Parallel Matrix Multiplication on Memristor-Based Computation-in-Memory Architecture

Adib Haron, Jintao Yu, Razvan Nane, Mottaqiallah Taouil, Said Hamdioui, Koen Bertels
Computer Engineering Laboratory, Department of Quantum Engineering
Delft University of Technology
Delft, The Netherlands

***Abstract*—One of the most important constraints of today's architectures for data-intensive applications is the limited bandwidth due to the memory-processor communication bottleneck. This significantly impacts performance and energy. For instance, the energy consumption share of communication and memory access may exceed 80%. Recently, the concept of Computation-in-Memory (CIM) was proposed, which is based on the integration of storage and computation in the same physical location using a crossbar topology and non-volatile resistive-switching memristor technology. To illustrate the tremendous potential of CIM architecture in exploiting massively parallel computation while reducing the communication overhead, we present a communication-efficient mapping of a large-scale matrix multiplication algorithm on the CIM architecture. The experimental results show that, depending on the matrix size, CIM architecture exhibits several orders of magnitude higher performance in total execution time and two orders of magnitude better in total energy consumption than the multicore-based on the shared memory architecture.**

I. Introduction

The communication cost (i.e., delay or energy) of data-intensive applications on existing computer architectures exceeds the computation cost [1]. The problem becomes more severe due to the growing gap between processor and memory speed [2] and the fast-growing big data applications [3]. This high communication cost is explained by two main reasons. First, existing computer architectures have limited bandwidth and high off-chip memory access latency that are attributed to the classical CMOS-based Von Neumann architectures [4]. Second, the algorithms for data-intensive applications spend most of the processing time to move data [5]. Consequently, the communication cost limits the scalability of today's architectures and reduce the opportunity to exploit higher degree of parallelism in algorithm due to the high impact on the total execution time and energy dissipation.

Minimizing the communication cost has been studied thoroughly in the past both at the architectural and the algorithmic levels. From an architectural perspective, three main parallel architectures have been developed to boost the computer performance: shared memory [6], [7], distributed memory [8], [9] and systolic array [10], [11]. For the shared memory architecture, the communication cost is mostly a consequence of the data movement throughout the memory hierarchy. Even though the entire data structures can be easily accessed, the performance is limited by the communication and memory bottleneck. For the distributed memory architecture, the high memory access latency is reduced as the memory is distributed across the platform instead of having a single large memory. However, this architecture incurs a high communication cost between processors. Similarly, the systolic array architecture reduces the memory access time. However, the pipeline processing limits the amount of parallelism that can be exploited.

From an algorithmic view, optimizing the algorithm according to a specific architecture can reduce the number of communication steps. For the shared memory architecture, the recursive array layout in [12] reduces false sharing and cache conflicts that avoid unnecessary communication between the main memory and the processors at the cost of extra addressing overhead. For distributed memory architectures, replication techniques were developed to reduce the communication steps at the cost of using extra memory to store matrix copies [13]. For systolic array architectures, the number of communication steps is reduced by loading the data onto a 2D mesh at the expense of more registers [14]. Algorithmic optimizations are limited by the existing architectural constraints while architectural improvements are limited by technology constraints. Therefore, to obtain further performance improvement beyond the above limitations, radical solutions are required that combine new architectures based on emerging technology with new algorithmic optimization.

To solve the communication issue, we use memristor-based CIM architecture [15] and propose a communication-efficient mapping scheme for parallel matrix multiplication algorithm. We alleviate the communication bottleneck by maximizing parallelism in communication while minimizing the communication distance. The main contributions of this paper are:

- A parallel programming model that bridge the gap between the parallel matrix multiplication algorithm and the memristor-based CIM architecture.
- A parallel cost model that is used to evaluate the performance of parallel matrix multiplication algorithm on memristor-based CIM architecture.
- Data and task mapping scheme for parallel matrix multiplication algorithm on memristor-based CIM architecture based on the Z-order traversal and H-tree topology.

The rest of this paper is organized as follows. Section II describes the memristor-based CIM architecture. Section III presents the parallel matrix multiplication algorithm. Section IV provides the CIM model of parallel computation. Section V shows our evaluation results. Section VI concludes the paper.

II. Memristor-based CIM Architecture

A memristor-based CIM architecture is a radically new computing architecture that integrates computation and memory in the same physical location, rather than Von Neumann architecture where the computation and memory are separated. Memristor-based CIM architecture [15] was shown to perform two to three orders of magnitude better than conventional architectures in computation, energy and area efficiency both for DNA sequencing and parallel additions.

Fig. 1 shows a memristor-based CIM architecture that consists of a computing-in-memory core, the secondary storage and the interconnection between them. The computing-in-memory core includes memory to hold the *working set*, loosely defined as the collection of information referenced by a program during its execution. This core memory is initialized with data originating from the secondary storage. The computing-in-memory core consists of two main parts: a memristor-based crossbar array and a controller. The array uses a grid structure to connect memristor devices using horizontal and vertical nano-wires. The controller, which is based on CMOS technology, provides appropriate control (voltage) signals to enable rows and columns in the crossbar to perform arithmetic logic operations. Furthermore, the controller is also responsible for memory read and write operations, and it enables communication within the crossbar array (through the vertical and horizontal nano-wires) or through its CMOS layer.

The major advantages of memristor-based CIM architectures are that a massive amount of parallel computation and communication operations can be performed within the high-density crossbar and that multiple memristor layers can be stacked on top of the CMOS layer. Furthermore, CIM architecture reduces the communication latency as the memories, and the computations are not separated. As a result, the computations can be executed locally without the need to move the data to off-chip processing units. Finally, the CIM architecture has the potential to improve performance while consuming little power and using a small area. These last benefits are possible due to the zero energy leakage, high scalability and high-density integration of the memristor device.

III. Parallel Matrix Multiplication Algorithm

Matrix multiplication is a key function of many data-intensive applications such as data recognition, data mining, and data synthesis [16]. This function contains a huge amount of parallelism that can be exploited. Equation (1) shows the mathematical definition of element C_{ij} (i.e., the output of matrix C on row i and column j) of the matrix multiplication algorithm. In this equation, A and B represent the input matrices and C the output matrix.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj} \tag{1}$$

Parallelism exploitation of the matrix multiplication can be classified by its degree of parallelism; 1D, 2D, and 3D parallel algorithms have been proposed. These algorithms impact the required number of parallel multiplications and additions. The 1D parallel algorithm computes n output rows of matrix elements in parallel by executing the multiply-add operation for each element sequentially. The 2D parallel algorithm operates on the entire output matrix n^2 in parallel and computing the multiply-add operation sequentially. The 3D parallel algorithm exploits the highest degree of parallelism n^3, by computing all multiplications in parallel followed by all additions using a binary tree. Table I shows the time complexity of sequential and parallel matrix multiplication algorithms and their computing resources requirement.

Table II summarizes the state-of-the-art of matrix multiplication implementations. The 1D parallel algorithm is typically implemented on multicores, FPGAs, and GPUs. These implementations commonly achieve a sustained performance of several GFLOPS and are mostly used for smaller matrix sizes (e.g., 2000). Besides, it is difficult to exploit higher degree of parallelism on these machines due to the communication cost. The 2D parallel algorithm has been implemented for larger matrix sizes (e.g., 10000) and are usually implemented on large distributed memory systems. The 3D parallel algorithm has been implemented for large matrix sizes (e.g., 30000) on supercomputers and achieves performances in TFLOPS range.

Based on the potential of CIM architecture, we focus on the 3D parallel matrix multiplication algorithm (see Algorithm 1) due to its maximum degree of parallelism. Many researchers have investigated the 3D parallel algorithm in various architectures such as hypercubes [23], multiprocessors [24], [25], massively parallel processing systems [13] and supercomputer clusters [22]. The 3D parallel algorithm focuses on an element-wise matrix distribution rather than the block-wise distribution used in the 2D parallel algorithm. For a square matrix sizes, the 3D parallel algorithm replicates n-copies of the input matrices A and B and distributes them to the particular memory locations for massively parallel computation.

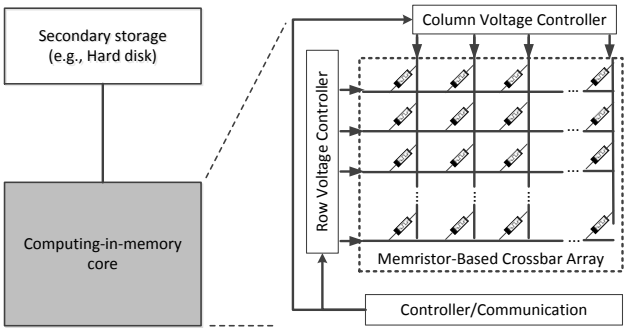

Fig. 1. Memristor-based CIM architecture

TABLE I
Matrix Multiplication (MM) Computational Complexity

Algorithm	Time complexity	Multipliers	Adders
Sequential MM	$\mathcal{O}(n^3)$	1	1
1D Parallel MM	$\mathcal{O}(n^2)$	n	n
2D Parallel MM	$\mathcal{O}(n)$	n^2	n^2
3D Parallel MM	$\mathcal{O}(log_2 n)$	n^3	n^3 - n^2

TABLE II
1D, 2D AND 3D MATRIX MULTIPLICATION IMPLEMENTATION

Machine/platform	Algorithm Classification	Matrix Size	Computing units	Data Type	Optimization	Throughput
Virtex-6475T [17]	1D	8	8	32 integer	unroll/pipe	10.0 GOPS
Virtex-7 XC7VX690T [18]	1D	512	512	16bit FP	energy	367 GFLOPS
Virtex-6 [19]	1D	500	500	Double-precision	Poly	2.3 GFLOPS
Virtex-6 LX240T [20]	1D	1024	1024	32 integer	Tiling	6 GOPS
GTX480 GPU [21]	1D	128	448	Single-precision	CUBLAS	541 GFLOPS
Intel Paragon system [9]	2D	10000	465	Double-precision	SUMMA	19.5 GFLOPS
Intel Paragon system [9]	2D	10000	465	Double-precision	PUMMA	11.6 GFLOPS
IBM POWERparallel[13]	3D	5000	216	Double-precision	PDGEEM	48.9 GFLOPS
IBM POWERparallel[13]	3D	5000	216	Double-precision	PZGEEM	77.9 GFLOPS
IBM Blue Gene [22]	3D	30000	8000	Double-precision	Type: A	4.0 TFLOPS
IBM Blue Gene [22]	3D	30000	8000	Double-precision	Type: B	4.8 TFLOPS
IBM Blue Gene [22]	3D	30000	8000	Double-precision	Type: C	5.6 TFLOPS

Algorithm 1 3D Parallel Matrix Multiplication Algorithm

Input: A, B, ▷ A and B are m-by-p and p-by-n
Output: C = $A{\cdot}B$ where C is m-by-n

1: **procedure** MATRIX MULTIPLY(INT m, INT n, INT p)
2: **for all** i, j, $k \in$ (1 to m, n, p) **do**
3: $C_{ijk} = A_{ik} \times B_{kj}$
4: **for all** i, $j \in$ (1 to m, n) **do**
5: **for** s = 1 to log_2 (p) **do**
6: **for all** k_{odd} = 1 to ($p/2^{s-1}$) **do**
7: $C_{ijk_{odd}} = C_{ijk_{odd}} + C_{ij(k_{odd}+2^{s-1})}$
8: $C_{ij} = C_{ij1}$

IV. CIM MODEL OF PARALLEL COMPUTATION

This section presents first the CIM parallel programming model for the memristor-based CIM architecture and thereafter the CIM parallel cost model used to evaluate the parallel matrix multiplication algorithm. Finally, we describe the data mapping and the task mapping schemes of the parallel matrix multiplication algorithm on the computing-in-memory core.

A. CIM Parallel Programming Model

The CIM parallel programming model is a process model that describes the basic operations and procedures to perform computations on a memristor-based CIM architecture. The model bridges the gap between the algorithmic and the architectural abstraction layers. Based on the working set on-chip location in the CIM architecture, we define the parallel programming model for CIM architecture as follows:

- *Map data:* Broadcast or scatter the input data from the secondary storage to the computing-in-memory core.
- *Compute and Communicate:* Compute the tasks in parallel and communicate between tasks in parallel interchangeably within the computing-in-memory core.
- *Readout data:* Gather the output data from the computing-in-memory core to the secondary storage.

Initially, the data set is located in the secondary storage. When the computation should be performed, the data is mapped on the computing-in-memory core for massively parallel computation. All the input data is placed on the computing-in-memory core in a particular location so that parallel computation and parallel communication between tasks can be executed efficiently. Lastly, the final output data is readout from the computing-in-memory core and stored back to the secondary storage.

We assume that there is a controller that can map and readout data between the secondary storage and the computing-in-memory core. The data can be mapped to any location within the computing-in-memory core due to the resistive-switching property of the memristor device. Therefore, we have full flexibility to map an enormous amount of data on the computing-in-memory core in designated places. Furthermore, we assume that we can perform any arithmetic operation on the data.

B. CIM Parallel Cost Model

The CIM parallel cost model is a mathematical model that is used to describe the cost of parallel matrix multiplication algorithm on memristor-based CIM architecture corresponding to the CIM parallel programming model. The CIM parallel cost parameters are the combination of four attributes:

- *Mapping cost*: The cost to move each *word* (α) from the secondary storage to computing-in-memory core.
- *Computational cost*: The cost to perform a single arithmetic function (β) within computing-in-memory core.
- *Communication cost*: The cost to move each *word* (γ) within a computing-in-memory core.
- *Readout cost*: The cost to move each *word* (δ) from computing-in-memory core to secondary storage.

Mapping cost involves the number of *words* (M) moving from the secondary storage to the computing-in-memory core. This cost is the summation of the data transfer and memory access costs. The data transfer cost (bandwidth) is the cost to move data from the secondary storage to the computing-in-memory core. The memory access cost (latency) is the cost to read data from secondary storage and to write data in computing-in-memory core.

Computational cost reflects the number of arithmetic operations (N) within the computing-in-memory core. The cost can be different between several arithmetic logic units (i) such as adder and multiplier. The cost depends on the mathematical model of memristor-based logic such as IMPLY logic [26].

Communication cost includes the number of *words* (O) moving within the computing-in-memory core. That is, the cost associated with the communication between tasks. This cost can be calculated after the data and the tasks are mapped on the computing-in-memory core. However, the basic cost of data movement on the computing-in-memory core can be divided into three categories: (and as illustrated in Fig. 2.)

- *zero-step*: The data stored in memristor A can be used for the next computation directly without the need to move the data.
- *one-step*: The data stored in memristor A need to move horizontally to memristor B for the next computation. It has the same cost if the data need to move vertically.
- *two-steps*: The data stored in memristor A need to move horizontally to memristor B and vertically to memristor C for the next computation.

Readout cost denotes the number of *words* (P) moving from the computing-in-memory core to the secondary storage. The cost is also associated with data transfer cost (bandwidth) and memory access cost (latency) but in the reverse direction.

With the above CIM parallel cost parameters, we can model the general total cost (T) of parallel matrix multiplication on memristor-based CIM architecture as shown in (2):

$$T = M \cdot \alpha + \sum_{i=1}^{n} N_i \cdot \beta_i + O \cdot \gamma + P \cdot \delta \qquad (2)$$

Two main strategies are adopted to reduce the total cost: First, we avoid unnecessary data transfer between the secondary storage and computing-in-memory core. Second, we reduce data movements within the computing-in-memory core via efficient data and task mapping schemes. These schemes are presented next.

C. Data Mapping Scheme

The main goal of the data mapping is to avoid unnecessary data movements between communicating tasks by increasing the data locality within the computing-in-memory core. An optimal data mapping prevents any additional communication cost. The data mapping specifies where the input data has to be loaded before computing the tasks. Therefore, the input data of the 3D matrix multiplication must be replicated and mapped onto the 2D computing-in-memory core, i.e., the 3D algorithmic index space must be converted to a 2D physical index space. Our data mapping scheme consists of two algorithms. The first algorithm, (see Algorithm 2) divides the disjoint tasks that produce output elements such as $c11$ and $c12$ into physical regions; each region is responsible for computing a single element of the output matrix. The second algorithm, (see Algorithm 3) is responsible for mapping the input data into each region.

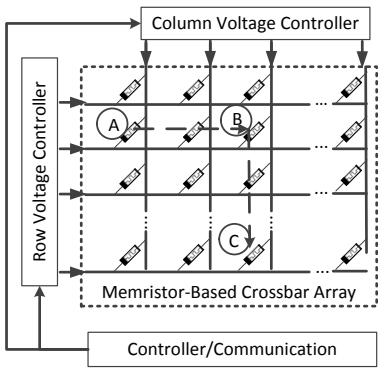

Fig. 2. Data movement on memristor-based crossbar array

Algorithm 2 Row Major-order Data Mapping Algorithm

Input: m, n, p ▷ input matrices are m-by-n and n-by-p
Output: LISTS ▷ two-dimensional array of coordinate lists

```
1: procedure MATRIXMAPPING(INT m, INT n, INT p)
2:     LISTS = new vector<INT_PAIR> * m * p
3:     INT ORDER = ceil(log4(n))
4:     for i = 0 to m − 1 do
5:         for j = 0 to p − 1 do
6:             RECURSIVE(LISTS[i][j], ORDER, 2*i, 2*j)
7:     return LISTS
```

Algorithm 3 Z-order Data Mapping Algorithm

Input: X ▷ X coordinate of current Z-traversal base
Input: Y ▷ Y coordinate of current Z-traversal base
Input: ORDER ▷ Recursive level
Output: INDEX ▷ Coordinate list following Z-order

```
1: procedure ADDTOINDEX(VECTOR<INT_PAIR> INDEX, INT X, INT Y)
2:     INDEX.append(INT_PAIR(X, Y))
3: procedure RECURSIVE(VECTOR<INT_PAIR> INDEX, INT ORDER, INT X, INT Y)
4:     if ORDER > 1 then
5:         RECURSIVE(INDEX,ORDER-1, 2*X, 2*Y)
6:         RECURSIVE(INDEX,ORDER-1,(2*(X+1), 2*Y)
7:         RECURSIVE(INDEX,ORDER-1,(2*X, 2*(Y+1))
8:         RECURSIVE(INDEX,ORDER-1,(2*(X+1), 2*(Y+1))
9:         return
10:    ADDTOINDEX(INDEX,X+1,Y)
11:    ADDTOINDEX(INDEX,X+2,Y)
12:    ADDTOINDEX(INDEX,X+1,Y+1)
13:    ADDTOINDEX(INDEX,X+2,Y+1)
```

Fig. 3 illustrates the results of both algorithms. It shows the data layout of the input matrices for a 16-by-16 matrix multiplication. The output elements are divided into regions by using the row-major ordering (from left to right) as presented in Algorithm 2 (see line 4 to 7); they are responsible for producing all the entries of the output matrix C. The data within a region is mapped using Z-order traversal; the arrows present the order in which the inputs are mapped; note that they follow a Z-order traversal as presented in Algorithm 3 (see line 5 to 13). Each calculated output element consists of the dot product of a row vector of matrix A and a column vector of matrix B. Therefore, two Z-order traversal are used for each region to map the 16 input elements both from a row of matrix A and a column of matrix B. Note that the Z-order for matrix B is applied after transposing it.

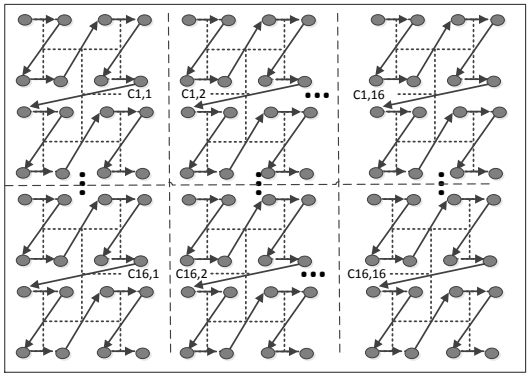

Fig. 3. Data mapping on the computing-in-memory core

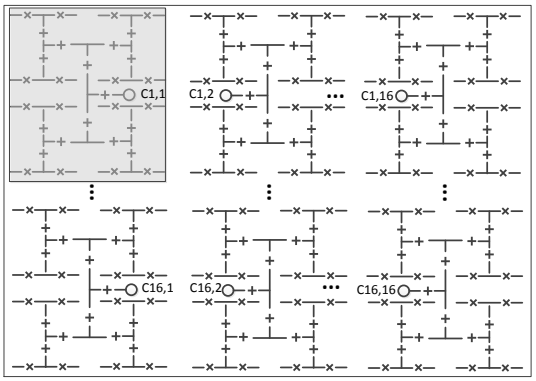

Fig. 4. Task mapping on the computing-in-memory core

The general relation between the Z-order depth and the matrix size is expressed by $ZOrder = log_4(n)$; it represents the number of hierarchical Z-shaped pattern levels that are used to map the input matrix data on the computing-in-memory core. Please note that if the matrix size is not a power of four, we can use a virtual padding technique to append zeros to the original matrices to force the order to a power of four. Alternatively, we could split the rounded Z-order into smaller Z-orders.

D. Task Mapping Scheme

The primary goal of the task mapping is to maximize parallelism in communication between tasks and minimize their communication distance. An efficient task mapping can alleviate the communication bottleneck (congestion) that reduce the communication cost. For the 3D matrix multiplication, the task mapping specifies where the additions and multiplications are executed and how these tasks communicate with each other within the computing-in-memory core.

The communication between tasks in 3D matrix multiplication is characterized by the binary tree structure. To map the tasks from the binary tree to the computing-in-memory core, we need a suitable arrangement of adders, multipliers, and their interconnection on the computing-in-memory core. Fig. 4 illustrates our proposed arrangement of the adders, multipliers, and their interconnection through the H-tree topology for a 16-by-16 matrix multiplication. The shaded rectangle is responsible for the computation of the first element of the output matrix. This block is repeated over the crossbar area to maximize the parallelism, and each block produces an element of the result matrix C_{ij}, represented by the circle shape. Applying the H-tree topology on the computing-in-memory core has the potential to reduce the communication cost as follows:

- *Bandwidth cost:* The H-tree topology provides a specific path for each communication between tasks (e.g., multiplications to additions). The communication can be performed in parallel without having congestions. Therefore, they apply zero bandwidth cost.
- *Latency and energy cost:* The H-tree topology allows the output data of multiplications can be used directly for the next computation (i.e., addition) without the need to move the data. In other words, The output data of multiplications can be stored directly as the input data for additions. Therefore, they apply *zero-step* data movement and thus, avoid the communication cost. However, long-distance communication between two tasks requires *one-step* data movement.

The recursiveness of H-tree topology provides efficient scaling when the matrix size grows. Therefore, an arbitrary number of matrix sizes can be mapped on the memristor crossbar using the H-tree topology. The H-tree topology is comprised of basic fractal units arranged in an H-pattern that consists of four multipliers and two adders. Each higher level fractal is constructed from an H-pattern of smaller level fractals. The general relation between the order (i.e., the depth or number of recursions) of the H-tree topology and matrix sizes is expressed by $HtreeOrder = log_2(n) - 1$.

The data mapping and task mapping are depicted in more details in Fig. 5 in which the layout presents half of the shaded area of Fig. 4 and belongs to the output C_{11}. The scheduling of the layout follows exactly as the Algorithm 1. Finally, because both the task mapping (H-tree topology) and the data mapping (Z-order traversal) are fixed, the compiler can compute the exact locations of the final results for the readout operations.

Please note that the purpose of the H-tree topology is to evaluate the performance of parallel matrix multiplication on the memristor-based CIM architecture. Therefore, the hardware implementation details are not discussed.

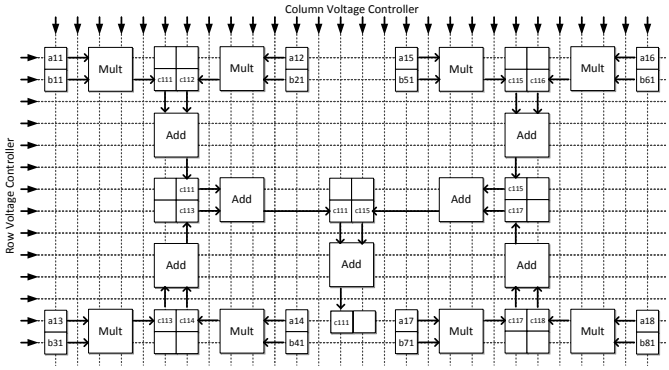

Fig. 5. Layout on the computing-in-memory core

TABLE III
DEVICE TECHNOLOGY AND ARCHITECTURAL PARAMETERS FOR MULTICORE AND CIM MODEL

	Multicore model	CIM model
	Device Technology	
Technology	FinFET	Memristor
Feature size	22 nm	5 nm [27]
Frequency	3.3 GHz	5.0 Ghz [28]
Delay	14 ps [29], [30]	200 ps (1/Frequency) [28]
Dynamic energy	0.00245 fJ (175 nW/gate [29]*Delay)	1 fJ [27]
Static energy	0.01297 fJ (42.83 nW/gate [27]*1/Frequency)	Zero leakage
Area	2.48×10^{-1} $u\text{m}^2$ [27]	1×10^{-4} $u\text{m}^2$ [27]
	Architecture	
Half adder	1-bit half adder (NAND gates)	1-bit IMPLY Half Adder [31], [26]
Delay	42 ps [3 cycles×14 ps]	1400 ps [7 cycles×200 ps]
Energy	0.077 fJ (Dynamic energy + Static energy)	1.75 fJ [0.25×7 switching×1fJ]
Area	12.40×10^{-1} $u\text{m}^2$ [5 gates×2.48×10^{-1} $u\text{m}^2$]	4×10^{-4} $u\text{m}^2$ [4 memristors×1×10^{-4} $u\text{m}^2$]
Full adder	1-bit full adder (NAND gates)	1-bit IMPLY Full Adder (N=1) [26]
Delay	84 ps [6 cycles×14 ps]	5800 ps [29N cycles×200 ps]
Energy	0.139 fJ (Dynamic energy + Static energy)	7.25 fJ [0.25×29N switching×1fJ]
Area	22.32×10^{-1} $u\text{m}^2$ [9 gates×2.48×10^{-1} $u\text{m}^2$]	$6\times10^{-4}u\text{m}^2$ [(3N+3) memristors×$1\times10^{-4}um^2$]
Adder	Rippled eight 4-bit Carry look ahead adder [32]	32-bit IMPLY Adder (N=32) [26]
Delay	252 ps [18 cycles×14 ps]	185600 ps [29N cycles×200 ps]
Energy	3.209 fJ (Dynamic energy + Static energy)	232 fJ [0.25×29N switching×1fJ]
Area	515.84×10^{-1} $u\text{m}^2$ [208 gates×2.48×10^{-1} $u\text{m}^2$]	$99\times10^{-4}u\text{m}^2$ [(3N+3) memristors×$1\times10^{-4}um^2$]
Multiplier	32-bit Wallace Tree Multiplier (N=32) [33]	32-bit Wallace Tree Multiplier (N=32) [33]
Delay	672 ps [48 cycles×14 ps]	46400 ps [232 cycles×200 ps]
Energy	127.71 fJ [Dynamic energy + Static energy]	6616 fJ [6616 switching×1fJ]
Area	20529.44×10^{-1} $u\text{m}^2$ [8278 gates×2.48×10^{-1} $u\text{m}^2$]	$5534\times10^{-4}u\text{m}^2$ [5534 memristors×1×10^{-4} $u\text{m}^2$]
Cache	8kB L1 cache/cluster	No Cache
Hit rate	0.98	
Load delay on hit	1 cycle	
Missed penalty	165 cycles [34]	
Dynamic energy	0.09737 nJ/cache [35]	Not Applicable
Static energy	0.00262 nJ/cache [35]	
Area	0.1098 mm^2 [35]	

V. EVALUATION RESULTS

This section compares the scalability performance of CIM architecture against the multicore. As the target is to measure the scalability for a large-scale matrix multiplication, models of parallel computation of the multicore and CIM architecture will be used. First, we describe for both architectures; their device technology and architectural parameters. Next, we validate the multicore model by comparing its performance with the IBM PowerLinux 7R2 multicore system for limited matrix sizes. Finally, we compare the CIM model against the validated multicore model for the extreme matrix sizes.

A. Multicore and CIM Model

Both the multicore and CIM models consist of two parameter sets: device technology and architecture. The multicore and CIM model are based on FinFET 22nm and memristor 5nm technology, respectively. The device technology parameters can be found in the upper part of Table III and are used to estimate the computational and communication cost such as adders, multipliers, and caches. The architectural parameters are tabulated in the lower part of Table III and are used to evaluate the scalability performance of parallel matrix multiplication algorithm.

The multicore model based on the load-and-store principle and is assumed to be a scalable multicore architecture, consisting of a number of clusters; each cluster has four adders, four multipliers, and 8kB of shared cache. For the communication cost, we assume a high cache hit rate (98%), independent of the matrix size. Moreover, we use a cache-simulation tools such as CACTI-P [35] to obtain the value of dynamic energy, static energy and area of the cache. The CIM model is assumed to be scalable as well, consisting of adders, multipliers and without any caches due to the concept of Computation-in-Memory. The computation cost of the adder and multiplier are based on the memristor-based material implication (IMPLY) logic [26].

We do not take into account the mapping and readout cost for both multicore and CIM architecture as we evaluate a single parallel algorithm only. Besides, both costs are more related to data transfer. Therefore, we only consider the computational and communication cost for parallel matrix multiplication algorithm. Both the multicore and CIM models are verified in Matlab.

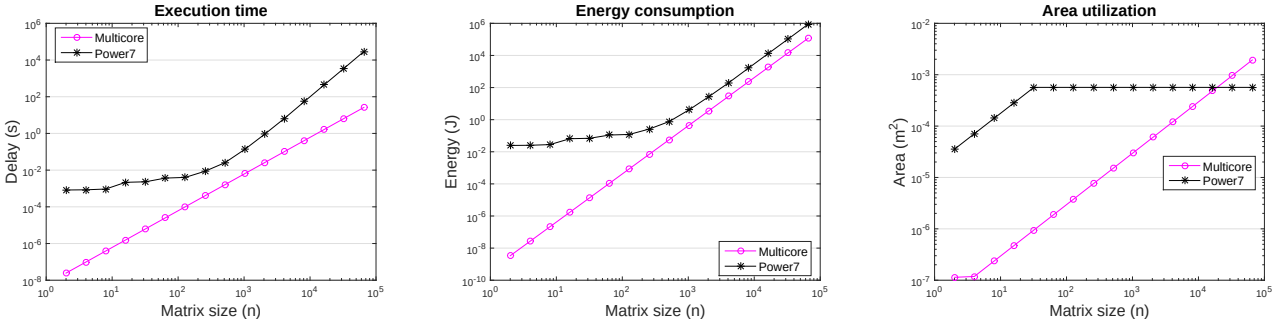

Fig. 6. Multicore (optimistic model) vs Power7 (actual platform)

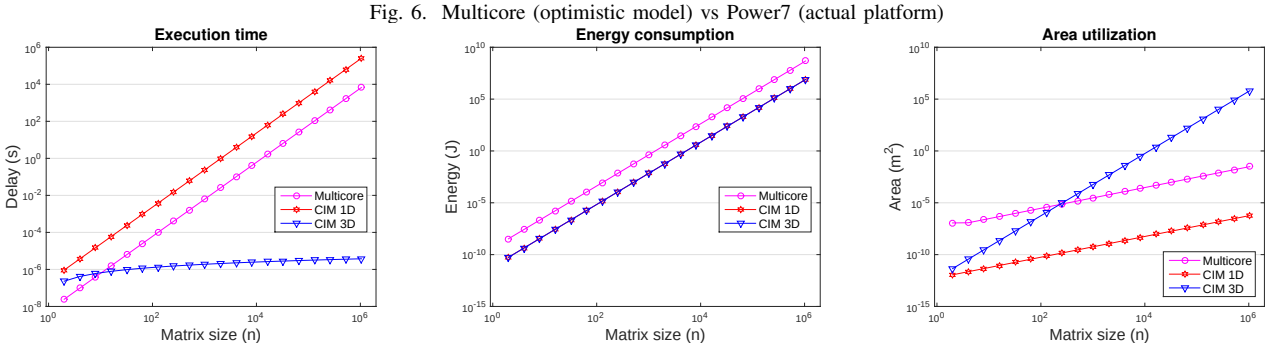

Fig. 7. Multicore vs CIM 1D vs CIM 3D (intrinsic parameters)

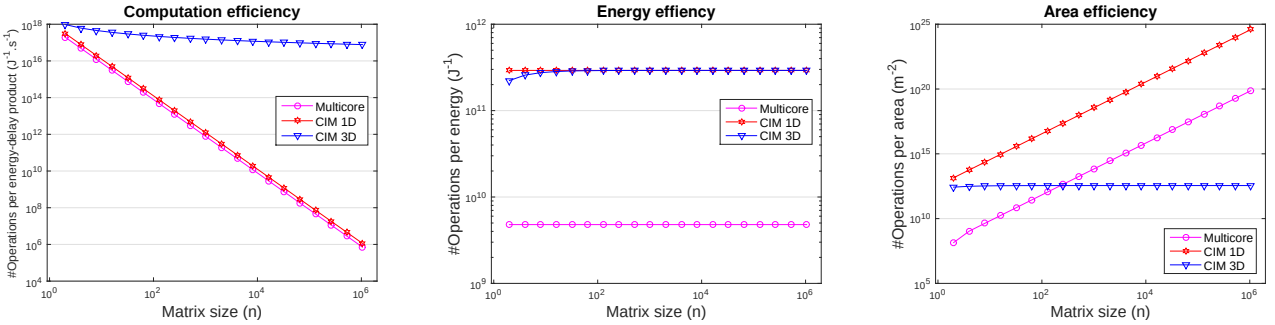

Fig. 8. Multicore vs CIM 1D vs CIM 3D (derived parameters)

B. Multicore Model Validation

The multicore model is validated by comparing it against the IBM PowerLinux 7R2 multicore (Power7) [36]. Fig. 6 shows their performances with respect to execution time, energy consumption, and the area utilization; both multicores are based on the similar 1D matrix multiplication. The multicore outperforms Power7 in delay and energy for all matrix sizes (i.e., 2^1 to 2^{16}) which is expected due to the simplified multicore model. For the area, the multicore model utilizes less area than Power7 for the relevant matrix sizes. Note that the multicore model scales with the matrix size without restrictions, while the Power7 has a fixed maximum area. The Power7 is an actual multicore platform that hardly scales due to the memory and communication bottlenecks. In particular, the cache is too small for a large data set and the bandwidth between processors and memory is limited. The microarchitecture of the chip is complex due to the control units, cache hierarchy, and on-chip interconnection. Meanwhile, the multicore model is based on an ideal shared memory architecture specifically tweaked to perform matrix multiplications. It consists of adders, multipliers, and caches only without any control units such as branch prediction unit. The communication cost of multicore is considered optimistic. Consequently, we conclude that our multicore model is an optimistic model of Power7 and, therefore, it is fair to compare against the CIM model.

C. CIM Performance Evaluation

The multicore model is compared against two CIM models: CIM 1D and CIM 3D. CIM 1D is based on the same 1D parallel matrix multiplication used in the multicore model, while CIM 3D is based on the 3D parallel matrix multiplication. The comparison allows us to conclude whether the performance gains come from a technology change or architectural improvements or a combination of them. Table IV summarizes the three models; it shows their time complexity, the number of multiplications and additions (operations) and the number of multipliers and adders (resources). The time complexity strongly determines the execution time, the number of operations is used to estimate the energy consumption, and the number of resources is used to estimate the area utilization.

Fig. 7 shows the execution time, energy consumption and area utilization of the three models for various matrix sizes

TABLE IV
MODELS AND THEIR COMPUTATIONAL COMPLEXITY

Model	Time complexity	# Multiplications & # Additions	# Multipliers & # Adders
Multicore	$\mathcal{O}(n^2)$	n^3 & n^3	n & n
CIM 1D	$\mathcal{O}(n^2)$	n^3 & n^3	n & n
CIM 3D	$\mathcal{O}(log_2 n)$	n^3 & n^3-n^2	n^3 & n^3-n^2

(i.e., 2^1 to 2^{20}). Multicore performs two orders of magnitude better with respect to delay than the CIM 1D due to the faster FinFET technology. However, the multicore performs two orders of magnitude worse with respect to energy than the CIM 1D due to the static and dynamic energy consumption of the cache. In addition, the multicore has five orders of magnitude larger area than the CIM 1D mostly due to the size of the cache. Generally, if the same class of algorithm is applied to both models, CIM model shows benefits in terms of energy and area only. Meanwhile, CIM 3D outperforms the multicore regarding delay by one to nine orders of magnitude improvement depending on the matrix size. CIM 3D gains its performance through the exploitation of massive parallelism even though memristor technology is slower than FinFET. Moreover, CIM 3D shows two orders of magnitude better energy consumption than the multicore for each matrix size. As a trade-off, CIM 3D requires one to seven orders of magnitude more resources depending on the matrix size than the multicore due to the resources needed by the 3D parallel matrix multiplication algorithm.

Fig. 8 depicts the efficiency of computation, energy, and area of the models derived from the intrinsic parameters from Fig. 7. With respect to computation efficiency (i.e., the number of operations per the energy-delay product), CIM 3D outperforms both multicore and CIM 1D immensely due to its remarkable delay performance. On average, CIM 3D shows six orders and five orders of magnitude higher than multicore and CIM 1D respectively. With respect to energy efficiency (i.e., the number of operations per energy), CIM 3D and CIM 1D have better energy efficiency than multicore due to their lower energy consumption. With respect to the area efficiency (the number of operations per area), CIM 3D has a constant efficiency due to the same change of the number of operations and the number of resources.

VI. CONCLUSION

We show the capability of memristor-based CIM architecture in exploiting the maximum amount of parallel matrix multiplication algorithm while minimizing their communication cost via our communication-efficient mapping scheme. The results clearly show that CIM architecture has the potential to outperform traditional shared-memory multicore architecture.

REFERENCES

[1] L. I. Millett *et al.*, *The Future of Computing Performance:: Game Over or Next Level?* NAP, 2011.

[2] C. A. Patterson *et al.*, *Getting Up to Speed:: The Future of Supercomputing*. NAP, 2005.

[3] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, 2014.

[4] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *COMMUN ACM*, 1978.

[5] G. Tao *et al.*, "Using MIC to accelerate a typical data-intensive application: the breadth-first search," in *IPDPSW*. IEEE, 2013.

[6] A. Heinecke and M. Bader, "Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms," in *MAW*. ACM, 2008.

[7] M. Gusev and S. Ristov, "A superlinear speedup region for matrix multiplication," *CCPE*, 2014.

[8] J. Choi *et al.*, "PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *CCPE*, 1994.

[9] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *CCPE*, pp. 255–274, 1997.

[10] I. Milovanović *et al.*, "Forty-three ways of systolic matrix multiplication," *IJCS*, 2010.

[11] G. Kuzmanov and W. M. Van Oijen, "Floating-point matrix multiplication in a polymorphic processor," in *ICFPT*. IEEE, 2007.

[12] S. Chatterjee *et al.*, "Recursive array layouts and fast matrix multiplication," *PDS*, 2002.

[13] R. C. Agarwal *et al.*, "A three-dimensional approach to parallel matrix multiplication," *IBM J RES DEV*, 1995.

[14] S. E. Bae *et al.*, "A faster parallel algorithm for matrix multiplication on a mesh array," *PCS*, 2014.

[15] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *DATE*. EDA Consortium, 2015.

[16] Y.-K. Chen *et al.*, "Convergence of recognition, mining, and synthesis workloads and its implications," *Proc. of the IEEE*, 2008.

[17] S. Skalicky *et al.*, "High level synthesis: Where are we? A case study on matrix multiplication," in *ReConFig*. IEEE, 2013.

[18] K. K. Matam and V. K. Prasanna, "Energy-efficient large-scale matrix multiplication on FPGAs," in *ReConFig*. IEEE, 2013.

[19] W. Zuo *et al.*, "Improving polyhedral code generation for high-level synthesis," in *CODES*. IEEE Press, 2013.

[20] Y. Liang *et al.*, "High-level synthesis: productivity, performance, and software constraints," *JECE*, 2012.

[21] E. S. Chung *et al.*, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *MICRO*. IEEE CS, 2010.

[22] M. D. Schatz, J. Poulson, and R. A. van de Geijn, "Scalable universal matrix multiplication algorithms: 2d and 3d variations on a theme," *TOMS*, 2012.

[23] J. Berntsen, "Communication efficient matrix multiplication on hypercubes," *PC*, 1989.

[24] S. L. Johnsson, "Minimizing the communication time for matrix multiplication on multiprocessors," *Parallel Computing*, 1993.

[25] H. Gupta and P. Sadayappan, "Communication efficient matrix multiplication on hypercubes," in *SPAA*. ACM, 1994.

[26] S. Kvatinsky *et al.*, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *VLSI*, 2014.

[27] (2010) ITRS Emerging Research Devices (ERD) report. [Online]. Available: http://www.itrs.net

[28] A. C. Torrezan *et al.*, "Sub-nanosecond switching of a tantalum oxide memristor," *Nanotechnology*, 2011.

[29] C. Meinhardt and R. Reis, "FinFET basic cells evaluation for regular layouts," in *LASCAS*, 2013.

[30] A. Muttreja *et al.*, "CMOS logic design with independent-gate FinFETs," in *ICCD*. IEEE, 2007.

[31] J. Borghetti *et al.*, " 'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, 2010.

[32] P. Behrooz, "Computer arithmetic: Algorithms and hardware designs," *Oxford University Press*, 2000.

[33] S. Kakde *et al.*, "Design of area and power aware reduced Complexity Wallace Tree multiplier," in *IEEE, ICPC*. IEEE, 2015.

[34] D. Levinthal, "Performance analysis guide for intel core i7 processor and intel xeon 5500 processors," *Intel Performance Analysis Guide*, 2009.

[35] S. Li *et al.*, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *ICCAD*. IEEE, 2011.

[36] B. Sinharoy *et al.*, "IBM POWER7 multicore server processor," *IBM J RES DEV*, 2011.