ComPRIMe: A Compiler for Parallel and Scalable ReRAM-based In-Memory Computing

Steffen Frerix*, Saeideh Shirinzadeh†, Saman Fröhlich*, Rolf Drechsler *†
*Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany
†Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
Email: {frerix, froehlich, drechsler}@uni-bremen.de, Saeideh.Shirinzadeh@dfki.de

***Abstract*—In-memory computing is a promising solution for the issue of memory bottleneck in current computing systems. ReRAM is a non-volatile memory technology which natively implements basic logic operations and therefore enables to perform computational tasks. This allows to realize post von Neumann computer architectures with merged memory and processor. In this paper, we propose a fully automated compiler using and-inverter graphs (AIGs) for a conventional in-memory computer architecture which supports parallel computation within regular ReRAM crossbar arrays. The proposed synthesis scheme optimizes crossbar mapping to increase parallelism and lower the number of memory reads and allocated ReRAM devices which results in considerable reductions in latency and area of in-memory implementations. Experimental results reveal minimum speed-ups of factor 2 compared to recent works while consuming a fraction of the ReRAM devices.**

I. Introduction

Today's von Neumann based computer architectures suffer from a large gap in performance of memory and processor due to substantially different improvement rates [1]. This condition known as *memory wall* is the primary issue in the performance of current computing systems challenged by applications dealing with big amounts of data. In-memory computing alleviates this problem by abolishing the need to the power hungry and time consuming communication process between memory and processor as the storage, computation, and updating are all performed in the same unit.

The intrinsic abrupt switching capability of non-volatile memory technology devices such as resistive random access memory (ReRAM) [2] is a promising candidate for realizing in-memory computing architectures within standard memory arrays. ReRAM is a nano-scale two-terminal memristive device [3] whose internal resistance can be switched between two high and low states designating binary states. Computation within ReRAM devices has been performed by means of different primitive logic operations. In [4], it was shown that material implication can be used in memristive devices to execute Boolean functions. Logic design within ReRAM arrays is also performed using memristor aided-logic (MAGIC) NOR gates with an arbitrary number of input variables which are each stored as a resistance value [5]. In [6], a majority operation of three with negation, i.e. $\mathrm{RM}_3 = \langle x, \bar{y}, z \rangle$, was shown to be natively executable in ReRAM based on the logical states of its terminals and internal resistance. RM_3 is used as the basic memristive operation in this work as it can be exploited efficiently on ReRAM crossbar arrays by applying appropriate voltage levels to bitlines and wordlines.

Besides non-volatility and resistive switching property, ReRAM provides high scalability, zero standby power, and CMOS compatibility. This allows implementation of massively parallelized hybrid in-memory architectures where computational tasks are performed within banks of regular memory arrays each corresponding to an independent processing unit. However, computing a function in such in-memory architectures is performed through sequences of instructions which have to be executed within a number of cycles. This is in contrast to conventional CMOS circuits where a combinational task is carried out by electrical interaction of components causing only a propagation delay. This makes it critical to manage computational and read instructions in an efficient manner by maximizing parallelism and therefore reducing the length of the instruction set, which is the motivation behind this paper.

In this paper, we propose an efficient automated compiler for an in-memory computer architecture based on regular resistive crossbar arrays. The proposed approach translates arbitrary Boolean functions represented by and-inverter-graphs (AIGs) into logic-in-memory instruction sets which are executable by applying appropriate voltage levels to bitlines and wordlines of a standard ReRAM crossbar. The proposed compilation procedure allows execution of bit-level parallel computational instructions and lowers the number of required reads considerably. Comparisons with the state-of-the-art show significant improvements with respect to latency and area of the resulting implementations.

II. Preliminaries

A. Graph-Based Representations

Graph-based representations of Boolean functions play an important role in synthesis. AIGs are a state-of-the-art structure for synthesis of Boolean functions used in this work. In order to keep this paper self-contained, we also briefly introduce MIGs.

1) AIGs: AIGs [7] are representations of Boolean networks. The edges represent wires between two input AND gates which correspond to the nodes. Additionally, the edges can be complemented to represent inverters between the nodes. Each terminal node of an AIG corresponds to an input of the boolean network, while each output node corresponds to an

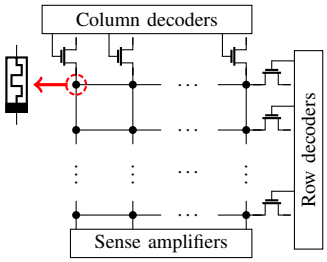


Fig. 1: ReRAM crossbar array

output. Due to their scalability, AIGs play an important role in logic synthesis.

2) MIGs: MIGs [8] are a graph-based representation of Boolean functions, where each node represents the three input majority operation

$$\langle x, y, z \rangle = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)\,.$$

Edges between nodes represent the connections between majority operations and can be complemented. MIGs contain any AND/OR/Inverter graphs including AIGs and allow for a compact representation.

B. Logic Synthesis with ReRAM

1) Manipulation of ReRAM cells: ReRAM cells are usually ordered in memory crossbar arrays, where each cell is connected to a bitline and a wordline. In order to change the inherent resistive value of a cell, a Voltage V has to be applied to the corresponding bit- and wordlines.

A memory crossbar array is depicted in Fig. 1. The columns correspond to bitlines and the rows to wordlines. In order to change the resistive state of a single ReRAM cell in such an array, the authors of [9] have introduced a $V/2$ scheme. To set a ReRAM cell into a low resistive state, the corresponding bitline is driven to $-V/2$ and $V/2$ is applied to the corresponding wordline using voltage drivers. In order to not change the resistive state of any other ReRAM cell in the crossbar array, all other bitlines and wordlines are driven to *GND*. This way, the voltage difference between the bitline and the wordline of the other cells is below V and thus not high enough to change their state. Putting a cell into a high resistive state can be done in a similar fashion by applying $V/2$ to the bitline and $-V/2$ to the wordline. $V/2$ and $-V/2$ can be interpreted as logic values 1 and 0, respectively.

2) Resistive Majority Operation RM_3: A single ReRAM cell can be seen as a two terminal device with the terminals *P* and *Q* and the internal resistance state *Z*. A low resistance state is identified with logic value 1, while a high resistance state is identified with logic value 0. In order to perform a resistive majority operation, logic values are applied as voltages to P and Q. This results in the computation of $\mathrm{RM}_3(P, Q, Z) = \langle P, \bar{Q}, Z \rangle$ and the result being stored as the new resistive value $Z_{res} = \mathrm{RM}_3(P, Q, Z)$ of the same cell (c.f. [6]). When placed in a crossbar array, the bitline is connected to Q and the wordline is connected to P. The device performing the computation is called *host cell*.

Being able to implement RM_3 allows ReRAM crossbar arrays to compute any given function since the RM_3 operation is universal. Of particular importance is the direct assignment of the value X to a ReRAM cell Y and the inversion of X into Y. Both can be performed by two RM_3 operations, respectively:

Inversion		Assignment	
$Y \leftarrow \mathrm{RM}_3(1, 0, Y)$	$//Y = 1$	$Y \leftarrow \mathrm{RM}_3(1, 0, Y)$	$//Y = 1$
$Y \leftarrow \mathrm{RM}_3(0, X, Y)$	$//Y = \overline{X}$	$Y \leftarrow \mathrm{RM}_3(X, 1, Y)$	$//Y = X$

III. Related Work

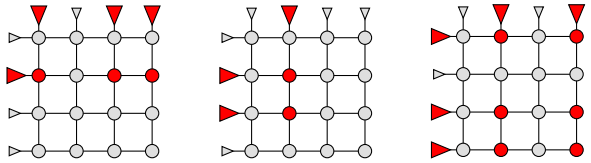
This section studies some of the state-of-the-art approaches for synthesis and optimization of in-memory computing systems using resistive crossbars.

A Programmable Logic-in-Memory (PLiM) computer architecture was proposed in [6]. PLiM is a fully-programmable in-memory computing system, which uses ReRAM as both memory and computational unit. It utilizes a lightweight controller which manages the operations performed on ReRAM arrays which are all implemented in majority-oriented logic (i.e. using RM_3 and complement operations).

In [10], an MIG-based compiler for the PLiM architecture is introduced. Since MIGs are not canonical, different MIGs for the same Boolean function result in different PLiM programs. The compiler proposed in [10] uses MIG rewriting techniques and a translation algorithm to optimize the PLiM program with respect to the expected number of PLiM instructions and required ReRAM cells. As the PLiM can handle only a single instruction at each cycle, the length of instruction set is strongly affected by the number of MIG nodes. Accordingly, the synthesis procedure in [10] first optimizes the target MIGs [11] and then selects a node traversal order to lower the costs in terms of latency and number of required devices. The compiler only considers computational instructions and does not aim at reducing the reads.

A synthesis approach for a very long instruction word (VLIW) architecture using crossbar ReRAM called ReVAMP was proposed in [12]. ReVAMP uses the RM_3 operation similarly to PLiM. ReVAMP supports both read and computational instructions and allows parallel execution of computations sharing one operand which have to be performed in the same wordline. Accordingly, a larger crossbar dimension with a higher number of columns/bitlines can potentially speed up the resulting in-memory operations. However, a large number of bitlines is not always useful as the maximum possible parallelization is limited by the number of concurrent shared operands. This effect has been studied in [12] for an MIG-based delay-focused crossbar mapping in which the latency is lowered by allocating sufficient area and number of ReRAM devices.

A framework for synthesis and in-memory mapping of logic execution (SIMPLE) was proposed in [13] which performs computations through MAGIC NOR gates [5]. MAGIC provides stateful logic and hence does not need read instructions. The latency in this case is only caused by computational steps. SIMPLE starts with an optimized netlist for NOR gates representing the target Boolean function and maps it to a memristive



(a) Wordline (b) Bitline (c) Mixed

Fig. 2: Examples for three kinds of parallel computation. Active devices are highlighted in red.

crossbar considering location and timing constraints optimally such that the latency is minimized. As this approach uses an exact algorithm, it has very low scalability so that the runtime for functions with less than 10 input variables is reported to take several days [14].

Most recently, MAGIC has been utilized in a heuristic design methodology which aims at latency, area and layout optimization [14]. The input and output devices for each gate are placed in the same column. Computation proceeds from the bottom right of the crossbar array to the upper left corner while trying to lower *copy* operations. In this case, mapping forms a semi-staircase placement of gates on the crossbar. This leaves a large number of spare devices. Although the approach aims at keeping the crossbar dimensions close to a square shape, it fails to efficiently use the allocated space.

IV. Parallel In-Memory Computing

A. Types of Parallelism

Parallel computations may be performed on a ReRAM array by addressing multiple wordlines or bitlines simultaneously. We say that a computation is *wordline parallel* if it uses one wordline and multiple bitlines; we call it *bitline parallel* if it uses one bitline and multiple wordlines; and we call it *mixed parallel* if it uses multiple wordlines and multiple bitlines.

Fig. 2 shows an example for each type of parallelism. Row and column drivers are represented by triangles; active devices are shown in red and slightly enlarged. The ReRAM cells in which a computation takes place are highlighted in red.

Mixed parallel computation has to deal with data distortion; even the small computation in Fig. 2c cannot be performed if only one of the six activated cells contains a value that must not be overwritten. Since the problem becomes more severe as more lines are activated, it is unfeasible to make efficient use of this parallelism in general. Bitline parallel computation cannot deal with inversions efficiently, which are central to most logic representations. While wordline parallel computation cannot directly parallelize assignment operations, it can parallelize inversion. Assignment can then be realized by double inversion. Since it offers the most versatility, we focus on enabling wordline parallelism in this paper.

B. Target Architecture

We target a generic in-memory computing architecture that is capable of performing wordline parallel computations. Fig. 3 illustrates the core components. We use a ReRAM array with wordsize w for data storage and computations as well as ρ many registers to keep values that are read from the array. The architecture is able to perform two actions;

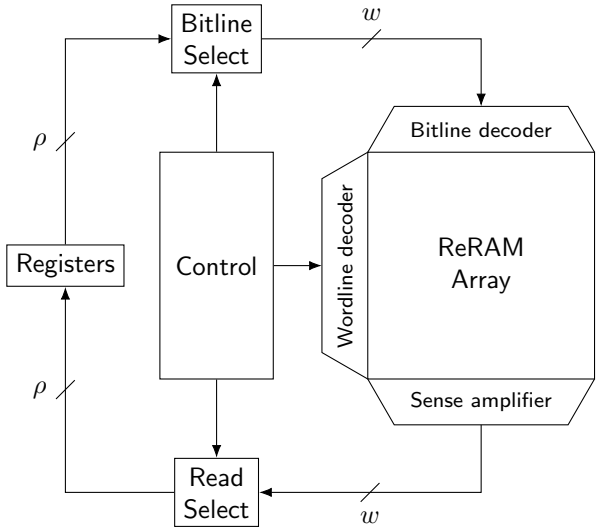


Fig. 3: Generic architecture for wordline parallelism

1) Read from the array: We access the bits of a whole word in one array access. The Read Select circuitry directs required bits to specified registers. In this way, one may accumulate information with multiple reads before engaging in computation.

2) Perform computation: The logical values stored in the registers are applied as voltages to the bitlines. Bitline Select chooses which bitlines are activated and which register contents are applied to them. In contrast to previous in-memory computation schemes, we only apply logical constants to the wordlines; therefore, we do not need circuitry to connect the wordlines to registers.

We deliberately abstract over the way that the instructions representing these actions are stored, fetched and managed. One may store instructions on the same array that is used for computation, as is done by PLiM [6]; or one may use a dedicated instruction memory that enables pipelining for better overall performance, as is done by ReVAMP [12]. Both the instruction size and the hardware cost of such an architecture are comparable to that of ReVAMP.

For simplicity, we will assume that the number of registers ρ is equal to the wordsize w in the remainder of this paper. This ensures that we do not have to worry about register limitations; we are prepared for the limit case in which the contents of w different ReRAM cells are read out and applied to the bitlines.

V. Compilation

A. Enabling parallelism

Synthesis for RM_3-based in-memory computing has made ample use of MIGs for both sequential as well as parallel architectures [10], [12]. When computing two MIG-nodes in parallel within a wordline we have to respect four constraints:

1) All children of both nodes must be computed. In particular, there must not be any data dependencies between the nodes.

2) The nodes must share a wordline operand.
3) The host cells of the nodes must be placed in the same wordline.
4) Since it is overwritten, the content of the host cells must not be necessary for any other computations.

While the first constraint can be dealt with by levelization, the last three constraints severely hinder parallelization efforts for general MIGs. We therefore propose the compiler ComPRIMe that focuses on enabling parallel computation by accommodating these constraints.

Every AIG can be represented as an MIG where each node has a constant zero child. This is an ideal situation for parallelization, since it completely alleviates Constraint 2. In contrast to previous approaches, ComPRIMe uses AIGs as logic representation for the synthesis of parallelly computed logic. When computing an AIG node, we always apply logical zero to the wordline.

Each node of the AIG is associated with a ReRAM cell on the crossbar which holds its content. In order to avoid initialization of a new device, previous approaches tried to reuse these cells as hosts to a computation. However, this reuse introduces sequential dependencies even within one level of an AIG in the form of Constraint 4. Moreover, the placement of a single node may determine the placement of nodes in several later levels that inherit its position. This inheritance obstructs satisfaction of Constraint 3. In general, while reusing cells as hosts is important to reduce copy operations, it is detrimental to enabling parallelism. Therefore, we abandon such reuse altogether. Instead, ComPRIMe allocates a new cell for each computation and initializes it with the necessary host value. In this way, we have full control over the placement of all computations and alleviate Constraint 4 completely.

This comes at the cost of extra work since we need to transfer the data of each host node into a new ReRAM cell. As was explained in Section IV-A, multiple inversions can be performed in parallel within a wordline. Therefore, ComPRIMe uses inversion as initialization operation.

B. Host and Bitline Operands

Each AIG node has two outgoing, possibly complemented edges. One of them will be used as host operand, i.e. loaded into the ReRAM cell which will perform the computation, and one will be used as bitline operand, i.e. applied as a voltage to the bitline. Since we use inversion as initialization operation, complemented edges are preferred as host operands. On the other hand, since resistive majority naturally inverts its bitline operand, it is beneficial for the second operand to be inverted as well; ideally all operands are inverted. Since this cannot be assumed for a general AIG, we need to perform extra computations to obtain the inverted values. Fortunately, these computations are inversions and can therefore be parallelized efficiently. Moreover, ComPRIMe takes care that each inverse value is computed at most once.

The decision which outgoing edge is used as host operand and which is used as bitline operand does not affect the number of computations but only the number of reads. As we focus on computations in this paper, we simply make this decision before any instruction is generated. The decision is made in an arbitrary fashion; as we are faced with a symmetric situation in which both host and bitline operands need to be read out, simple heuristics do not yield any benefit.

C. Managing Allocations

We wish to keep the number of required ReRAM cells as low as efficient parallelism allows. To this end, ComPRIMe keeps track of whether the content of a cell is still needed and frees the cell if possible. We call the set of free cells in a word a *hole*. The cells in a hole can be reused instead of allocating new ones. However, filling a small hole in a word is computationally inefficient since we forsake the opportunity to perform the computations in parallel with others. Therefore, we introduce a parameter h to the compilation; ComPRIMe will only try to reuse holes that are not smaller than h.

D. Placing Nodes

Each node in our logic representation (and potentially its inverse value) must be assigned to a ReRAM cell on the crossbar once it is to be computed. These placements majorly influence the efficiency of the computation; all nodes placed in the same word may be computed in parallel. Moreover, by placing nodes in the same word whose host operands reside on the same wordline, we reduce the read cost for initializations since their host operands can be read simultaneously.

Assume we are given a set N of nodes to be placed and further assume that their children have already been placed on the crossbar. We split N into equivalence classes $N_1, \ldots, N_n$, where all members of N_k have host inputs that reside on the same wordline. These are then again split into groups of size at most w. Next, we inspect holes on the crossbar. If a hole is larger than h, we try to fill it. A hole is filled greedily with fitting groups, where larger groups are considered before smaller ones. All groups that remain are placed on newly allocated words (with the same filling algorithm).

We assume that the primary inputs to the function we compute are present in the memory array that we are computing on. As we never reuse a ReRAM cell as host, computation of a function does not destroy its inputs. Therefore, primary inputs do not have to be copied. However, if they are not tightly packed into words, the number of reads may increase since the inputs cannot be read in parallel. It may then pay off to rearrange them into words. Assuming the worst case, in which each input must be read sequentially, the cost for this operation is $I + 5\left\lceil \frac{I}{w} \right\rceil$, where I is the number of primary inputs; we need to spend one array access on reading each input as well as four computations and one read per w many inputs on double inversions.

This preprocessing step is superfluous when processing wordlevel data, in particular with arithmetic functions. The output level of a function can be written into words again without additional cost, allowing us to chain multiple wordlevel functions without data movement.

E. Computing an AIG

The AIG levels are computed successively by following three steps for each one. First, place and compute the inverted

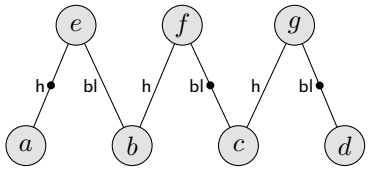


Fig. 4: Example AIG

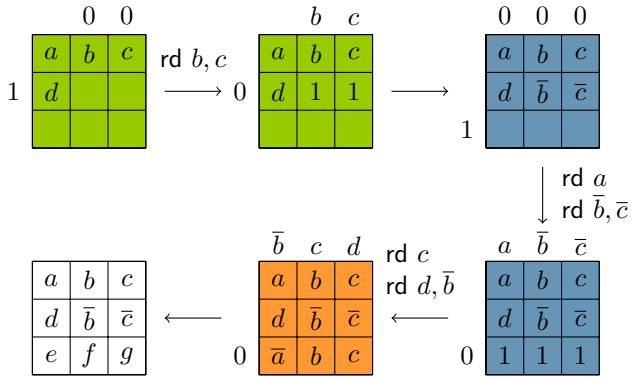


Fig. 5: Computation of the function in Fig. 4.

values necessary for this level. Second, place the nodes of the level and initialize the cells with the corresponding host operands. Third, compute the nodes themselves by scheduling the computations that have host cells in the same word in parallel. Reads are scheduled as required by the computations.

Example. We demonstrate the computation of an AIG level with an example. Consider the AIG in Fig. 4. We compute the upper level. To the left of each edge it is indicated whether the edge is used as host operand (h) or as bitline operand (bl). Fig. 5 illustrates the computation on a 3x3 ReRAM array by displaying the six intermediate states. Operands applied to bitlines and wordlines are indicated outside of the array. We start the computation on the top left, where the lower level of Fig. 4 is already present. The three steps discussed in Section V-E are indicated by the background colours green, blue and orange respectively.

1) Compute inverted values: The inverted values necessary for this level are $\overline{b}$ and $\overline{c}$. They will be placed in the second word. Logical one is written to the respective cells. Then we read out (rd) the values of b and c from the first word to write their inverses.

2) Initialize host cells: The three nodes to be computed will be placed in the third word, to which we write logical one. Then we read out the values of a, $\overline{b}$ and $\overline{c}$ before applying them to the bitlines. Since the operands are spread across two words, we need to spend two read operations.

3) Compute level: The bitline operands $\overline{b}$, c and d are read out with two reads and and applied to the bitlines.

VI. Experimental Results

All results shown use wordsize $w = 16$ and hole parameter $h = 12$, unless stated otherwise. We evaluate instruction sets generated by ComPRIMe in terms of delay and area. Delay (D) is expressed as the total number of array accesses, which may be computations (C) or reads (R). Area is expressed as total number of ReRAM devices needed by the program (Z). We compare our results with ReVAMP [12] as well as with a state-of-the-art synthesis approach using MAGIC [14]. Subscripts are used to indicate which approach produced a result, where c represents ComPRIMe, r represents ReVAMP and m represents the synthesis approach using MAGIC. To ensure a fair comparison we report D_c^*, which is the total delay produced by ComPRIMe increased by the worst case input arrangement cost discussed in Section V-D.

A. Comparison with ReVAMP

We evaluate ComPRIMe on benchmarks provided by EPFL[1] and compare with ReVAMP that ran the same benchmarks in [12]. Table I shows the results. AIG versions were included in the benchmarks and could be readily used for instruction generation with ComPRIMe. We achieve a significant speedup of factor 4.79 regarding computations and a speedup of 1.70 regarding reads. Taking input placement into account, the solution generated by ComPRIMe is faster by a factor of 2.32 while at the same time taking only about a third of the area. It is worth noting that while ReVAMP does take the cost for arranging inputs on the array into account, it does not account for the time it takes to read these values in the first place. These results further corroborate that the MIG, which is used by the ReVAMP synthesis as logic representation, fails to enable the parallel computational power that is provided by ReRAM array computing. Moreover, we see that the dynamic ReRAM allocation used by ComPRIMe provides significant area benefits over the static allocation used by ReVAMP.

B. Comparison with Staircase Synthesis using MAGIC

The results of evaluating ComPRIMe on the combinational functions of the ISCAS benchmark suite [15] are shown in Table II. The benchmarks were optimized with ABC [16] before instruction generation. We compare ComPRIMe with a state-of-the-art synthesis approach using MAGIC [14]. Each MAGIC operation takes two array accesses; one for initializing the output device and one for the actual computation. We report the total delay D_m in terms of array accesses. We can observe a speedup of factor 2.13 while using less than $\frac{1}{6}$ of the number of devices. This ratio only considers the ReRAM devices taking part in the computation. However, the Staircase synthesis has to allocate an area that can be more than an order of magnitude larger than the number of utilized ReRAM cells in order to compute the benchmarks [14]. Furthermore, it does not account for the cost of arranging inputs in the exact needed place on the array.

C. Scalability

Fig. 6 compares the performance of ComPRIMe for different wordsizes w and hole parameters h. The numbers shown are total delay D_c^* and total number of ReRAM devices Z_c needed for all benchmarks of Table I combined. The results show that ComPRIMe can be scaled up to achieve

[1] http://lsi.epfl.ch/mig

TABLE I: Comparison with ReVAMP

Benchmark	C_c	C_r	speedup	R_c	R_r	speedup	D_c^*	D_r	speedup	Z_c	Z_r	improv.
pci_spoci_ctrl	300	1523	507%	868	1328	152%	1283	2851	222%	656	1360	207%
revx	2639	14575	552%	7366	14004	190%	10035	28579	284%	512	9776	1909%
sqrt32	1414	4216	298%	1785	3948	221%	3241	8164	251%	240	2720	1133%
square	5125	30988	604%	17472	29880	171%	22681	60868	268%	6992	23264	332%
des_area	1356	5971	440%	2875	5639	196%	4714	11610	246%	1904	4880	256%
ac97_ctrl	2572	8803	342%	5579	7520	134%	11111	16323	146%	7392	14928	201%
hamming	986	3603	365%	2205	3450	156%	3456	7053	204%	624	2880	461%
tv80	2245	11219	499%	6696	10368	154%	9434	21587	228%	2912	9248	317%
mem_ctrl	2698	9497	352%	7709	8405	109%	11980	17902	149%	4128	10000	242%
i2c	306	1450	473%	754	1247	165%	1257	2697	214%	768	1328	172%
ss_pcm	114	313	274%	151	257	170%	406	570	140%	336	496	147%
usb_phy	118	538	455%	303	460	151%	574	998	173%	400	592	148%
max	1209	4431	366%	1527	4184	274%	3408	8615	252%	1536	5024	327%
spi	928	4615	497%	1960	4301	219%	3252	8916	274%	1504	4272	284%
sasc	183	602	328%	350	514	146%	711	1116	156%	560	912	162%
div16	1854	8375	451%	4741	7825	165%	6637	16200	244%	544	5472	1005%
MAC32	2642	15980	604%	8190	15363	187%	10958	31343	286%	3600	12544	348%
pci_bridge32	4393	23826	542%	11225	21914	195%	20237	45740	226%	9440	24736	262%
MUL32	2459	14047	571%	6220	13389	215%	8763	27436	313%	2384	12496	524%
systemcdes	712	3312	465%	1826	3090	169%	2952	6402	216%	1072	3120	291%
comp	6703	32297	481%	17450	31293	179%	24522	63590	259%	5600	18912	337%
systemcaes	2742	11100	404%	5836	10229	175%	9803	21329	217%	4720	11536	244%
simple_spi	227	930	409%	534	794	148%	959	1724	179%	560	1200	214%
usb_funct	3719	16054	431%	11649	14269	122%	17813	30323	170%	7840	16912	215%
$\sum$	47644	228265	479%	125271	213671	170%	190187	441936	232%	66224	198608	299%

TABLE II: Staircase comparison

Benchmark	D_c^*	D_m	speedup	Z_c	Z_m	improv.
c2670	955	1102	115%	464	1513	326%
c6288	2521	7502	297%	544	6369	1170%
c1355	311	472	151%	192	1035	539%
c7552	2002	4364	217%	720	4461	619%
c880	446	854	191%	208	889	427%
c3540	1090	2870	263%	416	2764	664%
c499	311	484	155%	192	1021	531%
c1980	449	1034	230%	192	1087	566%
c432	306	450	147%	112	405	361%
c5315	1831	2722	148%	640	3553	555%
$\sum$	10222	21854	213%	3680	23097	627%

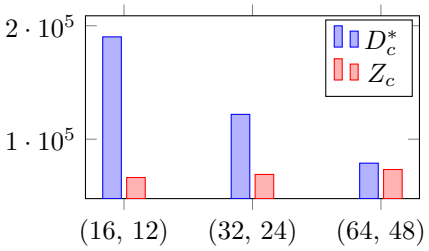


Fig. 6: Performance of ComPRIMe for different values of (w, h).

additional speedup, while the necessary number of ReRAM devices changes only insignificantly. All compilations ran in less than half a second per benchmark on an Intel Core i7-8550U processor with 24GB of RAM.

VII. Conclusion

This paper proposes ComPRIMe, a novel AIG-based compiler for ReRAM-based in-memory computing. The proposed compilation approach is highly scalable, supports parallel execution of computations, and efficiently maps them to ReRAM crossbar addressing both latency and area. We have identified three major constraints that reduce potential parallelism and have presented solutions to alleviate them to a large extend. Comparison of results with two state-of-the-art approaches with bit-level parallel computing ability shows significant speed and area improvements to both.

VIII. Acknowledgement

This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under contract number DR 287/35-1 and by the University of Bremens graduate school SyDe, funded by the German Excellence Initiative.

References

[1] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04, 2004, pp. 162–167.

[2] H. . P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai, "Metaloxide RRAM," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, June 2012.

[3] L. O. Chua and S. M. Kang, "Memristive devices and systems," *Proceedings of the IEEE*, vol. 64, no. 2, pp. 209–223, Feb 1976.

[4] J. Borghetti, G. S. Snider, P. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, pp. 873–876, 2010.

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

[6] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (PLiM) computer," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 427–432.

[7] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[8] L. Amar, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference (DAC)*, 2014, pp. 1–6.

[9] Y.-C. Chen, C. F. Chen, C. T. Chen, J. Y. Yu, S. Wu, S. L. Lung, and R. L. and, "An access-transistor-free (0T/1R) non-volatile resistance random access memory (RRAM) using a novel threshold switching, self-rectifying chalcogenide device," in *IEEE International Electron Devices Meeting 2003*, 2003, pp. 37.4.1–37.4.4.

[10] M. Soeken, S. Shirinzadeh, P. Gaillardon, L. G. Amar, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[11] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for rram-based in-memory computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1422–1435, 2017.

[12] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay, "Crossbar-constrained technology mapping for ReRAM based in-memory computing," *CoRR*, vol. abs/1809.08195, 2018. [Online]. Available: http://arxiv.org/abs/1809.08195

[13] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 225–232.

[14] A. Zulehner, K. Datta, I. Sengupta, and R. Wille, "A staircase structure for scalable and efficient synthesis of memristor-aided logic," in *Proceedings of Asia and South Pacific Design Automation Conference*, 2019, pp. 237–242.

[15] F. Brglez, "A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN," in *Int. Symposium on Circuits and Systems, Special Session on ATPG and Fault Simulation, June 1985*, 1985, pp. 663–698.

[16] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.