Skeleton-based design and simulation flow for Computation-in-Memory architectures

Yu, Jintao; Nane, Razvan; Haron, Adib; Hamdioui, Said; Corporaal, H; Bertels, Koen

DOI
10.1145/2950067.2950071

Publication date
2016

Document Version
Accepted author manuscript

Published in
2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)

Skeleton-Based Design and Simulation Flow for Computation-In-Memory Architectures

Jintao Yu Razvan Nane Adib Haron Said Hamdioui Henk Corporaal* Koen Bertels
Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
*Eindhoven University of Technology, Postbus 513, 5600 MB Eindhoven, The Netherlands
{J.Yu-1, R.Nane, M.A.B.Haron, S.Hamdioui, K.L.M.Bertels}@tudelft.nl
H.Corporaal@tue.nl

November 18, 2016

Abstract

Memristor-based Computation-in-Memory is one of the emerging architectures proposed to deal with Big Data problems. The design of such architectures requires a radically new automatic design flow because the memristor is a passive device that uses resistance to encode its logic value. This paper proposes a design flow for mapping parallel algorithms on the CIM architecture. Algorithms with similar data flow graphs can be mapped on the crossbar using the same template containing scheduling, placement, and routing information; this template is named *skeleton*. By configuring such a skeleton with different pre-designed circuits, we can build CIM implementations of the corresponding algorithms in that class. This approach does not only map an algorithm on a memristor crossbar, but also gives an estimation of its performance, area, and energy consumption. It also supports user-defined constraints and parallel SystemC simulation. Experimental results demonstrate the feasibility and the potential of the approach.

1 Introduction

Big Data Analytics is becoming increasingly difficult to solve using classical Von Neumann-based computer architectures because of limited bandwidth (due to memory-access bottlenecks), energy inefficiency and limited scalability (due to CMOS technology). Computation-in-Memory (CIM)-based [1, 2] architectures address the aforementioned problems by enabling in-memory computations using non-volatile memristor technology [3, 4]. They have huge potential and they could outperform the state-of-the-art with orders of magnitude [2, 5]. Exploring the potential of such architectures and appropriately evaluating their performance and scalability for larger applications require automatic flows and methods that efficiently map high-level algorithmic description to low-level memristor crossbar.

VLSI (Very-Large-Scale Integration) CAD (Computer Aided Design) flows for CMOS-based hardware solutions are not applicable to memristor-based CIM because of different signal propagation styles. In CMOS circuits, logic values are represented by the voltage; they propagate along wires implicitly and the propagation finishes within one clock cycle [6]. Because memristors are passive devices that use resistance to encode logic values, data has to be copied by controllers explicitly. This requires specific commands and several clock cycles; In addition, it depends not only on the relative positions of the source and sink [7], but also on their [1]orientations on the crossbar. In conventional VLSI CAD flows, placement and routing are performed based on the High-Level Synthesis (HLS) scheduling results [8]. However, in memristor-based CIM, placement and routing information is required before scheduling can be performed. As a consequence, a new methodology is required to appropriately design a memristor-based CIM architecture.

In this work, we propose a design and simulation flow that performs scheduling, placement, and routing simultaneously; the flow is based on the *algorithmic skeleton* [9] concept, or a *skeleton* in short. A skeleton provides an implementation template for a specific class of algorithms that have similar Data Flow Graph (DFG)s. It uses this knowledge for optimising communication and hides its implementation details from the user. Skeleton-based design flows have been used in parallel programming for supercomputers [10], GPU [11], grid structures [12], and hybrid architectures [13]. Benkrid et al. extended this concept into a *hardware skeleton* with placement information and applied it to FPGA (Field Programmable Gate

[1]Direction of source/sink input/output ports, i.e., North, South, East, West.

Arrays)-based designing [14, 15]. Hardware skeletons do not contain routing information since it can be generated by FPGA back-end tools. However, the routing information is an essential part of mapping algorithms on the memristor crossbar. We further extend the hardware skeleton concept with routing information and refer it as [2]*CIM skeleton*. This skeleton can be configured with different predesigned circuits for implementing corresponding algorithms. Furthermore, complex algorithms can be implemented by composing simple skeletons. The main contributions of this paper are:

- Extending the hardware skeleton concept by appending routing information. The extended skeleton integrate information about scheduling, placement, and routing for a class of algorithms.
- Developing a design and simulation flow for memristor-based CIM architecture based on the extended skeleton. By using a few carefully designed skeletons, many parallel algorithms can be implemented rapidly. Both a SystemC model and a layout are generated through the flow.

The rest of the paper is organized as follows. After introducing the CIM architecture in Section 2, Section 3 presents the design and simulation flow. Section 4 shows the experimental results for three study cases. Finally, Section 5 concludes the paper and discusses future research directions.

2 Background

Figure 1 shows the heterogeneous CIM/CPU computing scenario. The CPU works with memristor-based memory, including Resistive Random Access Memory (RRAM) and CIM. Besides of storing data, CIM also performs computing as an accelerator of CPU.

2.1 CIM Architecture

CIM architecture [2] consists of a memristor layer and a CMOS layer. The former is a dense crossbar with a memristor at each cross point while the latter is used to implement the controller as shown by Figure 1. Any part of the crossbar can be configured as either memory or logic, depending on the commands of the controller. The communication between logic and memories within the crossbar can be in any direction, as shown in the figure. Due to the high density of memristor technology [16, 17], a CIM chip could contain as many as 10^6 functional units. As a result, manually exploring the design space is impossible. To transfer data between functional units, we have two options. One is through the CMOS layer, which has a bandwidth limit. The other one is on the memristor crossbar, which is named as the *copy* operation [7]. The latter one is preferred, because it has higher parallelism. The state of a memristor can be copied to another in one cycle if they share the column or the row. Otherwise, this operation will take a minimum of two cycles and temporary registers will be needed. In Figure 2, A and C represent source memristors on the output ports of two multipliers while B and D are destinations memristors on two input ports of an adder. Since A and B share the row, copying the data from A to B needs only one cycle. The pseudo command of the controller is:

$S_1:\ Move\ (25,25)\ to\ (35,25)\ \{A \rightarrow B\}.$

The controller addresses the memristors with their coordinates. Different from this case, C and D are not in the same column or row. Thus, we need to divide the communication into two steps:

$S_1:\ Move\ (25,10)\ to\ (40,10)\ \{C \rightarrow E\}$

$S_2:\ Move\ (40,10)\ to\ (40,20)\ \{E \rightarrow D\}.$

[2]In the rest of the paper, *skeletons* refer to CIM skeletons.

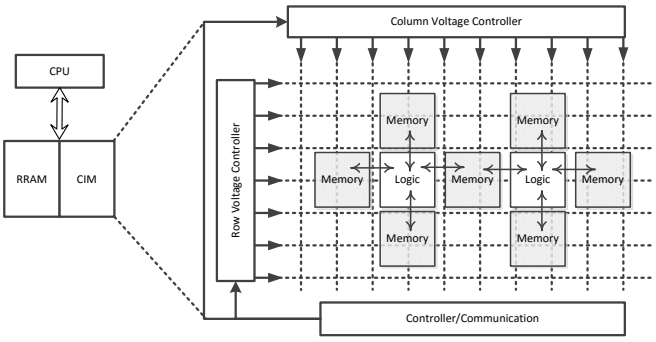


Figure 1: CIM/CPU Heterogeneous Computing and Memristor Crossbar Configuration

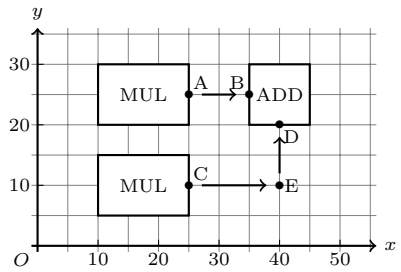


Figure 2: CIM Crossbar Communication

2.2 Requirements for a New Flow

The data transfer mechanism on memristor crossbar has a significant influence on the design flow, especially for scheduling. Scheduling is a process that assigns time steps to operations. If a design cannot meet the constraint, the compiler will allocate more resources and try to schedule again. Figure 3 illustrates the scheduling results of the vector inner product function: $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$. Here, $\vec{a} = (a_1, a_2, \ldots, a_n)$, $\vec{b} = (b_1, b_2, \ldots, b_n)^\intercal$, and the vector size $n = 4$. For explanation purposes, we set a latency constraint of 100

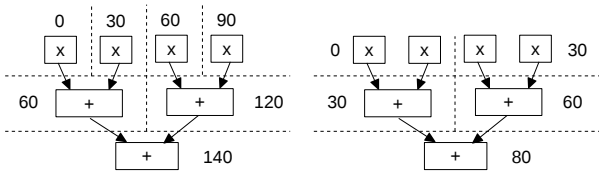


Figure 3: ASAP Scheduling of Inner Product

cycles for this function. We assume the multiplication's latency is 30 cycles and addition is 20. Now, if three adders and only one multiplier are allocated, the lowest latency of the function is 160 cycles. It can be achieved by using ASAP (As-Soon-As-Possible) scheduling algorithm. The start cycle of each operation is marked in Figure 3a. This scheduling cannot meet the constraint, so the compiler will allocate more resources and schedule again. Figure 3b shows the scheduling results with two multipliers and three adders. The overall latency is 100, which meets the constraint. From this case, we learn that scheduling validates resource allocation. Because the placement and routing are based on the allocated resources, they can only be performed after the scheduling process.

The scheduling process for memristor-based CIM also depends on routing results. Let A and B be two operations and B have a data dependency on A. For conventional CMOS technology, the data transfer from A to B is done once operation A finishes in no more than one cycle. Therefore, we can schedule B to the next time step after A without considering the routing. In CIM architecture, the data transfer needs one or more cycles, depending on the routing between them. As a result, the scheduler cannot decide the numbers of time steps between A and B without routing information. In conclusion, the scheduling, placement and routing depend on each other in CIM architecture. Hence, conventional design flows cannot be applied directly to CIM designing.

It is possible to adapt the conventional flow to CIM by iteratively executing it. However, even to get a suboptimal result, a long execution time is needed. The idea is to make assumptions on data transfers during the scheduling phase. Later, we convert these assumptions into constraints on placement and routing. If these cannot be met, then we need to increase the assumed costs and restart from scheduling. Placement and routing are time-consuming processes, so this iteration leads to a long time. Alternatively, we can make a trade-off between the quality of the solution and the execution time. In this case, the result is suboptimal.

Different from this approach, we solve the scheduling, placement and routing all together for a class of problems with a particular structure. We obtain the optimal solution without the need to iterate. This methodology is introduced next.

3 Skeleton-based Design and Simulation Flow

Figure 4 shows the overview of the complete CIM/SW compilation tool-chain, which consists of four components. At the highest level (Box 1 in the figure), the user programs an application in a high-level language, such as C, with annotations of using CIM library functions. These functions are the most time-consuming algorithms in the application, which are designed by library developers (Box 2). Their work is based on the support of skeleton designers and hardware designers, who provide the specification of fundamental skeletons (Box 3) and primitive circuits (Box 4) respectively.

In this paper, we address only the flow for the library developer, i.e., the CIM compiler. It translates algorithms into CIM implementations, including configuration files for the memristor crossbar and the controller circuits. In current research phase, we generate SystemC models for simulation, together with files that indicate the function-level layout. Section 3.1 introduces the compiler's working basis, i.e., primitive circuits and skeletons. Section 3.2 uses three examples to show the system generation process. Section 3.3 presents the support for parallel simulation.

3.1 Primitive Circuits and Skeletons

Primitive circuits form the basic structures with which we build higher-level functional blocks. They are crossbar memristor implementations of widely used operators, such as Boolean logic gates [18], adders [19], multipliers. We use SystemC models to represent them. Along with these models, the hardware designer also needs to provide the attributes of the primitive circuits, which are latency, energy, and area (i.e., width and height within the crossbar). They are used by the CIM compiler to calculate the attributes of generated functions. When a circuit is idle, it consumes no energy, due to the zero-leakage property of memristors [20]. We regard the boundary of a circuit as a rectangle. Its width and height are given by the number of memristors used in each side. We assume CIM works at a fixed frequency, and the attributes are evaluated at this frequency.

In this work, a skeleton consists of several nodes. It specifies the parallelism, communication, and synchronization of these nodes, without defining their functionality. These nodes can be configured as either primitive circuits or skeletons. When a node is configured as a primitive circuit, its functionality is decided. Configuring a node as a skeleton is called *skeleton nesting*. By using nesting repetitively, the function designer can build large and complex skeletons.

In Figure 4, the skeletons stored in the repository are provided by skeleton designers. They are called *funda-*

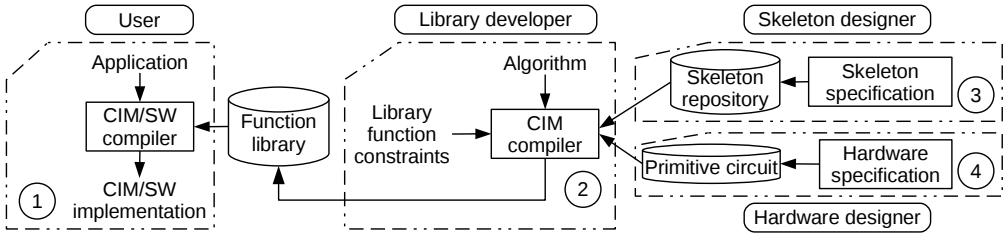


Figure 4: CIM Compilation Tool-chain

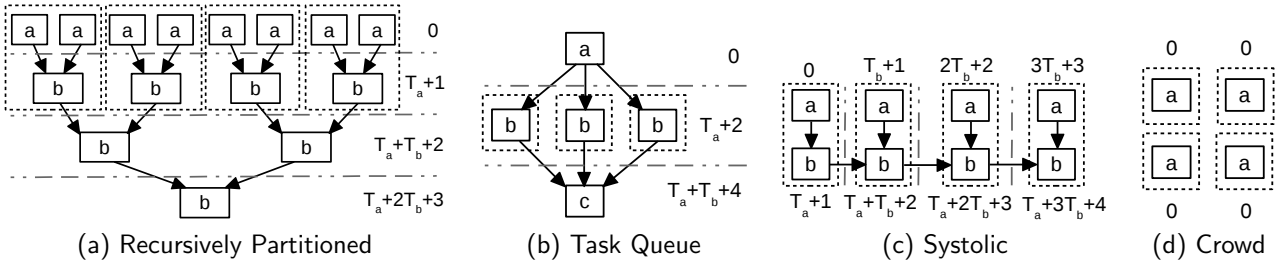


Figure 5: DFGs, Scheduling, and Parallel Simulation Support of Fundamental Skeletons

mental skeletons. The skeleton designer first needs to decide the set of fundamental skeletons, according to their expressiveness, reuse-ability, and designing difficulty. Then, he defines the scheduling, placement, and routing algorithms for each skeleton. When a skeleton is used to create library functions, the library designer does not need to care about its implementation details. We choose the fundamental skeleton set following the classification proposed by Campbell [21] and for which the DFGs are shown in Figure 5; the nodes with the same letters are configured with the same primitive circuit or skeleton. These fundamental skeletons are:

- **Recursively partitioned**. Problems are partitioned into a small size, and they are solved separately. After that, the solutions are collected in a recursive style.
- **Task queue**. These problems are solved by repeated concurrent execution of many instances of a task.
- **Systolic**. It consists of nodes that have data flowing between them and that may operate concurrently in a pipelined fashion.
- **Crowd**. Similar to the Systolic skeleton except for that there is no data flow between the concurrently operating nodes. A one-dimensional *Crowd* skeleton is an array of nodes while a two-dimensional one is a matrix.

To represent the layout, every skeleton is extended with a coordinate system. The placement of its nodes is performed under this system. A circuit rectangle may have eight different orientations, which is sufficient to be represented by an angle (0, 90, 180, and 270), and whether it reflects over the x-axis [22]. We use the coordinate of the bottom left corner as the position of the node. When skeletons are nested, their layout coordinate systems are also nested. Ports are also regarded as rectangles. Their placement is described under the coordinate system of a primitive circuit or a skeleton.

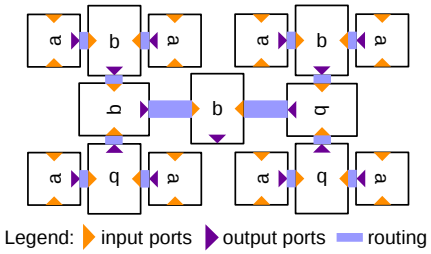


Figure 6: Three-level H-tree Layout

A skeleton is associated with a placement and routing algorithm. In the *Recursively partitioned* skeleton, nodes are arranged following the H-tree [23] pattern to minimize the communication cost. In this pattern, an output port is linked with a direct path to an input port if there is a data flow in between. Since all the corresponding bits share same rows or columns, their communication costs are just one cycle. Figure 6 shows a layout of a three-level *Recursively partitioned* skeleton as an example. In other skeletons, the nodes are placed next to each other in a matrix style.

Since a skeleton also defines the placement and routing algorithms, it knows the communication cost between operations during the scheduling phase. For instance, all the communication costs are only one cycle in *Recursively partitioned* skeletons while some of them are two cycles in *Task queue* skeletons. In Figure 5, the starting moments of the nodes' execution are marked besides the skeletons. T_x represents the latency of node x in these expressions.

3.2 System Generation

In functional programming languages, skeletons are higher-order functions. They take functions as parameters. Via these parameters, a skeleton's nodes are configured as primitive circuits or skeletons. If a skeleton is scalable, it also has parameters for configuring the size.

Suppose we want to implement the matrix multiplication algorithm:

$$AB = \begin{pmatrix} \vec{a_1}^\intercal & \vec{a_2}^\intercal & \cdots & \vec{a_n}^\intercal \end{pmatrix}^\intercal \begin{pmatrix} \vec{b_1} & \vec{b_2} & \cdots & \vec{b_n} \end{pmatrix}$$

$$= \begin{pmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} & \cdots & \vec{a_1} \cdot \vec{b_n} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} & \cdots & \vec{a_2} \cdot \vec{b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{a_n} \cdot \vec{b_1} & \vec{a_n} \cdot \vec{b_2} & \cdots & \vec{a_n} \cdot \vec{b_n} \end{pmatrix},$$

where $\vec{a_i}$ is a row vector of matrix A and $\vec{b_i}$ is a column vector of B. This is a complex algorithm that does not fit any fundamental skeleton. However, we can see that it contains repetitive patterns. Each element of the result matrix is an inner product of two vectors. Thus, we can divide it into two levels. The top level is a two-dimensional *Crowd* skeleton, because there are no data flows between these elements. The lower level is the vector inner product function. This function suits a *Recursively partitioned* skeleton, with "a" and "b" nodes in Figure 5 configured as multipliers and adders. They are predefined operators that can be found in the primitive circuit library.

To implement the matrix multiplication, we need to build the system bottom-up. First, we declare instances of the adder and the multiplier. Subsequently, we instantiate a *Recursively partitioned* skeleton based on these primitive circuit library elements. Constraints can be applied to the skeleton if necessary. Finally, we build a two-dimensional *Crowd* skeleton on top of the inner product and generate SystemC codes. We assume both matrices are 16×16, so the vector size of the inner product is also 16. Figure 7 represents the generated system. The symbols "×" and "+" stand for multipliers and adders while dashes between them are communication paths. Each subsystem, as shown in the dashed box, has a detailed layout following the H-tree pattern. If an application cannot fit any existing skeleton, it is necessary to develop a new one. In this case, the skeleton repository should be extended.

In a similar way, we can calculate all the results of discrete convolution on range $[-M,\ M]$:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m] \cdot g[m],\ n \in [-M,\ M],$$

where f and g are two functions, and n is a variable. we can use the *Systolic* skeleton as the low level, and instantiate a one-dimensional *Crowd* skeleton on top of that. An instance of the *Systolic* skeleton produces the result of one input value, so all the results can be acquired with multiple instances simultaneously.

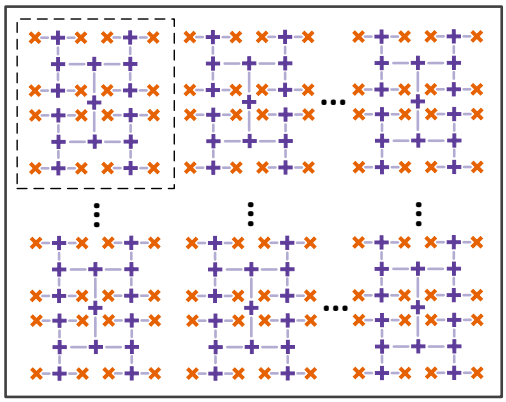


Figure 7: Multiplication of Two 16×16 Matrices

3.3 Parallel Simulation Support

The standard SystemC implementation [24] does not support parallelism. It limits the performance and scale of the simulation since the resources on a single machine are finite. Therefore, we add parallel simulation support in code generation for acceleration and for enlarging the system scale.

Different skeletons require different support strategies. Figure 5 illustrates one possible parallelization method for each skeleton. The parts marked with dotted boxes can be simulated in parallel by different threads (possibly on different machines). The sizes of these boxes are decided by the number of available threads. The communication between these threads is implemented with MPI (Message Passing Interface).

4 Experimental Results

To validate the design flow, we first demonstrate the approach for vector inner product. Thereafter, we analyze its scalability while considering not only the inner product but also matrix multiplication and convolution. Finally, the ability of the proposed approach to perform large simulation will be shown. It is worth noting that these experiments are performed on a high-performance computer with 20 Intel Xeon E5-2670 HT cores, running at 2.50 GHz.

4.1 Placement and Routing Results

We use the graphic output of inner product to show the placement and routing results. We configured the *Recursively partitioned* skeleton with multipliers and a sub-system, which is the combination of an adder and

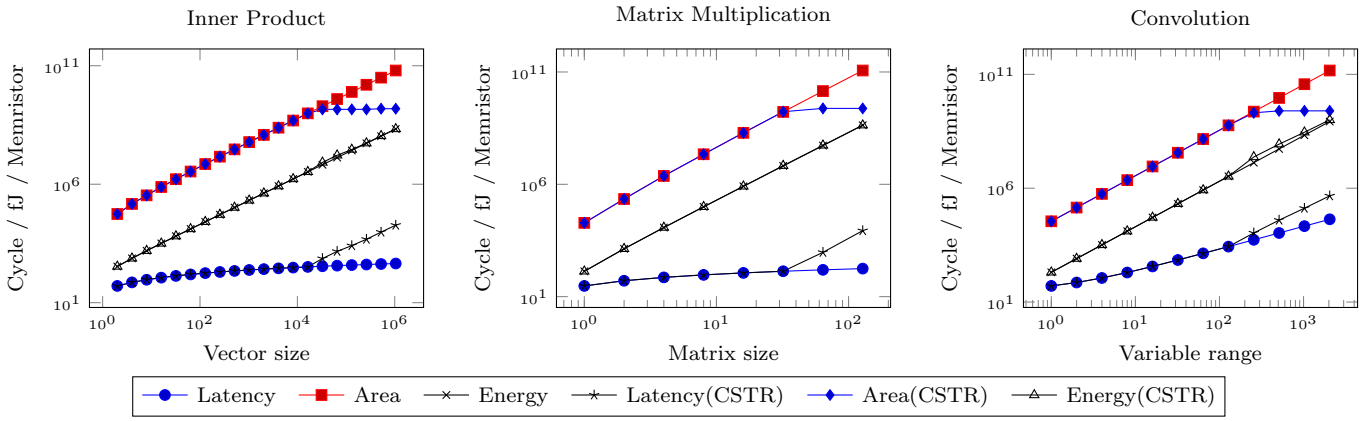


Figure 8: Experimental Results

Table 1: Primitive Circuit Attributes

Module	Latency/CC	Width	Height	Energy/fJ
Adder	20	80	100	67
Multiplier	30	120	160	134
Register	1	33	33	1

two registers. Registers are needed to change the orientations of the input ports so that the adders and the multipliers can be arranged in a H-tree style. The attributes of these primitive circuits are listed in Table 1 [2, 18, 19]; they are synthetic data used only for illustration purpose. Here, the latency is the number of clock cycles (CC) between the inputs and the corresponding output. The width and height are expressed in the number of memristors. The energy is valued for producing one (set of) result(s) in terms of femtojoule (fJ). Figure 9 shows the graphic output generated by our flow when the vector size is 16. Adders and multipliers are marked with "A" and "M" while registers are squares without labels. The input ports (orange triangles) are aligned with the output ports (violet triangles), and the circuit is mapped according to the H-tree pattern. The graphical output allows us to verify that the placement algorithm defined by the skeleton works correctly.

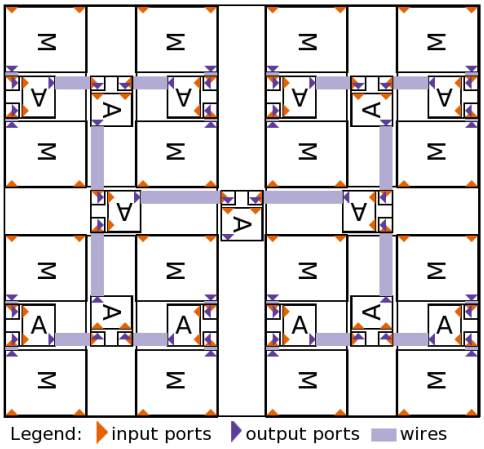


Figure 9: Graphic Output of Inner Product

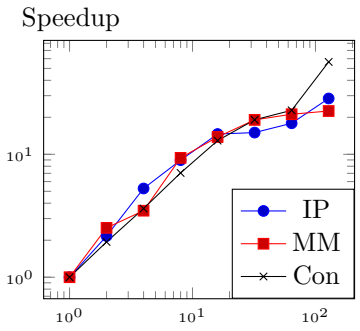
4.2 System Scaling

We varied the system sizes to evaluate the scaling capabilities without putting any constraint, and generated three cases: inner product, matrix multiplication, and convolution; we assume the matrices to be square $N \times N$. The results are shown in Figure 8 for Latency, Area, and Energy. The area is defined based on the total number of memristors used by the implementation. The time complexities of the inner product and matrix multiplication are $O(\log N)$ while that of the convolution is $O(N)$. They are confirmed by the experimental results.

Next, we put area constraints to investigate the flow's ability to handle them; the constraint specifies that the width and height should not exceed 50,000 memristors each. We have to point out that this is only a hypothetical setting. In reality, a CIM chip could be much bigger than this. These experiment results are also shown in Figure 8; they are marked with CSTR in the legend. Comparing these with those for which no constraints was assumed, we can see that the trends of latency and area change. The area stops growing, which shows that the constraint is applied. However, the latency grows in a polynomial manner due to hardware reuse.

4.3 Parallel Simulation

We enabled the parallel simulation support to examine its effect. First, we simulated the baselines which are based on sequential simulations. The results are shown on the right side of Figure 10; it lists the simulation sizes and the corresponding simulation time. The abbreviations IP, MM, and Con stand for Inner Product, Matrix Multiplication, and Convolution respectively. Thereafter, we fixed the system size and changed the number of MPI nodes. For each configuration, we performed the



Alg	Size	Base/s
IP	2^{18}	590.1
MM	64	275.4
Con	256	718.4

Figure 10: Speedup of Parallel Simulation

simulation ten times and calculated the average execution time after removing the maximum and minimum values. Figure 10 shows the speedup of each configuration over the sequential simulation as the baseline. The output data of all the parallel simulations are verified and found to match those of the sequential one. When MPI nodes are less than 16, the speedups are almost the same as the thread number. This result shows a good scalability.

5 Conclusion and Future Work

In this work, we explained why a skeleton-based flow is required for CIM, and we presented a high-level description of this flow with a focus on the collaboration between different designers. We extend hardware skeletons with routing information. An extended skeleton provides scheduling, placement, and routing algorithms for a class of problems that have similar structures. With composition operations, complex skeletons can be built from simple ones.

In future work, we will address each of the specific tool-chain compilers in detail.

References

[1] Y. Pershin and M. Ventra, "Memcomputing: A computing paradigm to store and process information on the same physical platform," in *IWCE*, 2014, pp. 1–2.

[2] S. Hamdioui, L. Xie, H. A. D. Nguyen *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *DATE*. EDA Consortium, 2015, pp. 1718–1725.

[3] L. O. Chua, "Memristor-the missing circuit element," *Circuit Theory, IEEE Transactions on*, vol. 18, no. 5, pp. 507–519, 1971.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart *et al.*, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[5] H. A. D. Nguyen, L. Xie, M. Taouil *et al.*, "Computation-in-memory based parallel adder," in *NANOARCH*, July 2015, pp. 57–62.

[6] C. L. Seitz, "System timing," *Introduction to VLSI systems*, pp. 218–262, 1980.

[7] L. Xie, H. A. D. Nguyen, M. Taouil *et al.*, "Interconnect networks for memristor crossbar," in *NANOARCH*, July 2015, pp. 124–129.

[8] S. H. Gerez, *Algorithms for VLSI design automation*. Wiley New York, 1999, vol. 8.

[9] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[10] M. Zandifar, M. Abdul Jabbar, A. Majidi *et al.*, "Composing algorithmic skeletons to express high-performance scientific applications," in ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 415–424.

[11] C. Nugteren and H. Corporaal, "Bones: An automatic skeleton-based c-to-cuda compiler for gpus," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 35:1–35:25, Dec. 2014.

[12] Y. Wang and Z. Li, "Gridfor: A domain specific language for parallel grid-based applications," *International Journal of Parallel Programming*, pp. 1–22, 2015.

[13] M. Goli and H. Gonzalez-Velez, "Heterogeneous algorithmic skeletons for fast flow with seamless

coordination over hybrid architectures," in *PDP*, Feb 2013, pp. 148–156.

[14] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid, "High level programming for fpga based image and video processing using hardware skeletons," in *FCCM '01*, March 2001, pp. 219–226.

[15] K. Benkrid and D. Crookes, "From application descriptions to hardware in seconds: a logic-based approach to bridging the gap," *VLSI*, vol. 12, no. 4, pp. 420–436, April 2004.

[16] K. Eshraghian, K. R. Cho, O. Kavehei *et al.*, "Memristor mos content addressable memory (mcam): Hybrid architecture for future high performance search engines," *VLSI*, vol. 19, no. 8, pp. 1407–1417, Aug 2011.

[17] E. Linn, R. Rosezin, S. Tappertzhofen *et al.*, "Beyond von neumann - logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, p. 305205, 2012.

[18] L. Xie, H. A. D. Nguyen, M. Taouil *et al.*, "Fast boolean logic mapped on memristor crossbar," in *ICCD*, Oct 2015, pp. 335–342.

[19] S. Kvatinsky, G. Satat, N. Wald *et al.*, "Memristor-based material implication (imply) logic: Design principles and methodologies," *VLSI*, vol. 22, no. 10, pp. 2054–2066, Oct 2014.

[20] H. Lee, Y. Chen, P. Chen *et al.*, "Low-power and nanosecond switching in robust hafnium oxide resistive memory with a thin ti cap," *EDL*, vol. 31, no. 1, pp. 44–46, Jan 2010.

[21] D. K. Campbell, "Clumps: a candidate model of efficient, general purpose parallel computation," Ph.D. dissertation, University of Exeter, 1994.

[22] M. A. Riepe and K. A. Sakallah, "Transistor level micro-placement and routing for two-dimensional digital VLSI cell synthesis," in ser. ISPD '99, New York, NY, USA: ACM, 1999, pp. 74–81.

[23] A. L. Fisher and H. Kung, "Synchronizing large systolic arrays," in *1982 Technical Symposium East*, 1982, pp. 44–52.

[24] Accellera Systems Initiative, SystemC. [Online]. Available: http://accellera.org/