing: Myth or reality? In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'17)*. 722–731.

[9] A. F. Harvey and Data Acquisition Division Staff. 1991. DMA fundamentals on various PC platforms, NATIONAL INSTRUMENTS.

[10] Kaiming He, Jian Sun, and Xiaoou Tang. 2010. Guided image filtering. In *European Conference on Computer Vision*. 1–14.

[11] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 372–385.

[12] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Proceedings of the IEEE 34th International Conference on Computer Design (ICCD'16)*. IEEE, 25–32.

[13] Giacomo Indiveri and Shih-Chii Liu. 2015. Memory and information processing in neuromorphic systems. *Proc. IEEE* 103, 8 (2015), 1379–1397.

[14] Muhammad Jazib Khan. 2020. Programmable Address Generation Unit for Deep Neural Network Accelerators. KTH royal institute of technology.

[15] Sandeep Kaur Kingra, Vivek Parmar, Che-Chia Chang, Boris Hudec, Tuo-Hung Hou, and Manan Suri. 2020. SLIM: Simultaneous logic-in-memory computing exploiting bilayer analog OxRAM devices. *Sci. Rep.* 10, 1 (2020), 1–14.

[16] Kazukuni Kitagaki, Takayoshi Oto, Tatsuhiko Demura, Yoshitsugu Araki, and Tomoji Takada. 1991. New address-generation-unit architecture for video signal processing. In *Visual Communications and Image Processing'91: Image Processing*, Vol. 1606. International Society for Optics and Photonics, 891–900.

[17] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. 2021. Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. IEEE, 43–56.

[18] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[19] Miguel Miranda, Francky Catthoor, Martin Janssen, and Hugo De Man. 1996. ADOPT: Efficient hardware address generation in distributed memory architectures. In *Proceedings of the 9th International Symposium on Systems Synthesis*. IEEE, 20–25.

[20] Miguel A. Miranda, Francky V. M. Catthoor, Martin Janssen, and Hugo J. De Man. 1998. High-level address optimization and synthesis techniques for data-transfer-intensive applications. *IEEE Trans. VLSI Syst.* 6, 4 (1998), 677–686.

[21] O. Mutlu. 2018. Processing data where it makes sense in modern computing systems: Enabling in-memory computation. In *Proceedings of the 7th Mediterranean Conference on Embedded Computing (MECO'18)*. 8–9.

[22] Hoang Anh Du Nguyen, Jintao Yu, Muath Abu Lebdeh, Mottaqiallah Taouil, Said Hamdioui, and Francky Catthoor. 2020. A classification of memory-centric computing. *ACM J. Emerg. Technol. Comput. Syst.* 16, 2 (2020), 1–26.

[23] Ardavan Pedram, Stephen Richardson, Mark Horowitz, Sameh Galal, and Shahar Kvatinsky. 2016. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Des. Test* 34, 2 (2016), 39–50.

[24] Laurent Pinchart. 2014. Mastering the DMA and IOMMU APIs. In *Proceedings of the Embedded Linux Conference*.

[25] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. 2020. Memory devices and applications for in-memory computing. *Nat. Nanotechnol.* 15, 7 (2020), 529–544.

[26] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, et al. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–197.

[27] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. IEEE, 273–287.

[28] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses. In *Proceedings of the 48th International Symposium on Microarchitecture*. 267–280.

[29] Abhairaj Singh, Sumit Diware, Anteneh Gebregiorgis, Rajendra Bishnoi, Francky Catthoor, Rajiv V. Joshi, and Said Hamdioui. 2021. Low-power memristor-based computing for edge-AI applications. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'21)*. 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401226

[30] C. David Wright, Peiman Hosseini, and Jorge A. Vazquez Diosdado. 2013. Beyond von-Neumann computing with nanoscale phase-change memory devices. *Adv. Funct. Mater.* 23, 18 (2013), 2248–2254.

[31] Amirreza Yousefzadeh, Gert-Jan van Schaik, Mohammad Tahghighi, Paul Detterer, Stefano Traferro, Martijn Hijdra, Jan Stuijt, Federico Corradi, Manolis Sifalakis, and Mario Konijnenburg. 2022. SENeCA: Scalable energy-efficient neuromorphic computer architecture. In *Proceedings of the IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS'22)*. IEEE.

[32] Fisher Yu and Vladlen Koltun. 2015. Multi-scale context aggregation by dilated convolutions. arXiv:1511.07122. Retrieved from https://arxiv.org/abs/1511.07122.

[33] Mahdi Zahedi, Muah Abu Lebdeh, Christopher Bengel, Dirk Wouters, Stephan Menzel, Manuel Le Gallo, Abu Sebastian, Stephan Wong, and Said Hamdioui. 2022. MNEMOSENE: Tile architecture and simulator for memristor-based computation-in-memory. *ACM J. Emerg. Technol. Comput. Syst.* 18, 3 (2022), 1–24.

Received July 2021; revised February 2022; accepted May 2022